Axiomatic Semantics Verification of a Secure Web Server

A Dissertation
Presented to the
Department of Computer Science
Brigham Young University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

by
Paul E. Black

February 1998

This dissertation by Paul E. Black is accepted in its present form by the Department of Computer Science of Brigham Young University as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Phillip J. Windley, Committee Chairman

Douglas M. Campbell, Committee Member

J. Kelly Flanagan, Committee Member

William A. Barrett, Committee Member

Parris K. Egbert, Committee Member

Date

Scott N. Woodfield, Graduate Coordinator

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Society is increasingly dependent on systems with computers as vital components. The software used in systems is increasing in complexity in two ways: systems have more components interacting with each other (more breadth) and systems are built on top of other, preexisting systems (more depth). Since computer systems are highly non-linear, an error in one part of the system may lead to a failure in a logically (or physically!) distant part of the system. The current state of practice in software production seems to be design, implement, and test until it is good enough or the project runs out of time. This is increasingly unable to deliver software of acceptable quality on an acceptable budget in many areas.

- Life critical systems: heart pacemakers, airline flight control systems, nuclear reactor shutdown systems.
- High volume systems: a bug in the control software of a laser printer could mean the manufacturer has to replace millions of ROMs all over the country.
- Security systems: systems connected to the Internet are potentially open to automated break-in attempts from millions of sites all over the world. A single break-in can be disastrous. Authentication in the core of an operating system can be critical, too.
- Complex systems: a complicated memory management algorithm may deliver needed performance, but simulation is not enough to assure the designers that it will work.

Formal methods help to manage this complexity, thereby increasing reliability and productivity, by providing formal models which may be checked and compared.

From the time people began writing programs for computers, some people have been concerned with the correctness of their programs and have advanced arguments to justify trust in algorithms or programs. In 1967 Floyd [20] presented a way to assign meaning to computer programs. Hoare introduced axiomatic semantics for formal program verification in his seminal paper "An Axiomatic Approach to Computer Programming" [30]. In the decade following, much was learned about how to verify computer programs, how to use formal methods in the program development process, and where difficulties arise. Manna and Waldinger's paper [35] is an excellent paper from that era. Since then inference rules have been developed for function calls [23], non-local branching [1], distributed [26] programs, functional programs [12], etc. Many of these were developed for simple, invented languages to clearly demonstrate a principle with a minimum of distracting detail.

Dijkstra ([17], pp. 56–57) advocates that code should be designed for clear verification even if that makes failure characteristics worse. However the most reliable systems are built with experience and conservative, almost pessimistic or even redundant, design. Formal methods do not provide guidance about *what* to build: only *how* to build it. Thus formal methods are an adjunct to experienced design. Formal methods consists of three parts: formal models, formal requirements, and formal means to relate the two (verification or proof, synthesis, etc.) [5]. Formal models can help development by unambiguously capturing the design, that is recording details precisely. Verification helps by insuring that the model of an implementation matches the requirements with no overlooked cases, no false analogies, or any hidden assumptions.

However not much code is formally verified today. Formal verification is often rejected as being impractical for "real code," as not applying to the real situations, or as not really helping during software development. De Millo, Lipton, and Perlis (speaking of manually reviewed verifications) said [15],

> [we believe] in the basic impossibility of verifying any system large enough and flexible enough to do any real-world task.

Thesis Statement

Computer-assisted, post-hoc formal verification can be applied to useful, production code written in real languages. Many people consider formal verification to be of little or no value for existing code and inapplicable to real code and real languages because of complications such as the following:

1. If expressions can have side effects (especially function calls), and semantics for a single expression can be arbitrarily complex.

2. Program constructs may have multiple exit points.

3. Using arrays, especially without language-defined bounds checks, has the potential to overwrite memory.

4. Behavior based on I/O operations must be specified with more complicated functions, instead of being able to use mathematics.

5. Operating system calls and library functions, especially those in Unix, have very complex behavior. Some are essentially nondeterministic.

6. Concepts such as security, integrity, and confinement must be formalized for Unix.

7. The Unix file systems and process attributes must be formally modeled.

We claim that that our formal verification of a secure HTTP ("web") server, which has all the complications noted above, tends to refute those concerns. We further show that the verification yielded insights into and improved our understanding of code, such as clarifying specifications and exposing assumptions.

In Chapter 2 we show how this work has grown from and differs from related work. Chapter 5 explains in detail our new approach which allows us to reason about expressions with side effects. We detail our formalization of C, Unix, inference rules, security properties, etc. in Chapter 6 pointing out areas we have not covered. The proof of security of the secure web server, `thttpd`, is outlined in Chapter 7, and we present our results, observations, and conclusions in Chapter 8. Appendix A is a software verification manual and documentation of the support code we used in the proof. Appendix C is the complete code of the secure web server.

CHAPTER 2

RELATED WORK

This dissertation builds on and is related to decades of work by many people. Their work may be divided, with some overlap, into three parts. Section 2.1 describes work on formalizing software semantics and proving theorems about code and algorithms. Section 2.2 relates a complimentary approach of automatically extracting the theorems needed to prove program correctness. Finally sec. 2.3 refers to work on expressions with side effects.

Here we need to introduce some notation. As is typical, we represent an inference rule as

$$\frac{\begin{array}{c}\vdash H_1\\ \vdots \\ \vdash H_n\end{array}}{\vdash C}$$

This means if all of the hypotheses $H_1, \ldots, H_n$ are true, one can conclude $\vdash C$. Any number of hypotheses may qualify one conclusion. Note that an axiom may be stated as an inference rule with no hypotheses, that is, a conclusion which is always true.

2.1 Reasoning About Programs

2.1.1 Floyd

In 1967 Floyd reported on a method to formally infer the correctness of programs. The method represents a program as a directed graph. Program statements are vertices, and possible control transfers are edges. Statements may have many multiple entries and exits. Predicates representing correctness conditions are attached to the edges. The semantic definition of a command c with k incoming or "entry" arcs and l exit arcs is a verification condition function, $V_c(P_1, \ldots, P_k, Q_1, \ldots, Q_l)$ where P_i is the predicate on the i^{th} entry and Q_j is the predicate on the j^{th} exit.

Floyd defines consistency and completeness for verification condition functions (V_c's) and gives theorems about combining V_c's which any set of inference rules for programs must obey. An example shows semantic definitions for restricted assignment, conditional, and other statements. The paper also defines the strongest verifiable proposition for an edge, the strongest verifiable consequence, etc., and shows how to prove termination.

Although the representation of programs as graphs may seem unnecessarily general in the day of structured programming, it may still be applicable. With exceptions and return and break statements, large applications are rarely purely structured.

2.1.2 Hoare

In a landmark paper in 1969 Hoare described a system of using axioms to assign meaning to programs [30]. The paper draws an analogy between this "axiomatic semantics" and axioms of arithmetic, and gives axioms (or axioms schemas) for assignment, consequence (pre-condition strengthening and post-condition weakening), sequence, and iteration ("while" loops) in a simple language. The language excludes side effects and procedure calls.

The paper also mentions the need to consider termination, but gives no guidance on how this might be done. Hoare spends considerable time arguing for the use of correctness proofs for reliability and formal specifications for documentation and portability. In spite of this, he recognizes that more work was needed [30, p. 579]:

> The practice of supplying proofs for nontrivial programs will not become widespread until considerably more powerful proof techniques become available, and even then will not be easy.

2.1.3 Dijkstra

Dijkstra proposed constructing programs by simultaneously deriving and verifying them from the termination condition rather than the "verification afterward" orientation of Floyd and of Hoare. His 1976 book [17] explains how to use his weakest precondition (wp) algebra and a simple nondeterministic specification language to develop software.

This "correct by construction" philosophy is seen today in such things as the Cleanroom Approach [18] and the SPARK Approach [4, p. vii]. The general scheme of computing some form of a weakest precondition from a statement and its postcondition is still often proposed as the bases of software verification.

2.1.4 Gordon

In 1989 Gordon embedded Hoare's axiomatic semantics for a simple language in HOL. The simple language did not have complicating features such as side effects or procedures calls. The package is called "prog_logic," and implements many of the concepts described in [21] about proofs, devising inference rules, and a verification condition generator.

Gordon's embedding is the foundation of axiomatic semantics in HOL upon which many extensions and applications to new situations are based. Although the book is oriented to manual correctness proofs, it is an excellent guide to implementing fundamental inference rules. It also discusses many of the subtle problems which may arise such as inconsistent or incomplete rules.

2.1.5 Gurevich

Gurevich presented [24] a new abstraction, similar to an operational semantics, called "evolving algebras" or "abstract state machines," in 1993. These descriptions can rather naturally describe many of the complex semantics of languages such as C.

The abstract state machines representation has been applied to many problems including software, hardware, protocols, and real-time systems.

2.2 Verification Condition Generators

2.2.1 Manna and Waldinger

In the late 1970's Manna and Waldinger, along with Levitt and Katz, implemented Floyd's method of generating verification conditions and extended it with code which proves most of the conditions automatically [34]. They also showed how many loop invariants can be generated automatically. Invariants are generated by one of two approaches: algorithmic, which generates invariants which are certainly true, but may not be sufficient, and heuristic, which generates invariants which may be helpful, but must be checked to see if they are true.

This work concentrates mostly on automating software verification as much as possible. Obviously, the more automated a verification system is, the more widely acceptable it is.

2.2.2 Homeier

In his 1995 Ph.D. dissertation, Homeier [32] began with an operational semantics of a highly simplified version of C. He then proved the correctness of an axiomatic semantics and several inference rules including new rules for function calls (including mutually recursive functions) and for instantiations of theorems about them. His inference rules also handle an operator with a simple side effect. He clearly explains the need to have separate assertion and programming languages and formalizes an embedding of programming language constructs in the assertion language.

From that formalization he wrote and verified a verification condition generator. Given a piece of code annotated with assertions at key points, a verification generator derives the set of theorems and implications which must be proved to prove the code correct. This approach has the advantage that the user need not be concerned with inference rules for each programming construct, such as assignment statements, loops, and function calls.

This work is noteworthy particularly for the following reasons.

1. The axiomatic semantics are proven correct from an operational (definitional) semantics.

2. The verification addresses some difficult constructs, such as mutually recursive functions.

3. The verification condition generator is proven correct.

2.2.3 Maurer

W. Douglas Maurer developed the Join-Point Method, a derivative of Floyd's inductive assertion method. In a 1996 paper [36], Maurer reports on an operational semantics of C and a verification condition generator based upon the theory. He proved that conditions only have to be established and proved at a number of "join points" in the execution graph of a program. All other conditions can be derived and checked completely automatically.

Maurer's on-going work continues to develop practical means of verifying production programs written in production languages. The verification is highly automated, too.

2.3 Handling Side Effects

One of the semantically difficult features which appears early and often in software verification is handling expressions which have side effects. This section reviews work on reasoning about languages with side effects.

2.3.1 Cunningham and Gilford

In [13] Cunningham and Gilford show how side effects in expressions can be handled by

1. Introducing an axiom schema for the result of evaluating a simple variable or constant, $P\ \{x\}\ P \wedge \Phi = x$, where Φ is the meaning of the expression and cannot appear in P.

2. Giving rules to infer the correctness of, say, assignment statements from simpler rules. For instance, when the right hand side of an assignment is evaluated first, the rule is

   $$\frac{P\{x\}Q \wedge \Phi = u, \quad Q\{y\}R^u_y \wedge \Phi = y}{P\{x \leftarrow y\}R}$$

   where u is a reference to x, and

3. Providing an inference rule schema for functions with any number of parameters.

The example inference rules assume that the order of evaluation of pieces of expressions is defined. They did not show how their approach could be applied to control statements such as conditionals or loops.

2.3.2 Kowaltowski

Kowaltowski gives [33] another method of reasoning about side effects in axiomatic semantics using unique variables to maintain intermediate results. He presents inference rules for simple variables and constants, assignments, unary and binary operators, conditional statements, and iterative statements (`while` loops). The assignment rule is

$$\frac{P\{E\}Q^x_\sigma}{P\{x := E\}Q}$$

where σ is a distinguished variable which is the result of computing expression E.

Cunningham and Gilford's rules allow for side effects in the "left hand side" of assignments, while Kowaltowski allows for relations other than equality between the result-carrying variable (Φ or σ) and other variables and constants.

2.3.3 Boehm

In [9] Boehm presents a logic which deals quite easily with side effects and aliasing. The formalization is similar to a denotational semantics in that the value of a programming expression can be transformed into a mathematical expression. Additionally inference rules allow one to reason directly about programming expressions like Dijkstra's weakest precondition calculus [17]. The logic maintains a strict separation between the assertion language and the programming language.

The logic deals with side effects and aliasing by having separate inference rules for the result value of an expression ("value" rules) and the effect of the expression on the state ("effect" rules). For instance, the value axiom for a boolean operator is

$$<a \oplus b> = <a> \oplus <a; b>$$

That is given an operator $\oplus$, the value of $a \oplus b$ is the value of a "combined with" ($\oplus$) the value of b after a is executed. The effect axiom is

$$<a \oplus b>t = <a; b>t$$

That is, the effect of $a \oplus b$ on state t is the effect of a then b on state t.

Since the logic defines *the* value of an expression in a state, it does have difficulty accommodating nondeterminism or concurrency.

2.3.4 Homeier

Homeier's work is mentioned in this section as well as the preceding section on verification condition generators. His language has an increment operator which has a side effect. The inference rules handle side effects by separately representing the value of an expression and its effect on the state.

The effect and "timing" of the increment operator is explicitly represented in the rules. It is not clear how this approach could be generalized to, say, user-defined functions with side effects.

2.3.5 Norrish

Norrish developed a nearly-complete model of the C language, called Cholera. This is an operational semantics model and is detailed in a 1997 report [38]. Norrish plans to prove an axiomatic semantics from this model and use a combined Cholera and axiomatic semantics system to verify software.

CHAPTER 3

A SECURE WEB SERVER

The subject of this study is a secure World Wide Web server called `thttpd`[1]. Much of this section comes from [7] and [11].

In June 1995, Management Analytics wrote a secure World Wide Web server called `thttpd` and gave thorough, though informal, specifications of security. The code consists of about 100 lines of C. Their

> ...solution to the security problem with servers is to design a secure server with security properties that can be explicitly demonstrated.

They list the general properties of interest as

- information integrity (no information on the server can be corrupted by outside users),
- availability of service (outside users cannot deny services to other users), and
- confidentiality (the server only provides information which is explicitly authorized for outside access).

3.1 Design Features

To assure these properties, they use the following design features.

- Small Size, so it is feasible to thoroughly examine for security properties,
- Confinement of Operating Privileges, using the Unix command `setuid` to run as a non-privileged user (named "www") so

  > ...the basic operating system protection features (e.g., access control, process separation, limited input and output capabilities) that are used by millions of people every day are used to protect the server ...

[1]It is named for HTTP, the HyperText Transfer Protocol, which is used for WWW transfers on the Internet. HTTP is similar to FTP, but is enhanced for Web interaction.

- Confinement of File System Access, using the Unix command `chroot` to avoid access to sensitive or special files, such as password or device files,
- Confinement of File Output, by only writing to a single, predefined log file,
- Confinement of Information Flow, by reading only one fixed size request from the user,
- Confinement of File Inputs, by only opening files which are owned by "www" and "world" readable.

They used proven technology (Unix and C) as a good base to provide a secure server. They felt this makes the server more portable than if it were implemented in, say, SPARK [4]. Also C compilers and Unix operating systems have had extensive operational experience, for more "real world" testing than seldom-used or invented systems have.

The redundant confinements give defense in depth. That is, it is unlikely that any one service or library call failure will comprise the security. The redundancy also makes the server resistant to inevitable operator errors.

After having been examined by dozens of people on the network, having run on the Internet for over a year, and being subjected to hundreds of thousands of unfriendly attack, no flaw has been found.

We feel this would be a good test subject for verification: small, unlikely to have gross errors, written for real use, but having many complications ignored in other work.

3.2 Alternative Subjects

Why get mired in the ugliness of an old, semantically complex language like C at all? Why not invent or use a clean language? Actual programs must be implemented to be used, and therefore depend on actual compilers and libraries, not models. Although a simple, invented language may be easier to verify, it cannot be as widely known or its compiler as widely available as a commercial language, such as C. Short of verifying the compiler, a commercial compiler is probably more reliable than a compiler for a novel language. Additionally verifying an actual program written in a commonly-used language raises interesting issues which are often avoided by definition or never even come to light with simple, invented languages.

Why not use a simple example program, one written to be easy to formally verify, instead of a production program? Programs must be engineered with redundancy in order to not fail in the presence of the inevitably failure in the library functions, operating system, or hardware. In spite of being thoroughly checked and tested, `thttpd` would be even more trusted after a formal verification which yields additional considerations. The web server was written to be quite simple, so we believe it is a good candidate for formal verification.

CHAPTER 4

APPROACH TO VERIFYING THTTPD

The final goal of verification must be to improve the software production process. In our opinion, the best general way to develop software today is using a modified Cleanroom method. This method is based on incremental development, verification by inspection at each increment, and testing for statistical properties rather than testing to debug [18]. In our case, we are interested in code which already exists, so we will use post-hoc verification instead. Not everyone can develop code from scratch in such a productive environment, so post-hoc verification is still useful. Additionally the Cleanroom method can only be fully applied when a complete and rather formal specification is available. In contrast post-hoc verification allows us to formally verify just certain critical properties while engineering those which are less critical.

After one has chosen post-hoc verification one can still ask what should be verified for high level code. Curzon [14] noted several different possibilities.

- One could compile high level (source) code into low level (assembler) code, and verify the low level code.
- One could verify the high level code, compile it, then prove a correspondence between the low and high level codes.
- One could verify the high level code and verify the compiler, i.e., that in every case the low level code it produces implements the high level code.

Curzon choose the third possibility and presented a verified compiler. We will not concern ourselves with a correct compilation or execution of C code. First, compilers and hardware are usually reliable, and well engineered software should minimize any possible exposure from lower-level failures. Second, one would have to verify each compiler and hardware implementation separately. Finally and most importantly, we can formalize parts of the ANSI C standard, not a particular compiler's implementation or hardware's execution.

We must choose a method of verification and model of computation, a verification support system which can support it, and some approach to formalize the specification.

4.1 Possible Verification Methods

Software is usually verified either by model checking or theorem proving. Model checkers explicitly compare the input/output mappings of two functions. In the simplest form, one may wish to decide if two boolean expressions were the same. One might generate and compare the truth tables for each expression. If they match, the functions are the same. Model checking takes this approach while exploiting symmetries and special conditions to reduce the actual number of cases checked. Model checking is appealing since it is decidable, that is, there are efficient, highly automated algorithms for proving correctness. With sophisticated programs, models as large as 10^{20} states have been checked [10].

One particular model which has been used to check security is that of Bell and LaPadula. It defines the idea of a "secure state" and possible transitions from state to state. Checking that all transitions from reachable secure states are to other secure states leads inductively to a proof that the system is secure.

Formal verification can only show the truth or falsity of propositions about models. Since we are limited by laziness (it isn't worth modeling everything which could conceivably happen), theoretical ignorance (we don't know everything), or practical ignorance (we haven't examined everything) [42] in modeling the real world, our models will never exactly correspond to every facet of reality. Because of this we must use probabilistic reasoning, such as fault trees or mean time to failure analysis, if we wished to assure ourselves that the code is good enough even though our components are not perfect.

Theorem proving approaches verification by formalizing [5]

- a model of computation,
- the specification, and
- rules of inference.

The model of computation dictates the axioms, that is, what the effect of atomic executions are. The rules of inference allow us to deduce theorems about larger pieces of code from axioms and previously proven theorems. The verification is then proving that (the model of) the implementation satisfies the specification.

4.2 Formal Reasoning

There are many models of computation: denotational semantics, operational semantics, abstract state machines [24], axiomatic semantics [30], etc. The first three models are definitional, that is, new predicates, functions, concepts, etc. are added by defining them in terms of existing relationships or concepts. The advantage is that the logic cannot be made inconsistent by adding definitions. Also language models are often closer to human understanding of semantics, and thus easier to validate and more likely to be correct. For example, it is relatively clear to define the effect of an assignment statement as a formal version of something like "`A = B + C;` means 'add the contents of B to the contents of C and put the result in A'."

Axiomatic semantics can introduce inconsistencies in the logic through the axioms added. In addition, the axioms may not be very intuitive and therefore harder to validate and get correct. For instance, the axiom (schema) for simple assignment statements is $\vdash \{\mathtt{Q}^{\mathtt{v}}_{\mathtt{expr}}\}\ \mathtt{v} = \mathtt{expr};\ \{\mathtt{Q}\}$. The notation $\mathtt{Q}^{\mathtt{v}}_{\mathtt{expr}}$ denotes `Q` with all free occurrences of v replaced by `expr`. See Sect. 4.4 for more detail.

The definitional semantics tends to develop a complete description of the behavior of a piece of code. Axiomatic semantics allows reasoning about just particular properties. For instance, if we are only interested in proving that the code maintains confidentiality, we need only reason about confidentiality and the behaviors which influence it. A definitional semantics approach would tend to generate the entire behavior, in some formalization, which must then be examined altogether to prove confidentiality.

4.3 Automated Formal Reasoning

Practically speaking, we need a significant degree of automation to handle the multitude of trivial details associated with applying inference rules. As the example in Sect. 4.4.2 shows, even simple proofs can be complicated. Complex inference rules, such as those introduced in Chap. 5, and lengthy conditions aggravate the problem. Fortunately much of the detail can be handled automatically.

The amount of automation can vary. The least automated approach is a proof checker. A proof checker takes

- axioms, theorems, and rules of inference, and
- a proof, or list of inference steps,

and verifies that each step is a valid application of the inference rules. Although it reduces the chance of error in a manually derived proof, a checker provides no help in finding a proof.

On the opposite end of the automation spectrum would be a fully automated theorem prover. Given a formal logic and the formal description of a property of interest, it would return "true" or "unproven" depending on whether it could be proven or not. Hopefully if it could not prove the property, it would give some guidance about how much it could prove or what it could not prove. In simpler logics, such as first order logic, this is practical. But it can be difficult or impossible to represent some interesting properties in simple logics. On the other hand a program which handles higher order logics and proves (nearly) all theorems presented to it is well beyond our current ability to create.

Nevertheless formalizing properties and models and proving theorems about them can be as enlightening as, and perhaps more useful in the long run than, an oracular "true" or "false." Also a significant level of automation is feasible. A theorem proving assistant can make sure that rules of inference are validly applied, keep track of goals and which theorems have or have not been proved, and can prove simpler theorems or inferences automatically. Once the model of computation is chosen, one must then choose a theorem proving assistant to use and logic in which to embed it. Some choices are Coq, Nuprl, PVS [39], HOL [22], or a special purpose program. Coq uses constructive (or "intuitionistic") logic, while Nuprl, PVS, and HOL use classical logic. If one chooses a specially written theorem proving assistant program, it must be written and validated. Since one of the assistant's main values is preventing logical errors, bugs in the logic of the assistant can be very damaging.

We were initially inclined to use HOL because we were familiar with it, we knew it could handle large proofs, and it was available[1]. Also it never had programming errors which led to incorrect proofs and has a very large user community. Since there was no compelling reason to change, we decided to do our work in HOL.

4.3.1 HOL: A Theorem Proving Environment

HOL is a theorem proving assistant which allows classical reasoning in higher order logic. It is probably the most mature theorem prover and has the largest user community. All theorems are derived directly or indirectly from five axioms and eight inference rules in a classical logic. The level of automation is relatively low: although simple inferences are proven automatically by built-in routines, the user must do a lot of the thought work directing which tactics and inferences to use at what time in a typical proof.

The user can easily extend the HOL environment. Since most extensions are implemented by adding new definitions, no inconsistencies can be introduced. (The user can declare theorems axiomatically, but this is discouraged.)

4.3.2 Forward and Backward Proofs

The programming language associated with HOL is SML [40]. Two styles of proofs are supported by the HOL environment: forward proofs and backward proofs. Forward proofs proceed from known axioms, definitions, and theorems by inference rules to the theorems which are the final goal. This is the style most often used in introductory mathematical text and geometric proofs.

Backward proofs begin by declaring the goal to be proved. The user then invokes "tactics" to break the goal into (hopefully) simpler subgoals. To be accepted by the backward proof system, each tactic must be justified with inference rules which would prove the goal given the subgoals. For instance, the conjunction inference rules says that if two propositions, P and Q are theorems, their conjunction, $P \wedge Q$ is a theorem. The tactic `CONJ_TAC` says that one may prove a goal of the form $P \wedge Q$ by proving the subgoals P and Q. As each tactic is invoked

[1] for free.

by the user, its justifying inference rule is recorded. When all subgoals have been completely proven, the subgoal package automatically executes all the inference rules to produce a theorem corresponding to the original goal.

We find the backward proof much easier. The user declares the initial goal, then works on breaking the goal down into simpler subgoals. The subgoal package and tactics keeps track of the many, tiny details such as what must be proved to justify an inference, variable renaming, universal or existential quantification, etc. Because of HOL's security, the user need have little concern about thinking through the entire proof from the beginning.

4.3.3 Tactics to Expedite The Proof

Rather than tediously executing each inference step, larger steps can be packaged into "tactics." A tactic is a small program which examines the current goal and applies particular primitive inference rules, pre-proved theorems, or other tactics. The underlying inference engine in HOL guarantees that each step is, indeed valid. Therefore an error in a tactic merely fails to do what was desired, and never results in an invalid inference.

These tactics or small programs prove simple theorems or advance the proof in specific ways. For instance, many assignment statements can be handled quite mechanically, and verifying a `while` loop always consists of forming the following subgoals.

1. Any side effects in the `while` condition establish an interim condition from the invariant.
2. The body of the loop reestablishes the invariant from the interim condition and the success of the test.
3. The invariant and the negation of the test imply the postcondition.

Tactics form a custom, higher level theorem proving language which is still as trustworthy as the underlying program. Just as judiciously designed subroutines make writing (or changing) the top level of a program easier, tactics make proofs much more succinct and easier to do. When there are many good tactics, common situations are handled automatically and the proof proceeds faster.

4.4 Axiomatic Semantics

We chose an axiomatic semantics to model the implementation language, C, of `thttpd` and to encode rules of inference. This section introduces a convention we use, gives a brief introduction to axiomatic semantics and the syntax we use to represent it, and gives an example. Finally we point out the kinds of errors which may arise using axiomatic semantics and how to address them.

The syntax

$$\vdash A$$

means that A is a theorem. We use the HOL convention that all free variables are assumed to be universally quantified. Variables may range over functions as well as simple types, and function application is implicit. Thus $\vdash Px$ means "for all P and x, $P(x)$ is true."

4.4.1 A Brief Introduction

From [7] there are two main types of statements in axiomatic semantics: partial correctness and total correctness. An axiomatic statement of partial correctness is

$$\vdash \{\texttt{Precondition}\}\ \texttt{Code}\ \{\texttt{Postcondition}\}$$

where `Precondition` and `Postcondition` are predicates on the state of the computation and `Code` is a fragment of code. The above means if `Code` is executed in a state which satisfies `Precondition`, then when it terminates, `Postcondition` will be true. In this case, "terminates" means that the code doesn't loop indefinitely or abort abnormally. For example,

$$\vdash \{y = 3\}\ \mathrm{x = y;}\ \{x = 3\}$$

Although we do not use it, a statement of total correctness is similar, but asserts that the computation always terminates, too. When we have nonstructured jumps, like `return` or `continue` statements or the `exit()` function in C, total correctness must also show *reachability*, or that execution may reach the code. The syntax for total correctness uses square brackets ([and]) instead of curly braces ({ and }). For instance, the above assignment always terminates, so we can prove the stronger

$$\vdash [y = 3]\ \mathtt{x = y;}\ [x = 3]$$

The axiom (or actually, axiom schema) for C assignment statements of the form `v = expr;` is the following:

$$\vdash \{\mathtt{Q}^{\mathtt{v}}_{\mathtt{expr}}\}\ \mathtt{v = expr;}\ \{\mathtt{Q}\} \tag{4.1}$$

as long as `expr` doesn't have any side effects [21, pp. 15–17] and `v` is not an alias for any variable in `Q` or `expr`. The notation $\mathtt{Q}^{\mathtt{v}}_{\mathtt{expr}}$ means `Q` with all free occurrences of `v` replaced by `expr`. For instance, $(\mathtt{g} * (\mathtt{h} + 1))^{\mathtt{g}}_{(\mathtt{i}-1)}$ is `((i-1)*(h+1))`. The assignment axiom means, if $\mathtt{Q}^{\mathtt{v}}_{\mathtt{expr}}$ is true, then when the assignment is done, `Q` is true.

Inference rules for axiomatic semantics have hypotheses and conclusions. For example, Equation 4.2 is the inference rule for sequential statements. It states that if executing $s1$ in state P establishes state R and executing $s2$ in R establishes Q, we can conclude that executing $s1; s2$ in state P establishes Q.

$$\frac{\begin{array}{c}\vdash \{P\}\ s1\ \{R\} \\ \vdash \{R\}\ s2\ \{Q\}\end{array}}{\vdash \{P\}\ s1; s2\ \{Q\}} \tag{4.2}$$

4.4.2 Example Inference in Axiomatic Semantics

Suppose we want to prove that the following code swaps the values of `x` and `y`. (This doesn't work if `x` and `y` are aliases.)

```
x = x + y;
y = x - y;
x = x - y;
```

or more formally, we wish to prove

$$\vdash \{\mathrm{x} = \mathcal{X} \wedge \mathrm{y} = \mathcal{Y}\}\ \mathrm{x} = \mathrm{x} + \mathrm{y};\ \mathrm{y} = \mathrm{x} - \mathrm{y};\ \mathrm{x} = \mathrm{x} - \mathrm{y};\ \{\mathrm{x} = \mathcal{Y} \wedge \mathrm{y} = \mathcal{X}\}$$

where we use the assertion language variables $\mathcal{X}$ and $\mathcal{Y}$ to represent the initial values of `x` and `y` respectively. We use a backward proof, that is, proceeding from the goal to axioms or theorems by inference rules.

We instantiate the sequence rule 4.2 to get

$$\frac{\vdash \{\mathrm{x} = \mathcal{X} \land \mathrm{y} = \mathcal{Y}\}\ \mathtt{x = x + y;\ y = x - y;}\ \{\mathrm{x} = \mathcal{X} + \mathcal{Y} \land \mathrm{y} = \mathcal{X}\} \qquad \vdash \{\mathrm{x} = \mathcal{X} + \mathcal{Y} \land \mathrm{y} = \mathcal{X}\}\ \mathtt{x = x - y;}\ \{\mathrm{x} = \mathcal{Y} \land \mathrm{y} = \mathcal{X}\}}{\vdash \{\mathrm{x} = \mathcal{X} \land \mathrm{y} = \mathcal{Y}\}\ \mathtt{x = x + y;\ y = x - y;\ x = x - y;}\ \{\mathrm{x} = \mathcal{Y} \land \mathrm{y} = \mathcal{X}\}}$$

In other words, we can conclude the goal if we can prove

$$\vdash \{\mathrm{x} = \mathcal{X} \land \mathrm{y} = \mathcal{Y}\}\ \mathtt{x = x + y;\ y = x - y;}\ \{\mathrm{x} = \mathcal{X} + \mathcal{Y} \land \mathrm{y} = \mathcal{X}\}$$

(which we will refer to as subgoal 1) and

$$\vdash \{\mathrm{x} = \mathcal{X} + \mathcal{Y} \land \mathrm{y} = \mathcal{X}\}\ \mathtt{x = x - y;}\ \{\mathrm{x} = \mathcal{Y} \land \mathrm{y} = \mathcal{X}\}$$

(subgoal 2).

The assignment axiom 4.1 gives us the theorem

$$\vdash \{(\mathrm{x} = \mathcal{Y} \land \mathrm{y} = \mathcal{X})^{\mathtt{x}}_{\mathtt{x-y}}\}\ \mathtt{x = x - y;}\ \{\mathrm{x} = \mathcal{Y} \land \mathrm{y} = \mathcal{X}\}$$

which simplifies to

$$\vdash \{\mathrm{x} - \mathrm{y} = \mathcal{Y} \land \mathrm{y} = \mathcal{X}\}\ \mathtt{x = x - y;}\ \{\mathrm{x} = \mathcal{Y} \land \mathrm{y} = \mathcal{X}\}$$

by the definition of replacement, and then

$$\vdash \{\mathrm{x} = \mathcal{Y} + \mathrm{y} \land \mathrm{y} = \mathcal{X}\}\ \mathtt{x = x - y;}\ \{\mathrm{x} = \mathcal{Y} \land \mathrm{y} = \mathcal{X}\}$$

using standard algebraic laws on the precondition. Because computer arithmetic is finite, the usual laws of arithmetic and algebra *don't* strictly hold. Since overflow is rarely[2] a problem, we model computer arithmetic as arithmetic on integers. Modeling real numbers is even more complex. Since $\mathrm{y} = \mathcal{X}$, the above proves subgoal 2.

We can instantiate the sequence rule again to prove subgoal 1.

$$\frac{\vdash \{\mathrm{x} = \mathcal{X} \land \mathrm{y} = \mathcal{Y}\}\ \mathtt{x = x + y;}\ \{\mathrm{x} = \mathcal{X} + \mathcal{Y} \land \mathrm{y} = \mathcal{Y}\} \qquad \vdash \{\mathrm{x} = \mathcal{X} + \mathcal{Y} \land \mathrm{y} = \mathcal{Y}\}\ \mathtt{y = x - y;}\ \{\mathrm{x} = \mathcal{X} + \mathcal{Y} \land \mathrm{y} = \mathcal{X}\}}{\vdash \{\mathrm{x} = \mathcal{X} \land \mathrm{y} = \mathcal{Y}\}\ \mathtt{x = x + y;\ y = x - y;}\ \{\mathrm{x} = \mathcal{X} + \mathcal{Y} \land \mathrm{y} = \mathcal{X}\}}$$

[2] Quantifying "rarely" is a prime example of the need for design and engineering trade-offs we alluded to in the Introduction, page 2, and Chapter 3, page 11. In fact, overflow caused the loss of the first Ariane 5 rocket [3, 2].

Now we need to prove $\vdash \{x = \mathcal{X} \wedge y = \mathcal{Y}\}\ x = x + y;\ \{x = \mathcal{X} + \mathcal{Y} \wedge y = \mathcal{Y}\}$ (subgoal 1.1) and $\vdash \{x = \mathcal{X} + \mathcal{Y} \wedge y = \mathcal{Y}\}\ y = x - y;\ \{x = \mathcal{X} + \mathcal{Y} \wedge y = \mathcal{X}\}$ (subgoal 1.2).

Subgoal 1.2 can be proved starting with

$$\vdash \{(x = \mathcal{X} + \mathcal{Y} \wedge y = \mathcal{X})^{y}_{x-y}\}\ y = x - y;\ \{x = \mathcal{X} + \mathcal{Y} \wedge y = \mathcal{X}\}$$

by the assignment rule, then

$$\vdash \{x = \mathcal{X} + \mathcal{Y} \wedge x - y = \mathcal{X}\}\ y = x - y;\ \{x = \mathcal{X} + \mathcal{Y} \wedge y = \mathcal{X}\}$$

by the definition of replacement, and finally

$$\vdash \{x = \mathcal{X} + \mathcal{Y} \wedge y = \mathcal{X} + \mathcal{Y} - \mathcal{X}\}\ y = x - y;\ \{x = \mathcal{X} + \mathcal{Y} \wedge y = \mathcal{X}\}$$

by algebra.

Subgoal 1.1 can be proved with

$$\vdash \{(x = \mathcal{X} + \mathcal{Y} \wedge y = \mathcal{Y})^{x}_{x+y}\}\ x = x + y;\ \{x = \mathcal{X} + \mathcal{Y} \wedge y = \mathcal{Y}\}$$

by the assignment rule, then

$$\vdash \{x + y = \mathcal{X} + \mathcal{Y} \wedge y = \mathcal{Y}\}\ x = x + y;\ \{x = \mathcal{X} + \mathcal{Y} \wedge y = \mathcal{Y}\}$$

by the definition of replacement.

4.5 Example Automated Proof

Lest the reader despair because of all the work and details needed to formally verify software, we show how the preceding example is automated using HOL and tactics we wrote.

Variables which are only in the assertion language (so-called "logical" variables) have names beginning with a caret (ˆ) after Homeier. Unfortunately for clarity, HOL uses a caret for antiquotation, so we define the names separately (as `logX` and `logY` with type ":num" for natural numbers) to include them using antiquotation. The HOL prompt is a hyphen (`-`), and commands end with a semicolon (;). We represent HOL types in this dissertation with a colon (:) and the type name. The value of the result follows "`val` *name* `=`" after the command.

```
- val logX = mk_var{Name="^X", Ty = :num};
val logX = (--`^X`--) :term

- val logY = mk_var{Name="^Y", Ty = :num};
val logY = (--`^Y`--) :term
```

Next we set a goal with the function `g`. The predicate `Partial` expresses partial correctness. Code is embedded in HOL as abstract syntax trees (see Sect. 6.2.2, page 47 for details). Briefly variables consist of a `Var` constructor, the variable name, a disambiguator (for scoping), and the C type.

```
- g(`Partial ((x=^logX)/\(y=^logY))
  (Seq
   (Simple
    (Assign(Vref(Var "x" 0 Int))
     (Binary (Lval(Vref(Var "x" 0 Int)))
           Add (Lval(Vref(Var "y" 0 Int))))))
   (Seq
    (Simple
     (Assign(Vref(Var "y" 0 Int))
      (Binary (Lval(Vref(Var "x" 0 Int)))
            Sub (Lval(Vref(Var "y" 0 Int))))))
    (Simple
     (Assign(Vref(Var "x" 0 Int))
      (Binary (Lval(Vref(Var "x" 0 Int)))
            Sub (Lval(Vref(Var "y" 0 Int))))))))
   ((y=^logX)/\(x=^logY))`);
val it =
  Status: 1 proof.
  1. Incomplete:
       Initial goal:
       (--`Partial ((x = ^X) /\ (y = ^Y))
             (Seq
                (Simple
```

```
            (Assign (Vref (Var "x" 0 Int))
              (Binary (Lval (Vref (Var "x" 0 Int))) Add
                (Lval (Vref (Var "y" 0 Int))))))
         (Seq
            (Simple
              (Assign (Vref (Var "y" 0 Int))
                (Binary (Lval (Vref (Var "x" 0 Int))) Sub
                  (Lval (Vref (Var "y" 0 Int))))))
           (Simple
             (Assign (Vref (Var "x" 0 Int))
               (Binary (Lval (Vref (Var "x" 0 Int))) Sub
                 (Lval (Vref (Var "y" 0 Int))))))))
       ((y = ^X) /\ (x = ^Y))`--)
```

The goal is solved automatically by repeatedly applying the tactic SEQ_ASSIGN_TAC. Each application of the tactic

1. separates the last statement from the sequence, if necessary, deriving an intermediate predicate from the postcondition, and

2. proves the last statement using the assignment rule.

```
- e(REPEAT SEQ_ASSIGN_TAC);
OK..
val it =
  Initial goal proved.
  |- Partial ((x = ^X) /\ (y = ^Y))
       (Seq
          (Simple
            (Assign (Vref (Var "x" 0 Int))
              (Binary (Lval (Vref (Var "x" 0 Int))) Add
                (Lval (Vref (Var "y" 0 Int))))))
         (Seq
            (Simple
              (Assign (Vref (Var "y" 0 Int))
```

```
          (Binary (Lval (Vref (Var "x" 0 Int))) Sub
            (Lval (Vref (Var "y" 0 Int))))))
      (Simple
        (Assign (Vref (Var "x" 0 Int))
          (Binary (Lval (Vref (Var "x" 0 Int))) Sub
            (Lval (Vref (Var "y" 0 Int)))))))
  ((y = ^X) /\ (x = ^Y))
```

4.6 Limitations of Axiomatic Semantics

Although axiomatic semantics is a high level, powerful means of reasoning about programs, it is open to a significant problem. Since inference rules and theorems about individual statements are introduced as axioms, the collection may be inconsistent. An inconsistency could allow us to conclude an absurd statement of partial correctness, such as $\vdash$ {T} x = 1; {x = 2}. In addition the form of axiomatic semantics makes it hard to be sure that a certain axiomatic semantics correctly reflects the language semantics. We especially point out that many different inference rules for procedures calls have been proposed through the years correcting limitations and inaccuracies of previous rules.

A good way to address both consistency and correctness problems is to begin with a lower level, definitional description such as operational semantics [38] or abstract state machines [24], then prove the axiomatic semantics from that. Both Norrish [37] and Homeier [32] began with operational semantics and built the axiomatic semantics on that base. Low level descriptions are more obviously related to the languages and handle complexities as side effects more cleanly. Therefore they are more likely to be correct than a higher level axiomatic semantics. Since they are definitional, they cannot introduce inconsistencies. An axiomatic semantics proved from a low level semantics is more likely to be correct and consistent.

We feel that the time and effort of this level of detail is not justified in this dissertation.

CHAPTER 5

INFERENCE RULES FOR SIDE EFFECTS

In Chap. 2 we saw that historically there have been two general approaches to handling programming languages whose statements may have side effects. The first is to separate theorems about the effect of statements on the program state from theorems about the result value of expressions or statements [9, 24, 32]. These semantics are similar to denotational or operational semantics and handle side effects quite easily.

The other approach is to analyze the subexpressions which cause side effects separately from the original expression and use unique variables which carry the result of side effects [13, 33]. Our approach is similar to these. In this chapter we show inference rules for handling side effects in various C statements such as assignments, conditionals, and loops. Our general approach can be used to create axiomatic semantics for other statements or languages with side effects in expressions.

Early versions of parts of this chapter are in [8].

5.1 Model Formality

Before presenting the inference rules, we must remind the reader of some concepts of formality. The real world, in which we actually run software, is incredibly complex. Effects which may influence what software actually does when it runs extend from a squirrel in the next state knocking out power to alpha particles flipping bits in memory, from a bug in the compiler to timing differences due to multitasking. We use models in order to formally reason about systems. Generally, the more accurate the model, the harder it is to reason about since more detail is included.

When reasoning about software, we speak of embedding the programming language in a formal system. The "depth" of embedding is a somewhat objective measure of how much detail is carried into the formal system. Another way to look at it is to consider how the semantic gap between a computer program and a formal representation is bridged. If most of the translation or bridging is done informally, as in Figure 5.1, it is a "shallow embedding." A "deep embedding" has most of the conversion specified formally, as in Figure 5.2.

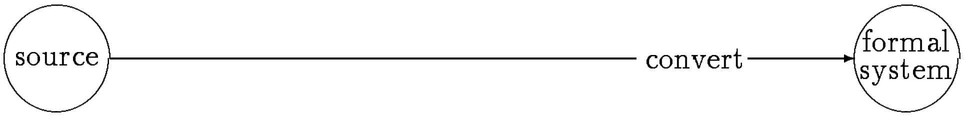

Figure 5.1: Informal Semantics in Shallow Embedding

Consider a piece of C code such as

```
if (x) x -= y;
```

A shallow embedding would have a conversion program which yields some equivalent formal representation, say $x \neq 0 \Rightarrow x' = x - y$. We can reason about the formal statement using preexisting inference rules such as modus ponens or those for arithmetic. But we must simply trust that our reasoning applies to the original code. Notice that the informal program embodies, for instance, the C rule that any non-zero value is treated as true and the meaning of the operator `-=`. There may be errors in the conversion, but we can't examine whether there are errors within this formal system. (We could, of course, verify the conversion program separately to assure ourselves that it will not mask bugs in source code.)

Figure 5.2: Formal Semantics in Deep Embedding

In a deep embedding, little translation is done informally or outside the formal system. For instance the above piece of C code might be rendered in an abstract syntax form like this.

```
If (Var "x" Int)
            (Simple (SubAssign (Var "x" Int) (Var "y" Int)))
```

The meaning of the piece of code is derived by associating form or syntax with semantics by formally defined rules in addition to those given in the formal system. For instance, a formal rule defines that expressions of the form $l- =r$ are equivalent to $l = l - r$.

With shallow embedding we can prove that `z = 0;` and `z = z-z;` have the same effect, namely, that `z` is assigned the value 0 regardless of its previous value. However we *cannot* prove that for all variables v the expressions v= 0; and v=v−v; have the same effect. In contrast a deep embedding has the tools to prove that for all variables v the expressions `Assign` v `(Const 0 Int)` and `Assign` v `(Binary` v `Sub` v`)` have the same effect.

Although an embedding may generally be said to be shallow or deep, there are many variations. For instance, an otherwise "shallow" embedding may represent function calls syntactically and have inference rules associated with them. A nominally "deep" embedding probably parses informally (a "shallow" embedding of the syntax) rather than carrying strings of characters and describing lexical analysis formally. If we wanted to reason about machine language programs which modify their instructions at run time, we could not even use syntactic abstraction: we would have to model uninterpreted memory contents. So the terms shallow and deep are relative terms depending on the verification.

5.2 A Rule for Preevaluation Side Effects

The assignment axiom for `v = expr;` is

$$\vdash \{\mathtt{Q}^{\mathtt{v}}_{\mathtt{expr}}\}\ \mathtt{v} = \mathtt{expr};\ \{\mathtt{Q}\}$$

as long as `expr` doesn't have any side effects ([21], pp. 15-17) and `v` is not an alias for anything in `Q` or `expr`. Since C statements may have side effects, this rule may not apply. As a simple example, the semantics of `a = 2 * ++b;` is well defined [25] (it is equivalent to the compound statement `++b; a = 2 * b;`), but the statement modifies the value of `b` as well as `a`.

To reason about complex statements, we introduce a general inference rule which derives the correctness of one statement from the correctness of a semantically equivalent statement.

$$\frac{\vdash \texttt{SEM\_EQ stm1 stm2} \qquad \vdash \{\texttt{pre}\}\ \texttt{stm1}\ \{\texttt{post}\}}{\vdash \{\texttt{pre}\}\ \texttt{stm2}\ \{\texttt{post}\}} \tag{5.1}$$

The predicate SEM_EQ is true if its two statement arguments are semantically equivalent. The inference rule means if

- two statements are semantically equivalent, and
- there is a partial correctness theorem for precondition, statement stm1, and postcondition

we can conclude an analogous partial correctness theorem for statement stm2.

We have not fully formally defined semantic equivalence. Although we have some inference rules for higher level terms, an SML function checks equivalence and specializes the definition.

We introduce the following rule to reason specifically about preevaluation side effects, that is, side effects which take place *before* the expression is evaluated.

$$\frac{\vdash \texttt{PreEval expr stm1 stm2}}{\vdash \texttt{SEM\_EQ (Seq (Simp expr) stm1) stm2}} \tag{5.2}$$

Seq is the abstract syntax constructor which creates a statement from a sequence of two statements. Simp converts any expression into a simple statement, which is allowed in C. The PreEval is a predicate which is true if extracting the preevaluation side effects expression expr from statement stm2 yields stm1.

Informally the rule means if stm2 can be separated into expr (which has all preevaluation side effects) and stm1, then expr (in a statement) followed by stm1 is semantically equivalent to stm2.

For example, we can derive theorems about the effect of `a = 2 * ++b;` from the sequence of simpler statements `++b; a = 2 * b;` from the instances of the inference rules

$$\frac{\vdash \texttt{PreEval}\quad \texttt{++b}\quad \texttt{a} = 2 * \texttt{b};\quad \texttt{a} = 2 * \texttt{++}\,\texttt{b};}{\vdash \texttt{SEM\_EQ}\ (\texttt{Seq}(\texttt{Simp}\ \ \texttt{++b})\ \ \texttt{a} = 2 * \texttt{b};)\ \ \texttt{a} = 2 * \texttt{++}\,\texttt{b};}$$

and

$$\frac{\vdash \texttt{SEM\_EQ}\,(\texttt{++}\,\texttt{b};\ \ \texttt{a} = 2 * \texttt{b};)\ \texttt{a} = 2 * \texttt{++}\,\texttt{b}; \qquad \vdash \{\texttt{P}\}\ \texttt{++}\,\texttt{b};\ \ \texttt{a} = 2 * \texttt{b};\ \{\texttt{Q}\}}{\vdash \{\texttt{P}\}\ \texttt{a} = 2 * \texttt{++}\,\texttt{b};\ \{\texttt{Q}\}}$$

Since we have a deep embedding, we can derive a single rule to separate preevaluation side effects using Rules 5.1 and 5.2.

$$\frac{\vdash \texttt{PreEval expr stm1 stm2} \qquad \vdash \{\texttt{pre}\}\ (\texttt{Seq (Simp expr) stm1})\ \{\texttt{post}\}}{\vdash \{\texttt{pre}\}\ \texttt{stm2}\ \{\texttt{post}\}} \tag{5.3}$$

Why add another inference rule just to separate side effects? Homeier's language, Sunrise [32], has an operator with a side effect, increment, which can occur in test expressions. He handles this by embedding the semantics of the operator in the inference rules. However functions, which have arbitrary semantics including side effects, can occur in loop or test expressions in C. Even statements without function calls can have multiple side effects using, say, increment and assignment operators. We take this more general approach to be able to separate a side effect from the expression in which it occurs.

Given the above preference for separation inference rules, why define two rules (one for preevaluation side effects and semantic equivalence and another for semantic equivalence and partial correctness) instead of the single rule 5.3? Two rules make future development easier since it breaks proofs and inferences into smaller pieces. The semantic equivalence of preevaluation separation can be proven from, say, a denotation semantics such as [38] without reference to the definition of partial correctness. And total correctness need only have one rule for semantic equivalence rather than a rule for preevaluation side effects, a rule for postevaluation side effects, a rule for conditionals with side effects, etc.

5.3 A Rule for Postevaluation Side Effects

C allows postevaluation side effects in expressions in addition to preevaluation side effects. The statement `a = 2 * b++;` is well defined, just as the preevaluation case. The statement can be broken down into the equivalent compound statement `a = 2 * b; b++;`.

The following rule allows us to reason about postevaluation side effects, that is, side effects which take place *after* the expression is evaluated.

$$\frac{\vdash \texttt{PostEval stm1 expr stm2}}{\vdash \texttt{SEM\_EQ (Seq stm1 (Simp expr)) stm2}} \tag{5.4}$$

The PostEval is a predicate which is true if extracting the postevaluation side effects expression expr from statement stm2 yields stm1.

Informally the rule means if stm2 can be separated into stm1 and expr (which has all postevaluation side effects), stm1 followed by expr (in a statement) is semantically equivalent to stm2.

Like the case for preevaluation side effects, we can derive a rule to separate postevaluation side effects using Rules 5.4 and 5.1.

$$\frac{\begin{array}{c}\vdash \texttt{PostEval stm1 expr stm2} \\ \vdash \{\texttt{pre}\}\ \texttt{(Seq stm1 (Simp expr))}\ \{\texttt{post}\}\end{array}}{\vdash \{\texttt{pre}\}\ \texttt{stm2}\ \{\texttt{post}\}} \tag{5.5}$$

5.4 Side Effects in Conditionals

The rules presented above are inadequate for control statements. For instance, suppose we were allowed to apply the postevaluation rule 5.5 to the following code.

```
if (b++ > 0) {
    t = b;
} else {
    e = b;
}
```

It would be transformed into this (note the postincrement afterward) which is *not* the same. The increment would be delayed until after the entire conditional statement.

```
if (b > 0) {
    t = b;
} else {
```

```
    e = b;
}
b++;
```

In the following sections we present inference rules for some control structures and indicate how the general approach could cover many other structures.

Conditionals are the simplest form of control statements for our purposes. Without side effects the inference rule is straight forward:

$$\frac{\begin{array}{c}\texttt{IS\_VALUE}\ \texttt{expr}\ \texttt{test} \\ \vdash \{\texttt{pre}\ \wedge\ \texttt{test}\}\ \texttt{thenCode}\ \{\texttt{post}\} \\ \vdash \{\texttt{pre}\ \wedge\ \texttt{\~{}test}\}\ \texttt{elseCode}\ \{\texttt{post}\}\end{array}}{\vdash \{\texttt{pre}\}\ \texttt{IfElse (expr) thenCode elseCode}\ \{\texttt{post}\}}$$

Notice we must use `IS_VALUE` to indicate the equivalence between `expr`, which is in the program language, and `test`, which is in the assertion language.

Any preevaluation side effects can be separated and handled with the preevaluation rule 5.3. However postevaluation side effects must be handled specially. Figure 5.3 shows the flow in a conditional statement with postevaluation side effects in the test expression.

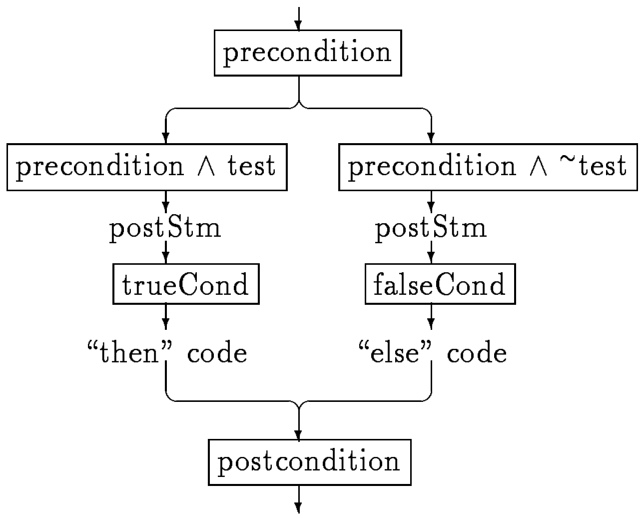

Figure 5.3: Control Flow in a Conditional

Rectangles are predicates on the program state. The pieces of text ‘postStm,’ ‘“then” code,’ and ‘“else” code’ show code execution. The sequence of events is

1. Find the test condition in the initial state (when the precondition is true),

2. Evaluate the postevaluation side effects, yielding new conditions, then

3. Evaluate the code in "then" or "else" branch, yielding the postcondition.

This is the corresponding inference rule.

$$\frac{\begin{array}{c}\vdash \texttt{SEM\_EQ (Seq (Simp expr) postStm)) (Simp ex)} \\ \vdash (\texttt{postStm} = \texttt{EmptyStmt}) \vee \\ \quad (\texttt{postStm} = (\texttt{Simp postSeEx}) \wedge \texttt{NoPreSE postSeEx}) \\ \vdash \texttt{IS\_VALUE expr test} \\ \vdash \{\texttt{pre} \wedge \texttt{test}\}\ \texttt{postStm}\ \{\texttt{trueCond}\} \\ \vdash \{\texttt{pre} \wedge \texttt{\~{}test}\}\ \texttt{postStm}\ \{\texttt{falseCond}\} \\ \vdash \{\texttt{trueCond}\}\ \texttt{thenCode}\ \{\texttt{post}\} \\ \vdash \{\texttt{falseCond}\}\ \texttt{elseCode}\ \{\texttt{post}\}\end{array}}{\vdash \{\texttt{pre}\}\ \texttt{IfElse (ex) thenCode elseCode}\ \{\texttt{post}\}} \tag{5.6}$$

Informally in order to prove the partial correctness of the conditional statement, we must prove the following:

- The original test expression code, `ex`, is split into a side effect free test expression, `expr`, followed by a statement for any postevaluation side effects `postStm`. (Any preevaluation side effects can be removed by Rule 5.3.)

- The postevaluation side effect statement `postStm` is empty (if there are no side effects), or it is a simple statement of the postevaluation side effects `postSeEx` with no preevaluation side effects.

- The code `expr` corresponds to `test` in the assertion language.

- Executing `postStm` with `test` true or false establishes the "true" or "false" conditions respectively.

- Executing the "then" and the "else" code establishes the post condition.

Typically most of these theorems are proven automatically, thus minimizing the user's work.

An inference rule for one-armed conditionals can be derived from the above rule and the rule which states that one-armed conditionals are semantically equivalent to two-armed conditionals with empty "else" cases (*If and IfElse with Empty*, Table 6.6, page 58).

5.5 Loops with Pre and Post Eval Side Effects

In simple languages the inference rule for a while loop, or backward jump, is straight forward:

$$\frac{\begin{array}{c}\vdash \texttt{IS\_VALUE expr test} \\ \vdash \{\texttt{invariant} \ \wedge \ \texttt{test}\}\ \texttt{body}\ \{\texttt{invariant}\}\end{array}}{\vdash \{\texttt{invariant}\}\ \texttt{while expr body}\ \{\texttt{invariant} \wedge \texttt{\textasciitilde test}\}} \tag{5.7}$$

In languages in which the test expression may have side effects, the rule is more complex. Note that neither the preevaluation (5.3) nor the postevaluation side effect rule (5.5) are valid. If one were allowed to use, say, the preevaluation rule, one could prove

```
while (pre-eval side-effects in expr)
    body
```

by proving

```
pre-eval side-effect;
while (expr)
    body
```

But in the second form, the side effect is not executed every loop! The flow of control in a `while` loop with pre- and postevaluation side effects is given in Figure 5.4.

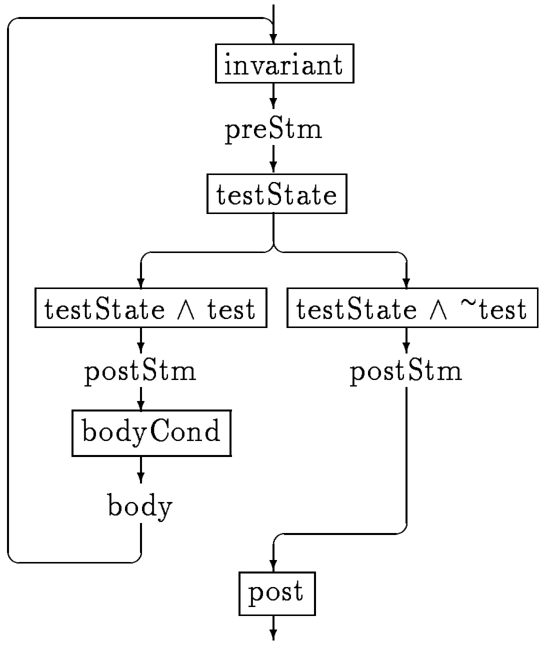

Figure 5.4: Control Flow in a While Loop

The inference rule for while statements is then

$$\dfrac{\begin{array}{l}\texttt{SEM\_EQ (Seq preStm (Seq (Simp testEx) postStm)) (Simp ex)} \\ \quad \vdash (\texttt{preStm} = \texttt{EmptyStmt}) \vee \\ \qquad (\texttt{preStm} = (\texttt{Simp preSeEx}) \wedge \texttt{NoPostSE preSeEx}) \\ \quad \vdash (\texttt{postStm} = \texttt{EmptyStmt}) \vee \\ \qquad (\texttt{postStm} = (\texttt{Simp postSeEx}) \wedge \texttt{NoPreSE postSeEx}) \\ \qquad\qquad \texttt{IS\_VALUE testEx test} \\ \qquad\quad \vdash \{\texttt{invariant}\}\ \texttt{preStm}\ \{\texttt{testState}\} \\ \qquad \vdash \{\texttt{testState} \wedge \texttt{test}\}\ \texttt{postStm}\ \{\texttt{bodyCond}\} \\ \qquad \vdash \{\texttt{testState} \wedge \texttt{\~{}test}\}\ \texttt{postStm}\ \{\texttt{post}\} \\ \qquad\quad \vdash \{\texttt{bodyCond}\}\ \texttt{body}\ \{\texttt{invariant}\}\end{array}}{\vdash \{\texttt{invariant}\}\ \texttt{While (ex) body}\ \{\texttt{post}\}} \tag{5.8}$$

In other words in order to prove the While loop correct, we must prove:

- Executing preStm, then the remaining test expression (testEx), then postStm is equivalent to the original test expression.
- The preevaluation (preStm) and postevaluation (postStm) side effects statements are either empty statements or are expressions with just pre- or postevaluation the side effects respectively.

- the code `testEx` corresponds to `test` in the assertion language.
- Executing `preStm` in the invariant condition establishes a test condition.
- Executing `postStm` with `test` true or false establishes the body or post conditions respectively.
- Executing body code in the body condition reestablishes the loop invariant.

We allow `preStm` and `postStm` to be the empty statement in case the original test expression has no side effects. To support our confidence in this rule, we note that when `expr` has no side effects, `preStm` and `postStm` are the empty statement. Therefore the test condition is the same as the invariant, the body condition is $\texttt{invariant} \wedge \texttt{test}$, and the post condition is $\texttt{invariant} \wedge \texttt{\~{}test}$. This reduces to the basic `while` loop rule (5.7). For higher assurance this and other inference rules should be proven from a simpler semantics, as we explained in Sect. 4.6.

The HOL tactic to reduce a `while` loop optionally takes a test condition and a body condition. The user can skip either or both if there are no side effects. The tactic also proves most conditions automatically. Thus the complexity of the rules are only exposed when necessary, and the user's work is minimized.

5.6 Other Looping Constructs and Limitations

Other looping constructs can be handled similarly. The `for` and `do...while` loops in C, `do...until` in Pascal, and `loop...begin...again` in Forth can be broken apart into side conditions and correctness conditions over pieces of code. Built-in tactics can keep track of where correctness conditions are needed and with regard to which expressions or pieces of code.

Directives which change the flow of control within loops, such as `break` and `continue` in C, can be handled with multiple post conditions as originally set forth in [1]. For example, a `break` statement would have a formalization something like this.

$$\vdash \{\texttt{pre}\}\ \texttt{break;}\ \{[\texttt{next} : \texttt{false}, \texttt{break} : \texttt{pre}]\}$$

In other words, the next sequential condition is "false" (control never arrives at the next statement), and the precondition of the `break` is the condition where the break control flow arrives.

The above rules are still not entirely adequate. A test expression may have multiple pre- or postevaluation side effects. For instance, `while (k=(j++, j+1) > 0) ....` This should be separated into the preevaluation side effects `j++; k=j+1;`, the test `k>0`, and, in this case, no postevaluation side effect. The rule only allows for one pre and post side effect expression. Either `NoPreSE` and `NoPostSE` should work on statements, not just expressions, the extraction of side effects will have to be more refined, or the abstract syntax should allow the comma (,) operator.

The inference rules given above are not valid in the presence of sequence points with side effects. Sequence points arise in C from logical OR's (`||`) and AND's (`&&`), which also have short circuit evaluation semantics, and the comma operator (`,`), among others. Consider the following code fragment. The variable `c` may or may not be incremented.

```
if (b++ || c++ > b) ...
```

Arbitrarily many sequence points may occur in an expression leading to arbitrarily branching control flows. We should be able to apply the general approach of deriving inference rules from control flows. This should yield inference rules to separate the additional intermediate states which arise.

CHAPTER 6

FORMAL DESCRIPTIONS AND MODELS

A formal verification needs a lot of background definition. Many things which are "understood" must be explicitly and formally defined. This is probably one reason formal methods, and formal verification in particular, are used little. But this is also one of its strengths: details are much less likely to be overlooked and assumptions are clearly stated.

This chapter presents the formalizations upon which we base the verification of `thttpd`. We begin with the formal specification of security in Sect. 6.1. That is, it is our formal definition of the statement "`thttpd` is secure." Section 6.2 presents our formalization of C: the abstract syntax tree (AST) representation, semantic rules, inference rules for partial correctness in axiomatic semantics, and correctness or well-formedness rules, and correspondence with the assertion language. Section 6.3 gives our formalization of relevant parts of Unix, and finally Sect. 6.4 is our formalization of C library functions and operating system calls. Because we have spent a lot of time finding and correcting silly errors while trying to get the formalization of calls correct, we include a subsection which is a guidebook on what to do and what not to do.

6.1 Formalizing the Specification

Programs must be verified with regard to something, e.g., a specification or properties to be checked. The security properties for `thttpd` are

- information integrity (no information on the server can be corrupted by outside users),
- confidentiality (the server only provides information which is explicitly authorized for outside access), and
- availability.

Although `thttpd` was designed to maximize availability (or minimize denial-of-service attacks), we felt this would be much more difficult to quantify and prove, so we leave it out of this verification. For instance, a user can repeatedly access the server causing the log file to grow until the disk is filled. Even though integrity and confidentiality are preserved (nothing on the disk is corrupted and the user didn't get any unauthorized information), http service (and, probably, most other services) are not available.

The above are *not* formal specifications. We must formalize them before we can reason about them. We consider the code to be secure if it has both confidentiality and information integrity. We introduce two predicates to express that the code has those properties: `hasConfidentiality` and `hasInfoIntegrity`. Confidentiality is with respect to the file system. Information integrity is with respect to some initial file system state (`preFSS`), the file system, and a log file, which may change as a result of the user's request. Thus the highest level predicate is as follows.

```
! code preFSS fs logfile.
    isSecure code preFSS fs logfile =
        hasConfidentiality code fs /\
        hasInfoIntegrity code preFSS fs logfile
```

6.1.1 Information Integrity

We formalize information integrity by equating "information on the server" with the contents of the file system and specify that "not corrupted" means not changed, deleted, moved, renamed or made inaccessible. These definitions are actually too strict, since the user can cause two types of changes.

1. Requests are logged in (added to) the log file.
2. The return output to the user, through `stdout`, is modeled as part of the file system, too.

Thus strictly speaking a user's action *can* cause a change on our system. So we refine the specification to say that the server preserves information integrity if it changes nothing on the file system, except possibly the log file or stdout. We define the predicate `hasInfoIntegrity code` to mean "this code maintains information integrity."

```
! code preFSS fs logfile.
  hasInfoIntegrity code preFSS fs logfile =
    Partial (SoFS preFSS fs) code
      (let finode = inodeNamed logfile in
         (!inode.~((inode=finode) \/ (inode=SYS_stdout)) ==>
             preFSS inode (getFile fs inode)))
```

6.1.2 Confidentiality

Confidentiality is defined as making sure that whatever information a user gets is explicitly authorized for outside access. For a web server the only user of interest is the user making the request. Local users (such as system administrators) have interaction other than through the web server code, so could violate confidentiality other ways. The only way information which reaches the requesting user is the output of the program. Thus we can say that a piece of code has confidentiality if all the output to the user (`stdout` in Unix) has confidentiality. Since we modeled `stdout` as just another "file" in the file system, we can use a single predicate over the file system to represent confidentiality.

```
! code fs.
   hasConfidentiality code fs =
      Partial (SoFS hasConfidentialityFile fs) code
         (SoFS hasConfidentialityFile fs)
```

In detail this states that a piece of code has confidentiality with respect to a file system if the following partial correctness can be proven. If the State of the File System (SoFS) has confidentiality for each file of the file system (fs), after the code is executed the file system still has confidentiality. The predicate `hasConfidentiality` represents that we only care about the contents of the "file" `stdout` since we assume a remote user can't access any other file (except to get its contents through this web server).

```
! inode ufile.
    hasConfidentialityFile inode ufile =
        (inode = SYS_stdout) ==> (nonConfidential ufile)

nonConfidentialFD (OPEN_fildesfn s n cwd root) =
    nonConfFileOwner (inodeNamed s) /\
    nonConfFilePerm (inodeNamed s) /\ (fsconfined cwd root)
```

We define the contents of a file to be nonconfidential recursively. A file descriptor refers to a nonconfidential file if it was opened with the correct constraints: the file ownership and permissions show it was meant for outside use and the current working directory and file system root are confined. Any information read from such a file descriptor is nonconfidential. We must also explicitly declare that constant strings from the program, such as error messages, are nonconfidential.

6.2 Formalizing C

The C programming language is enormously complex. This complexity gives it tremendous expressive power, but also requires a huge formalization. We only model features, aspects, properties, etc. which are necessary to our verification. Complete formalization would require many times more complexity and be (even more) difficult to work with. A complete formalization is a model of even the tiniest detail!

Our formalization of C has several components: an AST, semantics, inference, and correctness rules, and correspondence with assertion language expressions. We have partially formalized some features, thus a particular feature may have an AST representation, but no semantic rules.

As Homeier points out, axiomatic semantics uses two related languages: one to represent the program code and one to represent the assertion language. The first language is embodied in the AST and rules. The second language expresses the pre- and postconditions. Some differences are that the program language has variable declarations and assignments while the assertion language has universally and existentially quantified variables. The use of two related, but distinct, languages is often overlooked. For instance, the typical presentation of the inference rule for a conditional use the same object, `test`, in both assertions and code, as below.

$$\frac{\begin{array}{c}\vdash \{\texttt{pre} \ \wedge \ \texttt{test}\}\ \texttt{code}\ \{\texttt{post}\} \\ \vdash \{\texttt{pre} \ \wedge \ \texttt{\~{}test}\} \Rightarrow \texttt{post}\end{array}}{\vdash \{\texttt{pre}\}\ \texttt{If (test) code}\ \{\texttt{post}\}}$$

6.2.1 Scope of Formalization

Here we summarize the aspects of C which are not represented at all, partially represented, or full represented. To be fully represented, the aspect must be

- automatically parsed from C source code,
- deeply embedded in the AST, and
- have inference rules covering all semantics.

Most importantly, we assume that all programs to be verified are valid, i.e., compile without error. This allows us to ignore whole classes of possible lexical and semantic errors, such as undeclared variables.

Features Not Represented

Some of these may be partially supported, for instance defined in the abstract syntax, but we cannot reason about them.

- character escapes, except `\n` and `\0`
- long, short, unsigned, or floating point types
- functions as arguments or pointers-to-function
- hexadecimal or octal constants
- explicit storage classes (auto, extern, register, and static) or type qualifiers (const and volatile)
- type casts or conversions

- enumeration types
- unions or bit fields
- typedef
- pointer arithmetic
- dynamic memory allocation
- recursive function calls
- "for," "do...while," "goto," "break," "continue," and "switch" statements
- returning a value from a user-defined function
- the preprocessor (except for symbolic constants)

The preprocessor is not modeled, except for "symbolic constants" such as the #define below. We could have run all programs through the preprocessor first, but many common "functions" and "variables," such as `getc` and `stdout`, are actually macros and expand to complex expressions. Additionally such expansion would be tied to a single implementation. We chose instead to approximate #define statements similarly to global variables. We manually rewrote macros with arguments as functions.

```
#define name constant
```

We have no inference rules to derive the postcondition (return value) of a function from its body, particularly statements containing `return`'s. We did not model this since `thttpd` doesn't use "return" statements. Something like Homeier's entry and exit logics should suffice.

Since the standard axiomatic semantics deals exclusively with structured code, there is no provision for non-local jumps, such as the `break`, `continue`, and `return` statements or calls to `exit()`. We believe Arbib and Alagić's axiomatic semantics, which have multiple postconditions, could work.

Features Approximately or Partly Represented

We handle the following aspects in a program outside the formal system. Thus these are shallowly embedded (see Sect. 5.1).

- lexical analysis, including comments, operators, and string and character constants with escapes (\)
- operator precedence and associativity
- operator disambiguation (prefix vs. postfix for `++` and `--`, unary vs. binary for `&`, etc.)
- keyword identification
- identifier scope rules and declarations
- symbolic constants (#define name constant)

We approximate the following features.

- structures: only `int`-type fields, no user-defined `struct`'s
- type `int`: in the assertion language we model `int`'s as natural numbers (the non-negative integers, which are the HOL type `:num`)[1]
- other types: in the assertion language we model `char`'s as HOL ascii characters, and pointer-to-`char` and string types as HOL strings
- arrays: model as HOL arrays
- pointers: only dereference of address-of lvalue yields original lvalue
- varargs (functions which take variable number of arguments): The AST represents function parameters with a list[2]. The initial, fixed set of arguments are given directly. The rest are represented by a variable. For instance, `fprintf()` is declared as

[1] We discuss alternatives and consequences of this decision in Sect. 6.4.1.

[2] Since parameter types are deeply embedded, the list is a consistent HOL type: CVar. A shallow embedding would have caused problems since we would have had to combine elements of different types, e.g., `[format:string, x:num, y:num]`.

```
Func (Var "fprintf" 0 Int)
  (CONS (Var "stream" 0 (Ptr (Struct "FILE")))
   (CONS (Var "format" 0 (Ptr Char)) varargs))
```

Specific rules instantiate theorems about varargs functions, but they do not cover all cases.

The C language uses type casting extensively. The term "type casting" refers to coercing a value of some type to an equivalent value of another, related type. A simple example is assigning an integer value to a floating point variable. Operating system calls are often written using a specially defined type. For instance, `time` returns a value of `time_t` rather than `int`, and `strftime` takes and returns values of `size_t`.

To handle type casting, we would have to write a rather elaborate model of C types. Since `thttpd` does not depend on unreasonable assumptions about types, we work around the problem. For instance, we axiomatize system calls replacing the special types with `int`. No explicit type casting is used in `thttpd`.

Features Fully Represented

We fully model the following features of C.

- all operators, except operator-assignments (such as +=), the ternary operator (?:), logical "and" and "or" (`&&` and `||`), the comma (,) operator, and sizeof
- "if," "while," "return," empty, simple expression, and compound or block statements
- function and variable definition
- function calls
- default variable initialization

The remainder of this section generally proceeds from the lowest level, syntax, Sect. 6.2.2 through side effects, Sect. 6.2.3, semantically equivalent constructs, Sect. 6.2.4, well-formedness conditions, Sect. 6.2.5, to the highest level, inference rules, Sect. 6.2.6.

6.2.2 Abstract Syntax Tree

Rather than carry the complex concrete syntax of C programs into the formal system, a program parses the source and does some semantic analysis. The C "source code" about which we reason is an abstract syntax tree with keywords replaced by constructors and much implicit information, such as precedence, types, and scoping, made explicit.

The HOL representation of a definition is somewhat verbose and has many detail which are not pertinent. We present the essence of the definitions here. A new type definition is the type name, an equal sign (=), and a list of constructors separated by vertical bars (|). Each constructor is the constructor name and a list of (predefined) types. The predefined HOL types we use are "string" (lists of ascii characters) and "num" (natural numbers).

```
CConst = CCid  string |
         CCstr string |
         CCint num
```

C constants can be symbolic constants or #define names (CCid), string constants, or integer constants.

```
CType = Char | Int | Void |
        Ptr CType           |
        Array CType CConst |
        Struct string       |
        Funty CType CType
```

C types are either char, int, void, pointer, array, struct, or function. The type `Array` has the type of array elements and the size of the array. If the size is not known, for example a dynamically allocated array, the constant `CCid "UNK"` is used. Function types are curried. This is not a problem in practice since function names are represented by a special constructor which has a list of parameters.

We define two HOL constants, EOS and CR, to represent `'\0'` and `'\n'` respectively. Both are of type string.

```
Cvar = Var string num CType
```

A C variable is a name, a disambiguator, and a type. The disambiguator was first introduced by Homeier [32] and is used for scoping. For instance, the two different variables both named j in the fragment

```
int j;
{
    int j;
    k = k + j;
}
k = k + j;
```

could be translated as

```
(Var "j" 1 Int)
```

and

```
(Var "j" 0 Int)
```

The disambiguator is also used in inference rules to generator unique versions of variables.

```
Cbinop = Mul | Div | Mod  | Add | Sub | ShL | ShR |
         Eq  | NEq | Lt   | Gt  | LEq | GEq |
         And | Or  | BAnd | BOr | BXor
```

The binary operators are multiply (`*`), divide (`/`), modulo (`%`), add (`+`), subtract (`-`), shift left (`<<`), shift right (`>>`), equal (`=`), not equal (`!=`), less than (`<`), greater than (`>`), less than or equal (`<=`), greater than or equal (`>=`), logical AND (`&&`), logical OR (`||`), bitwise AND (`&`), bitwise OR (`|`), and bitwise exclusive OR (`^`).

```
Cunaryop = Not | BNot
```

The unary operators which take an expression are logical negation (`!`) and bitwise complement (`~`).

```
Clunaryop = AdrOf | PreInc | PostInc | PreDec | PostDec
```

To add semantic checking, we separately define unary operators which take an lvalue. They are address-of (&), prefix and postfix increment (++), and prefix and postfix decrement (--).

```
ParamList = PL CExpr ParamList | PLnull
```

A function call parameter list is a C expression and a parameter list or an empty list. This should really be defined as a HOL list of expressions, but the implementation was exceedingly complicated. We do, however, define a function which converts from a ParamList to a HOL list.

```
    (PL2CEL (PL h l) = (CONS h (PL2CEL l))) /\
    (PL2CEL PLnull = [])
```

```
Lvalue = Vref Cvar          |
         Aryref Cvar CExpr |
         Deref CExpr       |
         Field Lvalue Cvar |
         Arrow Lvalue Cvar
```

An lvalue is a variable reference, an array reference (including the array and a subscript), a dereference (*), a structure member access (.) (including the structure and the member), or a dereferencing structure member access (->).

```
CExpr = Const  CConst CType        |
        Lval   Lvalue              |
        Assign Lvalue CExpr        |
        OpAsgn Lvalue Cbinop CExpr |
        Binary CExpr  Cbinop CExpr |
        LUnary Clunaryop Lvalue    |
        Unary  Cunaryop  CExpr     |
        Condl  CExpr CExpr CExpr   |
        Comma  CExpr CExpr         |
        Call Cvar ParamList
```

A C expression is a constant, an lvalue, an assignment (`=`), an operator-assignment (such as `+=`), a binary operation, a unary operation taking an lvalue, a unary operation taking an expression, a ternary conditional (`?:`), a comma operation (`,`), or a function call.

We added operator-assignment, the ternary conditional, and the comma operator for future work. Although they are in the abstract syntax, we have no inference rules for them.

```
CStmt = EmptyStmt                  |
        Simple CExpr               |
        If     CExpr CStmt         |
        IfElse CExpr CStmt CStmt   |
        CWhile CExpr CStmt         |
        DoWhile CStmt CExpr        |
        Seq CStmt CStmt            |
        Block Cvar list CStmt      |
        Ret CExpr                  |
        EmptyRet                   |
        Break                      |
        Cont
```

A C statement is an empty statement (a lone `;`), simply an expression, an "if" or "one-armed" conditional, an "if..else" or "two-armed" conditional, a "while," a "do..while," a sequence (two succeeding statements), a compound or block, a "return" with an expression, a "return" without an expression, a "break," or a "continue."

As with some operators, "do..while," "break," and "continue" are in the abstract syntax, but we have no inference rules for them.

```
Cbody = SOMEBody CStmt | NOBody
```

A function body can either be a statement or nothing. The latter is needed to describe axioms for system calls or library functions for which we have no body.

```
Cfunction = Func Cvar (Cvar list) (Cvar list) Cbody
```

A function is the function name (which includes the return type), a list of parameters, a set (list) of global variables which may be referenced or modified, and an (optional) body.

```
Cfile = EndOfCFile             |
        Cfdcl Cvar list Cfile |
        Cffun Cfunction Cfile
```

We define a C file as a list of variable declarations and function definitions. We include this level rather than going to the level of the entire program since some scopes are confined to source files.

We wrote a program to translate from C source to HOL abstract syntax trees (AST). It is adapted from Paulson [40, chapter 9]. The parser has many enhancements to make it easier to debug grammars. Here is a nonsensical piece of C code and its corresponding AST.

```
void m;int cup;
y(){char h,q;FILE *a;if(0!=h)q=*a;else {t=m++;m=cup("joe");}}
```

```
Cfdcl [Var "m" 0 Void; Var "cup" 1 Int]
(Cffun
(Func (Var "y" 0 Int) [] []
 (SOMEBody
  (Block [Var "h" 0 Char; Var "q" 0 Char;
                        Var "a" 0 (Ptr (Struct"FILE"))]
   (IfElse (Binary (Const (CCint 0) Int) NEq
                  (Lval (Vref (Var "h" 0 Char))))
    (Simple (Assign (Vref (Var "q" 0 Char))
              (Lval (Deref
                  (Lval(Vref (Var "a" 0(Ptr(Struct"FILE"))))))))
    (Block [] (Seq
     (Simple (Assign(Vref(Var "t" 0 Void))
               (LUnary PostInc (Vref(Var "m" 0 Void)))))
     (Simple (Assign (Vref (Var "m" 0 Void))
```

```
                (Call (Var "cup" 1 Int)
                  (PL(Const(CCstr "joe")(Ptr Char))PLnull))))))))))
EndOfCFile
```

6.2.3 Side Effects

A significant difference between this work and others is the treatment of side effects. In this section we define many aspects of what side effects are, what are preevaluation or postevaluation side effects, etc. We have not formalized all aspects of side effects: some conversions are shallowly embedded, that is, the conversions are done by external programs rather than by rules within the formal system. (Because of the informality of some parts and the axiomatized inference rules, we doubt our system is error-free. For serious use, it should be formally proved from a lower-level description as we argued in Sect. 4.6.)

We begin by defining which C expressions and lvalues have no side effect, that is, those which don't change the program state, Table 6.1. Inference rules are in two forms.

1. Constructs which never changes the program state, e.g., constants.
2. Constructs which don't change the program state, but some component may, e.g., the index of an array reference.

All assignments, increments, and decrements change state. We are conservative and assume that any function call changes state.

Next we define which expressions and lvalues have no postevaluation side effects, Table 6.2. A postevaluation side effect is a change to the program state after the evaluation of the expression. The postincrement (v++) and postdecrement (`v--`) have postevaluation side effects.

As we showed in Chap. 5, we handle side effects by separating them from statements or expressions, and evaluating the semantically simpler equivalent. The predicate `PreEval` defines how to separate preevaluation side effects from statements. Informally the expression

```
PreEval lv e s1 s2
```

Variable Reference:

$$\frac{}{\vdash \text{NoSELv (Vref var)}}$$

Array Reference (var[c]):

$$\frac{\vdash \text{NoSE c}}{\vdash \text{NoSELv (Aryref var c)}}$$

*Dereference (*c):*

$$\frac{\vdash \text{NoSE c}}{\vdash \text{NoSELv (Deref c)}}$$

Structure Access (lv.var):

$$\frac{\vdash \text{NoSELv lv}}{\vdash \text{NoSELv (Field lv var)}}$$

Constant:

$$\frac{}{\vdash \text{NoSE (Const c t)}}$$

Lvalue Expression:

$$\frac{\vdash \text{NoSELv l}}{\vdash \text{NoSE (Lval l)}}$$

Binary Operation:

$$\frac{\vdash \text{NoSE c1} \qquad \vdash \text{NoSE c2}}{\vdash \text{NoSE (Binary c1 opr c2)}}$$

Address-Of (&lv):

$$\frac{\vdash \text{NoSELv lv}}{\vdash \text{NoSE (LUnary AdrOf lv)}}$$

Unary Operation:

$$\frac{\vdash \text{NoSE c}}{\vdash \text{NoSE (Unary opr c)}}$$

Table 6.1: Lvalues and Expressions Without Side Effects

Variable Reference:

$$\frac{}{\vdash \text{NoPostSELv (Vref var)}}$$

Array Reference (var[c]):

$$\frac{\vdash \text{NoPostSE c}}{\vdash \text{NoPostSELv (Aryref var c)}}$$

*Dereference (*c):*

$$\frac{\vdash \text{NoPostSE c}}{\vdash \text{NoPostSELv (Deref c)}}$$

Structure Access (lv.var):

$$\frac{\vdash \text{NoPostSELv lv}}{\vdash \text{NoPostSELv (Field lv var)}}$$

Empty Parameter List:

$$\frac{}{\vdash \text{NoPostSEPl PLnull}}$$

Nonempty Parameter List:

$$\frac{\vdash \text{NoPostSE ce} \qquad \vdash \text{NoPostSEPl prest}}{\vdash \text{NoPostSEPl (PL ce prest)}}$$

Function Call:

$$\frac{\vdash \text{NoPostSEPl plist}}{\vdash \text{NoPostSE (Call v plist)}}$$

Constant Expression:

$$\frac{}{\vdash \text{NoPostSE (Const c t)}}$$

Lvalue Expression:

$$\frac{\vdash \text{NoPostSELv l}}{\vdash \text{NoPostSE (Lval l)}}$$

Assignment:

$$\frac{\vdash \text{NoPostSELv l} \quad \vdash \text{NoPostSE e}}{\vdash \text{NoPostSE (Assign l e)}}$$

Binary Operation:

$$\frac{\vdash \text{NoPostSE c1} \quad \vdash \text{NoPostSE c2}}{\vdash \text{NoPostSE (Binary c1 opr c2)}}$$

Address-Of (&lv):

$$\frac{\vdash \text{NoPostSELv lv}}{\vdash \text{NoPostSE (LUnary AdrOf lv)}}$$

Preeval Increment (++lv):

$$\frac{\vdash \text{NoPostSELv lv}}{\vdash \text{NoPostSE (LUnary PreInc lv)}}$$

Preeval Decrement (--lv):

$$\frac{\vdash \text{NoPostSELv lv}}{\vdash \text{NoPostSE (LUnary PreDec lv)}}$$

Unary Operation:

$$\frac{\vdash \text{NoPostSE c}}{\vdash \text{NoPostSE (Unary opr c)}}$$

Table 6.2: Lvalues and Expressions With No Postevaluation Side Effects

means that the AST

```
Seq (Simple e) s1
```

has the same semantics as

```
s2
```

The lvalue, `lv`, holds the result of the expression, `e`. The lvalue cannot have any side effects, and the expression cannot have any *post*evaluation side effects.

We have only formalized part of `PreEval`, Table 6.3: the rest is shallowly embedded in an external routine. We use `NoSE` to ensure that the lvalue has no side effects, and we use `NoPostSE` to ensure that the expression has no postevaluation side effects.

Return Statement:

$$\frac{\vdash \mathrm{NoSELv\ l} \qquad \vdash \mathrm{NoPostSE\ e}}{\vdash \mathrm{PreEval\ l\ e\ (Ret\ cme)\ (Ret\ c)}}$$

Simple Statement:

$$\frac{\vdash \mathrm{NoSELv\ l} \qquad \vdash \mathrm{NoPostSE\ e}}{\vdash \mathrm{PreEval\ l\ e\ (Simple\ cme)\ (Simple\ c)}}$$

If Statement:

$$\frac{\vdash \mathrm{NoSELv\ l} \qquad \vdash \mathrm{NoPostSE\ e}}{\vdash \mathrm{PreEval\ l\ e\ (If\ cme\ s)\ (If\ c\ s)}}$$

If/Then/Else Statement:

$$\frac{\vdash \mathrm{NoSELv\ l} \qquad \vdash \mathrm{NoPostSE\ e}}{\vdash \mathrm{PreEval\ l\ e\ (IfElse\ cme\ s1\ s2)\ (IfElse\ c\ s1\ s2)}}$$

Table 6.3: Preevaluation Side Effect Separation

Finally we define which expressions and lvalues have no preevaluation side effects, Table 6.4. A preevaluation side effect is a state change before or during the expression evaluation. Assignment, function call, preincrement, and postincrement have preevaluation side effects.

The reader may question whether it is right to consider, say, assignment to be a preevaluation side effect. Strictly speaking "preevaluation side effect" is simply a name we give to any predicate, property, or phenomenon we choose. So the question reduces to whether our definition is consistent and whether it is useful.

Since `NoPreSE` is defined from the existing logic, rather than being asserted, we cannot introduce inconsistencies in the underlying formal logic. Since we do not explicitly define an inverse of `NoPreSE`, for instance, `PreSE`, we cannot introduce inconsistencies into our semantics by having an expression which is both `NoPreSE` and `PreSE`. However we do know, at a higher level, of relationships between the preceding `NoSE`, `NoPostSE`, and `NoPreSE`. We have proven `NoSE P` $\Rightarrow$ `NoPostSE P` and `NoSE P` $\Rightarrow$ `NoPreSE P`. (We tried to prove `NoPostSE P`$\wedge$`NoPreSE P` $\Rightarrow$ `NoSE P`, but have not been able to. It appears to be a problem with using induction on two different functions, `NoPostSE` and `NoPreSE`, at once.) At the highest level, we use these predicates in inference rules. The more confidence, we should prove that the inference rules with the predicates are consistent with lower level semantics (Sect. 4.6).

Judging whether the definition is useful is harder. Extreme definitions such as $\forall$`e. NoPreSE e` (every expression is `NoPreSE`) or $\forall$`e. ~NoPreSE e` (nothing is `NoPreSE`) are obviously not very useful, but the above definitions are not extreme. We have found that these predicates express what we want to express and let us prove what we want to prove (and believe to be correct). Therefore we are satisfied with the definitions.

The predicate `PostEval` defines how to separate postevaluation side effects from statements. Informally the expression

```
PostEval s1 e s2
```

means that the AST

```
Seq s1 (Simple e)
```

has the same semantics as

```
s2
```

The expression, `e`, cannot have any *pre*evaluation side effects.

Postevaluation side effects differ from preevaluation side effects in that the result of postevaluation side effects cannot be used in the statement. Thus there is no lvalue in the predicate. We use `NoPreSE` to partially formalize `PostEval`, Table 6.5. The table is much shorter than that for `PreEval` because postevaluation side effects are handled in special ways in control structures.

Variable Reference

$$\frac{}{\vdash \text{NoPreSELv (Vref var)}}$$

Array Reference (var[c]):

$$\frac{\vdash \text{NoPreSE c}}{\vdash \text{NoPreSELv (Aryref var c)}}$$

*Dereference (*c):*

$$\frac{\vdash \text{NoPreSE c}}{\vdash \text{NoPreSELv (Deref c)}}$$

Structure Access (lv.var):

$$\frac{\vdash \text{NoPreSELv lv}}{\vdash \text{NoPreSELv (Field lv var)}}$$

Unary Operation:

$$\frac{\vdash \text{NoPreSE c}}{\vdash \text{NoPreSE (Unary opr c)}}$$

Constant Expression:

$$\frac{}{\vdash \text{NoPreSE (Const c t)}}$$

Lvalue Expression:

$$\frac{\vdash \text{NoPreSELv lv}}{\vdash \text{NoPreSE (Lval lv)}}$$

Binary Operation:

$$\frac{\vdash \text{NoPreSE c1} \quad \vdash \text{NoPreSE c2}}{\vdash \text{NoPreSE (Binary c1 opr c2)}}$$

Address-Of (&lv):

$$\frac{\vdash \text{NoPreSELv lv}}{\vdash \text{NoPreSE (LUnary AdrOf lv)}}$$

Posteval Increment (lv++):

$$\frac{\vdash \text{NoPreSELv lv}}{\vdash \text{NoPreSE (LUnary PostInc lv)}}$$

Posteval Decrement (lv--):

$$\frac{\vdash \text{NoPreSELv lv}}{\vdash \text{NoPreSE (LUnary PostDec lv)}}$$

Table 6.4: Lvalues and Expressions With No Preevaluation Side Effects

Simple Statement:

$$\frac{\vdash \text{NoPreSE e}}{\vdash \text{PostEval (Simple cme) e (Simple c)}}$$

Table 6.5: Postevaluation Side Effect Separation

6.2.4 Semantic Equivalence

Rather than defining many additional partial correctness inference rules, we define semantic equivalence of certain statements, Table 6.6. This allows us to define total correctness, termination, or other predicates more succinctly. Informally if two statements are semantically equivalent, one may be replaced by the other in any circumstance and the semantics of the program is unchanged.

Reflexivity:

$$\frac{}{\vdash \text{SEM\_EQ s s}}$$

Symmetry:

$$\frac{\vdash \text{SEM\_EQ s1 s2}}{\vdash \text{SEM\_EQ s2 s1}}$$

Transitivity:

$$\frac{\begin{array}{l}\vdash \text{SEM\_EQ s1 s2}\\ \vdash \text{SEM\_EQ s2 s3}\end{array}}{\vdash \text{SEM\_EQ s1 s3}}$$

EmptyStmt Null (Left):

$$\frac{}{\vdash \text{SEM\_EQ (Seq EmptyStmt s) s}}$$

EmptyStmt Null (Right):

$$\frac{}{\vdash \text{SEM\_EQ (Seq s EmptyStmt) s}}$$

Sequence Associativity:

$$\frac{}{\begin{array}{l}\vdash \text{SEM\_EQ (Seq (Seq s1 s2) s3)}\\ \qquad\qquad \text{(Seq s1 (Seq s2 s3))}\end{array}}$$

Sequence Composition:

$$\frac{\vdash \text{SEM\_EQ r1 r2} \qquad \vdash \text{SEM\_EQ s1 s2}}{\vdash \text{SEM\_EQ (Seq r1 s1) (Seq r2 s2)}}$$

Preevaluation Side Effect Separation:

$$\frac{\vdash \text{PreEval lv expr s1 s2}}{\vdash \text{SEM\_EQ (Seq (Simple expr) s1) s2}}$$

Postevaluation Side Effect Separation:

$$\frac{\vdash \text{PostEval s1 expr s2}}{\vdash \text{SEM\_EQ (Seq s1 (Simple expr)) s2}}$$

If and IfElse with Empty:

$$\frac{}{\begin{array}{l}\vdash \text{SEM\_EQ (IfElse t s EmptyStmt)}\\ \qquad\qquad\qquad\qquad \text{(If t s)}\end{array}}$$

Table 6.6: Semantic Equivalence of Statements

6.2.5 Well-Formedness Conditions

Well-Formedness conditions are taken almost unchanged from Homeier [32]. They are a collection of consistency checks mostly related to function calls.

We begin with a series of auxiliary definitions. To infer anything about a function call from a partial correctness theorem on a function definition, we must be sure that the actual or calling parameters conform to the formal or declared parameters. We define the predicate `Ctypes_conform` to check that two C types conform. The predicates `is_Ptr`, `is_Array`, `is_Struct`, and `is_Funty` are true if the operand is a pointer, array, structure, or function respectively. Basically two types conform if they are the same primitive type, or they are structurally the same and their subtypes conform.

```
(Ctypes_conform Char t = (Char = t)) /\
(Ctypes_conform Int t = (Int = t))   /\
(Ctypes_conform Void t = (Void = t)) /\
(Ctypes_conform (Ptr st) t =
       ((is_Ptr t \/ is_Array t) =>
               (Ctypes_conform st (CTy_subty t)) | F)) /\
(Ctypes_conform (Array st1 sz) t =
       ((is_Ptr t \/ is_Array t) =>
               (Ctypes_conform st1 (CTy_subty t)) | F)) /\
(Ctypes_conform (Struct s1) t =
       (is_Struct t => (s1 = Struct_subty t) | F)) /\
(Ctypes_conform (Funty it ot) t =
       (is_Funty t => Ctypes_conform it (Funty_frmty t) /\
                     (Ctypes_conform ot (Funty_toty t)) | F))
```

Define that two variables have the same name and disambiguator and their types conform. From this definition we prove a theorem which is easier to use: it "decomposes" both variables at the same time.

```
TYPE_CONF (Var n d ty) v2 =
        (n = Cvar_name v2) /\ (d = Cvar_disamb v2)
                    /\ Ctypes_conform ty (Cvar_ v2)

TYPE_CONF (Var n d ty) (Var n2 d2 ty2) =
        (n = n2) /\ (d = d2) /\ Ctypes_conform ty ty2
```

Define that the first set is a subset of the second given type conformance.

```
(!a. IS_SUBSET_TYCONF [] a = T) /\
(!a h l. IS_SUBSET_TYCONF (CONS h l) a =
        (SOME_EL (TYPE_CONF h) a) /\ (IS_SUBSET_TYCONF l a))
```

Define set difference, that is, `SETDIFF a b` is a - b.

```
(!(b:'a list). SETDIFF [] b = []) /\
(!b h l. SETDIFF (CONS h l) b =
        let diffTail = (SETDIFF l b) in
                (IS_EL h b) => diffTail | (CONS h diffTail))
```

A list of variables is piece-wise distinct if no variable occurs more than once in the list [32, Sect. 10.1, p. 234]. The predicate `IS_EL` is true if the first operand is an element of the second operand, a list.

```
(DL [] = T) /\
(!h l. DL (CONS h l) = ~(IS_EL h l) /\ (DL l))
```

Currently we shallowly define the following. It is straight forward, but time consuming, to formally define them, as Homeier does. Since the two well-formedness checks are rarely, if ever, violated, we believe this informality does not really reduce the number of errors we might catch in programs.

- `WF_xs x`: A list of variables is well-formed if every variable in the list is well-formed [32, Sect. 10.4.2, p. 256].

- `WF_c stmt`: A statment, stmt, is well-formed if `While` statements have well-formed termination conditions, `Call` statements use variables reasonably (e.g., don't pass a global as a reference), and logical variables aren't used in expressions [32, Sect. 10.4.2, p. 258].

- `variants vs bs`: Generate unique variants (using disambiguators) of a set of variables, vs, with respect to a base set, bs [32, Sect. 10.1, pp 231–234].

- `logicals vs`: Generate logical (assertion language-only) versions of program variables. Logical variables are only used in assertions, and therefore cannot be changed. They are used as "initial" values.

- `GV_c stmt`: Return the global variables used by functions called in the statement, stmt.

- `FV_c stmt`: Return the free program variables of the statement, stmt.

- `FV_a e`: Return the free variables of an assertion (HOL) expression, e.

- `pairwiseEqual vl1 vl2`: Create a conjunction of pairwise equivalency of two lists of variables. For instance, if a, b, c, and d are variables, we can conclude the following.

$$\vdash \texttt{pairwiseEqual}\ [\mathsf{a};\mathsf{b}]\ [\mathsf{c};\mathsf{d}] \quad = \quad (\mathsf{a} = \mathsf{c}) \wedge (\mathsf{b} = \mathsf{d})$$

Rule 6.1 defines that a function specification is well-formed syntactically, e.g., all variables used in the body are either parameters or listed as globals. This corresponds to Homeier's WF_proc_syntax [32, p. 259]. Since we did not deal with recursive functions, we left out the check for recursive calls. It should be straight forward to add.

$$\frac{\begin{array}{c}\vdash \mathrm{x} = \mathrm{APPEND\ formls\ globls} \qquad \vdash \mathrm{x0} = \mathrm{logicals\ x} \\ \vdash \mathrm{WF\_xs\ x} \qquad \vdash \mathrm{DL\ x} \qquad \vdash \mathrm{WF\_c\ body} \\ \vdash \mathrm{IS\_SUBSET\ (GV\_c\ body)\ globls} \qquad \vdash \mathrm{IS\_SUBSET\ (FV\_c\ body)\ x} \\ \vdash \mathrm{IS\_SUBSET\_TYCONF\ (FV\_a\ pre)\ x} \\ \vdash \mathrm{IS\_SUBSET\_TYCONF\ (FV\_a\ post)\ (APPEND\ x\ x0)}\end{array}}{\vdash \mathrm{WF\_fn\_syntax\ pre\ (Func\ name\ formls\ globls\ (SOMEBody\ body))\ post}} \tag{6.1}$$

Rule 6.2 is the partial correctness condition for C functions. A well-formed function call, `WF_fnp`, serves as a "template" or pre-proved theorem for function calls. That is, a function call is verified using a `WF_fnp` theorem. Our `WF_fnp` corresponds to Homeier's WF_proc [32, p. 260]. Any return value is indicated by a predicate on the special variable `C_Result` in the postcondition. Informally the inference rule states that a function is partially correct if it is well-formed and the body can be proved partially correct. The `pairwiseEqual` represents variables set to the "initial" values of the actuals.

$$\frac{\begin{array}{c}\vdash \text{x} = \text{APPEND formls globls} \qquad \vdash \text{x0} = \text{logicals x} \\ \vdash \text{WF\_fn\_syntax pre} \\ \qquad\qquad (\text{Func name formls globls (SOMEBody body)) post} \\ \vdash \{(\text{pairwiseEqual x x0}) \wedge \text{pre}\}\ \text{body}\ \{\text{post}\}\end{array}}{\vdash \text{WF\_fnp pre (Func name formls globls (SOMEBody body)) post}} \tag{6.2}$$

6.2.6 Inference Rules for Partial Correctness

For reference, we present all the inference rules for partial correctness in Tables 6.7 and 6.8. We discussed in detail various rules which handle side effects in Chap. 5. The inference rules are repeated, with their corresponding HOL tactics and uses documented, in App. A. Here we will only make a few comments on the rules.

In the rule for expressions without side effects, the substitution is the $c \lhd [x := e]$ operator from Homeier.

The rule for expressions with side effects needs to formally specify the relation of the precondition on the postcondition. Some substitution-like relation, like that in the rule for expressions with side effects, should be mentioned.

In the array assignment rule, the array substitution replaces every `a` in `c` with a new function having the value `e` at location `i`. This corresponds to the $c \lhd [a[i] := e]$ operator from Homeier.

The block inference rule is incorrect for local variables which shadow outer variables. This should be easy to correct, for instance, see [21].

We derive the inference rules in Table 6.9 for efficiency in doing proofs. We also prove a more general inference rule for While which includes postcondition weakening. The postcondition weakening makes it easier to use in a tactic: the tactic can leave an implication subgoal if the computed postcondition doesn't match the goal.

6.2.7 C and the Assertion Language

We primarily use built-in HOL expressions for the assertion language. In some cases we deeply embed features of C to model them more accurately.

Precondition Strengthening:

$$\frac{\vdash \text{stronger} \Rightarrow \text{pre} \qquad \vdash \{\text{pre}\}\ \text{stm}\ \{\text{post}\}}{\vdash \{\text{stronger}\}\ \text{stm}\ \{\text{post}\}}$$

Conjunction:

$$\frac{\vdash \{\text{pre1}\}\ \text{stm}\ \{\text{post1}\} \qquad \vdash \{\text{pre2}\}\ \text{stm}\ \{\text{post2}\}}{\vdash \{\text{pre1}\wedge\text{pre2}\}\ \text{stm}\ \{\text{post1}\wedge\text{post2}\}}$$

Postcondition Weakening:

$$\frac{\vdash \{\text{pre}\}\ \text{stm}\ \{\text{post}\} \qquad \vdash \text{post} \Rightarrow \text{weaker}}{\vdash \{\text{pre}\}\ \text{stm}\ \{\text{weaker}\}}$$

Disjunction:

$$\frac{\vdash \{\text{pre1}\}\ \text{stm}\ \{\text{post1}\} \qquad \vdash \{\text{pre2}\}\ \text{stm}\ \{\text{post2}\}}{\vdash \{\text{pre1}\vee\text{pre2}\}\ \text{stm}\ \{\text{post1}\vee\text{post2}\}}$$

Expression, No Side Effects:

$$\frac{\vdash \text{NoSE e}}{\vdash \{\mathtt{post}^{\mathtt{e}}_{[\mathtt{c\_result}]}\}\ (\mathtt{Simple\ e})\{\text{post}\}}$$

Expression, Side Effects:

$$\frac{\vdash \text{NoSELv l}}{\vdash \{\text{pre}\}\ (\text{Simple (LUnary op l)})\ \{\text{post}\}}$$

Empty Statement:

$$\frac{}{\vdash \{\text{pre}\}\ \text{EmptyStmt}\ \{\text{pre}\}}$$

Block:

$$\frac{\vdash \{\text{pre}\}\ \text{stm}\ \{\text{post}\}}{\vdash \{\text{pre}\}\ (\text{Block llv stm})\ \{\text{post}\}}$$

Semantic Equivalence:

$$\frac{\vdash \text{SEM\_EQ stm1 stm2} \qquad \vdash \{\text{pre}\}\ \text{stm1}\ \{\text{post}\}}{\vdash \{\text{pre}\}\ \text{stm2}\ \{\text{post}\}}$$

Sequence:

$$\frac{\vdash \{\text{pre}\}\ \text{stm1}\ \{\text{mid}\} \qquad \vdash \{\text{mid}\}\ \text{stm2}\ \{\text{post}\}}{\vdash \{\text{pre}\}\ (\text{Seq stm1 stm2})\ \{\text{post}\}}$$

Assignment:

$$\frac{\vdash \text{NoSE e}}{\vdash \{\mathtt{post}^{\mathtt{e}}_{\mathtt{x}}\}\ (\text{Simple (Assign (Vref x) e)})\ \{\text{post}\}}$$

Array Assignment:

$$\frac{\vdash \text{NoSE i} \qquad \vdash \text{NoSE e}}{\vdash \{\mathtt{post}^{\mathtt{e}}_{\mathtt{a[i]}}\}\ (\text{Simple(Assign(Aryref a i) e)})\{\text{post}\}}$$

Table 6.7: Inference Rules for Partial Correctness, Part I

If/Then/Else (Two-Armed Conditional):

$$\frac{\begin{array}{c}\vdash (\texttt{postStm} = \texttt{EmptyStmt}) \vee \\ (\texttt{postStm} = (\texttt{Simp postSeEx}) \wedge \texttt{NoPreSE postSeEx}) \\ \vdash \texttt{SEM\_EQ (Seq (Simp expr) postStm)) (Simp ex)} \\ \vdash \texttt{IS\_VALUE expr test} \\ \vdash \{\texttt{pre} \;\wedge\; \texttt{test}\}\ \texttt{postStm}\ \{\texttt{trueCond}\} \\ \vdash \{\texttt{trueCond}\}\ \texttt{thenCode}\ \{\texttt{post}\} \\ \vdash \{\texttt{pre} \;\wedge\; \texttt{\~{}test}\}\ \texttt{postStm}\ \{\texttt{falseCond}\} \\ \vdash \{\texttt{falseCond}\}\ \texttt{elseCode}\ \{\texttt{post}\}\end{array}}{\vdash \{\texttt{pre}\}\ \texttt{IfElse (ex) thenCode elseCode}\ \{\texttt{post}\}}$$

While:

$$\frac{\begin{array}{c}\vdash (\texttt{preStm} = \texttt{EmptyStmt}) \vee \\ (\texttt{preStm} = (\texttt{Simp preSeEx}) \wedge \texttt{NoPostSE preSeEx}) \\ \vdash (\texttt{postStm} = \texttt{EmptyStmt}) \vee \\ (\texttt{postStm} = (\texttt{Simp postSeEx}) \wedge \texttt{NoPreSE postSeEx}) \\ \vdash \texttt{SEM\_EQ (Seq preStm (Seq (Simp testEx) postStm)) (Simp ex)} \\ \vdash \{\texttt{invariant}\}\ \texttt{preStm}\ \{\texttt{testState}\} \quad \texttt{IS\_VALUE testEx test} \\ \vdash \{\texttt{testState} \wedge \texttt{test}\}\ \texttt{postStm}\ \{\texttt{bodyCond}\} \\ \vdash \{\texttt{bodyCond}\}\ \texttt{body}\ \{\texttt{invariant}\} \\ \vdash \{\texttt{testState} \wedge \texttt{\~{}test}\}\ \texttt{postStm}\ \{\texttt{post}\}\end{array}}{\vdash \{\texttt{invariant}\}\ (\texttt{While ex body})\ \{\texttt{post}\}}$$

Function Call:

$$\frac{\begin{array}{c}\vdash \text{WF\_fnp pre (Func name vvals globls b) post} \\ \vdash \text{WF\_c (Simple (Call name ps))} \quad \vdash \text{vals} = \text{specialize\_varargs vvals ps} \\ \vdash \text{vals'} = \text{variants vals (APPEND (FV\_a qpost) globls)} \\ \vdash \text{y} = \text{APPEND vals globls} \quad \vdash \text{x} = \text{APPEND vals globls} \\ \vdash \text{x0} = \text{logicals x} \qquad \vdash \text{y0} = \text{logicals y} \\ \vdash \text{x0'} = \text{variants x0 (FV\_a qpost)} \\ \vdash \text{specpost} = \text{SUBS post (APPEND vals' x0') (APPEND vals y0)} \\ \vdash \text{postimpq} = \text{SUBS (specpost} \Rightarrow \text{qpost) newC\_Result genrC\_Result}\end{array}}{\begin{array}{l}\vdash \{(\texttt{pre}^{\texttt{vals'}}_{\texttt{vals}} \;\wedge\; (\forall \texttt{x.postimpq})^{\texttt{x}}_{\texttt{x0'}})^{\texttt{vals'}}_{\texttt{PL2CELps}}\} \\ \qquad\qquad \text{(Simple (Call name ps))}\ \{\text{qpost}\}\end{array}}$$

Table 6.8: Inference Rules for Partial Correctness, Part II

Sequence Assoc I:

$$\frac{\vdash \{\text{pre}\}\ (\text{Seq}\ (\text{Seq s1 s2})\ \text{s3})\ \{\text{post}\}}{\vdash \{\text{pre}\}\ (\text{Seq s1}\ (\text{Seq s2 s3}))\ \{\text{post}\}}$$

Sequence Assoc II:

$$\frac{\vdash \{\text{pre}\}\ (\text{Seq s1}\ (\text{Seq s2 s3}))\ \{\text{post}\}}{\vdash \{\text{pre}\}\ (\text{Seq}\ (\text{Seq s1 s2})\ \text{s3})\ \{\text{post}\}}$$

If/Then (One Armed Conditional):

$$\frac{\begin{array}{c}\vdash (\texttt{postStm} = \texttt{EmptyStmt}) \lor \\ \qquad (\texttt{postStm} = (\texttt{Simp postSeEx}) \land \texttt{NoPreSE postSeEx}) \\ \vdash \texttt{SEM\_EQ (Seq (Simp expr) postStm)) (Simp ex)} \\ \vdash \texttt{IS\_VALUE expr test} \\ \vdash \{\texttt{pre} \ \land \ \texttt{test}\}\ \texttt{postStm}\ \{\texttt{trueCond}\} \quad \vdash \{\texttt{trueCond}\}\ \texttt{code}\ \{\texttt{post}\} \\ \vdash \{\texttt{pre} \ \land \ \texttt{\~{}test}\}\ \texttt{postStm}\ \{\texttt{falseCond}\} \quad \vdash \texttt{falseCond} \Rightarrow \texttt{post}\end{array}}{\vdash \{\texttt{pre}\}\ \texttt{If (ex) code}\ \{\texttt{post}\}}$$

Table 6.9: Derived Partial Correctness Inference Rules

Variables which only appear in the assertion language are prefixed with a caret (^) and are referred to as "logical variables." Thus `^x` may appear in, say, a precondition, such as `^x` $>0 \Rightarrow$ `x` >0, but it would be incorrect (that is, violate well-formedness conditions, Sect. 6.2.5) to have `Simple (Assign (Vref (Var "^x" 0 Int)) (Const (CCint 0) Int))`.

We need to model C arrays in some detail in the assertion language. We define a HOL type which is a function, from numbers (indices) to values, and the size of the array. We create a symbolic numeric constant, `UNK`, to use if the array size is unknown. We define the array update function, `CA_PUT X i y`, which is a new array (function) and the original size. The new array is everywhere the same as the original array, X, except at i, where it has the value y. (This corresponds to [X|i:y] in [41, page 112].) Note that if the index is not in the range $[0, \mathtt{size})$, the value is undefined.

```
CA_PUT (CA f size) i y =
    CA (\j . (0 <= i /\ i < size) =>
                    ((i=j) => y | f j) | (@x.F))
       size
```

We actually prove and use a slightly more general version. It takes a variable rather than a constructor.

```
CA_PUT ar i y =
    CA (\j . (0 <= i /\ i < CA_SZ ar) =>
                    ((i=j) => y | ((CA_FN ar) j)) | (@x.F))
       (CA_SZ ar)
```

Indexing an array also checks the array boundaries. Note that we hardcoded the lower bounds of 0. This seems reasonable given our work in C, but it could be easily changed, at the cost of a little more work during proofs, if arrays could have a different lower bound.

```
CA_IDX (CA f size) i = ((0 <= i /\ i < size) => (f i) | (@x.F))
```

Upon casual inspection, one might object that this definition prevents us from proving results about indexing an array if the size is unknown. But this is entirely reasonable: if we know nothing about the size of an array, any access may be out of bounds, and thus, undefined! However if the size is even known symbolically and we can prove that the index is always less than this symbolic limit, e.g. `if (i<MAX) k=a[i];`, the bounds checks can be satisfied.

HOL type	*AST type*
`:ascii`	`Char`
`:num`	`Int`
`:bool`	`Int`
`:void`	`Void`
`:string`	`Ptr Char`
`:T ptr`	`Ptr T`
`:T list`	`Array T (Unknown size)`
`:T -> U`	`Funty T U`
`:'S`	`Struct "S"`
`:T #...# U`	`Struct "(T,...,U)prod"`

Table 6.10: Assertion Language Type to AST

AST type	*HOL type*
`Char`	`:ascii`
`Int`	`:num`
`Void`	`:void`
`Array Char`	`:string`
`Array T`	`:T list`
`Ptr Char`	`:string`
`Ptr T`	`:T ptr`
`Struct "S"`	`:'S`
`Struct "(T,...,U)prod"`	`:T #...# U`

Table 6.11: AST Type to Assertion Language

Although the conversion from assertion language (HOL) types to AST and back are generally symmetrical, a few remarks are in order.

- We define one HOL type constructor, `ptr`, and one new type, `void`.

- In Tables 6.10 and 6.11, we leave appropriate subtype conversions implicit. For instance, `Ptr T` actually converts to a `ptr` of the type which `T` converts to, or loosely, `:(ctype2hol T) ptr`.

- Since HOL types may carry the complete *structure* (names of component types), we represent them as an AST structure whose name is the print representation of the HOL type.

- It may appear that information is lost since both HOL types :num and :bool become AST type `Int`. However the context makes it clear when a :num should be a :bool. See note 1, page 68.

- Note that `Array Char` is converted to :string, but :string is converted to `Ptr Char`. This asymmetry reflects the C schizophrenia between arrays and pointers and our attempt to model C "strings" as HOL strings.

Converting C to the Assertion Language

Variants of AST variables with nonzero disambiguators are converted to HOL variables with an apostrophe and the disambiguator. Thus `Var "x" 2 Int` becomes `x'2:num`, but `Var "x" 0 Int` becomes `x:num`. To simplify the table, we only refer to a zero disambiguator, but nonzero disambiguators are handled, too.

`CHAR a` is defined as `HD (EXPLODE a)`. `EXPLODE` turns a HOL string into a list of HOL "ascii," that is, characters.

Here are explanatory notes for Table 6.12.

1. The function `castToBool` insures that the term is type :bool. There are three cases.

   (a) If the term is already :bool, it is returned.

   (b) If it is a variable, a variable of the same name, but of type :bool is returned.

   (c) Otherwise `term = 0` is returned reflecting C's rule that any nonzero value is true.

2. `2 EXP P` is 2 to the integer power `P`.

AST expression	*HOL expression*
Const (CCid "S") T	S:T (a symbolic constant)
Const (CCstr "S") Char	CHAR "S" (:ascii)
Const (CCstr "S") T	"S" (:string)
Const (CCint N) T	N (:num)
Var "S" 0 (Array T M)	CA (CA_FN (S:num->T)) M
Var "S" 0 T	S:T
Lval E	E
Vref E	E
Aryref A i	CA_IDX (EXPLODE A) i if A is type :string
Aryref A i	CA_IDX A i
Deref E	deref E
Field S f	(f:t1->t2) S where t1 is type of S and t2 is type of f
LUnary (AdrOf E)	adrof E
Unary (Not E)	~ E (note 1)
Binary E Add F	E + F
Binary E Sub F	E - F
Binary E Mul F	E * F
Binary E Div F	E DIV F
Binary E Mod F	E MOD F
Binary E ShL F	E * 2 EXP F (note 2)
Binary E ShR F	E DIV 2 EXP F (note 2)
Binary E Eq F	E = F
Binary E NEq F	~ (E = F)
Binary E Lt F	E < F
Binary E Gt F	E > F
Binary E LEq F	E <= F
Binary E GEq F	E >= F
Binary E And F	E $\wedge$ F
Binary E Or F	E $\vee$ F
Binary E BAnd F	BITAND E F (note 3)
Binary E BOr F	BITOR E F (note 3)
Binary E BXor F	BITXOR E F (note 3)

Table 6.12: AST Expression to Assertion Language

3. Bitwise functions are undefined. Their types are inferred from context. This means we can convert expressions with bitwise operators to the assertion language, but we can't prove anything about their value.

6.3 Formalizing the Unix Environment

6.3.1 The File System

A large part of the Unix environment is the permanent storage, or "file system." In Unix terms, a file system is just one volume, partition, or disk. However since we want to verify over everything which can be written to or read from, we lump all disks, etc. into one conceptual file system. In particular, the output to the user, stdout, isn't even a file, but is modeled as one. This corresponds well with the Unix convention of accessing I/O as a file, albeit with special drivers. At the level of semantics which we need to prove the security of `thttpd`, we only need a loose correspondence between a file path name, a unique file designator, and the contents of the file. This is in contrast to, say, Hayes' path-oriented description [27]. For example, we don't need detailed semantics of "parent directory" (the ".." entry), multiple hard links, etc. We make extensive use of incompletely defined functions in order to specify some relations between functions and their inputs and outputs without modeling all the details.

We define a file system as a set of files each having a unique identification number. The identification number corresponds roughly to an inode number. We model it as a CArray to deeply embed the notion of changing a file, which is an item of the array. Note that we don't explicitly associate directories, file names, or paths with files.

```
val unixFileSystem = :(^unixFile # num) CArray;
```

We use Windley's "records" package [43] to model a single file as a record of permissions and contents. (This creates a type constructor, `unixFile`.) We do not define the types `permission` or `fscontents` further. We do later define relationships between functions which operate on them.

```
create_record "unixFile"
   [("permissions", :'permission,
```

```
    ("contents",    :'fscontents];
```

For recursive function definitions, we need an inductive definition of files. We begin with an empty file and append strings to it.

```
new_constant{Name="empty_unixFile", Ty = :^unixFile};

new_constant{
    Name="appendFile",Ty = :string->^unixFile->^unixFile
};

!e f. ?!fn.
        (fn (empty_unixFile:^unixFile) = (e:'a)) /\
        (!n o. fn (appendFile n o) = f (fn o) n o)
```

We must distinguish between output to the user, stdout, and output to other files. For instance, writing to stdout cannot compromise information integrity (since stdout is not a part of the "real" file system) and anything written to stdout must nonconfidential (since the user sees it). We create a special constant to represent it in specifications and system axioms.

```
new_constant{Name="SYS_stdout", Ty = :num};
```

We incompletely define two functions. The first, `inodeNamed`, represents the mapping from names to file (roughly, inode) numbers. Since the way names map to files doesn't affect security, its type (and testing equality between instances) is all the specification we need. The second function, `getFile`, maps numbers to files in a file system. These can be thought of as primitive or abstract "look up" functions.

```
new_constant{Name = "inodeNamed", Ty = :string->num};
new_constant{
  Name="getFile",Ty = :^unixFileSystem->num->^unixFile
};
```

We model the state of a file system as a predicate from inodes to files. This predicate may be thought of as a snapshot of the contents of each file.

```
! P fs . SoFS P fs = (!inode. P inode (getFile fs inode))
```

Since we are verifying how code *changes* state, rather than what the state absolutely is, we can phrase our partial correctness conditions relatively. Informally the condition will be something like this: "if the file system begins in some state P and we run code C, the state will still be P for all files except perhaps for the log file." This looks something like the following.

$$\vdash \{\texttt{SoFS P fs}\}\ \texttt{C}$$
$$\{\forall \texttt{inode.inode} \neq \texttt{logfile} \Rightarrow \texttt{P inode (getFile fs inode)}\}$$

Finally we define a distinguished constant meaning *the* file system.

```
val SYS_FileSystem = (--`SYS_FileSystem:^unixFileSystem`--);
```

To define operating system calls, Sect. 6.4, and use them in proofs, we partially define additional predicates and relations. The function `inodeOf` returns the file number of the file indicated by a `FILE` pointer. It is a logical or conceptual function in that the mapping is not an actual data structure available in Unix. Indeed, the notion we use of uniquely identifying files by number is only loosely embodied in Unix i-nodes.

```
new_constant{Name = "inodeOf", Ty = :'FILE->num};
```

To partially specify its semantics, we define that the inodeOf of a "handle" of an inode is that inode. We also define that `fopen()` never returns stdout.

```
|- ! inode type.
    inodeOf (deref (FOPEN_handlefn inode type)) = inode


|- ! inode type.
    ~(inodeOf (deref (FOPEN_handlefn inode type)) = SYS_stdout
```

Similarly to `inodeOf` we define that function `inodeOfFileDes` returns the file number of the file indicated by a file descriptor. We also define that file descriptor 1 is stdout.

```
new_constant{Name = "inodeOfFileDes", Ty = :num->num};

|- inodeOfFileDes 1 = SYS_stdout
```

To prove file system confinement, we must formalize some Unix behavior related to relative paths. Rather than define directories, path names, root directories, parents, current working directory, and so forth, we just axiomatize some relations. The predicate `resolvePath` is intended to represent the path resolution: `.` means the current working directory (cwd), `..` is cwd's parent unless cwd is root, `/...` starts from root, etc. The auxiliary predicates `dirParent` and `resPathRest` are place holders for the rest of the functionality which we don't need to define for the proof: `dirParent` returns the parent of a directory and `resPathRest` does everything else (relative and absolute paths).

```
new_constant{Name = "resPathRest",
    Ty = :string->string->string->string};
new_constant{Name = "dirParent",
    Ty = :string->string};

! root cwd path .
    resolvePath root cwd path = (
        (path = ".") => cwd |
        (path = "..") =>
            ((cwd = root) => root | dirParent cwd) |
        resPathRest root cwd path)
);
```

6.3.2 Process ID and Permissions

In Unix systems[3], processes have identification numbers pertinent to our work: user ID (UID), representing the person running the program, and group ID (GID), representing the person's group affiliation. To allow a program to run as a surrogate for the owner, a process, which is created with the ID of the process starting it, may change or set its UID to its owner under some circumstances. To

[3] Much of this section was taken from [8, App. B].

keep track of this, each process actually has three IDs: the real ID, the "effective" ID (the ID which it is using), and a "saved" ID. We model all of these as global values which can only be set through the appropriate system calls.

Files also have a UID and a GID, and have three sets of permissions:

- those for the file owner ("user"),
- those for people in the same group as the file ("group"), and
- those for any other person in the world ("other").

If the process' UID matches the file UID, user permissions are checked to authorize the operation. If the UID's don't match, but the GID's match, group permissions are checked. If neither UID's nor GID's match, the "other" or world permissions are checked.

Although permissions are typically used hierarchically, it need not be so. Thus a file may be readable by everyone on the system except the owner, if it has read permission for others but no read permission for the owner.

6.4 Operating System and Library Calls

For completeness, we must consider the axioms of input/output calls, miscellaneous operating system calls and library functions, for a proof is no better than the axioms upon which it is based.

In this section we explain some of the problems and nuances of axiomatizing system calls, Sect. 6.4.1, and leave the details for Appendix B. We also explain the assumption we made to prove security, Sect. 6.4.2, and give a guidebook with hints on formalizing functions, Sect. 6.4.3.

6.4.1 Nuances of Axiomatizing System Calls

Negative Numbers

Several operating system calls return -1 to indicate an error which `thttpd` sometimes acts on. However throughout the rest of the formalization the built-in natural numbers (HOL `num`'s) serve as a good model of C `int`'s. To handle error checking for system calls, we took a short cut of partially defining `int_neg` to represent the negative or unary minus operator. Here is the definition.

```
new_constant{Name = "int_neg", Ty = :num->num};
```

We wrote some axioms to prove theorems containing `int_neg`.

```
(* int_neg is one-to-one *)
|- ! i i' . (int_neg i = int_neg i') = (i = i')

(* int_neg returns 0 for 0 *)
|- int_neg 0 = 0

(* int_neg is never greater than 0 *)
|- !i . ~(int_neg i > 0)
```

Although these axioms lead to an inconsistency, as shown below, we believe our proof is valid.

1. `|- (int_neg 0=int_neg 1)=(0=1)` by axiom that `int_neg` is one-to-one.
2. `|- (int_neg 0=int_neg 1) = F` by 1 and arithmetic.
3. `|- (0=int_neg 1) = F` by 2 and axiom that `int_neg` returns 0 for 0.
4. `|- ~(int_neg 1 > 0)` by axiom that `int_neg` is never greater than 0.
5. `|- int_neg 1 <= 0` by 4 and theorem NOT_GREATER.
6. `|- int_neg 1 = 0` by 5 and theorem LESS_EQ_0 (since HOL uses natural numbers).
7. `|- (0=0) = F` by 3 and 6.
8. `|- T = F` by 7 and arithmetic.
9. `|- F` by 8 and boolean logic.

Why not model `int`'s as integers? When we began, there was not a good package in HOL to handle arithmetic on integers, as there was for the built-in `num`'s. Also computers only handle a range of integers (with overflow handling differing from language to language), so integers are not a precise model, either. Since the program used negative numbers only marginally, we felt the minor benefit would not be worth the effort. We were wrong.

Modeling negative numbers as natural numbers unavoidably introduced an inconsistency. We did not redo our model because it would take extensive rework and the actual effect on the proof would be small.

The reader may ask, why not use a special type to represent the return value from `chdir()`, `fprintf()`, `open()`, etc.? In some cases, the return value is not used at all, which is no problem. In other cases, the return value is only compared with zero (`if (0 != fn()) ...`). We should be able to accommodate this by adding another implicit "type casting" to the conversion from AST to assertion language (Sect. 6.2.7). However there are cases where the return value is saved in a variable for later use. Using a special type implies changing the types of some variables because of the context of their use. This would be somewhat complex, ad hoc, and strays from the C model.

If any changes are made in this facet, we recommend modeling at least with full integers rather than more limited measures like modeling with a special type or natural numbers. The complexity of the model and the proofs increases very little and may, in fact, decrease. Even though integers are not a precise model, another proof phase could show that with certain assumptions about the size of `int`'s, there is no overflow.

Unstructured Flow

We don't model nonstructured jumps well, so the axiom for `exit()` is particularly poor. As coded `exit()` produces a totally undefined state. This allows us to prove theorems for "filter" conditionals, such as the following.

$$\vdash \{\mathtt{T}\}\ \mathtt{if}\ \ (\mathtt{x} < 0)\ \ \mathtt{exit}(1);\ \{x \geq 0\}$$

The postcondition follows if the test fails. To cover the case that the test succeeds, we must prove the following, which we can now.

$$\vdash \{\mathtt{T} \wedge x < 0\}\ \mathtt{exit}(1);\ \{x \geq 0\}$$

Unfortunately the current coding of `exit()` prevents us from analyzing the final state of the program formally. It also allows us to prove the following.

$$\vdash \{\mathtt{T}\}\ \mathtt{exit}(1);\ \ \mathtt{y} = \mathtt{x};\ \{y \geq 0\}$$

Since we can prove this intermediate.

$$\vdash \{\mathtt{T}\}\ \mathtt{exit}(1);\ \{x \geq 0\}$$

Therefore total correctness must be more specific in that the statement not only terminate, but execution reaches the final statement (reachability).

Instead `exit()` should be coded to express a transfer of control to the end of the program. One possible remedy is to use multiple post conditions, as suggested in [1] and refined for C in [38]. The `exit()` call, and thus `main()` and `thttpd` itself, would have `F` as the "sequence" postcondition and the constraint as the "exit" postcondition. We would then have to prove that execution actually reaches the points of interest, as opposed to the unreachable `y=x;` above, in addition to partial correctness.

Varargs

We approximate "varargs" (a function with a variable number of arguments) by giving the fixed components, in this case `stream` and `format`, then a special variable, `varargs`. The special variable is bound to the actuals by `specialize_varargs` in the function call inference rule, Table 6.8, page 64. However we cannot specify, for instance, that `scanf()` may change the memory indicated by the pointers.

Description Precision

The call axioms are *not* complete descriptions by any stretch of the imagination. They model enough of the function for our proof. In many cases we can make the description accurate (correct), but not precise (incomplete) by using underdefined functions.

Consider our call axiom for `read()`. This function tries to read up to `nbyte` bytes from the file descriptor, `filedes`, into a buffer, `buf`.

```
|- WF_fnp (T)
    (Func (Var "read" 0 Int)
     [Var "fildes" 0 Int; Var "buf" 0 (Ptr Char);Var "nbyte" 0 Int]
     [Var "errno" 0 Int] NOBody)
    ((C_Result = int_neg 1) /\ (?READ_errno.errno=READ_errno) \/
     (C_Result = 0) \/
     ~(^fildes = int_neg 1) /\ (C_Result > 0) /\
                (C_Result = readSpec(^fildes, ^buf, ^nbyte)))
```

The return value, `C_Result`, and the final state of the buffer and the file descriptor depend upon many factors:

1. whether the file descriptor is valid, and it was opened for reading,
2. whether the file is capable of seeking,
3. the mode, if it is a `STREAMS` file,
4. exactly when and which type of interrupt (signal, in Unix) is received, if one is received,
5. whether the descriptor or object are marked for non-blocking I/O, and
6. how many bytes are left before the end of the file.

However, to prove that `thttpd` is secure, little of this matters. The axiom only distinguishes between the three cases of an error return, `(C_Result = int_neg 1)` $\wedge$ `(?READ_errno.errno = READ_errno)`, an end of file return, `(C_Result = 0)`, and a successful read, `~(^fildes = int_neg 1)` $\wedge$ `(C_Result > 0)` $\wedge$ `(C_Result = readSpec(^fildes, ^buf, ^nbyte))`. The function `readSpec()` is underspecified. We only give the number of arguments, and, implicitly via HOL type inference, the return type and types of arguments. Nothing else about `readSpec()` is formalized. This underspecification lets us formalize the parts which are needed for the proof, while saving enormous amounts of time and keeping the specification simpler to understand. The axiom has limited precision, but is still accurate.

Of course, in the future another program or proof of a different property of this program may need a more precise or detailed formalization. Then `readSpec()`

may be defined in more detail. The general question of how one would provide a completely accurate description, yet easily extract those parts which are relevant to the proof is open.

6.4.2 An Assumption

The correctness of `thttpd` depends on a specific fact about the parameters passed to a starting program. In C a program starts by the system calling a particular function, `main()`. It is passed an array of strings, called `argv` in `thttpd`, and the size of the array, called `argc`.

Assumption 1 (argc_Arraysizeargv) *CA_SZ argv = argc*

In `main()` we depend on the fact that `argc` is, indeed, the size of the array, `argv`. With this assumption, we can infer that the accesses to `argv` in the following `thttpd` code are valid.

```
if (argc>1) strncpy(remotehost,argv[1],MAXSIZE);
       else strcpy(remotehost,"nowhere");
if (argc>2) strncpy(remoteuser,argv[2],MAXSIZE);
       else strcpy(remoteuser,"nobody");
```

6.4.3 A Guidebook to Formalizing System and Library Calls

We must write axioms for operating system calls and library functions instead of proving theorems since we don't have source code. Even if we did, the theorems would be tied to a particular implementation. We can view the axioms as specifications. However even figuring out how to express some behaviors can be difficult. This section has some hints for formalizing calls.

Verify Skeleton Code

The first suggestion is to write a piece of skeleton code and verify it to get the "form" right. For example, the `chdir()` function may succeed or fail. If it succeeds, it sets the current working directory and returns 0. If it fails, the global variable `errno` is set and -1 is returned. The following code roughly embodies this behavior.

```
int chdir(char *path)
{
    if (nondeterministic==0)
        C_Result=0;
    else
        C_Result=1;
    if (C_Result==0)
        SYS_cwd=path;
    else
        errno=CHDIR_errno;
}
```

The code is simple enough to prove quickly, yet, it models several features which are important to our verification.

Level of Detail

Formalize only as much as necessary for the proof. Begin with underdefined functions, general definitions, or assumptions, and only define them in more detail as needed for the proof. It is not uncommon to only need a few high-level theorems about a function.

At one time our proof depended on the fact that the time formatted to `YY/MM/DD HH:MM:SS` is always a string less than 20 characters since we copy it into a buffer of 20 characters. More formally, `strlen(strftimeSpec("%Y/%m/%d %T",s))<20`. (In fact, the string is always 18 characters. Since we ignore many possible memory overwrite problems, our final proof doesn't depend on on this anymore.)

Here was an interesting trade-off: we could have specified in great detail how `strftime` works, how it formats `%Y` and `%m`, etc., and proven the above as a theorem. However since the details were never needed and since it is a very minor point in the proof, we chose instead to make it an assumption. It turns out that subsequent changes in the proof eliminated the need for the theorem altogether.

Logical vs. Program Variables in `WF_fnp`

Only program variables may appear in function call (`WF_fnp`) preconditions. Logical variables may not be used. For instance, in `fprintf()` we have a precondition on the state of the file system.

Postconditions may contain both program variables and logical variables. A logical variable denotes the value of the variable at the time of entrance. A program variable denotes the value of the variable at the time of exit. The postcondition expresses the logical relation between these two sets of values, thus describing the effect of calling the procedure [32, pp. 68–69].

Formal Parameters in Postconditions

Postconditions should be written in terms of logical versions of pass-by-value formal parameters, not program versions. Suppose we have the following function.

```
int INC(int p)
{
    return (p+1);
}
```

The following is a theorem (a brief version of the AST is shown).

$$\texttt{WF\_fnp}\ \{\texttt{T}\}\ (\texttt{Func(Var "INC" Int)}\ [\texttt{Var "p" Int}]\ []\ldots)$$
$$\{\texttt{C\_Result} = \texttt{\^{}p} + 1\}$$

Notice that we use the logical version of the formal parameter, that is `^p` instead of just program version, which would be `p`. If we do not use the logical version, the actual is not substituted for the formal in the Call rule A.3.

Global Variables in Conditions

Global variables can be included directly in preconditions. In postconditions, they should be coded as above: values at entrance are denoted by logical variables, values at exit are denoted by program variables. Given the following function,

```
int aglobal;

int accum(int p)
{
    aglobal = aglobal + p;
}
```

We can prove the following theorem.

$$\begin{aligned}&\texttt{WF\_fnp}\ \{\texttt{T}\}\ (\texttt{Func}(\texttt{Var}\ \texttt{"accum"}\ \texttt{Int})\ [\texttt{Var}\ \texttt{"p"}\ \texttt{Int}]\\ &\qquad\qquad [\texttt{Var}\ \texttt{"aglobal"}\ \texttt{Int}]\ldots)\\ &\quad \{\texttt{aglobal} = \hat{}\texttt{aglobal} + \hat{}\texttt{p}\}\end{aligned}$$

Failures or Multiple Results

Most operating system calls have multiple, distinct outcomes. For instance, `fprintf()` may write to a file on disk, or, if the disk is full, it may fail. From the point of view of the program, the calls have nondeterministic results.

Nondeterminism may be modeled as a disjunction of possible postconditions. For example, `fprintf()` returns the number of bytes written if it succeeds or a negative number if it fails. It can be partially modeled as

$$\begin{aligned}&\vdash \{\texttt{T}\}\ \texttt{fprintf}(\texttt{fp}, \texttt{format}, \ldots);\\ &\quad \{(\texttt{C\_Result} < 0) \vee (\texttt{C\_Result} \geq 0 \wedge \textit{changes to fp})\}\end{aligned}$$

CHAPTER 7

THE PROOF

We finally come to the proof itself. Unfortunately it is not clear how best to present the proof. The fully formal, machine-mediated proof is about 3,300 lines of HOL commands in 18 files. But the purpose of a proof is to convey information, usually to increase one's confidence in a theorem [15]. We also wanted to demonstrate that a proof can help one gain additional *insight* into a program, its assumptions, domain of correct operation, limitations, etc., in addition to increasing confidence in its correctness.

Instead of the complete proof, this chapter is a summary of the proof. We give the significant theorems and lemmas leading up to the theorem of security. For the proof of each theorem we give conditions for each C code statement, informal explanations of the steps of the proof, and various notes. We hope to present enough detail to show the intermediate conditions we found useful and when and how we used axioms, definitions, and theorems. Section 7.2 explains the actual software architecture of `thttpd` and gives a very high level outline of the proof: the theorems we prove and the order of presentation. The http server was designed with redundant features to be secure even in the presence of some failures. Section 7.1 discusses possible future work about proving feature independence.

The top-level theorem of security is a conjunction of the highest level properties of confidentiality and information integrity. In other words, we can prove that `thttpd` is secure if we show that it has confidentiality *and* information integrity as mentioned in Chapter 3. Section 7.3 is the proof that `thttpd` maintains confidentiality, and Sect. 7.4 is the proof of information integrity. Section 7.5, gives the almost-trivial proof that `thttpd` is secure, that is, that calling `thttpd`'s `main()` routine maintains security.

7.1 Proving Design Features Independently

The properties of confidentiality and information integrity are established in `thttpd` by multiple, redundant design features. One could imagine exploring this intentional redundancy by proving that a property holds when only one or two of the features hold [7]. For instance confidentiality is established by just file input confinement (reading only files which are owned by "www" and "world" readable) *or* just file system access confinement (using `chroot()` to only access files in the "web" area). Unfortunately this is beyond the scope of this "simple" proof.

One possible approach for the future is to prove weaker high level properties. For instance, rather than formally defining confidentiality as file input *and* file system confinement, we could define it to be file input *or* file system confinement. We could then prove that confidentiality is maintained given either feature alone.

However, since the code actually has both properties, we must make sure we prove them independently. For instance, we may not notice that we are proving file input confinement piece by piece while proving file system confinement. Hoffman gives [31] the following method to determine feature independence. For each of n features p_i, find the smallest set of axioms $A(p_i)$ upon which that feature depends. Feature i is independent of other features if and only if its set of axioms is not a subset of any other features' axioms, that is, $A(p_i) \not\subset \bigcup_{j=1,\ldots,i-1,i+1,\ldots,n} A(p_j)$. If no feature's axioms is a subset of the others, the features are independent.

If we were concerned about particular failures, we could model those by writing modified system axioms incorporating those failures. Exploring possible failures is probably more profitable through fault trees or some other probabilistic analysis rather than formal proofs.

7.2 Overview

Although the code for the latest version of `thttpd` is given in App. C, we explain here the structure of the code, since the proof is similarly structured. We then briefly relate each of the major theorems.

7.2.1 Structure of `thttpd`

The function call tree of `thttpd` is given in Fig. 7.1. Strictly speaking `LOG2` and `LOG4` are macros, not functions, but the way they are written allows us to treat them functions. When a request arrives at the server, `thttpd` is started. The `main()` routine establishes confinements, parses the input, and checks that

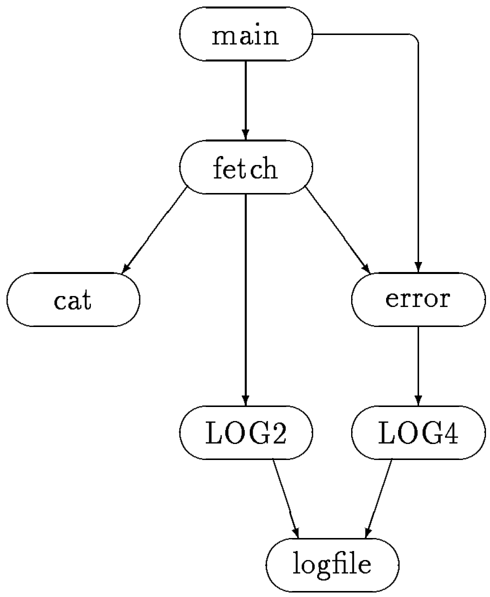

Figure 7.1: Function Call Tree for `thttpd`

the request is a "get." If everything is okay, it calls `fetch()` which first checks security for the requested file. If the file is public, `fetch()` calls `cat()`, logs the request, and exits. `cat()` opens the file, copies the contents to the user, then closes it. The function `error()` returns an error message to the user, logs it, and exits. `LOG2` and `LOG4` open the log file, call `logfile()` to write a timestamp, then they write their message and close the log.

As we mentioned above, we prove one theorem that `thttpd` maintains confidentiality and another theorem that it maintains information integrity. Each theorem proceeds by proving a confidentiality or information integrity property theorem for each function beginning from lowest level of the function call hierarchy to the highest level. Table 7.1 summarizes the confidentiality theorems, or briefly, that users cannot access information not marked for the public. Table 7.2 summarizes the information integrity theorems, or, that users cannot cause `thttpd` to change the file system, except for the log file. In the cases of `fetch()` and `error()`, we actually prove the function never returns, however the table shows the penultimate condition.

We almost always use backward proofs. That is, we state the theorem to be proved, and break it down into simpler and simpler sufficient conditions or subgoals. Each simplification step must be, of course, justified by our inference rules.

`logfile()`	If the file pointer passed is not `stdout`, confidentiality is maintained.
`LOG2`	Confidentiality is maintained.
`LOG4`	Confidentiality is maintained.
`error()`	If the error message passed is not confidential, confidentiality is maintained.
`cat()`	If the named file has a public owner and permission and the file system is confined, confidentiality is maintained.
`fetch()`	If the file system is confined, confidentiality is maintained.
`main()`	Confidentiality is maintained.

Table 7.1: Confidentiality Theorems

`logfile()`	Nothing is changed, except possibly the file passed.
`LOG2`	Nothing is changed, except possibly the log file.
`LOG4`	Nothing is changed, except possibly the log file.
`error()`	Nothing is changed, except possibly the log file and `stdout`.
`cat()`	Nothing is changed, except possibly `stdout`.
`fetch()`	Nothing is changed, except possibly the log file and `stdout`.
`main()`	The function never returns.

Table 7.2: Information Integrity Theorems

The proof of each theorem proceeds from the beginning of the function to the end, emulating in some aspects the flow of control. Since we use axiomatic semantics, steps are annotated primarily by giving the postcondition for each statement. (The precondition is the preceding step's postcondition.) We give many details in the first proofs, and skip progressively more in later proofs for brevity.

7.3 Proving Confidentiality

7.3.1 `logfile()`

Theorem 1 (WF_fnp_cf_logfile)

WF_fnp
(SoFS hasConfidentialityFile SYS_FileSystem $\wedge$
~(inodeOf (deref F) = SYS_stdout))

(Func (Var "logfile" 0 Void)
[Var "F" 0 (Ptr (Struct "FILE"))]
[Var "errno" 0 Int;
Var "remotehost" 0 (Array Char (CCid "BUFSIZE"));
Var "remoteuser" 0 (Array Char (CCid "BUFSIZE"));
Var "timestamp" 0 (Array Char (CCint 64));
Var "TZ" 0 (Struct "TZstruct");
Var "SYS_FileSystem" 0 (Struct array of files)]
(SOMEBody ...))

(SoFS hasConfidentialityFile SYS_FileSystem)

The predicate `WF_fnp` takes three arguments: a precondition, a function declaration, and a postcondition. The function declaration is

1. the function name and return type,
2. the parameter list[1],
3. a list of all the global variables which this function might read or change, either directly or through functions it calls, and

[1]Square brackets ([and]) delimit a list.

4. the AST of the body code of the function (elided for brevity). Instead the original C code is shown at each step of the proof.

Informally this theorem says, if the file system has confidentiality (State of File System hasConfidentiality) and we won't be writing to `stdout` (the file pointer F is not `stdout`), then when `logfile()` finishes, the file system will still have confidentiality. The predicate `SoFS` applies the predicate `hasConfidentiality-File`, in this case, to every file in the file system, `SYS_FileSystem`.

Proof.

By rule 6.2, page 62, we reduce this to proving that the call is syntactically correct and the partial correctness of the body block, with formals initialized to actuals. By rule 6.1 and the function call axioms SYS_fprintf, SYS_localtime, SYS_strftime, and SYS_time, we prove that the call is syntactically correct[2]. Then by the block inference rule in Table 6.7, page 63, we reduce the proof of the body block to a proof of the body (code).

Since the proof of every function uses rules 6.2 and 6.1 and the body inference rule similarly, in future proofs we will only mention the function call axioms and theorems used.

1. `t=time(NULL);`

   Postcondition:

   ```
   (F = ^F) /\ (SoFS hasConfidentialityFile SYS_FileSystem) /\
   ~(inodeOf (deref F) = SYS_stdout) /\
   (?some_time.t = some_time)
   ```

   The first three clauses follow from the precondition. (The first clause shows the initialization of the formal variable with a logical value representing the actual variable's value.) The last follows from SYS_time with the following intermediate condition and assignment.

[2]The primary concern is that all global variables used by called functions are listed as global variables used by this function.

```
(F = ^F) /\ (SoFS hasConfidentialityFile SYS_FileSystem) /\
~(inodeOf (deref F) = SYS_stdout) /\
(?some_time.C_Result = some_time)
```

Note that we do not exclude the possibility that `time()` returns -1. Fortunately `localtime()` seems to behave acceptably with a negative time.

2. `strftime(timestamp, 20, "%Y/%m/%d %T", localtime(&t));`

   Postcondition:

   ```
   (F = ^F) /\ (SoFS hasConfidentialityFile SYS_FileSystem) /\
   ~(inodeOf (deref F) = SYS_stdout)
   ```

   First, we separate the effect of the call to `localtime()` with the following intermediate condition.

   ```
   (F = ^F) /\ (SoFS hasConfidentialityFile SYS_FileSystem) /\
   ~(inodeOf (deref F) = SYS_stdout) /\
   (?tsptr.C_Result = tsptr)
   ```

   The first three follow from the precondition. The last follows from SYS_localtime, that is, the result is a pointer to a time structure. The postcondition follows from the intermediate and SYS_strftime.

3. `fprintf(F,"%s %s %s ",remotehost,remoteuser,timestamp);`

   The postcondition is the function postcondition. It follows from SYS_fprintf specialized with the predicate `hasConfidentialityFile`, the definition of `hasConfidentialityFile`, and the precondition that `F` is not `stdout`.

7.3.2 `LOG2`

Theorem 2 (WF_fnp_cf_LOG2)
WF_fnp
(SoFS hasConfidentialityFile SYS_FileSystem)
(Func (Var "LOG2" 0 Void)

[Var "x" 0 (Ptr Char); Var "y" 0 (Ptr Char)]
[Var "SYS_FileSystem" 0 (Struct array of files);
Var "remotehost" 0 (Array Char (CCid "BUFSIZE"));
Var "remoteuser" 0 (Array Char (CCid "BUFSIZE"));
Var "timestamp" 0 (Array Char (CCint 64));
Var "TZ" 0 (Struct "TZstruct"); Var "errno" 0 Int;
Var "SYS_cwd" 0 (Ptr Char); Var "SYS_root" 0 (Ptr Char);
Var "WWWlog" 0 (Ptr Char)]
(SOMEBody ...))
(SoFS hasConfidentialityFile SYS_FileSystem)

Informally, if the file system is confidential, then when `LOG2` finishes, the file system will still be confidential.

Proof:

The top level correctness is proved with the theorem WF_fnp_cf_logfile and the axioms SYS_fopen, SYS_fprintf, and SYS_fclose.

1. `F = fopen(WWWlog,"a+");`

   Postcondition:

   ```
   (SoFS hasConfidentialityFile SYS_FileSystem) /\
   ((F = NULL) \/
    (?FOPEN_handlefn . F =
           FOPEN_handlefn(inodeNamed WWWlog) "a+"))
   ```

   The first clause follows from the precondition. The second clause follows from SYS_fopen with the following intermediate condition and assignment.

   ```
   (SoFS hasConfidentialityFile SYS\_FileSystem) /\
   ((C_Result = NULL) \/
    (?FOPEN_handlefn . C_Result =
           FOPEN_handlefn(inodeNamed WWWlog) "a+"))
   ```

2. `if (F != NULL) {logfile(F);fprintf(F,x,y);}`

   Postcondition:

   ```
   SoFS hasConfidentialityFile SYS_FileSystem
   ```

   We use `IFTHEN_TAC` and `BLOCK_TAC` to get to the body of the conditional. We prove it as follows.

   (a) Strengthen the precondition to say the file system is confidential and F is not stdout.

   ```
   (SoFS hasConfidentialityFile SYS_FileSystem) /\
   ~(inodeOf (deref F) = SYS_stdout)
   ```

   We prove that the stronger precondition follows with the axiom fopenHandle_not_stdout.

   (b) `logfile(F);`

   Postcondition:

   ```
   (SoFS hasConfidentialityFile SYS_FileSystem) /\
   ~(inodeOf (deref F) = SYS_stdout)
   ```

   This follows from the precondition and WF_fnp_cf_logfile.

   (c) `fprintf(F,x,y);`

   Postcondition is the `if` statement postcondition. It follows from the definitions of SoFS and hasConfidentialityFile and SYS_fprintf.

3. `fclose(F);`

   The postcondition is the function postcondition. It follows from SYS_fclose.

7.3.3 `LOG4`

Theorem 3 (WF_fnp_cf_LOG4)
WF_fnp
(SoFS hasConfidentialityFile SYS_FileSystem)
(Func (Var "LOG4" 0 Void)

[Var "x" 0 (Ptr Char); Var "y" 0 (Ptr Char);
Var "z" 0 (Ptr Char); Var "w" 0 (Ptr Char)]
[Var "SYS_FileSystem" 0 (Struct array of files);
Var "remotehost" 0 (Array Char (CCid "BUFSIZE"));
Var "remoteuser" 0 (Array Char (CCid "BUFSIZE"));
Var "timestamp" 0 (Array Char (CCint 64));
Var "TZ" 0 (Struct "TZstruct"); Var "errno" 0 Int;
Var "SYS_cwd" 0 (Ptr Char); Var "SYS_root" 0 (Ptr Char);
Var "WWWlog" 0 (Ptr Char)]
(SOMEBody ...))
(SoFS hasConfidentialityFile SYS_FileSystem)

Proof:

The proof of `LOG4` is nearly identical to that of `LOG2`.

7.3.4 `error()`

Theorem 4 (WF_fnp_cf_error)
WF_fnp
(SoFS hasConfidentialityFile SYS_FileSystem $\wedge$ *nonConfidentialS s)*
(Func (Var "error" 0 Int)
[Var "s" 0 (Ptr Char)]
[Var "SYS_FileSystem" 0 (Struct array of files);
Var "remotehost" 0 (Array Char (CCid "BUFSIZE"));
Var "remoteuser" 0 (Array Char (CCid "BUFSIZE"));
Var "timestamp" 0 (Array Char (CCint 64));
Var "TZ" 0 (Struct "TZstruct");
Var "errno" 0 Int; Var "WWWlog" 0 (Ptr Char);
Var "SYS_root" 0 (Ptr Char); Var "SYS_cwd" 0 (Ptr Char);
Var "bs1" 0 (Array Char (CCid "BUFSIZE"));
Var "name" 0 (Array Char(CCid "BUFSIZE"))]
(SOMEBody ...))
(F)

Informally, if the file system is confidential and the string passed is not confidential, then `error()` never finishes. This is a poor postcondition, but we cannot express it better since we cannot give a condition for a transfer of control other than the next sequential statement. As we explain in Sect. 6.2.1, this may be expressed with a more elaborate model. For now we settle for manually inspecting that the final condition is "the file system is still confidential."

Proof:

The top level correctness is proved by the theorem WF_fnp_cf_LOG4 and the axioms SYS_exit and SYS_printf.

1. ```
   printf("HTTP/1.0 302 Found\n");
   printf("Server: ManAl/0.1\n");
   printf("MIME-version: 1.0\n");
   printf(REDIRECT);
   printf("Content-type: text/html\n");
   printf("<HEAD><TITLE>Document moved</TITLE></HEAD>\n");
   printf("<BODY><H1>Document moved</H1>\n");
   printf(ERRORLINE);
   ```

   Postcondition:

   ```
   SoFS hasConfidentialityFile SYS_FileSystem /\
   nonConfidentialS s
   ```

   We prove these all with the same postcondition using the axiom SYS_printf, the definition of SoFS, hasConfidentialityFile, and nonconfidentiality of fixed strings and parameters.

2. ```
   printf("(%s) </BODY>\n",s);
   ```

   Postcondition:

   ```
   SoFS hasConfidentialityFile SYS_FileSystem /\
   nonConfidentialS s
   ```

We prove this similarly, except we use the precondition that the string passed is nonconfidential, so writing it doesn't compromise confidentiality.

3. `LOG4("Error:%s - %s %s\n",s,bs1,name);`

   Postcondition:

   ```
   SoFS hasConfidentialityFile SYS_FileSystem
   ```

   We prove this from the precondition and the theorem WF_fnp_cf_LOG4. We note that the precondition to the last command shows that the file system is still confidential, so we are satisfied, even though we cannot formally prove what we want.

4. `exit(1);`

   The postcondition is the function postcondition. We prove it with the axiom SYS_exit.

7.3.5 `cat()`

Theorem 5 (WF_fnp_cf_cat)

WF_fnp
(SoFS hasConfidentialityFile SYS_FileSystem ∧
nonConfFileOwner (inodeNamed s) ∧
nonConfFilePerm (inodeNamed s) ∧
fsconfined SYS_cwd SYS_root)
(Func (Var "cat" 0 Void)
[Var "s" 0 (Array Char (CCid "UNK"))]
[Var "SYS_FileSystem" 0 (Struct array of files);
Var "SYS_root" 0 (Ptr Char); Var "SYS_cwd" 0 (Ptr Char);
Var "SYS_stdout" 0 Int; Var "errno" 0 Int;
Var "bs2" 0 (Array Char (CCid "BUFSIZE"))]
(SOMEBody ...))
(SoFS hasConfidentialityFile SYS_FileSystem)

Informally, if

- the file system is confidential,
- the file named "s" is for outside viewing (the owner and permissions show that it is nonconfidential), and
- the current working directory and file system root are properly confined,

then when `cat()` finishes, the file system will still be confidential.

Proof:

Top level correctness is proved with the axioms SYS_open, SYS_read, SYS_write, and SYS_close.

1. `i=open(s,0);`

   Postcondition:

   ```
   SoFS hasConfidentialityFile SYS_FileSystem /\
   (~(i = int_neg 1) ==> nonConfidentialFD i)
   ```

   The second clause of the postcondition says if the open succeeds, `i` is a nonconfidential file descriptor. We prove this by separating the call to `open()` with the intermediate condition below, using the axiom SYS_open, the definition of nonConfidentialFD, and assignment.

   ```
   SoFS hasConfidentialityFile SYS_FileSystem /\
   (~(C_Result = int_neg 1) ==> nonConfidentialFD C_Result)
   ```

2. `while ((n=read(i,bs2,MAXSIZE)) > 0) write(1,bs2,n);`

   Postcondition:

   ```
   SoFS hasConfidentialityFile SYS_FileSystem /\
   (~(i = int_neg 1) ==> nonConfidentialFD i)
   ```

   We prove this using the following as an intermediate condition (that is, the test state in Fig. 5.4) after the side effects in the test expression. We break the `while` loop into proving the cases outlined below: termination, test expression yields the test state, and the body reestablishes the invariant.

```
SoFS hasConfidentialityFile SYS_FileSystem /\
(~(i = int_neg 1) ==> nonConfidentialFD i) /\
(n > 0 ==> nonConfidentialS bs2)
```

Notice we added the condition that says, in essence, anything in the buffer `bs2` is confidential if something was read. The specification of `read()` allows it to arbitrarily not read characters. Since information would only be written to the user, and possibly be of concern, if something was read, this seemed easier than defining that the buffer is initialized to nonconfidential information.

(a) We prove the termination condition with standard logic analysis.

(b) Using the axioms SYS_read and `int_neg` is never greater than 0, the definition that anything read from a nonconfidential file descriptor is nonconfidential, assignment, and the following intermediate condition, we prove that the "test state" follows from test expression.

```
SoFS hasConfidentialityFile SYS\_FileSystem /\
(~(i = int_neg 1) ==> nonConfidentialFD i) /\
(C_Result > 0 ==> nonConfidentialS bs2)
```

(c) We are left with showing that executing the body (the `write()` call) reestablishes the invariant. We use the axiom SYS_write and the definitions of SoFS, hasConfidentialityFile, nonConfidential, and nonConfidentialS.

3. `close(i);`

   We prove that executing `close()` results in the function postcondition using the axiom SYS_close.

7.3.6 `fetch()`

Theorem 6 (WF_fnp_cf_fetch)

WF_fnp

(SoFS hasConfidentialityFile SYS_FileSystem ∧ fsconfined SYS_cwd SYS_root)

(Func (Var "fetch" 0 Void)
[ ]
[Var "bs1" 0 (Array Char (CCid "BUFSIZE"));
Var "bs2" 0 (Array Char (CCid "BUFSIZE"));
Var "name" 0 (Array Char (CCid "BUFSIZE"));
Var "remotehost" 0 (Array Char (CCid "BUFSIZE"));
Var "remoteuser" 0 (Array Char (CCid "BUFSIZE"));
Var "timestamp" 0 (Array Char (CCint 64));
Var "CHECKUSER" 0 Int; Var "WWWlog" 0 (Ptr Char);
Var "TZ" 0 (Struct "TZstruct"); Var "errno" 0 Int;
Var "buf" 0 (Struct "stat"); Var "SYS_stdout" 0 Int;
Var "SYS_FileSystem" 0 (Struct array of files);
Var "SYS_cwd" 0 (Ptr Char); Var "SYS_root" 0 (Ptr Char)]
(SOMEBody ...))
(F)

Informally if the file system has confidentiality and the current working directory and file system root is confined to the "web space," then when `fetch()` finishes, confidentiality is maintained. Since the function exits rather than returns, our model forces us to specify the postcondition as "impossible" (false).

Proof:

We prove top level correctness with the theorems WF_fnp_cf_cat, WF_fnp_cf_error, and WF_fnp_cf_LOG2, and the axioms SYS_geteuid, SYS_exit, SYS_stat, SYS_S_ISREG.

1. `staterr=stat(name,&(buf));`

   Postcondition:

   ```
   SoFS hasConfidentialityFile SYS_FileSystem /\
   fsconfined SYS_cwd SYS_root /\
   ((staterr = 0) /\ (buf=statSpec name SYS_cwd SYS_root) \/
    (staterr = int_neg 1))
   ```

We prove this using the axiom SYS_stat, the intermediate condition below, the axiom that the dereference of an address-of yields the original, buf in this case, and assignment.

```
SoFS hasConfidentialityFile SYS_FileSystem /\
fsconfined SYS_cwd SYS_root /\
((C_Result = 0) /\ (buf=statSpec name SYS_cwd SYS_root) \/
 (C_Result = int_neg 1))
```

2. `if (staterr != 0) error("Can't stat file");`

   Postcondition:

   ```
   SoFS hasConfidentialityFile SYS_FileSystem /\
   fsconfined SYS_cwd SYS_root /\
   (buf = statSpec name SYS_cwd SYS_root)
   ```

   We begin by breaking the conditional into two cases: the skip or exit case, if the test is false, and the body case. We prove the exit case with the lemma that int_neg 1 is not 0. We prove the body with WF_fnp_cf_error and the axiom that the constant string `"Can't stat file"` is nonconfidential.

3. ```
   if (0 == S_ISREG(buf.st_mode))
                               error("Can't fetch directories");
   ```

   Postcondition:

   ```
   SoFS hasConfidentialityFile SYS_FileSystem /\
   fsconfined SYS_cwd SYS_root /\
   (buf = statSpec name SYS_cwd SYS_root)
   ```

   Although the test is more complicated and takes more work, the proof of this statement is similar. The proof of the test uses the axiom SYS_S_ISREG and the condition is unchanged. Since we don't really model directories, this line of code doesn't contribute anything to the condition.
   ```

4.
```
if (CHECKUSER==1)
    if (buf.st_uid != geteuid())
        error("Not owner of file");
```

Postcondition:

```
SoFS hasConfidentialityFile SYS_FileSystem /\
fsconfined SYS_cwd SYS_root /\
(buf = statSpec name SYS_cwd SYS_root) /\
nonConfFileOwner (inodeNamed name)
```

The `thttpd` server may be installed to check or not check that server process owns the file. So the definition of nonConfFileOwner is if `CHECKUSER` is 1, the file owner must match the process owner, otherwise every file is assumed to have nonconfidential file ownership.

The proof proceeds much the same as before, breaking the conditionals into cases, and uses the axioms SYS_geteuid and the constant string is nonconfidential, the definition of nonConfFileOwner, and the theorem WF_fnp_cf_error.

5.
```
if (0 != (S_IROTH & buf.st_mode)) {
    cat(name); LOG2("cat %s\n",name); exit(1);
}
```

Postcondition:

```
SoFS hasConfidentialityFile SYS_FileSystem
```

Since `cat()` sends the file to the user, we no longer need the qualifiers, so the postcondition is simply that the file system (still) has confidentiality. The proof begins by breaking the conditional into cases. The exit condition is easily proved from the precondition. The body is proved as follows.

(a) `cat(name);`

Postcondition:

```
SoFS hasConfidentialityFile SYS_FileSystem
```

Once again, after `cat()` is done, we don't need the other conditions. The biggest challenge is showing that the preconditions of `cat()` are satisfied. First, we strengthen the precondition to the following.

```
SoFS hasConfidentialityFile SYS_FileSystem /\
fsconfined SYS_cwd SYS_root/\
nonConfFileOwner (inodeNamed name) /\
nonConfFilePerm (inodeNamed name)
```

Then the definition of nonConfFilePerm lets us prove the strengthening is valid, and the theorem WF_fnp_cf_cat finishes this step.

(b) `LOG2("cat %s\n",name);`

The postcondition remains the confidentiality of the file system. We prove this with WF_fnp_cf_LOG2.

(c) `exit(1);`

We can easily prove the overall postcondition since SYS_exit has a post-condition of false.

6. `error("Access Denied");`

Using the theorem WF_fnp_cf_error and the nonconfidentiality of the constant string, we prove the function postcondition.

7.3.7 `main()`

Theorem 7 (`main()` Confidential)

WF_fnp
(SoFS hasConfidentialityFile SYS_FileSystem)
(Func (Var "main" 0 Int)
[Var "argc" 0 Int; Var "argv" 0 (Array (Ptr Char) (CCid "UNK"));
Var "envp" 0 (Array (Ptr Char) (CCid "UNK"))]
[Var "bs1" 0 (Array Char (CCid "BUFSIZE"));
Var "bs2" 0 (Array Char (CCid "BUFSIZE"));
Var "bs3" 0 (Array Char (CCid "BUFSIZE"));
Var "line" 0 (Array Char (CCid "BUFSIZE"));

Var "DOCHROOT" 0 Int; Var "buf" 0 (Struct "stat");
Var "name" 0 (Array Char (CCid "BUFSIZE"));
Var "remotehost" 0 (Array Char (CCid "BUFSIZE"));
Var "remoteuser" 0 (Array Char (CCid "BUFSIZE"));
Var "timestamp" 0 (Array Char (CCint 64));
Var "CHECKUSER" 0 Int; Var "WWWlog" 0 (Ptr Char);
Var "TZ" 0 (Struct "TZstruct"); Var "errno" 0 Int;
Var "SYS_euid" 0 Int; Var "SYS_ruid" 0 Int;
Var "SYS_suid" 0 Int; Var "SYS_stdout" 0 Int;
Var "SYS_FileSystem" 0 (Struct array of files);
Var "SYS_cwd" 0 (Ptr Char); Var "SYS_root" 0 (Ptr Char)]
(SOMEBody ...))
(F)

Assuming argc_Arraysizeargv (Assumption 1) that is, that `argc` is the size of the array `argv`.

Proof:

We prove the top level with the theorems WF_fnp_cf_error and WF_fnp_cf_fetch, and the axioms SYS_chdir, SYS_chroot, SYS_read, SYS_setuid, SYS_sscanf, SYS_strcat, SYS_strcpy, SYS_strlen, SYS_strncpy, and SYS_strncasecmp.

1. `if (0 != chdir(WWWDIR)) error("cannot cd");`

   Postcondition:

   ```
   SoFS hasConfidentialityFile SYS_FileSystem /\
   (SYS_root = ^SYS_root) /\ (SYS_euid = ^SYS_euid) /\
   (SYS_ruid = ^SYS_ruid) /\ (SYS_suid = ^SYS_suid) /\
   (SYS_cwd = WWWDIR)
   ```

   The first clause follows from the precondition. The next four, root, suid, euid, and ruid, are the initial values of globals. The last clause is established by the `chdir()`. We use the following intermediate condition after `chdir()` is called, but before the test is done, and the axiom SYS_chdir.

```
SoFS hasConfidentialityFile SYS_FileSystem /\
(SYS_root = ^SYS_root) /\ (SYS_euid = ^SYS_euid) /\
(SYS_ruid = ^SYS_ruid) /\ (SYS_suid = ^SYS_suid) /\
((C_Result = 0) /\ (SYS_cwd=WWWDIR) \/
 (C_Result = int_neg 1) /\ (SYS_cwd=^SYS_cwd))
```

We prove the exit condition of the conditional with the lemma $0 \neq$ `int_neg` 1. We prove the body with the theorem WF_fnp_cf_error and the axiom that the constant string is nonconfidential.

2.
```
if (DOCHROOT == 1)
        if (chroot(".") != 0)
                error("Cannot change root directory to .");
```

Postcondition:

```
SoFS hasConfidentialityFile SYS_FileSystem /\
(SYS_ruid = ^SYS_ruid) /\ (SYS_suid = ^SYS_suid) /\
(SYS_euid = ^SYS_euid) /\ (SYS_cwd = WWWDIR) /\
((DOCHROOT = 1) ==> (SYS_root = WWWDIR))
```

The first five clauses follow from the precondition. The final clause is established by this statement. The first conditional is easily disposed of. To prove the second conditional, we separate the call to `chroot()` with the following intermediate condition and the axiom SYS_chroot.

```
SoFS hasConfidentialityFile SYS_FileSystem /\
(SYS_cwd = WWWDIR) /\ (SYS_euid = ^SYS_euid) /\
(SYS_ruid = ^SYS_ruid) /\ (SYS_suid = ^SYS_suid) /\
((C_Result = 0) /\ (SYS_root=WWWDIR) \/
 (C_Result = int_neg 1))
```

We satisfy the exit condition `SYS_root=WWWDIR` with the definition of resolving a path, Sect. 6.3.1, page 73. In particular, it says the `"."` is the current working directory, `SYS_cwd`. The exit condition is proved with the lemma `int_neg` $1 \neq 0$. The call to `error()` in the body is proved as usual.

Note: After this point in the proof, we use the more abstract property that the file system is confined. We simplify the condition to the following by strengthening the precondition. This is justified by the definition of `fsconfined`.

```
SoFS hasConfidentialityFile SYS_FileSystem /\
fsconfined SYS_cwd SYS_root
```

3. `if (0 != setuid(WWWUID)) error("setUID failed");`

   Postcondition:

   ```
   SoFS hasConfidentialityFile SYS_FileSystem /\
   fsconfined SYS_cwd SYS_root /\ (SYS_euid = WWWUID)
   ```

   The first two clauses follow from the precondition. The third clause follows from the success of `setuid()`. The intermediate condition after the call is the following. We prove it with SYS_setuid and the lemma `int_neg` $1 \neq 0$. We prove the body as usual.

   ```
   SoFS hasConfidentialityFile SYS_FileSystem /\
   fsconfined SYS_cwd SYS_root /\
   ((0 = C_Result) ==> (SYS_euid = WWWUID))
   ```

The next nine code statements deal with getting the remote user's name, the file name, etc. Other than potential memory overwrites, these have little to do with confidentiality, so the condition is unchanged. We group them and include a few notes about applicable theorems and axioms.

4. ```
   if (argc>1) strncpy(remotehost,argv[1],MAXSIZE);
   else        strcpy(remotehost,"nowhere");
   if (argc>2) strncpy(remoteuser,argv[2],MAXSIZE);
   else        strcpy(remoteuser,"nobody");
   ```

   Postcondition:

   ```
   SoFS hasConfidentialityFile SYS_FileSystem /\
   fsconfined SYS_cwd SYS_root /\ (SYS_euid = WWWUID)
   ```
   ```

We divide the conditionals into two parts and prove them with SYS_strncpy, SYS_strcpy, and the precondition.

5.
```
remotehost[MAXSIZE]='\0';
remoteuser[MAXSIZE]='\0';
read(0,line,MAXSIZE);
line[MAXSIZE]='\0';
sscanf(line, "%s %s %s", bs1, name, bs2);
```

   The same postcondition is used throughout:

```
SoFS hasConfidentialityFile SYS_FileSystem /\
fsconfined SYS_cwd SYS_root /\ (SYS_euid = WWWUID)
```

   We separate each statement with the sequence rule, and prove them with the preconditions and the assignment rule, SYS_read, or SYS_sscanf.

6.
```
if ((name[0] != '\0') && (name[strlen(name)-1] == '\r'))
                                      name[strlen(name)-1]='\0';
if ((name[0]=='/') && ((name[1]=='\0') || (name[1]==' ')))
                                      strcpy(name,WWWDefaultFile);
```

   Same postcondition for both:

```
SoFS hasConfidentialityFile SYS_FileSystem /\
fsconfined SYS_cwd SYS_root /\ (SYS_euid = WWWUID)
```

   We use the postcondition as the intermediate condition, too, and prove these conditionals from the precondition and SYS_strlen, SYS_strcpy, and assignment.

7.
```
if (DOCHROOT!=1) {
     strcpy(bs3,WWWDIR);strcat(bs3,name);strcpy(name,bs3);
}
```

   Postcondition:

```
SoFS hasConfidentialityFile SYS_FileSystem /\
fsconfined SYS_cwd SYS_root /\ (SYS_euid = WWWUID)
```

This statement prepends the WWW directory to the file name requested if we did not change the root. We prove the postcondition from the precondition, SYS_strcpy, and SYS_strcat.

8. `if (strncasecmp(bs1,"get",5) == 0) fetch();`

   Postcondition:

   ```
   SoFS hasConfidentialityFile SYS_FileSystem /\
   fsconfined SYS_cwd SYS_root /\ (SYS_euid = WWWUID)
   ```

   The postcondition is unchanged and is used for the intermediate condition after `strncasecmp()`. If the user requests a "get," this calls `fetch()` to check that the file is public and return a copy to the user. The precondition is (finally) necessary here for `fetch()` to maintain confidentiality. We prove the postcondition with SYS_strncasecmp and WF_fnp_cf_fetch.

9. `error("Unknown request");`

   The postcondition is the function postcondition. We prove it using WF_fnp_cf_error and the confidentiality of a constant string.

By examination we are satisfied that confidentiality was maintained through the end of the program.

7.3.8 `thttpd` has Confidentiality

Theorem 8 (`thttpd` Confidential)
hasConfidentiality
(Simple
(Call (Var "main" 0 Int)
(PL (Lval (Vref (Var "argc" 0 Int)))
(PL (Lval (Vref (Var "argv" 0 (Array (Ptr Char) (CCid "UNK")))))
(PL (Lval (Vref (Var "envp" 0 (Array (Ptr Char) (CCid "UNK")))))
PLnull)))))
SYS_FileSystem

Assuming argc_Arraysizeargv (Assumption 1) that is, that `argc` is the size of the array `argv`. Informally, a call to `main()` will not violate the confidentiality of the file system.

Proof:

This follows from the definition of `hasConfidentiality` (Sect. 6.1.2, page 41) and Theorem 7, `main()` Confidential.

7.4 Proving Information Integrity

7.4.1 `logfile()`

We mention a lemma which is a variant of $\tilde{}a \wedge (a \vee b) \Rightarrow b$. It was hard to prove in context, so we proved it separately.

Lemma 9 *preFSS inode (getFile fileSys inode)*

Assuming ~*(inode = inodeOf (deref ˆF))* and *!inode. (inode = inodeOf (deref ˆF))* ∨ *preFSS inode (getFile fileSys inode)*

Theorem 10 (WF_fnp_ii_logfile)

WF_fnp
(SoFS preFSS SYS_FileSystem)
(Func (Var "logfile" 0 Void)
  [Var "F" 0 (Ptr (Struct "FILE"))]
  [Var "errno" 0 Int;
    Var "remotehost" 0 (Array Char (CCid "BUFSIZE"));
    Var "remoteuser" 0 (Array Char (CCid "BUFSIZE"));
    Var "timestamp" 0 (Array Char (CCint 64));
    Var "TZ" 0 (Struct "TZstruct");
    Var "SYS_FileSystem" 0 (Struct array of files)]
  (SOMEBody . . .))
(let finode = inodeOf (deref ˆF) in
    (!inode. (inode = finode) ==>
      preFSS inode (getFile SYS_FileSystem inode)))

Informally this says, if the file system is in some state (`preFSS`), then when `logfile()` finishes, everything will be the same except perhaps for the file pointer passed in (`F`).

Proof:

We prove the top level with the axioms SYS_fprintf, SYS_localtime, SYS_strftime, SYS_time.

1. `t=time(NULL)`

   Postcondition:

   ```
   (F = ^F) /\ (SoFS preFSS SYS_FileSystem) /\
   (?some_time.t = some_time)
   ```

   The first two clauses follow from the precondition. The last follows from SYS_time with the following intermediate condition and assignment.

   ```
   (F = ^F) /\ (SoFS preFSS SYS_FileSystem) /\
   (?some_time.C_Result = some_time)
   ```

2. `strftime(timestamp, 20, "%Y/%m/%d %T", localtime(&t));`

   Postcondition:

   ```
   (F = ^F) /\ (SoFS preFSS SYS_FileSystem)
   ```

   First, we separate the effect of the call to `localtime()` with the following intermediate condition.

   ```
   (F = ^F) /\ (SoFS preFSS SYS_FileSystem) /\
   (?tsptr.C_Result = tsptr)
   ```

   The first two clauses follow from the precondition. The last follows from SYS_localtime. The postcondition follows from the intermediate and SYS_strftime.

3. `fprintf(F,"%s %s %s ",remotehost,remoteuser,timestamp);`

   The postcondition is the function postcondition. We prove this with Lemma 9 and SYS_fprintf.

7.4.2 LOG2

As in `logfile()`, we found a condition hard to prove in the middle, so we proved the following lemma.

Lemma 11 *!a b.a* $\vee$ *b* = ~*a* $\Rightarrow$ *b*

Theorem 12 (WF_fnp_ii_LOG2)
WF_fnp
(SoFS preFSS SYS_FileSystem)
(Func (Var "LOG2" 0 Void)
  [Var "x" 0 (Ptr Char); Var "y" 0 (Ptr Char)]
  [Var "SYS_FileSystem" 0 (Struct array of files);
    Var "remotehost" 0 (Array Char (CCid "BUFSIZE"));
    Var "remoteuser" 0 (Array Char (CCid "BUFSIZE"));
    Var "timestamp" 0 (Array Char (CCint 64));
    Var "TZ" 0 (Struct "TZstruct"); Var "errno" 0 Int;
    Var "SYS_cwd" 0 (Ptr Char); Var "SYS_root" 0 (Ptr Char);
    Var "WWWlog" 0 (Ptr Char)]
  (SOMEBody . . .))
(let finode = inodeNamed WWWlog in
    (!inode. (inode = finode) $\Rightarrow$
        preFSS inode (getFile SYS_FileSystem inode)))

Informally, when `LOG2` finishes, nothing is changed, except possibly the WWW log file.

Proof:

We use the theorem WF_fnp_ii_logfile and the axioms SYS_fopen, SYS_fprintf, and SYS_fclose.

1. `F = fopen(WWWlog,"a+");`

   Postcondition:

```
SoFS preFSS SYS_FileSystem /\
((F = NULL) \/
 (?FOPEN_handlefn.F = FOPEN_handlefn (inodeNamed WWWlog) "a+"))
```

The intermediate condition is the postcondition, except having `C_Result` instead of `F`. We prove it with SYS_fopen.

2. `if (F != NULL) {logfile(F);fprintf(F,x,y);}`

   Postcondition:

   ```
   let finode = inodeNamed WWWlog in
       (!inode.~(inode = finode) ==>
           preFSS inode (getFile SYS_FileSystem inode))
   ```

   We break the conditional into two parts, and prove the exit condition with the definition of SoFS. Next we begin to prove the body by strengthening the precondition to state that the file system is unchanged and F is a handle of the WWW log file.

   ```
   (SoFS preFSS SYS_FileSystem) /\
   (?FOPEN_handlefn.F =
       FOPEN_handlefn(inodeNamed WWWlog) "a+")
   ```

   We prove the two function calls in the body like this.

   (a) `logfile(F);`

   Postcondition:

   ```
   (?FOPEN_handlefn.F =
       FOPEN_handlefn(inodeNamed WWWlog) "a+") /\
   (let finode = inodeNamed WWWlog in
       (!inode.~(inode = finode) ==>
           preFSS inode(getFile SYS_FileSystem inode)))
   ```

   After `logfile()` the log file may have changed, but everything else is unchanged. We prove this with WF_fnp_ii_logfile, the definition of SoFS, and the axiom that the handle returned by `fopen()` refers to that file (Sect. 6.3.1, page 72).

(b) `fprintf(F,x,y);`

The postcondition is the statement postcondition. We prove this with the definition of SoFS, SYS_fprintf, Lemma 11, and the `fopen()` handle axiom again.

3. `fclose(F);`

The postcondition is the function postcondition. We prove it with SYS_fclose.

7.4.3 `LOG4`

Theorem 13 (WF_fnp_ii_LOG4)

WF_fnp
(SoFS preFSS SYS_FileSystem)
(Func (Var "LOG4" 0 Void)
[Var "x" 0 (Ptr Char); Var "y" 0 (Ptr Char);
Var "z" 0 (Ptr Char); Var "w" 0 (Ptr Char)]
[Var "SYS_FileSystem" 0 (Struct array of files);
Var "remotehost" 0 (Array Char (CCid "BUFSIZE"));
Var "remoteuser" 0 (Array Char (CCid "BUFSIZE"));
Var "timestamp" 0 (Array Char (CCint 64));
Var "TZ" 0 (Struct "TZstruct"); Var "errno" 0 Int;
Var "SYS_cwd" 0 (Ptr Char); Var "SYS_root" 0 (Ptr Char);
Var "WWWlog" 0 (Ptr Char)]
(SOMEBody ...))
(let finode = inodeNamed WWWlog in
(!inode. (inode = finode) ==>
preFSS inode (getFile SYS_FileSystem inode)))

Informally, when `LOG4` finishes, nothing is changed, except possibly the WWW log file.

Proof:

The proof of `LOG4` is essentially identical to that of `LOG2`.

7.4.4 `error()`

Once again we prove a simple lemma because it was difficult to do within the main proof.

Lemma 14 $!a\,b\,c.\verb|~|a \Rightarrow \verb|~|b \Rightarrow c = \verb|~|(b \vee a) \Rightarrow c$

Theorem 15 (WF_fnp_ii_error)
WF_fnp
(SoFS preFSS SYS_FileSystem)
(Func (Var "error" 0 Int)
[Var "s" 0 (Ptr Char)]
[Var "SYS_FileSystem" 0 (Struct array of files);
Var "remotehost" 0 (Array Char (CCid "BUFSIZE"));
Var "remoteuser" 0 (Array Char (CCid "BUFSIZE"));
Var "timestamp" 0 (Array Char (CCint 64));
Var "TZ" 0 (Struct "TZstruct");
Var "errno" 0 Int; Var "WWWlog" 0 (Ptr Char);
Var "SYS_root" 0 (Ptr Char); Var "SYS_cwd" 0 (Ptr Char);
Var "bs1" 0 (Array Char (CCid "BUFSIZE"));
Var "name" 0 (Array Char(CCid "BUFSIZE"))]
(SOMEBody ...))
(F)

Strictly speaking we prove that nothing is executed after `error()` finishes. However we inspect the proof to see that information integrity is preserved in the penultimate condition.

Proof:

We prove top level correctness with WF_fnp_ii_LOG4, SYS_exit, and SYS_printf.

1. `printf("HTTP/1.0 302 Found\n");`
   `printf("Server: ManAl/0.1\n");`
   `printf("MIME-version: 1.0\n");`
   `printf(REDIRECT);`
   `printf("Content-type: text/html\n");`
   `printf("<HEAD><TITLE>Document moved</TITLE></HEAD>\n");`
   `printf("<BODY><H1>Document moved</H1>\n");`
   `printf(ERRORLINE);`

   We use the same postcondition for every statement which is that stdout may have changed since `printf()` changes stdout, and it is modeled as a part of the file system.

   ```
   SoFS (\i u.~(i=SYS_stdout)==>preFSS i u) SYS_FileSystem
   ```

   For regularity, we first strengthen the precondition, then do the same proof for each call. We prove each call with SYS_printf and somewhat involved rewrites.

2. `LOG4("Error:%s - %s %s\n",s,bs1,name);`

   The postcondition is that stdout or the log file may have changed.

   ```
   SoFS (\i u. ~((i=SYS_stdout) \/ (i=inodeNamed WWWlog))==>
                                               preFSS i u) SYS_FileSystem
   ```

   We prove this with WF_fnp_ii_LOG4, SoFS, and Lemma 14.

3. `exit(1);`

   By examination, information integrity has been maintained up to this call. We prove this with SYS_exit.

7.4.5 `cat()`

Here again we need a simple lemma within the proof.

Lemma 16 $a \vee ({\sim}a \Rightarrow b) = {\sim}a \Rightarrow b$

Theorem 17 (WF_fnp_ii_cat)

WF_fnp
(SoFS preFSS SYS_FileSystem)
(Func (Var "cat" 0 Void)
[Var "s" 0 (Array Char (CCid "UNK"))]
[Var "SYS_FileSystem" 0 (Struct array of files);
Var "SYS_root" 0 (Ptr Char); Var "SYS_cwd" 0 (Ptr Char);
Var "SYS_stdout" 0 Int; Var "errno" 0 Int;
Var "bs2" 0 (Array Char (CCid "BUFSIZE"))]
(SOMEBody ...))
(!inode. (inode = SYS_stdout) ⇒
preFSS inode (getFile SYS_FileSystem inode))

Informally, `cat()` does not change any file, except perhaps stdout.

Proof:

The top level goal is reduced using SYS_open, SYS_read, SYS_write, and SYS_close.

1. `i=open(s,0);`

   Postcondition;

   ```
   SoFS preFSS SYS_FileSystem
   ```

   This follows from the precondition and SYS_open.

2. `while ((n=read(i,bs2,MAXSIZE)) > 0) write(1,bs2,n);`

   The postcondition is that all files are the same, except possibly stdout.

   ```
   !inode.~(inode = SYS_stdout) ==>
               preFSS inode (getFile SYS_FileSystem inode)
   ```

   We prove this in three main steps: we set up the loop invariant, prove the `read()` establishes the test intermediate condition, and prove that the body (the `write()`), reestablishes the invariant.

(a) We set up the following weaker loop invariant.

```
!inode.~(inode = SYS_stdout) ==>
             preFSS inode (getFile SYS_FileSystem inode)
```

This is valid by precondition strengthening and the definition of SoFS.

(b) We prove that the loop postcondition follows by reducing the loop with the `while` inference rule. We use the invariant as the condition after the test. The termination condition follows directly from the invariant. We prove the test, which calls `read()`, produces the condition with SYS_read.

(c) `write(1,bs2,n);`

We prove that the `write()` in the body reestablishes the invariant using the definition of SoFS, SYS_write, the axiom that file descriptor 1 is stdout (Sect. 6.3.1, page 73), and Lemma 16.

3. `close(i);`

We prove that the postcondition, which is the function postcondition, follows with SYS_close.

7.4.6 `fetch()`

Theorem 18 (WF_fnp_ii_fetch)

WF_fnp
(SoFS preFSS SYS_FileSystem)
(Func (Var "fetch" 0 Void)
  [ ]
  [Var "bs1" 0 (Array Char (CCid "BUFSIZE"));
    Var "bs2" 0 (Array Char (CCid "BUFSIZE"));
    Var "name" 0 (Array Char (CCid "BUFSIZE"));
    Var "remotehost" 0 (Array Char (CCid "BUFSIZE"));
    Var "remoteuser" 0 (Array Char (CCid "BUFSIZE"));
    Var "timestamp" 0 (Array Char (CCint 64));
    Var "CHECKUSER" 0 Int; Var "WWWlog" 0 (Ptr Char);

Var "TZ" 0 (Struct "TZstruct"); Var "errno" 0 Int;
Var "buf" 0 (Struct "stat"); Var "SYS_stdout" 0 Int;
Var "SYS_FileSystem" 0 (Struct array of files);
Var "SYS_cwd" 0 (Ptr Char); Var "SYS_root" 0 (Ptr Char)]
(SOMEBody ...))
(F)

Again strict conformance to our model only allows us to prove a postcondition of false, since `fetch()` exits.

Proof:

We prove the top level call is well-formed with WF_fnp_ii_cat, WF_fnp_ii_error, WF_fnp_ii_LOG2, SYS_geteuid, SYS_exit, SYS_stat, and SYS_S_ISREG. We simplify the function body precondition to the following.

```
SoFS preFSS SYS_FileSystem
```

1. `staterr=stat(name,&(buf));`

   The postcondition states that the file system is unchanged and either there was an error (`staterr` = -1) or `buf` has the status of file "name" given the current working directory and root.

   ```
   SoFS preFSS SYS_FileSystem /\
   ((staterr = 0) /\ (buf = statSpec name SYS_cwd SYS_root) \/
    (staterr = int_neg 1))
   ```

   The first clause follows from the precondition. The next clause follows from SYS_stat and the axiom that the dereference of an address-of yields the original, `buf` in this case.

2. `if (staterr != 0) error("Can't stat file");`

   The postcondition is that the file system is unchanged and `buf` has the status of file "name."

```
SoFS preFSS SYS_FileSystem /\
(buf = statSpec name SYS_cwd SYS_root)
```

We prove the false condition with the lemma that int_neg 1 is not 0. We prove the body with WF_fnp_ii_error.

3. `if (0 == S_ISREG(buf.st_mode))`

   ```
                            error("Can't fetch directories");
   ```

   The postcondition remains the same since fetching directories has no effect on information integrity. We prove the statement with the same condition for the intermediate condition after the test, SYS_S_ISREG for the test, and WF_fnp_ii_error.

4. ```
   if (CHECKUSER==1)
       if (buf.st_uid != geteuid())
           error("Not owner of file");
   ```

   Since we are concerned with information integrity, not confidentiality, the postcondition is still the same.

   ```
   SoFS preFSS SYS_FileSystem /\
   (buf = statSpec name SYS_cwd SYS_root)
   ```

   This follows from the precondition, SYS_geteuid, and WF_fnp_ii_error.

5. ```
   if (0 != (S_IROTH & buf.st_mode)) {
       cat(name); LOG2("cat %s\n",name); exit(1);
   }
   ```

   Since the state of `buf` was only needed until this statement, the postcondition is the simpler "file system is unchanged." Although `cat()` may change stdout, and `LOG2` may change the log file, execution is stopped by `exit()`, so the postcondition is not unreasonable.

   ```
   SoFS preFSS SYS_FileSystem
   ```

   We begin by breaking up the conditional, and proving the body.

(a) `cat(name);`

The postcondition is that nothing in the file system changes, except possibly stdout.

```
SoFS (\i u.~(i=SYS_stdout)==>preFSS i u) SYS_FileSystem
```

This follows from WF_fnp_ii_cat and the definition of SoFS.

(b) `LOG2("cat %s\n",name);`

The postcondition is that everything but possibly stdout and WWWlog is unchanged.

```
SoFS (\i u.~((i=SYS_stdout) \/ (i=inodeNamed WWWlog))
                                     ==>preFSS i u) SYS_FileSystem
```

We prove this from the precondition, WF_fnp_ii_LOG2, the definition of SoFS, and Lemma 14.

(c) `exit(1);`

We prove that the postcondition follows from SYS_exit.

6. `error("Access Denied");`

   The function postcondition follows from WF_fnp_ii_error.

   We see by inspection that information integrity was maintained through the end of the function.

7.4.7 `main()`

Theorem 19 (`main()` Integrity)

WF_fnp
(SoFS preFSS SYS_FileSystem) (Func (Var "main" 0 Int)
[Var "argc" 0 Int; Var "argv" 0 (Array (Ptr Char) (CCid "UNK"));
Var "envp" 0 (Array (Ptr Char) (CCid "UNK"))]
[Var "bs1" 0 (Array Char (CCid "BUFSIZE"));
Var "bs2" 0 (Array Char (CCid "BUFSIZE"));
Var "bs3" 0 (Array Char (CCid "BUFSIZE"));
Var "line" 0 (Array Char (CCid "BUFSIZE"));

Var "DOCHROOT" 0 Int; Var "buf" 0 (Struct "stat");
Var "name" 0 (Array Char (CCid "BUFSIZE"));
Var "remotehost" 0 (Array Char (CCid "BUFSIZE"));
Var "remoteuser" 0 (Array Char (CCid "BUFSIZE"));
Var "timestamp" 0 (Array Char (CCint 64));
Var "CHECKUSER" 0 Int; Var "WWWlog" 0 (Ptr Char);
Var "TZ" 0 (Struct "TZstruct"); Var "errno" 0 Int;
Var "SYS_euid" 0 Int; Var "SYS_ruid" 0 Int;
Var "SYS_suid" 0 Int; Var "SYS_stdout" 0 Int;
Var "SYS_FileSystem" 0 (Struct array of files);
Var "SYS_cwd" 0 (Ptr Char); Var "SYS_root" 0 (Ptr Char)]
(SOMEBody ...))
(F)

Assuming argc_Arraysizeargv (Assumption 1), that is, that `argc` is the size of the array `argv`.

Proof:

We prove the well-formedness of the top level goal using WF_fnp_ii_error, WF_fnp_ii_fetch, SYS_chdir, SYS_chroot, SYS_read, SYS_setuid, SYS_sscanf, SYS_strcat, SYS_strcpy, SYS_strlen, SYS_strncpy, and SYS_strncasecmp. We begin the proof of the body with the file system in some state and initial values for several important global variables.

```
SoFS preFSS SYS_FileSystem /\
(SYS_root = ^SYS_root) /\ (SYS_euid = ^SYS_euid) /\
(SYS_ruid = ^SYS_ruid) /\ (SYS_suid = ^SYS_suid)
```

1. `if (0 != chdir(WWWDIR)) error("cannot cd");`

   The postcondition adds that the current working directory is the WWW directory.

   ```
   SoFS preFSS SYS_FileSystem /\ (SYS_cwd = WWWDIR) /\
   (SYS_root = ^SYS_root) /\ (SYS_euid = ^SYS_euid) /\
   (SYS_ruid = ^SYS_ruid) /\ (SYS_suid = ^SYS_suid)
   ```

The intermediate condition after calling chdir() is the precondition with either the current working directory successfully changed to the WWW directory or the current working directory unchanged.

```
SoFS preFSS SYS_FileSystem /\
(SYS_root = ^SYS_root) /\ (SYS_euid = ^SYS_euid) /\
(SYS_ruid = ^SYS_ruid) /\ (SYS_suid = ^SYS_suid) /\
((C_Result = 0) /\ (SYS_cwd=WWWDIR) \/
 (C_Result = int_neg 1) /\ (SYS_cwd = ^SYS_cwd))
```

We prove these with SYS_chdir, the lemma that int_neg 1 is not 0, and WF_fnp_ii_error.

2.
```
if (DOCHROOT == 1)
        if (chroot(".") != 0)
                error("Cannot change root directory to .");
```

The postcondition adds that the root directory is the WWW directory (if that option is configured).

```
SoFS preFSS SYS_FileSystem /\ (SYS_cwd = WWWDIR) /\
(SYS_euid = ^SYS_euid) /\ (SYS_ruid = ^SYS_ruid) /\
(SYS_suid = ^SYS_suid) /\
((DOCHROOT = 1) ==> (SYS_root = WWWDIR))
```

The first five clauses follow from the precondition. If DOCHROOT is not 1, the last condition follows. Otherwise it follows from SYS_chroot, the definition that "." means the current working directory (which is WWWDIR from the second clause) in path resolution, the lemma that 0 is not int_neg 1, and WF_fnp_ii_error.

3.
```
if (0 != setuid(WWWUID)) error("setUID failed");
```

The postcondition here, and in fact, through the end of the program is that the file system is unchanged and the current working directory and the root directory are WWWDIR.

```
SoFS preFSS SYS_FileSystem /\ SYS_cwd = WWWDIR) /\
((DOCHROOT = 1) ==> (SYS_root = WWWDIR))
```

We use SYS_setuid and WF_fnp_ii_error to prove that the postcondition follows.

The next nine code statements deal with getting the remote user's name, the file name, etc. Other than potential memory overwrites, these have little to do with information integrity, so the condition is unchanged. We group them and include a few notes and theorems and axioms we used.

4. ```
   if (argc>1) strncpy(remotehost,argv[1],MAXSIZE);
   else        strcpy(remotehost,"nowhere");
   if (argc>2) strncpy(remoteuser,argv[2],MAXSIZE);
   else        strcpy(remoteuser,"nobody");
   ```

   The postcondition is unchanged. For both statements we divide the conditionals into two parts and prove them with SYS_strncpy, SYS_strcpy, and the precondition.

5. ```
   remotehost[MAXSIZE]='\0';
   remoteuser[MAXSIZE]='\0';
   read(0,line,MAXSIZE);
   line[MAXSIZE]='\0';
   sscanf(line, "%s %s %s", bs1, name, bs2);
   ```

   The same postcondition is used throughout.

   ```
   SoFS preFSS SYS_FileSystem /\ SYS_cwd = WWWDIR) /\
   ((DOCHROOT = 1) ==> (SYS_root = WWWDIR))
   ```

   We separate each statement with the sequence rule, and prove them with the preconditions and the assignment rule, SYS_read, or SYS_sscanf.

6. ```
   if ((name[0] != '\0') && (name[strlen(name)-1] == '\r'))
                                      name[strlen(name)-1]='\0';
   if ((name[0]=='/') && ((name[1]=='\0') || (name[1]==' ')))
   ```
   ```

```
strcpy(name,WWWDefaultFile);
```

We maintain the same postcondition throughout.

```
SoFS preFSS SYS_FileSystem /\ SYS_cwd = WWWDIR) /\
((DOCHROOT = 1) ==> (SYS_root = WWWDIR))
```

We use the postcondition as the intermediate condition, too, and prove these conditionals from the precondition and SYS_strlen, SYS_strcpy, and assignment.

7. ```
   if (DOCHROOT!=1) {
       strcpy(bs3,WWWDIR);strcat(bs3,name);strcpy(name,bs3);
   }
   ```

   Postcondition:

   ```
   SoFS preFSS SYS_FileSystem /\ SYS_cwd = WWWDIR) /\
   ((DOCHROOT = 1) ==> (SYS_root = WWWDIR))
   ```

   This statement prepends the WWW directory to the file name requested if we did not change the root. We prove the postcondition from the precondition, SYS_strcpy, and SYS_strcat.

8. `if (strncasecmp(bs1,"get",5) == 0) fetch();`

   Postcondition:

   ```
   SoFS preFSS SYS_FileSystem /\ SYS_cwd = WWWDIR) /\
   ((DOCHROOT = 1) ==> (SYS_root = WWWDIR))
   ```

   The postcondition is unchanged and is used for the intermediate condition after `strncasecmp()`. If the user requests a "get," this calls `fetch()` to check that the file is public and return a copy to the user. Since `fetch()` exits, we can prove the postcondition with SYS_strncasecmp and WF_fnp_ii_fetch.

9. `error("Unknown request");`

   We use the program postcondition. We prove it with WF_fnp_ii_error.

By examination we are satisfied that `thttpd` maintains information integrity throughout the end of the program.
   ```

7.4.8 `thttpd` has Information Integrity

Theorem 20 (`thttpd` Integrity)

hasInfoIntegrity
  (Simple
    (Call (Var "main" 0 Int)
      (PL (Lval (Vref (Var "argc" 0 Int)))
        (PL (Lval (Vref (Var "argv" 0 (Array (Ptr Char) (CCid "UNK")))))
          (PL (Lval (Vref (Var "envp" 0 (Array (Ptr Char) (CCid "UNK")))))
            PLnull)))))
preFSS SYS_FileSystem log

Assuming argc_Arraysizeargv (Assumption 1), that is, that `argc` is the size of the array `argv`. Informally, this means nothing in the file system is changed except perhaps the log file and stdout.

Proof:

This follows from the definition of `hasInfoIntegrity` (Sect. 6.1.1, page 41) and Theorem 19, `main()` Integrity.

7.5 Proving `thttpd` Secure

We come to the main thrust of the dissertation: proof that `thttpd` is secure.

Theorem 21 (`thttpd` Secure)

isSecure
  (Simple
    (Call (Var "main" 0 Int)
      (PL (Lval (Vref (Var "argc" 0 Int)))
        (PL (Lval (Vref (Var "argv" 0 (Array (Ptr Char) (CCid "UNK")))))
          (PL (Lval (Vref (Var "envp" 0 (Array (Ptr Char) (CCid "UNK")))))
            PLnull)))))
preFSS SYS_FileSystem log

Assuming argc_Arraysizeargv (Assumption 1), that is, that `argc` is the size of the array `argv`. Informally, this means that executing `thttpd` will not violate security with respect to some initial condition (`preFSS`) of the file system, including some log file.

Proof:

This follows from the definition of `isSecure` (Sect. 6.1, page 40), Theorem 8, `thttpd` Confidential, and Theorem 20, `thttpd` Integrity.

QED.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

Section 8.1 gives the results from our verification, that is, what we uncovered, while Sect. 8.2 notes limitations, primarily in the formalizations. Section 8.3 lists some areas of possible future work, including our proposal for a practical software verification system, and we finish with our conclusions in Sect. 8.4.

8.1 Verification Results

As expected, we found no serious bugs in `thttpd`. The formal verification did yield some interesting results. In this section we report the areas of concern and give our analyses of each one. None of these were noted in the detailed code walkthrough [11].

8.1.1 Assumption

The only explicit assumption we needed to make is Assumption 1, page 79, that `argc` passed to `main()` is, indeed, the number of arguments or the size of `argv`.

Analysis

We could have included this as part of the formal description, but instead wanted to expose it as an assumption. Since it is guaranteed as part of Unix program invocation, it should not invalidate any conclusions.

8.1.2 Formal Specifications

The original report on `thttpd` [11] gave the properties of interest as information integrity, "that the information residing in the server is not corrupted," availability of service, and confidentiality, "that the service only provides information ...that is explicitly authorized for outside access." We formalized these definitions.

Making formal specifications brought out some interesting details explicitly.

1. Causing information to be written to the log file or `stdout` is allowed, otherwise information on the server is corrupted.

2. Error messages and other fixed strings in the program are implicitly authorized for outside access, otherwise confidentiality is violated.

We also noticed while doing the proof that the log file must be in the server "space" (so it can be opened after the `chroot()`), but must not have permissions for users to request it. Otherwise, remote users could find out what other users are requesting, and information could be passed from one user to another. Passing could be done as follows: the first user requests files where the file names are the information to be passed. The information given as file names is logged. The other user gets the log file and reads the requests of the first user. This important configuration detail is included in the installation instructions, but not in the original report.

Analysis

None of these compromise security. Rather the formalizations help us understand more exactly what we mean by "security."

8.1.3 Ill-formed Code Constructs

The following shouldn't cause problems in practise, but are formally ill-formed. Every installation should check that these won't cause problems.

Calling `fclose()` with `NULL`

The macros `LOG2`, `LOG3`, and `LOG4` call `fclose` whether or not the file open succeeds. If the open fails, `fclose(NULL)` is executed. The manual page doesn't clearly say what `fclose()` does in that situation, but suggests that it checks for a valid stream. Interestingly, in a test of calling `fclose(NULL)` on HP-UX Release 9.0: August 1992, `fclose()` returned -1 (or, `EOF`) indicating an error, but did not set the global `errno`.

`time()` may fail in `logfile()`

The manual page says on failure `time()` returns `(time_t) -1`, but doesn't detail when failures may occur. The only error listed is if the parameter passed "points to an illegal address." Since `NULL` is hardcoded, this shouldn't cause a problem. However the call `localtime(& (time_t) -1)` could conceivably cause a core dump.

Calling `exit` with no parameter

The functions `error()` and `fetch()` call `exit()` with no parameter. However `exit()` is defined as taking one parameter: a status. In the best case, some compiler-dependent status is returned when the program exits. In the worst case, which is highly unlikely given the typical way of implementing function calls in C, the incorrect invocation could cause arbitrary functionality.

Unused Declarations

Lastly we found unused bits of code. Although it should not cause a problem, the macro `LOG3()` is defined, but never used. Also the function `logfile()` declares, but does not use, the variable `tloc`.

Analysis

It is remotely possible that these could cause core dumps. Flanagan points out [19] that a core dump may lead to a breach of confidentiality if someone requests the file `/core` and a previous core file has the following properties.

- The core file is in the web area. This is likely since core dumps are usually in the current directory, and `thttpd` changes to the root of the web area.
- The core file is owned by "web." This is likely since core file are usually owned by the executing process, and `thttpd` sets the UID to "web."
- The core file is "other" readable. This is unlikely since core file are usually readable only be the owner.

Thus it does not seem likely that these problems could compromise security. However, for greater assurance, the following steps could be taken.

1. `thttpd` prevents core files by setting the maximum core file to 0, say, with `setrlimit()`.

2. Every installation checks that core files are readable only by the owner.

8.1.4 No Check that `open()` Succeeds

The function which actually returns the contents of the requested file is `cat()`. It is only called from one other routine, `fetch()`, which checks conditions before calling `cat()`, which opens the file, but doesn't check for success. However `fetch()` misses an unusual condition which may cause the `open()` to fail. The implementation of the functions is given here for reference.

```
void    cat(char s[])
{int    i,n;FILE *F;
i=open(s,0);
while ((n=read(i,bs2,MAXSIZE)) > 0) write(1,bs2,n);
close(i);}

/* if www owns it, it can be put - else, forget it */
void fetch()
{int staterr;
staterr=stat(name,&(buf));
/* can't stat the file - die */
if (staterr != 0) error("Can't stat file");
if (0 == S_ISREG(buf.st_mode)) error("Can't fetch directories");
if (CHECKUSER==1) if (buf.st_uid != geteuid())
        error("Not owner of file"); /* don't own it - die */
if (0 != (S_IROTH & buf.st_mode)) {
        cat(name); LOG2("cat %s\n",name);exit(1);
} /* Send it*/
error("Access Denied");}
```

The calling routine, `fetch()`, checks that the file

1. exists (is accessible),
2. is a regular file (not a directory, pipe, etc.),
3. is owned by the special user "www," and
4. is readable by "other" users (the least privileged).

In the Unix permission system, processes must have permissions for the class to which they belong. If `thttpd` runs as user "www" and the file is owned by "www," the file must be readable by "owner." In this case, permissions for "other" users have no effect.

Therefore if a file is set up according to directions except that it is not "owner" readable, `cat()` fails to deliver the file (`read()` returns -1 and the `while` loop exits), but the log file indicates a successful delivery.

We found the problem while verifying confidentiality. We had to show that if `open()` failed (and returns -1 as the file descriptor), the contents of `bs2` were nonconfidential. However the axiom for `read()` originally stated that the buffer might be modified regardless of an error file descriptor. We strengthened the postcondition to say that if something was read, the file descriptor could not have been -1.

Analysis

This error has no effect on security, but the log file may be incorrect.

8.1.5 No Repeated Call to `write()`

The system documentation [29] seems to say that the call to `write()` within `cat()` may be interrupted.

> If *write()* is interrupted by a signal after it successfully writes some data, it will return the number of bytes written.

If this occurs, all the bytes read may not be written, but the log file indicates a successful delivery. This could be avoided by looping to repeatedly call `write()`

until all bytes are written or using a higher level interface, such as `putchar()` or `printf()`.

We found the problem while verifying confidentiality. We had to show that if `write()` failed, nothing confidential was written. However the axiom for `write()` originally guaranteed nothing about the file system in the case of a failure. While rereading the manual page describing `write()`, we noticed the above problem. We strengthened the postcondition to say that if -1 were returned, the file system is unchanged.

Analysis

This error has no effect on security, but the file delivered to the user may be missing some information with no indication of a problem in the log file.

8.2 Limitations and Restrictions

This proof is a simultaneous investigation of formal verification of software using axiomatic semantics, verification and formalization of the C language and operating system calls, and verification and formalization of a secure web server. As such in some cases we demonstrate the feasibility of approaches without following them fully. This section lists the limitations and short comings of our formalizations.

We did not prove total correctness, that is, reachability and termination in addition to partial correctness. Although informal inspection leads us to believe that `thttpd` terminates, total correctness will surely be important in proving availability.

Our model uses natural numbers. To be more rigorous we could use integers and prove there are no overflows. The only negative integer used, -1, is a return values, and we handle it with the underspecified function, `int_neg`. Although our implementation leads to an inconsistency (see Sect. 6.4.1 for details), we believe our proof is valid.

We did not formalize all of C. For example, we ignored floating point arithmetic, the C preprocessor, unrestricted pointers (pointer casts) and pointers-to-function, storage rules for structures and enumerations (`enum`), `do while` and `for` loops, scoping rules for multiple source files and nested blocks. We did not implement inference rules for recursive or mutually recursive function, but believe Homeier's logic would suffice. Likewise we abstract away or ignore many parts of Unix, such as I/O buffering, directories, and memory allocation.

We ignored the C macro preprocessor almost entirely. The subject, `thttpd`, used three macros with parameters, `LOG2`, `LOG4`, and `S_ISREG`. We treated them as functions, which is adequate for this code. We treated macros without parameters as special constants. In general macros must be expanded (running `cpp` expands *all* macros, including system macros) before translating into the abstract syntax or the preprocessor "language" must be formalized. Since preprocessing occurs before parsing, the formalization would have to be very low level (line oriented tokens) and the parsing done in the formal system.

Currently `exit()` is modeled as $\vdash \{P\}$ `exit();` $\{\mathrm{F}\}$. This is not unreasonable since control never passes to subsequent statements, equivalent to nontermination. However since $F \Rightarrow$ `anything`, we must informally inspect the proof to satisfy ourselves that the final constraints of functions really are established.

Similarly we have not given inference rules for values from `return` statements. Homeier's entry logic may provide a way to express them.

Representation of arrays is problematic. Part of the problem arises from the similarity in C between pointers and arrays. Compounding that is our choice to represent arrays as compound types carrying the value function *and* the array size. Lastly it is tempting to represent arrays of characters as HOL strings. Thus array variables are variously represented as `:'a list`, `:'a ptr`, `CA (CA_FN v) (CA_IDX v)`, `:string`, and `Struct "(ascii)CArray"`.

Although we have not explicitly addressed memory overwrites, which may arise from storing into an array beyond its boundary, we believe our axiomatization is conservative. The array read and write functions, `CA` and `CA_PUT`, are undefined if the index is outside the array boundary. Therefore there should be no need for a separate proof addressing memory.

There is no formal mechanism for top level proof. Currently we prove theorems for each routine separately. The top level goal is that a call to a routine named `main()` is secure. In particular the environment in which the function theorems are proved (in `WF_fnp_TAC`) is not required to be the same as the functions used in CALL tactics.

```
isSecure (main (int argc, char *argv[], char *envp[]))
```

The answer would be to create the environment from the complete code listing and system axioms, then draw only from that. The top level proof would look something like the following.

```
isSecure
    [WF_fnp chdir ... ,
     WF_fnp chroot ... ,
     WF_fnp geteuid ... ,
         ...    ]
    logfile (FILE *F) { ...
    LOG2 (char *x, ...
    LOG4 (char *x, ...
    error (char *s) { ...
    cat (char s[]) { ...
    fetch () { ...
    main (int argc, ...
```

The set of definitions, theorems, axioms, and tactics we have is, not surprisingly, far from a commercial or even particularly usable product or system. It

- lacks a good user interface,
- is not packaged (portable, installation script, single tar file, etc.), and
- lacks a tutorial and other examples.

Additionally the tactics aren't as general purpose as possible.

Although it has been a source of errors in the past, we have not established our axiomatic semantics and inference rules from a denotational semantics. This is important to avoid subtle errors in the inference rules and axioms.

The example code is still relatively small compared with, say, an operating system kernel, so we lack management tools for proofs of large, complex programs.

Large programs have rich, complicated behavior, so typically have large conditions, and we need to prove many properties about them. We manually handled a proof of two properties over eight functions with one assumption. However we surely need something more if we have, say many properties and dozens of assumptions and functions.

We only explored one program and verification method in detail. To be able to validly draw conclusions about the superiority (or inferiority) of this method, we must verify several different programs or try several verification methods.

8.3 Future Work

In this section we note possible directions for future work.

8.3.1 Inference Rules

The inference rules for `if` and `while` statement in Chapter 5 show how side effects can be characterized. It is straight forward to extend that to `for` and `do ...while` loops and operators with sequence points, such as `||` and `&&`. The inference rules must also handle `break` and `continue` statements in loops, perhaps along the lines of Homeier's entry and exit logics.

Most likely the structure of statements in axiomatic semantics must be extended to admit multiple post conditions after [1] to reason about exit and return conditions.

We must develop a whole set of inference rules for termination and reachability, to prove total correctness in the presence of nonlocal jumps.

Also, the inference rules should be proven correct from a lower level logic, such as operational semantics [38] or abstract state machines [24].

8.3.2 Numbers, Pointers, and Types

Modeling computer integer arithmetic with natural numbers is quite unrealistic, and, in our case leads to a contradiction. At a minimum integers should be the default model with a means of proving there are no overflows. Additionally some programs need rational or real arithmetic for floating point numbers.

The current simple modeling of arrays and pointers must be extended to follow the C model more closely, and must be able to model dynamically allocated data structures at a minimum.

C uses type conversion often and subtly. A set of inference rules to represent and reason about explicit and implicit type conversions will be necessary.

8.3.3 Function Calls

We do not model functions with a variable number of parameters well. If the output of, say, a `printf()` statement is important to the proof, the current methods must be extended.

We axiomatize only a small part of the functionality of operating system and library calls. Full descriptions will likely be huge. Extracting the information needed for a particular proof could be very tedious. More research is needed to either establish that this is not a problem or to come up with a workable approach.

8.3.4 A Complete Verification System

The inference rules, parsers, tactics, etc. we developed to verify `thttpd` by no means constitute a broadly useful software verification system. However our experience has earned us insight into what *would* be needed. As first proposed in [8], we believe the following elements are both necessary and sufficient.

1. A library of examples of design formalizations and examples of how to formalize common programming patterns. A formal specification is the first step in verification [5] and can, in itself, be of great benefit [16]. But finding a formalization and avoiding lapses can be hard. For instance, the specification of sorting in an early version of [21] could have been trivially satisfied by setting all the values to zero: it didn't specify that values at the end are a permutation of beginning values.

2. A high level model of the language along with rules of inference, such as axiomatic semantics. As explained in Sect. 4.6, page 26 the logic must be proven correct from a low level, definitional semantics to minimize errors.

3. Formal models of the environment. This begins, of course, with the programming language, but includes standard libraries, operating system routines, network services, etc. Larger programs typically use other services rather than being stand-alone entities.

4. A powerful, highly automated theorem proving environment. This corresponds to PVS [39], very powerful tactics in HOL, verification condition generators [32, 36], etc. An environment which finds loop invariants and proves most lower level theorems automatically allows a lower entry training cost and less user time. Additionally there must be a good user interface, probably graphical-based.

A system like this could be as widely used as compilers, revision control systems, or project management tools are today. It is an indictment of the state of the art that verifying a mundane two page program in a common language is a doctoral dissertation. Much of the theoretical foundation is in place; it does however require a huge amount of software development.

8.4 Conclusions

As we proposed in the thesis statement (Sect. 1), we applied computer-assisted, post-hoc formal verification to useful, production code written in a widely used language. In spite of very complex semantics of the language, we have developed additional insights into the code and increased our confidence in its security.

Many aspects of program verification have long since been established and are well known. However formal verification, particularly by axiomatic semantics, is not widely practiced. This dissertation adds details and approaches to deal with real-world problems rather than being limited to simple languages and examples. We developed and published inference rules for languages with side effects in expressions, methods of deeply embedding a semantically complex language, tactics for verifying software, a formalization of some security properties, and security models of parts of the Unix file system, processes, and system calls.

Since `thttpd` is a typical C program in many ways, we feel that this single successful verification strongly supports our belief that verification of real programs in real languages is practical. Dijkstra's call to sacrifice good software design practices when needed to ease verification is less urgent since well designed software can be verified just as easily. Although no panacea or "silver bullet," formal verification can and, in our opinion, will play an increasing important role in producing high quality software.

APPENDIX A

A SOFTWARE VERIFICATION MANUAL

This chapter is a manual for software verification using the formalizations and tactics developed for the dissertation. It mixes a tutorial with a reference manual for tactics. We assume the reader is generally familiar with axiomatic semantics and HOL.

Section A.1 explains the preliminaries of translating source code and specifications into a goal to solve in HOL, while Sect. A.2 gives some hints and general suggestions about how to prove goals, especially those related to software verification. Sections A.3, A.4, A.5, and A.6 present inference rules and related tactics beginning at simple expressions and ending with functions. In the last section, Sect. A.7, we explain other, new tactics which we have found useful.

A.1 Verifying C Code

A good way to learn to verify is to begin with small fragments of C code. However for complete programs, one verifies C functions and programs. We begin by explaining how to verify C code, then show the additional steps to verify C functions.

The first step is to translate C code into an equivalent abstract syntax tree. It is probably quicker to hand-translate a short piece of code if one is familiar with AST forms. For longer pieces of code or if one is beginning, we have a translator which helps job. Specific instructions on how to load and run the translator are in file `c2holReadme.sml`. All the necessary files can be found at the following URL.

```
http://hissa.ncsl.nist.gov/~black/Disser/
```

The translator lexically analyzes the code, and does some semantic parsing, such as assigning types to variable references. The translator was written as a simple aid to create abstract syntax trees, not a finished tool, and there are many things the translator cannot handle or does wrong. Thus before running the translator, the user must make the following changes to the source code.

1. Make sure the code compiles without error. (The translator lacks many error checks.)

2. Remove comments.

3. Remove #ifdef's. This implies the user must decide which version to verify.

4. Replace #define macros which have parameters. If the macro is simple, the user can replace it with a function. If not, the user must do some preprocessing.

5. Remove #includes. Add appropriate declarations textually.

Be sure that declarations for global variables and function return values precede the source code to be translated. In the resultant AST, the user may need to do the following fix ups.

1. Correct or "cast" types, especially system types which are equivalent to `int`.

2. Correct disambiguators so variable references are correct.

3. Add to each function the list of global variables which the function, or its called functions, use.

4. Correct operator precedence or association errors.

If a variable has type `UnknownType`, add a declaration and translate again.

When one has an abstract syntax tree representation of the code, one must express convert side conditions and specifications into preconditions and postconditions. This is often an interactive process strengthening the precondition, adding assumptions, or weakening the postcondition as one gets stuck with unprovable goals. Finally the goal can be recorded as a theorem when it is proved.

A.2 Hints for Program Verification

This section is a record of some problems we ran into and how we solved them. This is not a complete tutorial on how to prove theorems in HOL by any means. But it does include some "advanced" tips particularly relevant to goals which arise in software verification.

A.2.1 Level of Proof

After many decades of improvements, computers are often fast enough to be used lavishly in proofs. Rather than spending 20 minutes crafting a minimal proof of some subgoal which takes milliseconds of CPU time, a user can often employ a general tactic. The general tactic is wasteful, often trying dozens of possibilities before hitting upon a method to solve a goal and taking 30 seconds of CPU time. But the user completes the step in two minutes rather than 20. Try the most powerful tactics first. If it finishes in a reasonable time, you're done. If not, you can work out a more intelligent proof.

As computers get faster, the "brute force" approach to theorem proving will make sense more often. Users can spend more time on formalizing properties and problems and less on proving theorems. However there will always be theorems which can't be solved automatically and there will always be a need for more powerful, efficient tactics to solve higher level theorems, so the basic tactics can't be ignored.

A.2.2 How to Prove Goals

Suppose you have a goal such as the following.

```
(--`(arSz = CA_SZ ar) /\ arSz > 0 /\ (j = 0) /\ (max = 0) ==>
    j <= arSz /\ (CA_SZ ar = arSz) /\ arSz > 0 /\
    (j > 0 ==> (?n. max = CA_IDX ar n)) /\
    (!n. 0 <= n /\ n < j ==>
                max >= CA_IDX (CA (CA_FN ar) arSz) n)`--)
```

These have the form $a \wedge \ldots \wedge k \Rightarrow p \wedge \ldots \wedge x$ and tend to arise when proving that one condition implies another, for instance, when strengthening preconditions. I find these difficult even to read, let alone prove.

A good first step is to break it into pieces with STRIP_THEN_REWRITE_TAC. The definition is

```
val STRIP_THEN_REWRITE_TAC =
        REPEAT STRIP_TAC THEN ASM_REWRITE_TAC [];
```

This usually results in a few much simpler subgoals. Applied to the above goal, we get these three subgoals.

```
(--`0 >= CA_IDX (CA (CA_FN ar) (CA_SZ ar)) n`--)
____________________________
    (--`arSz = CA_SZ ar`--)
    (--`arSz > 0`--)
    (--`j = 0`--)
    (--`max = 0`--)
    (--`0 <= n`--)
    (--`n < j`--)

(--`?n. 0 = CA_IDX ar n`--)
____________________________
    (--`arSz = CA_SZ ar`--)
    (--`arSz > 0`--)
    (--`j = 0`--)
    (--`max = 0`--)
    (--`j > 0`--)

(--`0 <= CA_SZ ar`--)
____________________________
    (--`arSz = CA_SZ ar`--)
    (--`arSz > 0`--)
    (--`j = 0`--)
    (--`max = 0`--)
```

In fact, all these goals could be proved by SOLVE_TAC (see Sec. A.7.1). So the original goal could be proved with the following tactic.

```
e (STRIP_THEN_REWRITE_TAC THEN SOLVE_TAC);
```

However many times subgoals can't be proven automatically, so we shall continue without SOLVE_TAC for now.

Working on these subgoals can often reveal missing assumptions, some rewrite which should be done before breaking up the goal, or incorrectly entered predicates (missing parentheses).

If examination doesn't show any obvious errors, one must try to prove the subgoals by hand. There are two general ways to prove these: show that the goal follows from the assumptions or show a contradiction in the assumptions.

Consider the first subgoal (HOL's style of printing shows the first subgoal to be proved last).

```
(--`0 <= CA_SZ ar`--)
____________________________
    (--`arSz = CA_SZ ar`--)
    (--`arSz > 0`--)
    (--`j = 0`--)
    (--`max = 0`--)
```

We see `CA_SZ ar` in the goal and the same value in the top assumption, where it is equal to `arSz`: this is a possibility. Looking around, we see `arSz > 0` in the next assumption. Clearly if `arSz > 0` and `arSz = CA_SZ ar`, `0 <= CA_SZ ar`, so we have a direction to proceed. How can we do that with HOL tactics?

The first assumption will rewrite instances of `arSz`, not `CA_SZ ar`. We can "bring down" (undischarge) `arSz > 0`, rewrite with assumptions, then ARITH_TAC should solve the goal. The following tactic solves this goal.

```
e (UNDISCH_TAC (--`arSz > 0`--) THEN
   ASM_REWRITE_TAC [] THEN ARITH_TAC);
```

Consider the next subgoal.

```
(--`?n. 0 = CA_IDX ar n`--)
____________________________
    (--`arSz = CA_SZ ar`--)
    (--`arSz > 0`--)
    (--`j = 0`--)
    (--`max = 0`--)
    (--`j > 0`--)
```

Note there is nothing in the assumptions about the values of CA_IDX (unless something can be rewritten with its definition). However we see that `j = 0` and `j > 0` are both assumed. This is a contradiction.

To demonstrate the contradiction we could try to undischarge `j > 0` and rewrite with assumptions (to get `0 > 0 ==> ...`) or we could use `j > 0 ==> ~(j = 0)` (proved with ARITH_TAC) with IMP_RES_TAC.

I tried the first method, but ARITH_TAC didn't prove the goal. Rather than trying to look up a theorem about `~(0 > 0)`, I backed up and tried the second method. The following tactic solved the goal.

```
e (IMP_RES_TAC (prove((--`j > 0 ==> ~(j = 0)`--), ARITH_TAC)));
```

One can often get ARITH_TAC to prove a convenient lemma even when it fails to help on the larger goal.

The last goal also cannot be proved true (directly).

```
  (--`0 >= CA_IDX (CA (CA_FN ar) (CA_SZ ar)) n`--)
  ____________________________
      (--`arSz = CA_SZ ar`--)
      (--`arSz > 0`--)
      (--`j = 0`--)
      (--`max = 0`--)
      (--`0 <= n`--)
      (--`n < j`--)
```

However we again see `j = 0` and `n < j` which are contradictory. We'll try the approach which worked for the previous goal. Indeed the following tactic solves the goal.

```
e (IMP_RES_TAC (prove((--`n < j ==> ~(j = 0)`--), ARITH_TAC)));
```

Finally we can examine all the tactics which solved the subgoals to see if there is anything in common. Since there isn't, we just combine them into one complete tactic.

```
e (STRIP_THEN_REWRITE_TAC THENL [
```

```
    UNDISCH_TAC (--`arSz > 0`--) THEN
                ASM_REWRITE_TAC [] THEN ARITH_TAC,
    IMP_RES_TAC (prove((--`j > 0 ==> ~(j = 0)`--), ARITH_TAC)),
    IMP_RES_TAC (prove((--`n < j ==> ~(j = 0)`--), ARITH_TAC))
]);
```

A.2.3 HOL Error: Invalid Tactic

Problem: using CALL_TAC or some other tactic causes a HOL error with only the error message `Invalid tactic`.

Example: While trying to prove file integrity of the entire `thttpd`, we tried to use the proof of `main()`. Even the HOL handler didn't yield much information.

```
e (CALL_TAC WF_fnp_fi_main) handle e=>Raise e;

Exception raised at Tactical.VALID:
Invalid tactic
```

Solution: In this case, the `WF_fnp_fi_main` used some assumptions which we had not included in the overall proof. We added the assumptions, started over, and the proof succeeded.

A.2.4 When Rewrites Don't Work

Problem: a rewrite or match which should work just doesn't.

Example: A simple rewrite should solve the following goal, but it doesn't.

```
 (--`x + 2 = y + 2`--)
 ----------------------------
     (--`x = y`--)

- e (ASM_REWRITE_TAC []);

OK..
  (--`x + 2 = y + 2`--)
  ----------------------------
```

```
(--`x = y`--)
```

(Your case is probably much more complex than this.)

Solution: Check that the types actually match. In the above, x is a string in the assumption, not a number. In complex goals, there may be unnoticed typographical errors, too. The function WHY_NOT reports such possible mismatches. It reports the following situations in the current (top) goal.

1. variables or constants with the same name, but different types, for instance (--`gamma:num`--) and (--`gamma:string`--).

2. variables or constants with the same type, but similar names, for instance isQueue and isQueu. It does not consider names of two characters or less (too many x's and y's would be reported) nor "misspellings" of the initial characters (because of names like xSize and ySize).

Problem: a reduction, such as, SUBSE isn't replaced.

Example: We try to prove a call to fprintf, but end up with a goal with SUBSE in it. It represents binding formals to actuals.

```
1 subgoal:
(--`Partial
 ((F = ^F) /\ ~(inodeOf (deref F) = SYS_stdout) /\
  SoFS hasConfidentialityFile SYS_FileSystem)
 (Simple (Call (Var "fprintf" 0 Int)
   (PL (Lval (Vref (Var "F" 0 (Ptr (Struct "FILE")))))
   (PL (Const (CCstr "%s %s %s %s %s %d %d ") (Ptr Char))
   (PL (Lval (Vref (Var "remotehost" 0
                    (Array Char (CCid "BUFSIZE")))))
   (PL (Lval (Vref (Var "remoteuser" 0
                    (Array Char (CCid "BUFSIZE")))))
   (PL (Lval (Vref (Var "daemonname" 0
                    (Array Char (CCid "BUFSIZE")))))
   (PL (Lval (Vref (Var "remotedata" 0
                    (Array Char (CCid "BUFSIZE")))))
```

```
   (PL (Lval (Vref (Var "timestamp" 0 (Array Char (CCint 64)))))
   (PL (Call (Var "getpid" 0 Int) PLnull)
   (PL (Call (Var "getppid" 0 Int) PLnull) PLnull))))))))))
 ((F = ^F) /\ ~(inodeOf (deref F) = SYS_stdout) /\
  SoFS hasConfidentialityFile SYS_FileSystem)`--)

- e(CALL_TAC
     (SPEC (--`hasConfidentialityFile:num->^unixFile->bool`--)
       (GEN (--`preFSS:num->^unixFile->bool`--) SYS_fprintf)) THEN
    PURE_REWRITE_TAC [SoFS,COND_EXPAND,hasConfidentialityFile] THEN
    BETA_TAC THEN STRIP_THEN_REWRITE_TAC THEN
                                (SOLVE_TAC ORELSE ALL_TAC));

1 subgoal:
(--`SUBSE
 (!SYS_FileSystem errno.
  (?FPRINTF_error. C_Result5 = int_neg FPRINTF_error) /\
  inSomeCasesOf C_Result5 (?FPRINTF_errno.errno = FPRINTF_errno) /\
  (!inode. (inode = SYS_stdout) ==>
    nonConfidential (getFile SYS_FileSystem inode)) \/
    (C_Result5 = 0) /\
    (!inode. (inode = SYS_stdout) ==>
        nonConfidential (getFile SYS_FileSystem inode)) \/
     C_Result5 > 0 /\
      (!inode. (~(inode = inodeOf (deref stream)) \/
        (?prev.
           ((inode = SYS_stdout) ==> nonConfidential prev) /\
            (appendFile (printfSpec format vargs) prev =
             getFile SYS_FileSystem inode))) /\
           ((inode = inodeOf (deref stream)) \/
            ((inode = SYS_stdout) ==>
             nonConfidential (getFile SYS_FileSystem inode)))) ==>
        (!inode. (inode = SYS_stdout) ==>
```

```
          nonConfidential (getFile SYS_FileSystem inode)))
[Var "stream" 0 (Ptr (Struct "FILE")); Var "format" 0(Ptr Char);
 Var "%%genvar%%4880" 0 (Array Char (CCid "BUFSIZE"));
 Var "%%genvar%%4879" 0 (Array Char (CCid "BUFSIZE"));
 Var "%%genvar%%4878" 0 (Array Char (CCid "BUFSIZE"));
 Var "%%genvar%%4877" 0 (Array Char (CCid "BUFSIZE"));
 Var "%%genvar%%4876" 0 (Array Char (CCint 64));
 Var "%%genvar%%4875" 0 Int; Var "%%genvar%%4874" 0 Int]
[Lval (Vref (Var "F" 0 (Ptr (Struct "FILE"))));
 Const (CCstr "%s %s %s %s %s %d %d ") (Ptr Char);
 Lval (Vref (Var "remotehost" 0
                         (Array Char(CCid "BUFSIZE"))));
 Lval (Vref (Var "remoteuser" 0
                         (Array Char(CCid "BUFSIZE"))));
 Lval (Vref (Var "daemonname" 0
                         (Array Char(CCid "BUFSIZE"))));
 Lval (Vref (Var "remotedata" 0
                         (Array Char(CCid "BUFSIZE"))));
 Lval (Vref (Var "timestamp" 0 (Array Char (CCint 64))));
 Call (Var "getpid" 0 Int) PLnull;
 Call (Var "getppid" 0 Int) PLnull]`--)
----------------------------
 (--`F = ^F`--)
 (--`~(inodeOf (deref F) = SYS_stdout)`--)
 (--`!inode. (inode = SYS_stdout) ==>
          nonConfidential (getFile SYS_FileSystem inode)`--)
```

Solution: Notice that the last two parameters are actually function calls. Side effects should be separated first. In this case, SEP_CALL_TAC works.

A.3 Simple Expressions

We shall begin with expressions. However, since expressions can have significant side-effects, handling them becomes quite involved.

A.3.1 Assignment Expressions

In C assignments are expressions, not statements. However correctness conditions are given over statements. Fortunately any expression can be a stand-alone statement, so the distinction should not cause problems.

Here is the inference rule for simple assignment without side effects.

$$\frac{\begin{array}{c}\texttt{pre} = \texttt{post}^{\texttt{x}}_{\texttt{e}} \\ \texttt{NoSE e}\end{array}}{\vdash \{\texttt{pre}\}\ (\texttt{Simple}\ (\texttt{Assign}\ (\texttt{Vref x})\ \texttt{e}))\ \{\texttt{post}\}}$$

The first condition states that the precondition (pre) is the postcondition with all x's replaced by the expression e. The second condition states that the expression has no side effects. If these two conditions are met, one can conclude the partial correctness of the assignment statement. A complete explanation of *why* this is the correct form for assignment can be found in [21].

Tactics

ASSIGN_TAC : tactic

SYNOPSIS

Prove goal of assignment to a variable or an array element.

KEYWORDS

tactic, assignment.

DESCRIPTION

Prove a goal consisting of a lone assignment statement of the form `Simple (Assign (Vref x) e)` or `Simple (Assign (Aryref a i) e)`. The weakest precondition is computed from the goal's postcondition and the assignment statement. The computed precondition is strengthened using `PRE_STRENGTHEN_TAC` to match the goal's precondition and the resulting implication subgoal is proved if possible.

Subgoals: The precondition strengthening may leave a subgoal.

FAILURE

```
LUnary PreInc (Vref (Var "B" 0 Int))
uncaught exception Fail: cexpr2hol SKIMP 2
```

The `cexpr2hol SKIMP 2` means that the routines can't produce an HOL expression for the C expression. This is often because the the expression has side effects, such as a function call or, in this case, an increment operator. First use one of the side effect inference rules to separate out the side effect, then use ASSIGN_TAC.

EXAMPLE

Prove that after `x = 1;` $x > 0$.

```
1 subgoal:
  (--`Partial T (Simple (Assign (Vref (Var "x" 0 Int))
                                          (Const (CCint 1) Int)))
        (x > 0)`--)
- e (ASSIGN_TAC);
OK..
Goal proved.
```

SEE ALSO

`SEQ_ASSIGN_TAC`, `FWD_SEQ_ASSIGN_TAC`, `PRE_SIDE_EFFECT_TAC`, `POST_SIDE_EFFECT_TAC`, `PRE_STRENGTHEN_TAC`.

`SEQ_ASSIGN_TAC : tactic`

SYNOPSIS

Separate and prove the last statement, an assignment.

KEYWORDS

tactic, assignment, sequence.

DESCRIPTION

Separate the last statement, which must be an assignment, and prove correctness. The intermediate condition is the weakest precondition computed from the final assignment. If the goal is just an assignment, ASSIGN_TAC is called, so that REPEAT SEQ_ASSIGN_TAC works.

Subgoals: All statements preceding the last assignment are left. If the statement is just an assignment, precondition strengthening may leave an implication.

FAILURE

See ASSIGN_TAC.

EXAMPLE

Prove swapping two variables, x and y, using a temporary variable, r. From [21, page 17].

```
1 subgoal:
  (--`Partial ((x = ^X) /\ (y = ^Y))
        (Seq
           (Simple (Assign (Vref (Var "r" 0 Int))
                            (Lval (Vref (Var "x" 0 Int)))))
        (Seq
           (Simple (Assign (Vref (Var "x" 0 Int))
                            (Lval (Vref (Var "y" 0 Int)))))
           (Simple (Assign (Vref (Var "y" 0 Int))
                            (Lval (Vref (Var "r" 0 Int)))))))
        ((x = ^Y) /\ (y = ^X))`--)
- e (REPEAT SEQ_ASSIGN_TAC)
OK..
Goal proved.
```

SEE ALSO

ASSIGN_TAC, FWD_SEQ_ASSIGN_TAC, TL_SEQUENCE_TAC, PRE_SIDE_EFFECT_TAC, POST_SIDE_EFFECT_TAC, PRE_STRENGTHEN_TAC.

FWD_SEQ_ASSIGN_TAC : tactic

SYNOPSIS

Separate and prove the first statement, a variable initialization.

KEYWORDS

tactic, assignment, sequence.

DESCRIPTION

Separate the first statement, which must be an assignment, try to deduce an appropriate postcondition, and try prove it. In other words, add a variable initialization to the precondition. The variable must not be in the precondition.

Subgoals: All statements following the first assignment are left. If the statement is just an assignment, precondition strengthening may leave an implication.

FAILURE

```
uncaught exception Fail: FWD_SEQ_ASSIGN_TAC: cannot handle lhs
in precondition (yet)
```

The variable being assigned is already in the precondition. For instance, this tactic handles

```
  (--`Partial (a > 7)
       (Simple (Assign (Vref (Var "max" 0 Int))
                            (Lval (Vref (Var "a" 0 Int)))))
     (a > 7 /\ (max = a))`--)
```

but not the following since it has `max < 0` in the precondition.

```
  (--`Partial (a > 7 /\ max < 0)
        (Simple (Assign (Vref (Var "max" 0 Int))
                               (Lval (Vref (Var "a" 0 Int)))))
      (a > 7 /\ (max = a))`--)
```

Use PRE_STRENGTHEN_TAC to get rid of the `max < 0` in the precondition first.

EXAMPLE

At the beginning of a proof of division by repeated subtraction, "initialize" `r` to `x` and `q` to `0`.

```
1 subgoal:
(--`Partial ((^X = x) /\ (^Y = y))
    (Seq (Simple (Assign (Vref (Var "r" 0 Int))
                         (Lval (Vref (Var "x" 0 Int)))))
    (Seq (Simple (Assign (Vref (Var "q" 0 Int))
                         (Const (CCint 0) Int)))
    (CWhile (Binary (Lval (Vref (Var "y" 0 Int))) LEq
                 (Lval (Vref (Var "r" 0 Int))))
      (Seq (Simple (Assign (Vref (Var "r" 0 Int))
                     (Binary (Lval (Vref (Var "r" 0 Int))) Sub
                       (Lval (Vref (Var "y" 0 Int))))))
      (Simple (Assign (Vref (Var "q" 0 Int))
                    (Binary (Lval (Vref (Var "q" 0 Int))) Add
                      (Const (CCint 1) Int))))))))
        (r < y /\ (x = r + y * q) /\ (^X = x) /\ (^Y = y))`--)
- e (REPEAT FWD_SEQ_ASSIGN_TAC);
1 subgoal:
(--`Partial ((^X = x) /\ (^Y = y) /\ (r = x) /\ (q = 0))
    (CWhile (Binary (Lval (Vref (Var "y" 0 Int))) LEq
             (Lval (Vref (Var "r" 0 Int))))
      (Seq (Simple (Assign (Vref (Var "r" 0 Int))
                 (Binary (Lval (Vref (Var "r" 0 Int))) Sub
                   (Lval (Vref (Var "y" 0 Int))))))
```

```
(Simple (Assign (Vref (Var "q" 0 Int))
          (Binary (Lval (Vref (Var "q" 0 Int))) Add
            (Const (CCint 1) Int)))))
  (r < y /\ (x = r + y * q) /\ (^X = x) /\ (^Y = y))`--)
```

USES

With REPEAT this automatically handles a series of variable initializations.

SEE ALSO

ASSIGN_TAC, SEQ_ASSIGN_TAC, SEQUENCE_TAC, PRE_SIDE_EFFECT_TAC, POST_SIDE_EFFECT_TAC, PRE_STRENGTHEN_TAC.

A.3.2 Expressions With No Effect

Our method of handling side effects is to transform the original code into an equivalent sequence which passes results by logical variables. For instance, to prove

$$\vdash \{\texttt{PRE}\}\ \texttt{if}\ (\texttt{++a}\ <\ \texttt{MAX})\ \ldots\ \{\texttt{POST}\}$$

we can prove the following subgoals.

$$\vdash \{\texttt{PRE}\}\ \texttt{++a}\ <\ \texttt{MAX}\ \{\texttt{PRE} \wedge (\texttt{C\_Result} = (\texttt{a} < \texttt{MAX}))\}$$
$$\vdash \{\texttt{PRE} \wedge (\texttt{C\_Result} = (\texttt{a} < \texttt{MAX}))\}\ \texttt{if}\ (\texttt{C\_Result})\ \ldots\ \{\texttt{POST}\}$$

To avoid infinite regress, the result of the test is only equated with a variable in the postcondition: we don't add another assignment statement, such as `C_Result = a < MAX`. After separating the preincrement, we have the following subgoal. (The preincrement is a separate subgoal which establishes PRE.)

$$\vdash \{\texttt{PRE}\}\ \texttt{a}\ <\ \texttt{MAX}\ \{\texttt{PRE} \wedge (\texttt{C\_Result} = (\texttt{a} < \texttt{MAX}))\}$$

The following inference rule allows us to reason about expressions which have no side effects, as `a < MAX` above, except to assign the result to a logical variable.

$$\frac{\begin{array}{c}\texttt{pre} = \texttt{post}_{\texttt{e}}^{\texttt{C\_Result}} \\ \texttt{NoSE e}\end{array}}{\vdash \{\texttt{pre}\}\ \texttt{Simple e}\ \{\texttt{post}\}}$$

EXPR_TAC : tactic

SYNOPSIS

Prove an expression which has no side effects.

KEYWORDS

tactic, expression.

DESCRIPTION

Prove a simple expression which may leave results in a logical variable. The expression cannot have side effects. If necessary, the computed precondition is strengthened to match the goal's precondition. The resulting implication subgoal is proved if possible.

Subgoals: The precondition strengthening may leave a subgoal.

FAILURE

```
uncaught exception Fail: multiple C_Result's in post: use
GENERAL_EXPR_TAC
```

If the postcondition has more than one C_Result, the user must specify which one to use.

```
unsolved goals
```

This HOL error may mean that the expression has side effects and `NoSE e` could not be proved.

EXAMPLE

```
1 subgoal:
  (--`Partial ((b = a) /\ a < ^c + 1 /\ (c = ^c))
        (Simple
          (Binary (Lval (Vref (Var "a" 0 Int))) Add
```

```
            (Lval (Vref (Var "b" 0 Int))))
        ((b = a) /\ a < ^c + 1 /\ (c = ^c)
                        /\ (C_Result1 = a + b))`--)
- e (EXPR_TAC);
OK..
Goal proved.
```

SEE ALSO

GENERAL_EXPR_TAC, ASSIGN_TAC, LUNARY_TAC, PRE_SIDE_EFFECT_TAC, POST_SIDE_EFFECT_TAC, PRE_STRENGTHEN_TAC.

GENERAL_EXPR_TAC : (string -> tactic)

SYNOPSIS

Prove an expression which has no side effects.

KEYWORDS

tactic, expression.

DESCRIPTION

Prove a simple expression which leave results in a variable. The expression cannot have side effects. If necessary, the computed precondition is strengthened to match the goal's precondition and the resulting implication subgoal is proved if possible. The result is bound to a variable named by the operand.

Subgoals: The precondition strengthening may leave a subgoal.

FAILURE

`unsolved goals`

This HOL error may mean that the expression has side effects and `NoSE e` could not be proved.

EXAMPLE

```
1 subgoal:
  (--`Partial ((b = a) /\ a < c /\ (c = C_Result2))
        (Simple
          (Binary (Lval (Vref (Var "a" 0 Int))) Add
            (Lval (Vref (Var "b" 0 Int)))))
        ((b = a) /\ a < c /\ (c = C_Result2)
                          /\ (C_Result1 = a + b))`--)
- e (GENERAL_EXPR_TAC "C_Result1");
OK..
Goal proved.
```

USES

This is a general form of EXPR_TAC for writing other tactics (binding a result to a particular variable) or when there is more than one result.

SEE ALSO

EXPR_TAC, ASSIGN_TAC, LUNARY_TAC, PRE_SIDE_EFFECT_TAC, POST_SIDE_EFFECT_TAC, PRE_STRENGTHEN_TAC.

A.3.3 Side Effect Expressions

The C language has operators which change the value of variables. We have inference rules which isolate expressions with these operators, but we have not deeply embedded the semantics of the operations. We use shallow (i.e., defined by SML routines) axiom introduction to create basic theorems about these.

$$\frac{\texttt{IS\_AXIOM}\ \mathsf{pre}\ \mathsf{stm}\ \mathsf{post}}{\vdash \{\mathsf{pre}\}\ \mathsf{stm}\ \{\mathsf{post}\}}$$

LUNARY_TAC : tactic

SYNOPSIS

Prove an increment or decrement statement.

KEYWORDS

tactic, statement.

DESCRIPTION

Prove the result of a statement consisting of just `++` or `--`. That is, the code must be

```
Simple (LUnary op lval)
```

where op is `PreInc`, `PostInc`, `PreDec`, or `PostDec`. The precondition is strengthened to be the original postcondition with the variable replaced by its new value (see ASSIGN_TAC). The postcondition is weakened to assign the result to a new logical variable. The strengthening and weakening implications are proved if possible.

Subgoals: It leaves no subgoal unless the postcondition does not follow from the precondition.

FAILURE

```
expected operator not found: LUnary
```

The statement is not in the form `Simple (Lunary ...)`. Simplify the statement first.

EXAMPLE

Show that `b` is incremented.

```
1 subgoal:
  (--`Partial ((b = B) /\ B < 1830)
        (Simple (LUnary PreInc (Vref (Var "b" 0 Int))))
        ((b = B + 1) /\ B < 1830)`--)
- e (LUNARY_TAC);
OK..
Goal proved.
```

SEE ALSO

EXPR_TAC, ASSIGN_TAC, PRE_SIDE_EFFECT_TAC, POST_SIDE_EFFECT_TAC, PRE_STRENGTHEN_TAC, POST_WEAKEN_TAC.

GENERAL_LUNARY_TAC : (string -> term -> tactic)

SYNOPSIS

A general form of LUNARY_TAC.

KEYWORDS

tactic, statement.

DESCRIPTION

This is a general form of LUNARY_TAC. It is only used internally now. It may be useful if the result of the expression must be used.

SEE ALSO

LUNARY_TAC.

A.3.4 The Empty Statement

The empty statement doesn't change anything. Thus the inference rule for it is simple.

$$\frac{}{\vdash \{\texttt{pre}\}\ \texttt{EmptyStmt}\ \{\texttt{pre}\}}$$

SKIP_TAC : tactic

SYNOPSIS

Prove the empty statement.

KEYWORDS

tactic, skip, empty, statement.

DESCRIPTION

Prove the empty statement. This weakens the postcondition to match it with the precondition.

Subgoals: The postcondition weakening may leave a subgoal.

FAILURE

Always succeeds if the statement is `EmptyStmt`.

EXAMPLE

```
1 subgoal:
  (--`Partial
        ((b = a) /\ a < ^c + 1 /\ (c = ^c)
         /\ (C_Result7 = a + b) /\ C_Result7 < 2 * c)
        EmptyStmt
        ((b = a) /\ a < ^c /\ (c = ^c))`--)
- e (SKIP_TAC);
Goal proved.
```

USES

Empty statements occasionally occur in loops or conditionals.

SEE ALSO

`EXPR_TAC`, `LUNARY_TAC`, `POST_WEAKEN_TAC`.

A.4 Statement Manipulation and Conditions

A.4.1 Replacing Statements with Semantically Equivalents

A statement may be replaced with a semantically equivalent one. This allows us to use this one, general inference rule for many types of statement transformations.

$$\frac{\begin{array}{c}\vdash \texttt{SEM\_EQ stm1 stm2} \\ \vdash \{\texttt{pre}\}\ \texttt{stm1}\ \{\texttt{post}\}\end{array}}{\vdash \{\texttt{pre}\}\ \texttt{stm2}\ \{\texttt{post}\}} \tag{A.1}$$

This inference rule is primarily used to separate side effects and reassociate sequences. Here are the inference rules for pre- and postevaluation side effects.

$$\frac{\texttt{PreEval lv expr s1} \quad \texttt{s2}}{\texttt{SEM\_EQ (Seq (Simple expr) s1) s2}}$$

$$\frac{\texttt{PostEval s1 expr} \quad \texttt{s2}}{\texttt{SEM\_EQ (Seq s1 (Simple expr)) s2}}$$

That is, a sequence is semantically equivalent to another statement if the sequence is a proper pre- or postevaluation separation.

Tactics

PRE_SIDE_EFFECT_TAC : (term -> term -> tactic)

SYNOPSIS

Separate an expression with a preevaluation side effect.

KEYWORDS

tactic, expression, preevaluation, side effect.

DESCRIPTION

Separates the second term, a C expression in AST form, from the current goal statement and introduces the first term, an intermediate condition, as the condition between the two new subgoals. The expression must not have any postevaluation side effects.

Side effects can only be separated from Simple, If, IfElse, or Ret statements. EmptyStmt, EmptyRet, Break, and Cont statements never have expressions. Use GENERAL_WHILE_TAC or WHILE_TAC for CWhile statements, one of the sequence tactics, such as SEQUENCE_TAC, for Seq statements, and BLOCK_TAC for Block statements.

Subgoals: The first subgoal is the expression which was separated. The second subgoal is the original statement with the expression replaced by a reference value.

FAILURE

```
expr is: LUnary PreInc (Vref (Var "b" 0 Int))
stmt is: Simple (Assign (Vref (Var "a" 0 Int)) (LUnary PostInc
                     (Vref (Var "b" 0 Int))))
uncaught exception Fail: expr not found in stmt in
sep_preeval_expr
```

The expression passed to PRE_SIDE_EFFECT_TAC was not found in the statement. (Note the expression has PreInc and the statement has PostInc above.) Change the expression to match the piece which you want to separate from the statement.

```
LUnary PostInc (Vref (Var "b" 0 Int))
semeqTactics.sep_preeval_expr: cannot prove that expr doesn't
have post-eval side effects
```

The expression probably has some postevaluation side effects. If so, separate a smaller piece which only has preevaluation side effects or use POST_SIDE_EFFECT_TAC instead.

```
semeqTactics.sep_preeval_expr: stmt must be (Simple e),
(If e s), (IfElse e s1 s2), or (Ret e)
```

Side effects can only be separated from Simple, If, IfElse, or Ret statements. Use another tactic to simplify the statement first.

EXAMPLE

Prove that after `a=++b;` both `a` and `b` are the original `B` plus 1.

```
1 subgoal:
  (--`Partial (b = B)
        (Simple (Assign (Vref (Var "a" 0 Int))
            (LUnary PreInc (Vref (Var "b" 0 Int)))))
        ((a = B + 1) /\ (b = B + 1))`--)


- e (PRE_SIDE_EFFECT_TAC (--`b = B + 1`--)
(--`LUnary PreInc (Vref (Var "b" 0 Int))`--));


OK..
2 subgoals:
  (--`Partial (b = B + 1)
        (Simple (Assign (Vref (Var "a" 0 Int))
                        (Lval (Vref (Var "b" 0 Int)))))
        ((a = B + 1) /\ (b = B + 1))`--)

  (--`Partial (b = B)
        (Simple (LUnary PreInc (Vref (Var "b" 0 Int))))
        (b = B + 1)`--)
```

SEE ALSO

POST_SIDE_EFFECT_TAC, GENERAL_WHILE_TAC, WHILE_TAC, BLOCK_TAC, SEQUENCE_TAC, LUNARY_TAC.

GENERAL_PRE_SIDE_EFFECT_TAC : (term -> string -> tactic)

SYNOPSIS

A general form of PRE_SIDE_EFFECT_TAC.

KEYWORDS

tactic, side effect, statement.

DESCRIPTION

This is a pure form of PRE_SIDE_EFFECT_TAC: it doesn't do the SEQUENCE_TAC. It is only used internally.

SEE ALSO

`PRE_SIDE_EFFECT_TAC`.

`POST_SIDE_EFFECT_TAC : (term -> term -> tactic)`

SYNOPSIS

Separate an expression with a postevaluation side effect.

KEYWORDS

tactic, expression, postevaluation, side effect.

DESCRIPTION

Separates the second term, a C expression in AST form, from the current goal statement and introduces the first term, an intermediate condition, as the condition between the two new subgoals. The expression must not have any preevaluation side effects.

Side effects can only be separated from `Simple` statements. `EmptyStmt`, `EmptyRet`, `Break`, and `Cont` statements never have expressions. Use one of the sequence tactics for `Seq` statements and BLOCK_TAC for `Block` statements. Because the semantics are so complex, we have no inference rules for other statements, yet.

Subgoals: The first subgoal is the original statement with the expression replaced by a reference value. The second subgoal is the separated expression.

FAILURE

```
expr is: LUnary PreInc (Vref (Var "b" 0 Int))
stmt is: Simple (Assign (Vref (Var "a" 0 Int)) (LUnary PostInc
(Vref (Var "b" 0 Int))))
uncaught exception Fail: expr not found in stmt in
sep_posteval_expr
```

The expression passed to POST_SIDE_EFFECT_TAC was not found in the statement. (Note the expression has `PreInc` and the statement has `PostInc` above.) Change the expression to match the piece which you want to separate from the statement.

```
LUnary PreInc (Vref (Var "b" 0 Int))
semeqTactics.sep_posteval_expr: cannot prove that expr doesn't
have pre-eval side effects
```

The expression probably has some preevaluation side effects. If so, separate a smaller piece which only has postevaluation side effects or use PRE_SIDE_EFFECT_TAC instead.

```
semeqTactics.sep_preeval_expr: stmt must be (Simple (Assign l
expr))
```

Side effects can only be separated from `Simple` assignment statements. Use another tactic to simplify the statement first or extend the inference rules.

EXAMPLE

Prove that after `a=b++;` a equals the original `B` and `b` equals the original `B` plus 1.

```
1 subgoal:
  (--`Partial (b = B)
        (Simple (Assign (Vref (Var "a" 0 Int))
            (LUnary PostInc (Vref (Var "b" 0 Int)))))
        ((a = B) /\ (b = B + 1))`--)
```

```
- e (POST_SIDE_EFFECT_TAC
(--`(a = B) /\ (b = B)`--)
(--`LUnary PostInc (Vref (Var "b" 0 Int))`--));
OK..
2 subgoals:
  (--`Partial ((a = B) /\ (b = B))
        (Simple (LUnary PostInc (Vref (Var "b" 0 Int))))
        ((a = B) /\ (b = B + 1))`--)

  (--`Partial (b = B)
        (Simple (Assign (Vref (Var "a" 0 Int))
                        (Lval (Vref (Var "b" 0 Int)))))
        ((a = B) /\ (b = B))`--)
```

SEE ALSO

PRE_SIDE_EFFECT_TAC, GENERAL_WHILE_TAC, WHILE_TAC, BLOCK_TAC, SEQUENCE_TAC, LUNARY_TAC.

GENERAL_POST_SIDE_EFFECT_TAC : (term -> tactic)

SYNOPSIS

A general form of POST_SIDE_EFFECT_TAC.

KEYWORDS

tactic, side effect, statement.

DESCRIPTION

This is a pure form of POST_SIDE_EFFECT_TAC: it doesn't do the SEQUENCE_TAC. It is only used internally.

SEE ALSO

PRE_SIDE_EFFECT_TAC.

`SEM_EQ_TAC : (term -> tactic)`

SYNOPSIS

Reduce a semantic equivalence goal.

KEYWORDS

tactic, semantic equivalence.

DESCRIPTION

This is only present for completeness.

USES

I have never used it.

A.4.2 Sequence Rule

The sequence rule states that executing two statements in sequence is the same as executing them separately with a matching intermediate condition.

$$\frac{\begin{array}{c}\vdash \{\mathtt{pre}\}\ \mathtt{stm1}\ \{\mathtt{interim}\} \\ \vdash \{\mathtt{interim}\}\ \mathtt{stm2}\ \{\mathtt{post}\}\end{array}}{\vdash \{\mathtt{pre}\}\ (\mathtt{Seq\ stm1\ stm2})\ \{\mathtt{post}\}}$$

Tactics

`SEQUENCE_TAC : (term -> tactic)`

SYNOPSIS

Separate the first statement from a sequence.

KEYWORDS

tactic, sequence.

DESCRIPTION

Split the first statement from a sequence. The operand is the intermediate condition.

Subgoals: The first subgoal is the first statement in the sequence. The second subgoal is the other statements.

FAILURE

```
Exception raised at Tactical.THEN:
Tactical.THEN: Resolve.MATCH_MP_TAC: No match
```

Could be applying SEQUENCE_TAC to a statement which is not a `SEQ`.

EXAMPLE

The following two `If` statements find the maximum of three numbers (when `max` is already assigned the value of the first, `a`). Split them into subgoals.

```
1 subgoal:
  (--`Partial (max = a)
        (Seq
           (If (Binary (Lval (Vref (Var "b" 0 Int))) Gt
                (Lval (Vref (Var "max" 0 Int))))
             (Simple (Assign (Vref (Var "max" 0 Int))
                 (Lval (Vref (Var "b" 0 Int))))))
          (If (Binary (Lval (Vref (Var "c" 0 Int))) Gt
               (Lval (Vref (Var "max" 0 Int))))
            (Simple (Assign (Vref (Var "max" 0 Int))
                (Lval (Vref (Var "c" 0 Int)))))))
        (((max = a) \/ (max = b) \/ (max = c)) /\
         max >= a /\ max >= b /\ max >= c)`--)

- e (SEQUENCE_TAC (--`((max=a)\/(max=b))
                         /\ max >= a /\ max >= b`--));
```

```
OK..
2 subgoals:
  (--`Partial (((max = a) \/ (max = b)) /\
              max >= a /\ max >= b)
        (If (Binary (Lval (Vref (Var "c" 0 Int))) Gt
             (Lval (Vref (Var "max" 0 Int))))
          (Simple (Assign (Vref (Var "max" 0 Int))
              (Lval (Vref (Var "c" 0 Int))))))
        (((max = a) \/ (max = b) \/ (max = c)) /\
          max >= a /\ max >= b /\ max >= c)`--)

  (--`Partial (max = a)
       (If (Binary (Lval (Vref (Var "b" 0 Int))) Gt
             (Lval (Vref (Var "max" 0 Int))))
          (Simple (Assign (Vref (Var "max" 0 Int))
              (Lval (Vref (Var "b" 0 Int))))))
       (((max = a) \/ (max = b)) /\ max >= a /\ max >= b)`--)
```

SEE ALLSO

TL_SEQUENCE_TAC, SEQ_ASSIGN_TAC, FWD_SEQ_ASSIGN_TAC.

TL_SEQUENCE_TAC : (term -> tactic)

SYNOPSIS

Separate the last statement from a sequence.

KEYWORDS

tactic, sequence.

DESCRIPTION

Split the last statement from a sequence. The operand is the intermediate condition.

Subgoals: The first subgoal is all but the last statement in the sequence. The second subgoal is the last statement.

USES

This is used to implement SEQ_ASSIGN_TAC.

SEE ALSO

`SEQUENCE_TAC, SEQ_ASSIGN_TAC, FWD_SEQ_ASSIGN_TAC.`

A.4.3 Precondition Strengthening

The precondition of a partial correctness theorem can always be replaced with a logically stronger condition.

$$\frac{\begin{array}{c}\texttt{pre} \Rightarrow \texttt{weaker} \\ \vdash \{\texttt{weaker}\}\ \texttt{stm}\ \{\texttt{post}\}\end{array}}{\vdash \{\texttt{pre}\}\ \texttt{stm}\ \{\texttt{post}\}}$$

Tactic

`PRE_STRENGTHEN_TAC : (term -> tactic)`

SYNOPSIS

Prove a goal with a weaker precondition.

KEYWORDS

tactic, precondition, stronger.

DESCRIPTION

Replace the current goal with a goal having a weaker precondition, the operand. Since we're doing backward proofs, the inference rule for precondition strengthening leads to a goal with a *weaker* precondition.

Subgoals: The first subgoal is the implication. The second subgoal is the statement with the weaker precondition.

FAILURE

Always succeeds if the goal is the proper form.

EXAMPLE

Weaken the precondition so it matches the assumption. The first subgoal can be solved with TAUT_TAC, and the second with ASM_REWRITE_TAC.

```
1 subgoal:
  (--`Partial ((p /\ ~s) /\ ~s) C2 q`--)
  ----------------------------
      (--`Partial (p /\ ~s) C2 q`--)
- e (PRE_STRENGTHEN_TAC (--`p /\ ~s`--));
OK..
2 subgoals:
  (--`Partial (p /\ ~s) C2 q`--)
  ----------------------------
      (--`Partial (p /\ ~s) C2 q`--)

  (--`(p /\ ~s) /\ ~s ==> p /\ ~s`--)
  ----------------------------
      (--`Partial (p /\ ~s) C2 q`--)
```

USES

This is often used to get the precondition in the right form, say as the invariant of a `while` loop, or to drop extraneous conditions.

SEE ALSO

`POST_WEAKEN_TAC`.

A.4.4 Postcondition Weakening

The postcondition of a partial correctness theorem can always be replaced with a logically weaker condition.

$$\frac{\begin{array}{c}\texttt{stronger} \Rightarrow \texttt{post}\\ \vdash \{\texttt{pre}\}\ \texttt{stm}\ \{\texttt{stronger}\}\end{array}}{\vdash \{\texttt{pre}\ \}\texttt{stm}\ \{\texttt{post}\}}$$

Tactic

POST_WEAKEN_TAC : (term -> tactic)

SYNOPSIS

Prove a goal with a stronger postcondition.

KEYWORDS

tactic, postcondition, weaker.

DESCRIPTION

Replace the current goal with a goal having a stronger postcondition, the operand. Since we're doing backward proofs, the inference rule for postcondition weakening leads to a goal with a *stronger* postcondition.

Subgoals: The first subgoal is the implication. The second subgoal is the statement with the stronger postcondition.

FAILURE

Always succeeds if the goal is the proper form.

EXAMPLE

Strengthen the postcondition so it matches an `LUnary` axiom. The first subgoal can be solved with TAUT_TAC, and the second with LUNARY_TAC.

```
1 subgoal:
  (--`Partial (b = B) (Simple (LUnary PostInc
                                  (Vref (Var "b" 0 Int)))
             (b = B + 1)`--)
```

```
- e (POST_WEAKEN_TAC (--`(b = B + 1)
                          /\ (C_Result2 = b - 1)`--));
OK..
2 subgoals:
  (--`Partial (b = B) (Simple (LUnary PostInc
                              (Vref (Var "b" 0 Int))))
        ((b = B + 1) /\ (C_Result2 = b - 1))`--)

  (--`(b = B + 1) /\ (C_Result2 = b - 1) ==> (b = B + 1)`--)
```

USES

This is rarely used explicitly in proofs. It is often used to implement other tactics to match postconditions.

SEE ALSO

PRE_STRENGTHEN_TAC.

A.4.5 Partial Correctness Conjunction

The conditions for two partial correctness theorems of the same code can be AND'd together.

$$\frac{\begin{array}{c}\vdash \{\mathtt{pre1}\}\ \mathtt{stm}\ \{\mathtt{post1}\} \\ \vdash \{\mathtt{pre2}\}\ \mathtt{stm}\ \{\mathtt{post2}\}\end{array}}{\vdash \{\mathtt{pre1}\ \wedge\ \mathtt{pre2}\}\ \mathtt{stm}\ \{\mathtt{post1}\ \wedge\ \mathtt{post2}\}}$$

There is no special tactic for this inference rule because it hasn't been used in proofs, yet.

A.4.6 Partial Correctness Disjunction

The conditions for two partial correctness theorems of the same code can be OR'd together.

$$\frac{\begin{array}{c}\vdash \{\mathtt{pre1}\}\ \mathtt{stm}\ \{\mathtt{post1}\} \\ \vdash \{\mathtt{pre2}\}\ \mathtt{stm}\ \{\mathtt{post2}\}\end{array}}{\vdash \{\mathtt{pre1}\ \vee\ \mathtt{pre2}\}\ \mathtt{stm}\ \{\mathtt{post1}\ \vee\ \mathtt{post2}\}}$$

There is no special tactic for this inference rule because it hasn't been used in proofs, yet.

A.5 Conditional and Loop Statements

A.5.1 Two-armed Conditionals

We can conclude the partial correctness of an If-Then-Else statement if both branches establish the postcondition. More particularly, the following must be true.

1. If the precondition and the test holds, after the "then" code finishes, the postcondition is true.
2. If the precondition and the complement of the test holds, after the "else code finishes, the postcondition is true.
3. The value of the test expression in the assertion language is "test."

$$\frac{\begin{array}{c}\texttt{IS\_VALUE expr test} \\ \vdash \{\texttt{pre} \wedge \texttt{ test}\}\ \texttt{stm1}\ \{\texttt{post}\} \\ \vdash \{\texttt{pre} \wedge \texttt{ \~{}test}\}\ \texttt{stm2}\ \{\texttt{post}\}\end{array}}{\vdash \{\texttt{pre}\}\ (\texttt{IfElse expr stm1 stm2})\ \{\texttt{post}\}} \tag{A.2}$$

This is the rule we originally implemented to prove `thttpd`. It is much simpler than the more complete Rule 5.6 given in Sect. 5.4, page 34. Since `thttpd` does not have any `if` statements with postevaluation side effects in the tests, we have not implemented the inference rule or written a tactic for it.

Tactics

`IFTHENELSE_TAC : tactic`

SYNOPSIS

Separate a conditional into subgoals for each branch.

KEYWORDS

tactic, conditional.

DESCRIPTION

Separate an `IfElse` goal into a subgoal for the true or "then" branch and a subgoal for the false or "else" branch. The correct logical test is generated from the code's test expression.

Subgoals: The first subgoal is the "true" or "then" branch. The second subgoal is the "false" or "else" branch.

FAILURE

```
Call (Var "strlen" 0 Int) (PL (Lval (Vref
         (Var "bs1" 0 (Array Char (CCid "BUFSIZE"))))) PLnull)


uncaught exception Fail: cexpr2hol SKIMP 4
```

You may have a side effect in the test. Use PRE_SIDE_EFFECT_TAC or SEP_CALL_TAC to separate the side effect first.

EXAMPLE

Prove `if (a > b) max = a; else max = b;`. Both goals can be resolved with ASSIGN_TAC.

```
1 subgoal:
  (--`Partial T
        (IfElse
           (Binary (Lval (Vref (Var "a" 0 Int))) Gt
                (Lval (Vref (Var "b" 0 Int))))
            (Simple (Assign (Vref (Var "max" 0 Int))
                         (Lval (Vref (Var "a" 0 Int)))))
            (Simple (Assign (Vref (Var "max" 0 Int))
                         (Lval (Vref (Var "b" 0 Int))))))
        (((max = a) \/ (max = b)) /\ max >= a /\ max >= b)`--)
- e (IFTHENELSE_TAC);
OK..
2 subgoals:
```

```
(--`Partial (T /\ ~(a > b))
     (Simple (Assign (Vref (Var "max" 0 Int))
                   (Lval (Vref (Var "b" 0 Int)))))
      (((max = a) \/ (max = b)) /\ max >= a /\ max >= b)`--)

(--`Partial (T /\ a > b)
     (Simple (Assign (Vref (Var "max" 0 Int))
                   (Lval (Vref (Var "a" 0 Int)))))
      (((max = a) \/ (max = b)) /\ max >= a /\ max >= b)`--)
```

SEE ALSO

IFTHEN_TAC, GENERAL_IFTHENELSE_TAC.

GENERAL_IFTHENELSE_TAC : (term -> tactic)

SYNOPSIS

A general form of IFTHENELSE_TAC.

KEYWORDS

tactic, conditional.

DESCRIPTION

This is a general form of IFTHENELSE_TAC to explicitly provide the test condition.

Subgoals: The first subgoal is the "true" or "then" branch. The second subgoal is the "false" or "else" branch.

FAILURE

```
Call (Var "strlen" 0 Int) (PL (Lval (Vref
        (Var "bs1" 0 (Array Char (CCid "BUFSIZE"))))) PLnull)

uncaught exception Fail: cexpr2hol SKIMP 4
```

You may have a side effect in the test. Use PRE_SIDE_EFFECT_TAC or SEP_CALL_TAC to separate the side effect first.

EXAMPLE

Prove the following [21, exercise 14, pp 34 & 35].

$$\frac{\vdash \{\mathtt{P} \land \mathtt{S}\}\ \mathtt{C1}\ \{\mathtt{Q}\} \qquad \vdash \{\mathtt{P} \land \mathtt{\sim S}\}\ \mathtt{C2}\ \{\mathtt{Q}\}}{\vdash \{\mathtt{P}\}\ \mathtt{if\ S\ then\ C1\ else\ if\ \sim S\ then\ C2}\ \{\mathtt{Q}\}}$$

```
1 subgoal:
  (--`Partial p (IfElse Sc C1 (If (Unary Not Sc) C2)) q`--)
  ____________________________
      (--`IS_VALUE Sc s`--)
      (--`IS_VALUE (Unary Not Sc) (~s)`--)
      (--`Partial (p /\ s) C1 q`--)
      (--`Partial (p /\ ~s) C2 q`--)
- e (GENERAL_IF_THENELSE_TAC (--`s:bool`--));
OK..
3 subgoals:
  (--`Partial (p /\ ~s) (If (Unary Not Sc) C2) q`--)
  ____________________________
      (--`IS_VALUE Sc s`--)
      (--`IS_VALUE (Unary Not Sc) (~s)`--)
      (--`Partial (p /\ s) C1 q`--)
      (--`Partial (p /\ ~s) C2 q`--)

  (--`Partial (p /\ s) C1 q`--)
  ____________________________
      (--`IS_VALUE Sc s`--)
      (--`IS_VALUE (Unary Not Sc) (~s)`--)
      (--`Partial (p /\ s) C1 q`--)
      (--`Partial (p /\ ~s) C2 q`--)

  (--`IS_VALUE Sc s`--)
```

```
  ----------------------------
      (--`IS_VALUE Sc s`--)
      (--`IS_VALUE (Unary Not Sc) (~s)`--)
      (--`Partial (p /\ s) C1 q`--)
      (--`Partial (p /\ ~s) C2 q`--)
- e (ASM_REWRITE_TAC []);
OK..
Goal proved.

Remaining subgoals:
  (--`Partial (p /\ ~s) (If (Unary Not Sc) C2) q`--)
  ----------------------------
      (--`IS_VALUE Sc s`--)
      (--`IS_VALUE (Unary Not Sc) (~s)`--)
      (--`Partial (p /\ s) C1 q`--)
      (--`Partial (p /\ ~s) C2 q`--)

  (--`Partial (p /\ s) C1 q`--)
  ----------------------------
      (--`IS_VALUE Sc s`--)
      (--`IS_VALUE (Unary Not Sc) (~s)`--)
      (--`Partial (p /\ s) C1 q`--)
      (--`Partial (p /\ ~s) C2 q`--)
- e (ASM_REWRITE_TAC []);
OK..
Goal proved.

Remaining subgoals:
  (--`Partial (p /\ ~s) (If (Unary Not Sc) C2) q`--)
  ----------------------------
      (--`IS_VALUE Sc s`--)
      (--`IS_VALUE (Unary Not Sc) (~s)`--)
      (--`Partial (p /\ s) C1 q`--)
```

```
      (--`Partial (p /\ ~s) C2 q`--)
- e (GENERAL_IFTHEN_TAC (--`~s`--));
OK..
3 subgoals:
  (--`Partial ((p /\ ~s) /\ ~s) C2 q`--)
  ____________________________
      (--`IS_VALUE Sc s`--)
      (--`IS_VALUE (Unary Not Sc) (~s)`--)
      (--`Partial (p /\ s) C1 q`--)
      (--`Partial (p /\ ~s) C2 q`--)

  (--`(p /\ ~s) /\ ~~s ==> q`--)
  ____________________________
      (--`IS_VALUE Sc s`--)
      (--`IS_VALUE (Unary Not Sc) (~s)`--)
      (--`Partial (p /\ s) C1 q`--)
      (--`Partial (p /\ ~s) C2 q`--)

  (--`IS_VALUE (Unary Not Sc) (~s)`--)
  ____________________________
      (--`IS_VALUE Sc s`--)
      (--`IS_VALUE (Unary Not Sc) (~s)`--)
      (--`Partial (p /\ s) C1 q`--)
      (--`Partial (p /\ ~s) C2 q`--)
```

USES

It is useful for proving general theorems or implementing other tactics.

SEE ALSO

IFTHENELSE_TAC.

A.5.2 One-armed Conditionals

We can conclude the partial correctness of an If-Then statement if the branch and the failed test establish the postcondition. More particularly, the following must be true.

1. If the precondition and the test holds, after the "then" code finishes, the postcondition is true.

2. If the precondition and the complement of the test holds, the postcondition is true.

3. The value of the test expression in the assertion language is "test."

$$\frac{\begin{array}{c}\texttt{IS\_VALUE expr test} \\ \vdash \{\texttt{pre} \wedge \texttt{ test}\}\ \texttt{code}\ \{\texttt{post}\} \\ \vdash \texttt{pre} \wedge \texttt{ \~{}test} \Rightarrow \texttt{post}\end{array}}{\vdash \{\texttt{pre}\}\ (\texttt{IfElse expr code})\ \{\texttt{post}\}}$$

This inference rule is proved from the If-Then-Else rule (A.2, page 171) and the semantic equivalence of If-Then and If-Then-Else with an empty "else" clause.

Tactics

IFTHEN_TAC : tactic

SYNOPSIS

Separate a conditional into subgoals for each branch.

KEYWORDS

tactic, conditional.

DESCRIPTION

Separate an `If` goal into a subgoal for the true or "then" branch and an implication for the false case. The correct logical test is generated from the code's test expression.

Subgoals: The first subgoal is the implication. The second subgoal is the true or "then" branch. This is opposite the order of subgoals in IFTHEN-ELSE_TAC. The implication (when the test is false) comes first since it is typically easier to prove than the true branch.

FAILURE

```
Call (Var "strlen" 0 Int) (PL (Lval (Vref
         (Var "bs1" 0 (Array Char (CCid "BUFSIZE"))))) PLnull)


uncaught exception Fail: cexpr2hol SKIMP 4
```

You may have a side effect in the test. Use PRE_SIDE_EFFECT_TAC or SEP_CALL_TAC to separate the side effect first.

EXAMPLE

Prove the following finds the maximum of two variable: `max = a; if (max < b) max = b;`. The implication can be solved by ARITH_TAC, and the "then" branch can be solved by ASSIGN_TAC.

```
1 subgoal:
  (--`Partial (max = a)
        (If
           (Binary (Lval (Vref (Var "max" 0 Int))) Lt
             (Lval (Vref (Var "b" 0 Int))))
          (Simple (Assign (Vref (Var "max" 0 Int))
                        (Lval (Vref (Var "b" 0 Int))))))
        (((max = a) \/ (max = b)) /\ max >= a /\ max >= b)`--)
- e (IFTHEN_TAC);
OK..
2 subgoals:
  (--`Partial ((max = a) /\ max < b)
        (Simple (Assign (Vref (Var "max" 0 Int))
                      (Lval (Vref (Var "b" 0 Int)))))
```

```
(((max = a) \/ (max = b)) /\ max >= a /\ max >= b)'--)

(--'(max = a) /\ ~(max < b) ==>
    ((max = a) \/ (max = b)) /\ max >= a /\ max >= b'--)
```

SEE ALSO

IFTHENELSE_TAC, GENERAL_IFTHEN_TAC.

GENERAL_IFTHEN_TAC test : tactic

SYNOPSIS

A general form of IFTHEN_TAC.

KEYWORDS

tactic, conditional.

DESCRIPTION

This is a general form of IFTHEN_TAC to explicitly provide the test condition.

Subgoals: The first subgoal is the implication. The second subgoal is the true or "then" branch. This is opposite the order of subgoals in GENERAL_IFTHENELSE_TAC. The implication (when the test is false) comes first since it is typically easier to prove than the true branch.

FAILURE

```
Call (Var "strlen" 0 Int) (PL (Lval (Vref
        (Var "bs1" 0 (Array Char (CCid "BUFSIZE"))))) PLnull)

uncaught exception Fail: cexpr2hol SKIMP 4
```

You may have a side effect in the test. Use PRE_SIDE_EFFECT_TAC or SEP_CALL_TAC to separate the side effect first.

EXAMPLE

See GENERAL_IFTHENELSE_TAC for a similar example.

USES

It is useful for proving general theorems or implementing other tactics.

SEE ALSO

`IFTHEN_TAC`, `GENERAL_IFTHENELSE_TAC`.

A.5.3 While Loops

This inference rule handles `while` loops which may have preevaluation or postevaluation side effects in the test. To handle side effects, we separate them from the test itself. The diagram in Figure 5.4, page 36, shows the flow of control and the conditions on each arc. The statement `preStm` has all the preevaluation side effects of the test, and the statement `postStm` has all the postevaluation side effects. If there are no preevaluation or no postevaluation side effects, `preStm` or `postStm` may be the empty statement. The `IS_VALUE` predicate does not allow side effects, so enforces that the test expression has no side effects. Notice that since the postevaluation side effects take place after the test is evaluated, `postStm` is found on both the "false" and "true" cases.

$$\frac{\begin{array}{c} \vdash (\mathtt{preStm} = \mathtt{EmptyStmt}) \lor \\ \qquad (\mathtt{preStm} = (\mathtt{Simp\ preSeEx}) \land\ \mathtt{NoPostSE\ preSeEx}) \\ \vdash (\mathtt{postStm} = \mathtt{EmptyStmt}) \lor \\ \qquad (\mathtt{postStm} = (\mathtt{Simp\ postSeEx}) \land\ \mathtt{NoPreSE\ postSeEx}) \\ \vdash \mathtt{SEM\_EQ\ (Seq\ preStm\ (Seq\ (Simp\ testEx)\ postStm))\ (Simp\ ex)} \\ \vdash \mathtt{IS\_VALUE\ testEx\ test} \\ \vdash \{\mathtt{invariant}\}\ \mathtt{preStm}\ \{\mathtt{testState}\} \\ \vdash \{\mathtt{testState}\ \land\ \mathtt{test}\}\ \mathtt{postStm}\ \{\mathtt{bodyCond}\} \\ \vdash \{\mathtt{bodyCond}\}\ \mathtt{body}\ \{\mathtt{invariant}\} \\ \vdash \{\mathtt{testState}\ \land\ \mathtt{\sim test}\}\ \mathtt{postStm}\ \{\mathtt{postCond}\} \\ \vdash \mathtt{postCond} \Rightarrow \mathtt{post} \end{array}}{\vdash \{\mathtt{invariant}\}\ (\mathtt{CWhile\ ex\ body})\ \{\mathtt{post}\}}$$

This inference rule is slightly stronger then inference rule (5.8) given in Sect 5.5, page 36. This rule has postcondition weakening built-in (`postCond` $\Rightarrow$ `post`). If the computed postcondition is not the same as the postcondition in the subgoal, the built-in weakening lets the tactic leave the implication as a subgoal rather than just failing.

This more-useful rule is proved from the basic inference rule and postcondition weakening.

Tactics

`WHILE_TAC : tactic`

SYNOPSIS

Separate a `while` loop without side effects into subgoals.

KEYWORDS

tactic, while, loop.

DESCRIPTION

This can only be used if the test expression has no side effects. If there are side effects, use GENERAL_WHILE_TAC instead. This tactic breaks a `while` loop into an implication of the exit condition and partial correctness that the body reestablishes the invariant.

Subgoals: There is always one to prove the body reestablishes the invariant. There may also be one to prove the exit condition implies the postcondition.

FAILURE

```
preeval side effects: LUnary PreInc (Vref (Var "a" 0 Int))
```

The loop test has a preevaluation side effect. Use GENERAL_WHILE_TAC.

```
posteval side effects: Binary (LUnary PostInc(Vref(Var "a" 0 Int)))
  Add (LUnary PostInc (Vref (Var "b" 0 Int))
```

The loop test has a postevaluation side effect. Use GENERAL_WHILE_TAC.

EXAMPLE

Prove the following loop. It is part of a proof of a division-by-repeated-subtraction algorithm.

$$\vdash \{\mathtt{r} = \mathtt{x} \wedge \mathtt{q} = 0\}\ \mathtt{while}\,(\mathtt{y} <= \mathtt{r})\ \{\mathtt{r} = \mathtt{r} - \mathtt{y};\ \mathtt{q} = \mathtt{q} + 1;\}$$
$$\{\mathtt{r} < \mathtt{y} \wedge \mathtt{x} = \mathtt{r} + (\mathtt{y} * \mathtt{q})\}$$

```
Remaining subgoals:
(--`Partial (x = r + y * q)
      (CWhile (Binary (Lval (Vref (Var "y" 0 Int))) LEq
                      (Lval (Vref (Var "r" 0 Int))))
        (Seq
          (Simple (Assign (Vref (Var "r" 0 Int))
                    (Binary (Lval (Vref (Var "r" 0 Int)))
                      Sub (Lval (Vref (Var "y" 0 Int))))))
          (Simple (Assign (Vref (Var "q" 0 Int))
                    (Binary (Lval (Vref (Var "q" 0 Int)))
                      Add (Const (CCint 1) Int))))))
        (r < y /\ (x = r + y * q))`--)
- e (WHILE_TAC);
OK..
2 subgoals:
(--`Partial ((x = r + y * q) /\ y <= r)
     (Seq
       (Simple (Assign (Vref (Var "r" 0 Int))
                    (Binary (Lval (Vref (Var "r" 0 Int)))
                      Sub (Lval (Vref (Var "y" 0 Int))))))
       (Simple (Assign (Vref (Var "q" 0 Int))
                    (Binary (Lval (Vref (Var "q" 0 Int)))
                      Add (Const (CCint 1) Int))))))
        (x = r + y * q)`--)

  (--`(x = r + y * q) /\ ~(y <= r) ==>
                           r < y /\ (x = r + y * q)`--)
```

SEE ALSO

GENERAL_WHILE_TAC, PRE_STRENGTHEN_TAC.

GENERAL_WHILE_TAC:(term option -> term option -> tactic)

SYNOPSIS

Separate a `while` loop into subgoals for each piece.

KEYWORDS

tactic, while, loop.

DESCRIPTION

Given the precondition as the loop invariant, reduce a `while` loop to a minimal set of subgoals needed to prove it.

The first "optional" operand is a test state condition. The second "optional" operand is the posttest condition. If the test expression has preevaluation side effects, a test state must be provided, for example, `(SOME (--‘(b=a) / a < C+1 / (c=C)‘--))`. If the test has no pre side effects, pass `NONE`. Likewise if the test has postevaluation side effects, a body condition must be given, for example, `(SOME (--‘(b=a) / a < C / (c=C)‘--))`, otherwise pass `NONE`.

Note that the existing precondition is used as the loop invariant. A PRE_STRENGTHEN_TAC may be needed to to get the precondition in the right form. See the example below.

Subgoals: This leaves from one to four subgoals depending on what, if any, side effects are in the loop test expression and whether the exit condition matches the postcondition. The possible goals are

1. partial correctness of preevaluation effects,
2. the exit condition implies the postcondition,
3. partial correctness of postevaluation effects giving the body precondition, and
4. partial correctness that the body reestablishes the invariant.

FAILURE

```
test has preeval side effects, so a
test state condition is required
```

The test expression has preevaluation side effects, but NONE was passed as the test state (first operand). You must give a condition to be used after the preevaluation side effects are applied and before the test is done.

```
test has posteval side effects, so a
body state condition is required
```

The test expression has postevaluation side effects, but NONE was passed as the body condition (second operand). You must give a condition to be used after the postevaluation side effects are applied and before the body is evaluated.

EXAMPLE

Prove a loop which has both pre- and postevaluation side effects in the test. It copies c to a and b. The original code and conditions are these.

```
{c=^c /\ ^c>0}
    a=0;
    b=0;
    while(++a + b++ + 1 < 2 *c)
        ;
{a=^c /\ b=^c}
```

The first two statements can be proved with REPEAT FWD_SEQ_ASSIGN_TAC.

```
1 subgoal:
(--`Partial ((b = a) /\ a < ^c /\ (c = ^c))
    (CWhile
       (Binary
          (Binary
             (Binary (LUnary PreInc (Vref (Var "a" 0 Int)))
                Add
```

```
                    (LUnary PostInc (Vref (Var "b" 0 Int))))
          Add (Const (CCint 1) Int))
       Lt
         (Binary (Const (CCint 2) Int)
               Mul (Lval (Vref (Var "c" 0 Int)))))
     EmptyStmt)
  ((a = ^c) /\ (b = ^c))`--)
- e (GENERAL_WHILE_TAC
           (SOME (--`(b+1=a) /\ a<^logc+1 /\ (c=^logc)`--))
           (SOME (--`(b = a) /\ a < ^logc /\ (c=^logc)`--)));
4 subgoals:
  (--`Partial ((b = a) /\ a < ^c /\ (c = ^c)) EmptyStmt
        ((b = a) /\ a < ^c /\ (c = ^c))`--)

  (--`Partial (((b + 1 = a) /\ a < ^c + 1 /\ (c = ^c)) /\
                                        (a + b) + 1 < 2 * c)
        (Simple (LUnary PostInc (Vref (Var "b" 0 Int))))
        ((b = a) /\ a < ^c /\ (c = ^c))`--)

  (--`Partial
        (((b + 1 = a) /\ a < ^c + 1 /\ (c = ^c)) /\
                                        ~((a + b) + 1 < 2 * c))
        (Simple (LUnary PostInc (Vref (Var "b" 0 Int))))
        ((a = ^c) /\ (b = ^c))`--)

  (--`Partial ((b = a) /\ a < ^c /\ (c = ^c))
        (Simple (LUnary PreInc (Vref (Var "a" 0 Int))))
        ((b + 1 = a) /\ a < ^c + 1 /\ (c = ^c))`--)
```

As a side note, the first three goals are proved with LUNARY_TAC, and the final goal is proved with SKIP_TAC.

USES

If you are not sure what conditions to use, try the underspecified `GENERAL_WHILE_TAC (SOME (--‘testState:bool‘--)) (SOME (--‘bodyCond:bool‘--))`. This will show the subgoals which will result. You can examine where `testState` and `bodyCond` appear to decide what they should be, then backup the proof and insert the actual conditions.

SEE ALSO

`PRE_STRENGTHEN_TAC`, `WHILE_TAC`.

A.6 Blocks and Functions

A.6.1 Blocks and Local Variables

Blocks define scope. In C they may also introduce local variables. Since we had no code with local variables (except function bodies), we use this simplistic (and wrong!) inference rule even though it does not handle local variables. A correct rule can be found in [21].

$$\frac{\vdash \{\mathtt{pre}\}\ \mathtt{stm}\ \{\mathtt{post}\}}{\vdash \{\mathtt{pre}\}\ (\mathtt{Block\ llv\ stm})\ \{\mathtt{post}\}}$$

Tactics

`BLOCK_TAC : tactic`

SYNOPSIS

Reduce a `Block` goal.

KEYWORDS

tactic, block, local variable.

DESCRIPTION

Strip the `Block` construct ignoring any local variables.

Subgoals: The body of the block.

FAILURE

Always succeeds if the statement is a `Block`.

EXAMPLE

```
1 subgoal:
  (--`Partial ((p = ^p) /\ (g = ^g))
        (Block []
          (Simple
            (Assign (Vref (Var "g" 0 Int))
              (Binary (Lval (Vref (Var "p" 0 Int))) Add
                (Const (CCint 1) Int)))))
        (g = ^p + 1)`--)
- e (BLOCK_TAC);
OK..
1 subgoal:
  (--`Partial ((p = ^p) /\ (g = ^g))
        (Simple
          (Assign (Vref (Var "g" 0 Int))
            (Binary (Lval (Vref (Var "p" 0 Int))) Add
              (Const (CCint 1) Int))))
        (g = ^p + 1)`--)
```

SEE ALSO

`CALL_TAC`, `IFTHEN_TAC`, `WHILE_TAC`.

A.6.2 Function Calls

Functions or procedures are a powerful means of structuring code. For reasonable verification, we must have a means of proving the correctness of a function call based on some kind of pre-proved theorem for the function being called. The theorem must include necessary preconditions and guaranteed postconditions in terms of the formal parameters and global variables.

The inference rule for a function call must do some kind of renaming, similar to assignment, since formal parameters are bound to values. However function calls are semantically complex: parameters may be bound to arbitrary expressions, and the function may access or change the global state. Function calls rules have been proposed, but many are too limited or have later been shown to be incorrect. We adapt our function call inference rule from Homeier [32, p. 107].

We specialized the rule to remove provisions for call-by-name parameters and to specialize for functions with variable numbers of arguments (`varargs`).

$$\frac{\begin{array}{c}\vdash \text{WF\_fnp } \{\text{pre}\} \text{ (Func name formals globls b) } \{\text{post}\} \\ \vdash \text{WF\_c (Simple (Call name ps))} \quad \vdash \text{vals} = \text{spec\_varargs formals ps} \\ \vdash \text{vals'} = \text{variants vals (APPEND (FV\_a qpost) globls)} \\ \vdash \text{y} = \text{APPEND vals globls} \quad \vdash \text{x} = \text{APPEND vals globls} \\ \vdash \text{x0} = \text{logicals x} \qquad \vdash \text{y0} = \text{logicals y} \\ \vdash \text{x0'} = \text{variants x0 (FV\_a qpost)} \\ \vdash \text{specpost} = \text{post} \lhd \text{(APPEND vals' x0') (APPEND vals y0)} \\ \vdash \text{postimpq} = (\text{specpost} \Rightarrow \text{qpost}) \lhd \text{newC\_Result genrC\_Result}\end{array}}{\begin{array}{l}\vdash \{(\mathtt{pre}^{\mathtt{vals'}}_{\mathtt{vals}} \ \wedge\ (\forall \mathtt{x}.\mathtt{postimpq})^{\mathtt{x}}_{\mathtt{x0'}})^{\mathtt{vals'}}_{\mathtt{PL2CELps}}\} \\ \qquad\qquad\qquad\qquad \text{(Simple (Call name ps)) } \{\text{qpost}\}\end{array}} \tag{A.3}$$

The operator $\lhd$ substitutes items from the first list with items from the second list. The $\forall$`x`.... binds all the variables in `x` to prevent variable capture.

Tactics

`CALL_TAC : (thm -> tactic)`

SYNOPSIS

Prove a function call from a $\mathtt{WF}_{\mathtt{fnp}}$ theorem.

KEYWORDS

tactic, function, call.

DESCRIPTION

Prove a function call given a well-formedness theorem for the function. This tactic strengthens the precondition to match the computed precondition.

Subgoals: This leaves an implication from the strengthening.

FAILURE

```
Exception raised at partialTactics.GENERAL_CALL_TAC:
Call var and WF_fnp Func var differ
```

The function being called and the well-formedness theorem function are different. The types may be different.

```
WF_fnp post has multiple C_Result's
```

The tactic doesn't know which `C_Result` variable to specialize as the function's return value.

EXAMPLE

Here is a well-formedness theorem for a function which sets a global variable, g, to the parameter, plus one.

```
|- WF_fnp T
   (Func (Var "f" 0 Void) [Var "p" 0 Int] [Var "g" 0 Int]
     (SOMEBody (Block []
       (Simple
         (Assign (Vref (Var "g" 0 Int))
           (Binary (Lval (Vref (Var "p" 0 Int))) Add
             (Const (CCint 1) Int)))))))
   (g = ^p + 1)
```

The following proves that after the call, g is one more than the value of the actual parameter, or $\vdash \{\mathtt{T}\}\ \mathtt{f(k);}\ \ \{\mathtt{g} = \mathtt{k} + 1\}$.

```
1 subgoal:
 (--`Partial T
       (Simple
         (Call (Var "f" 0 Void)
           (PL (Lval (Vref (Var "k" 0 Int))) PLnull)))
       (g = k + 1)`--)

- e (CALL_TAC WF_fnp_f);

1 subgoal:
  (--`T ==> T /\ (!p'1' g. (g = p + 1) ==> (g = p + 1))`--)
```

This can be proved with ASM_REWRITE_TAC.

SEE ALSO

GENERAL_CALL_TAC, SEP_CALL_TAC, WF_fnp_TAC.

GENERAL_CALL_TAC : (thm -> string -> tactic)

SYNOPSIS

A general form of CALL_TAC.

KEYWORDS

tactic, function, call.

DESCRIPTION

This allows one to give the new name of the function result variable.

Subgoals: This tactic leaves an implication from the precondition strengthening.

FAILURE

See CALL_TAC.

EXAMPLE

See CALL_TAC.

USES

This is only used in writing other tactics.

SEE ALSO

`CALL_TAC`, `SEP_CALL_TAC`, `WF_fnp_TAC`.

`SEP_CALL_TAC : (term -> thm -> tactic)`

SYNOPSIS

Separate and prove a function call.

KEYWORDS

tactic, call, function, side effect.

DESCRIPTION

Separate a function call as a preevaluation side effect and prove the call with the $\mathtt{WF}_{\mathtt{fnp}}$ theorem. The term operand is the intermediate condition after the function call is evaluated and before the simplified expression.

Subgoals: The first subgoal is an implication of the weakest precondition by the original precondition. The second subgoal is the expression, with the call replaced by a new variable, is left.

FAILURE

See PRE_SIDE_EFFECT_TAC, CALL_TAC, and SEQUENCE_TAC.

EXAMPLE

The function `f()` sets the global variable gv to six and returns four. Here is its well-formedness axiom.

```
|- WF_fnp T (Func (Var "f" 0 Int) [] [Var "gv" 0 Int] NOBody)
     ((C_Result = 4) /\ (gv = 6))
```

We use this to prove $\vdash \{\mathrm{T}\}$ $\mathtt{a} = \mathtt{f}();$ $\{(\mathtt{a} = 4) \wedge (\mathtt{gv} = 6)\}$.

```
a subgoal:
(--`Partial T
     (Simple
       (Assign (Vref (Var "a" 0 Int))
                     (Call (Var "f" 0 Int) PLnull)))
     ((a = 4) /\ (gv = 6))`--)

- e (SEP_CALL_TAC (--`(C_Result=4) /\ (gv=6)`--) f_thm);

2 subgoals:
  (--`Partial ((C_Result11 = 4) /\ (gv = 6))
        (Simple
          (Assign (Vref (Var "a" 0 Int))
            (Lval (Vref (Var "C_Result11" 0 Int)))))
        ((a = 4) /\ (gv = 6))`--)

  (--`T ==>
      T /\
      (!gv.
        (C_Result11 = 4) /\ (gv = 6) ==>
        (C_Result11 = 4) /\ (gv = 6))`--)
```

These can be solved with STRIP_THEN_REWRITE_TAC and ASSIGN_TAC respectively. There is another, similar example in WF_fnp_TAC.

SEE ALSO

CALL_TAC, PRE_SIDE_EFFECT_TAC, SEQUENCE_TAC, WF_fnp_TAC.

A.6.3 Verifying C Functions

The highest level of C code which we can prove at once is the function. We have no rules for reasoning about files, which may have static variables and functions. The aim at this level is to specify necessary preconditions, a function declaration with formal parameters and the body, guaranteed postconditions, and prove a theorem about calls to that function.

The first step is to translate the source code into an abstract syntax tree. See Sect. A.1 for details. Add a list of global variables accessed, that is, either used or potentially changed. Choose pre- and postconditions, and set a $\mathtt{WF}_{\mathtt{fnp}}$ goal. First use WF_fnp_TAC to split the goal into a proof of the syntactic correctness of the declaration and the proof of the body, then use WF_fn_syn_TAC to prove syntactic correctness. The proof of the body uses the preceding tactics to prove code correctness.

A.6.4 Function Correctness

We prove function correctness by proving that the function is declared properly, or has syntax correctness, and that the body code has partial correctness. We use Homeier's definition of function correctness: $\mathtt{WF}_{\mathtt{fnp}}$. The subscript "fnp" means "function, partial correctness." The operator $\vec{=}$ expresses the initial values of the formal parameters and global variables, or the conjunction of pairwise equivalence of two lists. The term $x \vec{=} y$ means $x_1 = y_1 \wedge x_2 = y_2 \wedge \ldots x_n = y_n$. Here is the inference rule to establish the partial correctness of a function declaration from syntactic correctness and partial correctness of the body (with globals and formals "initialized").

$$\frac{\begin{array}{c}\vdash \mathrm{x} = \mathrm{APPEND\ formals\ globls} \qquad\qquad \vdash \mathrm{x0} = \mathrm{logicals\ x} \\ \vdash \mathtt{WF}_{\mathtt{fn\_syntax}}\ \mathrm{pre\ (Func\ name\ formals\ globls\ (SOMEBody\ body))\ post} \\ \vdash \{(\mathtt{x}\ \vec{=}\ \mathtt{x0})\ \wedge\ \mathtt{pre}\}\ \mathrm{body}\ \{\mathrm{post}\}\end{array}}{\vdash \mathtt{WF}_{\mathtt{fnp}}\ \mathrm{pre\ (Func\ name\ formals\ globls\ (SOMEBody\ body))\ post}}$$

Tactic

WF_fnp_TAC : tactic

SYNOPSIS

Reduce the goal of function partial correctness.

KEYWORDS

tactic, function.

DESCRIPTION

Prove the correctness of a function call from syntactic correctness, or well-formedness, of the declaration and from partial correctness of the body. The body partial correctness has formal parameters and global variables equated to logical variables representing initial values.

See Sect. 6.4.3, page 79, "A Guidebook to Formalizing System and Library Calls," for details on specifying $\mathtt{WF}_{\mathtt{fnp}}$ theorems.

Subgoals: The first subgoal is syntac correctness or $\mathtt{WF}_{\mathtt{fn\_syntax}}$ since it is easiest to prove. The second subgoal is the partial correctness of the body.

FAILURE

Always succeeds if the goal is WF_fnp.

EXAMPLE

This is an extensive example of the complete proof of a simple function to illustrate the different aspects of WF_fnp_TAC as well as WF_fn_syn_TAC.

We begin with a function which will be called. The function, `const1()`, always returns 1 and sets a global variable, `a_global`, to 6. Here is its partial correctness axiom.

```
|- WF_fnp T
    (Func (Var "const1" 0 Int) [] [Var "a_global" 0 Int] NOBody)
       ((C_Result = 1) /\ (a_global = 6))
```

We will prove the following goal: a function, f(), sets another global variable, b_global to its parameter, p, plus the value of const1().

$$\vdash \mathtt{WF}_{\mathtt{fnp}}\ \mathtt{T}\ \mathtt{(f(p)\{b\_global=p+const1();\}}\ \mathtt{(b\_global=\hat{~}p+1)}$$

We begin with the above goal, then reduce function partial correctness (WF_fnp_TAC), and prove syntactic correctness (WF_fn_syn_TAC). Notice that since const1() references a_global and f() calls const1(), f() must list a_global, too, even though it is never explicitly referenced. Notice also in the subgoal to prove correctness of the body that the formals and globals are bound to logical variables representing initial values. Since the proof of syntactic correctness may take several seconds and to help the user diagnosis proof errors, WF_fn_syn_TAC reports as it works on each of the major hypotheses.

```
Initial goal:
 (--`WF_fnp T
    (Func (Var "f" 0 Void) [Var "p" 0 Int]
       [Var "b_global" 0 Int; Var "a_global" 0 Int]
      (SOMEBody (Block []
        (Simple (Assign (Vref (Var "b_global" 0 Int))
          (Binary (Lval (Vref (Var "p" 0 Int))) Add
                  (Call (Var "const1" 0 Int) PLnull)))))))
   (b_global = ^p + 1)`--)

- e(WF_fnp_TAC);

2 subgoals:
(--`Partial ((p = ^p) /\ (b_global = ^b_global) /\
             (a_global = ^a_global) /\ T)
    (Block []
      (Simple (Assign (Vref (Var "b_global" 0 Int))
        (Binary (Lval (Vref (Var "p" 0 Int))) Add
                (Call (Var "const1" 0 Int) PLnull)))))
    (b_global = ^p + 1)`--)
```

```
(--`WF_fn_syntax T
    (Func (Var "f" 0 Void) [Var "p" 0 Int]
       [Var "b_global" 0 Int; Var "a_global" 0 Int]
      (SOMEBody (Block []
        (Simple (Assign (Vref (Var "b_global" 0 Int))
          (Binary (Lval (Vref (Var "p" 0 Int))) Add
                  (Call (Var "const1" 0 Int) PLnull)))))))
    (b_global = ^p + 1)`--)

- e (WF_fn_syn_TAC [WF_fnp_const1]);

(trying to prove IS_SUBSET (GV_c body) globls ...)
(trying to prove IS_SUBSET (FV_c body) x ...)
(trying to prove IS_SUBSET_TYCONF (FV_a pre) x ...)
(trying to prove IS_SUBSET_TYCONF (FV_a post) (APPEND x x0) ...)
Goal proved.
|- WF_fn_syntax T
     (Func (Var "f" 0 Void) [Var "p" 0 Int]
        [Var "b_global" 0 Int; Var "a_global" 0 Int]
       (SOMEBody
         (Block []
           (Simple
             (Assign (Vref (Var "b_global" 0 Int))
               (Binary (Lval (Vref (Var "p" 0 Int))) Add
                 (Call (Var "const1" 0 Int) PLnull)))))))
     (b_global = ^p + 1)

Remaining subgoals:
(--`Partial
    ((p = ^p) /\ (b_global = ^b_global) /\
     (a_global = ^a_global) /\ T)
    (Block []
```

```
    (Simple (Assign (Vref (Var "b_global" 0 Int))
      (Binary (Lval (Vref (Var "p" 0 Int))) Add
              (Call (Var "const1" 0 Int) PLnull)))))
  (b_global = ^p + 1)`--)
```

We now use BLOCK_TAC to reduce the block and SEP_CALL_TAC to separate the call and introduce an intermediate condition. Notice that we only need to "carry" the initial value of one variable, the parameter p and the result of the function call. SEP_CALL_TAC leaves an implication from the original precondition to the precondition computed from the function call. STRIP_THEN_REWRITE_TAC proves the implication, and ASSIGN_TAC finishes the proof.

```
- e (BLOCK_TAC);

1 subgoal:
  (--`Partial
    ((p = ^p) /\ (b_global = ^b_global) /\
     (a_global = ^a_global) /\ T)
    (Simple (Assign (Vref (Var "b_global" 0 Int))
        (Binary (Lval (Vref (Var "p" 0 Int))) Add
                (Call (Var "const1" 0 Int) PLnull))))
    (b_global = ^p + 1)`--)

- e(SEP_CALL_TAC (--`(C_Result = 1) /\ (p = ^p)`--) WF_fnp_const1);

2 subgoals:
  (--`Partial ((C_Result1 = 1) /\ (p = ^p))
        (Simple
          (Assign (Vref (Var "b_global" 0 Int))
            (Binary (Lval (Vref (Var "p" 0 Int))) Add
              (Lval (Vref (Var "C_Result1" 0 Int))))))
        (b_global = ^p + 1)`--)

  (--`(p = ^p) /\ (b_global = ^b_global) /\
```

```
                      (a_global = ^a_global) /\ T ==>
      T /\ (!a_global.
        (C_Result2 = 1) /\ (a_global = 6) ==>
                      (C_Result2 = 1) /\ (p = ^p))`--)

- e (STRIP_THEN_REWRITE_TAC);

Goal proved.
  (--`(p = ^p) /\ (b_global = ^b_global) /\
                      (a_global = ^a_global) /\ T ==>
      T /\ (!a_global.
        (C_Result2 = 1) /\ (a_global = 6) ==>
                      (C_Result2 = 1) /\ (p = ^p))`--)

Remaining subgoals:
  (--`Partial ((C_Result1 = 1) /\ (p = ^p))
        (Simple
          (Assign (Vref (Var "b_global" 0 Int))
            (Binary (Lval (Vref (Var "p" 0 Int))) Add
              (Lval (Vref (Var "C_Result1" 0 Int))))))
        (b_global = ^p + 1)`--)

- e (ASSIGN_TAC);

  Initial goal proved.
|- WF_fnp T
   (Func (Var "f" 0 Void) [Var "p" 0 Int]
      [Var "b_global" 0 Int; Var "a_global" 0 Int]
     (SOMEBody (Block []
       (Simple (Assign (Vref (Var "b_global" 0 Int))
         (Binary (Lval (Vref (Var "p" 0 Int))) Add
                 (Call (Var "const1" 0 Int) PLnull)))))))
   (b_global = ^p + 1)
```

While preparing this example, I initially forgot to carry through the initial value of p. After ASSIGN_TAC left an implication which needed it, I backed up, expanded the intermediate condition in the SEP_CALL_TAC and finished the proof.

SEE ALSO

`CALL_TAC, SEP_CALL_TAC, WF_fn_syn_TAC.`

A.6.5 Function Syntactic Correctness

To prove function correctness, we must check several syntactic correctness or well-formedness conditions. We adapt Homeier's WF_proc_syntax [32, p. 259] for our well-formedness condition: $\mathtt{WF}_{\mathtt{fn\_syntax}}$. The subscript "fn_syntax" means "function, syntactic correctness."

Here are the conditions. The operator $\overset{\text{type}}{\subset}$ means "subset, allowing for type conformance." We approximate C's implicit type casting and automatic type promotion by allowing matching as long as types conform.

1. None of the formal parameters or global variables are logicals ($\mathtt{WF}_{\mathtt{xs}}\ \mathtt{x}$).
2. No variable occurs more than once in the formals or globals ($\mathtt{DL\ x}$).
3. All globals of functions called are listed in the globals $(\mathtt{GV}_{\mathtt{c}}\ \mathtt{body}) \subset \mathtt{globls}$.
4. All free variables (i.e., not local variables) in the body are formals or globals $(\mathtt{FV}_{\mathtt{c}}\ \mathtt{body}) \subset \mathtt{x}$.
5. All free variables in the precondition are formals or globals, given type conformance $(\mathtt{FV}_{\mathtt{a}}\ \mathtt{pre}) \overset{\text{type}}{\subset} \mathtt{x}$.
6. All free variables in the postcondition are formals or globals or logical versions of them (the x0), given type conformance $(\mathtt{FV}_{\mathtt{a}}\ \mathtt{post}) \overset{\text{type}}{\subset} (\mathtt{APPEND\ x\ x0})$.

$$\frac{\begin{array}{c}\vdash \mathrm{x} = \mathrm{APPEND\ formals\ globls} \qquad \vdash \mathrm{x0} = \mathrm{logicals\ x} \\ \vdash \mathtt{WF}_{\mathtt{xs}}\ \mathtt{x} \qquad \vdash \mathrm{DL\ x} \\ \vdash (\mathtt{GV}_{\mathtt{c}}\ \mathtt{body}) \subset \mathtt{globls} \qquad \vdash (\mathtt{FV}_{\mathtt{c}}\ \mathtt{body}) \subset \mathrm{x} \\ \vdash (\mathtt{FV}_{\mathtt{a}}\ \mathtt{pre}) \overset{\text{type}}{\subset} \mathtt{x} \qquad \vdash (\mathtt{FV}_{\mathtt{a}}\ \mathtt{post}) \overset{\text{type}}{\subset} (\mathtt{APPEND\ x\ x0})\end{array}}{\vdash \mathtt{WF}_{\mathtt{fn\_syntax}}\ \mathrm{pre\ (Func\ name\ formals\ globls\ (SOMEBody\ body))\ post}}$$

WF_fn_syn_TAC : (thm list -> tactic)

SYNOPSIS

Prove the syntactic correctness of a function.

KEYWORDS

tactic, function.

DESCRIPTION

Prove various "lemmas" about the syntactic correctness of a function, such as, all globals are declared and parameters have unique names. The operand is a list of $\mathtt{WF}_{\mathtt{fnp}}$ theorems, one for each function called in the body. With these lemmas, the goal is proved.

See Sect. 6.4.3, page 79, "A Guidebook to Formalizing System and Library Calls," for details on specifying $\mathtt{WF}_{\mathtt{fnp}}$ theorems.

Subgoals: None.

FAILURE

When WF_fn_syn_TAC fails, it prints a lot of text about what it is working on and how far it could get to help figure out what the problem is and how to correct it. These error messages leave out extraneous parts.

```
(trying to prove IS_SUBSET (GV_c body) globls ...)
No WF_fnp theorem for const1
Could not prove IS_SUBSET (GV_c body) globls
```

A function is called, in this case `const1()`, but no matching $\mathtt{WF}_{\mathtt{fnp}}$ theorem is given. Either the theorem was missed or the return value of the theorem function is different from the return value declared in the body. Supply the missing theorem or correct the declaration.

```
(trying to prove IS_SUBSET (GV_c body) globls ...)
Could not prove IS_SUBSET (GV_c body) globls
>>> These are missing or don't have exactly the same type
[Var "a_global" 0 Int]
```

A called function references a global variable, in this case `a_global`, but it is not listed in the globals for this function. Either it is missing or the type is different. Add the missing variable or correct the type declaration.

```
(trying to prove IS_SUBSET (GV_c body) globls ...)
(trying to prove IS_SUBSET (FV_c body) x ...)
Could not prove IS_SUBSET (FV_c body) x
>>> These are missing or don't have exactly the same type
[Var "free" 0 Int]
```

A free variable in the body, named `free` in this instance, is not declared as a global or a parameter. If it is a global, add it to the list of referenced globals or correct the type. If it is a parameter, correct the name or type.

```
(trying to prove IS_SUBSET (GV_c body) globls ...)
(trying to prove IS_SUBSET (FV_c body) x ...)
(trying to prove IS_SUBSET_TYCONF (FV_a pre) x ...)
Could not prove IS_SUBSET_TYCONF (FV_a pre) x
>>> These are missing or don't have exactly the same type
[Var "free" 0 Int]
```

A free variable in the precondition, named `free` here, is not declared as a parameter or a global. Add it to the globals, correct the name or type, or make it a program variable (no leading caret) instead of a logical variable. Logical variables are not allowed in the precondition.

```
(trying to prove IS_SUBSET (GV_c body) globls ...)
(trying to prove IS_SUBSET (FV_c body) x ...)
(trying to prove IS_SUBSET_TYCONF (FV_a pre) x ...)
(trying to prove IS_SUBSET_TYCONF (FV_a post) (APPEND x x0) ...)
```

```
Could not prove IS_SUBSET_TYCONF (FV_a post) (APPEND x x0)
>>> These are missing or don't have exactly the same type
[Var "free" 0 Int]
```

A free variable in the postcondition, named `free` here, is not declared as a parameter or a global. Add it to the globals or correct the name or type. Logical versions of parameters or globals, representing their initial values, are allowed in the postcondition.

EXAMPLE

See the example for WF_fnp_TAC.

SEE ALSO

`WF_fnp_TAC`.

A.7 Generally Useful Tactics

This section documents some general tactics which may be broadly useful. They are divided into three groups in corresponding sections: Sect. A.7.1 tactics to simplify or solve goals, Sect. A.7.2 tactics to undischarge assumptions, and Sec. A.7.3 tactics for arithmetic. Any or all of these may be subsumed by the improved tactics in newer HOL's such as MESON_TAC.

A.7.1 Tactics to Simplify or Solve Goals

Proofs in axiomatic semantics tend to carry a conjunction of a lot of conditions. Inference rules often involve one extended condition implying another where most of the conditions can be trivially satisfied. So there are often goals which look something like $a \wedge b \wedge c \wedge D \Rightarrow a \wedge b \wedge c \wedge E$.

`STRIP_THEN_REWRITE_TAC : tactic`

SYNOPSIS

Strip quantifiers and implications, then rewrite with assumptions.

KEYWORDS

tactic, strip, rewrite.

DESCRIPTION

This does REPEAT STRIP_TAC THEN ASM_REWRITE_TAC [] which usually greatly simplifies the goal and sometimes solves it altogether. Even when preliminary tactics should be executed before it, say expanding a definition, it can be a useful diagnostic aid by showing what needs to be proved and under what conditions or assumptions.

Subgoals: May solve the goal or leave any number of subgoals.

FAILURE

The tactic never fails. Since the tactic uses general rewriting, it may cause an infinite loop, but this has never happened in practice.

EXAMPLE

The following goal is the implication which establishes the precondition of a call to `printf()`.

```
1 subgoal:
  (--`!prev.
        nonConfidentialS (printfSpec "%s %s %s " vargs) ==>
        ((inode = SYS_stdout) ==>
         nonConfidential (getFile SYS_FileSystem inode)) ==>
        C_Result1 > 0 ==>
        (~(inode = SYS_stdout) \/
         ((inode = SYS_stdout) ==> nonConfidential prev) /\
         (appendFile (printfSpec "%s %s %s " vargs) prev =
          getFile SYS_FileSystem' inode)) /\
        ((inode = SYS_stdout) \/
         ((inode = SYS_stdout) ==>
          nonConfidential (getFile SYS_FileSystem' inode))) ==>
```

```
       (inode = SYS_stdout) ==>
       nonConfidential (getFile SYS_FileSystem' SYS_stdout)`--)

- e (STRIP_THEN_REWRITE_TAC);

4 subgoals:
  (--`nonConfidential (getFile SYS_FileSystem' SYS_stdout)`--)
  ____________________________
      (--`nonConfidentialS (printfSpec "%s %s %s " vargs)`--)
      (--`(inode = SYS_stdout) ==>
          nonConfidential (getFile SYS_FileSystem inode)`--)
      (--`C_Result1 > 0`--)
      (--`(inode = SYS_stdout) ==> nonConfidential prev`--)
      (--`appendFile (printfSpec "%s %s %s " vargs) prev =
          getFile SYS_FileSystem' inode`--)
      (--`(inode = SYS_stdout) ==>
          nonConfidential (getFile SYS_FileSystem' inode)`--)
      (--`inode = SYS_stdout`--)

  (--`nonConfidential (getFile SYS_FileSystem' SYS_stdout)`--)
  ____________________________
      (--`nonConfidentialS (printfSpec "%s %s %s " vargs)`--)
      (--`(inode = SYS_stdout) ==>
          nonConfidential (getFile SYS_FileSystem inode)`--)
      (--`C_Result1 > 0`--)
      (--`(inode = SYS_stdout) ==> nonConfidential prev`--)
      (--`appendFile (printfSpec "%s %s %s " vargs) prev =
          getFile SYS_FileSystem' inode`--)
      (--`inode = SYS_stdout`--)

  (--`nonConfidential (getFile SYS_FileSystem' SYS_stdout)`--)
  ____________________________
      (--`nonConfidentialS (printfSpec "%s %s %s " vargs)`--)
```

```
    (--`(inode = SYS_stdout) ==>
        nonConfidential (getFile SYS_FileSystem inode)`--)
    (--`C_Result1 > 0`--)
    (--`~(inode = SYS_stdout)`--)
    (--`(inode = SYS_stdout) ==>
        nonConfidential (getFile SYS_FileSystem' inode)`--)
    (--`inode = SYS_stdout`--)

(--`nonConfidential (getFile SYS_FileSystem' SYS_stdout)`--)
____________________________
    (--`nonConfidentialS (printfSpec "%s %s %s " vargs)`--)
    (--`(inode = SYS_stdout) ==>
        nonConfidential (getFile SYS_FileSystem inode)`--)
    (--`C_Result1 > 0`--)
    (--`~(inode = SYS_stdout)`--)
    (--`inode = SYS_stdout`--)
```

SEE ALSO

REPEAT (HOL), STRIP_TAC (HOL), ASM_REWRITE_TAC (HOL).

LIFT_QUANT_TAC : tactic

SYNOPSIS

Move quantifiers outward as much as possible.

KEYWORDS

tactic, quantifier, existential, universal.

DESCRIPTION

Move all universal and existential quantifiers as far outward as possible. This way we can deal with them all at once rather than encountering them at different, odd times in the proof. It is defined as follows.

```
val LIFT_QUANT_TAC =
 CONV_TAC (REDEPTH_CONV (
    AND_EXISTS_CONV ORELSEC AND_FORALL_CONV ORELSEC
    OR_EXISTS_CONV ORELSEC OR_FORALL_CONV ORELSEC
    LEFT_AND_EXISTS_CONV ORELSEC LEFT_AND_FORALL_CONV ORELSEC
    LEFT_IMP_EXISTS_CONV ORELSEC LEFT_IMP_FORALL_CONV ORELSEC
    LEFT_OR_EXISTS_CONV ORELSEC LEFT_OR_FORALL_CONV ORELSEC
    RIGHT_AND_EXISTS_CONV ORELSEC RIGHT_AND_FORALL_CONV ORELSEC
    RIGHT_IMP_EXISTS_CONV ORELSEC RIGHT_IMP_FORALL_CONV ORELSEC
    RIGHT_OR_EXISTS_CONV ORELSEC RIGHT_OR_FORALL_CONV));
```

Subgoals: It leaves the goal, which may have been changed.

FAILURE

Never fails.

EXAMPLE

See the example in UNDISCH_ALL_TAC. We use UNDISCH_ALL_TAC to undischarge all assumptions, then we use this tactic to sweep all quantifiers outward at once in preparation for unifying.

USES

This may be used after UNDISCH_ALL_TAC to simplify and unify all the assumption at once.

SEE ALSO

`UNDISCH_ALL_TAC` and many HOL conversions such as `AND_EXISTS_CONV`, `OR_FORALL_CONV`, `LEFT_AND_EXISTS_CONV`, `RIGHT_IMP_FORALL_CONV`.

ESTAB_TAC : (term -> tactic)

SYNOPSIS

Establish the term from the assumptions.

KEYWORDS

tactic, assumption.

DESCRIPTION

This tries to add the term to the assumption list using various of the assumptions and built-in tactics.

ESTAB_TAC tries each assumption by itself, then pairs of assumptions, then triples of assumptions with ARITH_CONV and TAUT_CONV to establish the term. If it does, it adds it to the assumption list with ASSUME_TAC. The heart of this mechanism is also used by INCONSIST_TAC. It is not algorithmically elegant, but it saves the user time.

Subgoals: None.

FAILURE

Fails if the term cannot be added.

EXAMPLE

Suppose you have the following goal.

```
(--`j > 0`--)
------------------------
(--`j >= arSz`--)
(--`!n. n < j ==> max >= f n`--)
(--`max = 0`--)
(--`arSz > 0`--)
(--`^fnn ar = arSz`--)
(--`j <= arSz`--)
```

Some inspection shows that we could establish `j > 0` by from `j >= arSz`, `arSz > 0`, and `j <= arSz`. The following tactic solves the above goal.

```
e (ESTAB_TAC (--`j > 0`--) THEN ASM_REWRITE_TAC []);
```

USES

Rather than picking through the assumption list or working out a long mathematical proof, this may be used to establish useful assumptions to rewrite or solve the goal.

SEE ALSO

INCONSIST_TAC, SOLVE_TAC.

INCONSIST_TAC : tactic

SYNOPSIS

Try to solve the goal by finding an inconsistency.

KEYWORDS

tactic, assumption.

DESCRIPTION

This tactic tries to solve a goal by proving an inconsistency in the assumptions.

The implementation is to try to establish F (false) (see ESTAB_TAC for details), then solve the goal with CONTRA_TAC. If that doesn't work, it enriches the assumption list with equalities to find an inconsistency with an inequality.

Subgoals: None.

FAILURE

Fails if it cannot solve the goal.

EXAMPLE

```
(--`someFunction n = 0`--)
----------------------
(--`n < j`--)
(--`0 <= n`--)
```

```
(--`j = 0`--)
```

Inspection suggests a proof by contradiction using the assumptions `n < j` and `j = 0`. The following tactic solves this goal.

```
e (INCONSIST_TAC)
```


ACKNOWLEDGEMENT

The idea and implementation of solving a goal by enriching the assumption list with equalities to find an inconsistency with an inequality is due to Robert Beers (beers@lal.cs.byu.edu).

SEE ALSO

`ESTAB_TAC`, `SOLVE_TAC`, `CONTRA_TAC` (HOL).

`SOLVE_TAC : tactic`

SYNOPSIS

Try to solve a goal by several different general approaches.

KEYWORDS

tactic.

DESCRIPTION

This is a general purpose tactic for solving goals, especially those which arise in software proofs with axiomatic semantics. It tries a series of tactics to solve the goal. It may take a few seconds, but these days user time is often more valuable than computer time. It tries the following ways.

1. Establish the goal from the assumptions.

2. Solve by proving an inconsistency in the assumptions.

3. If the goal is a = b, establish equalities which unify a and b.

4. If an assumption and the goal are b and `a ==> b'`, establish a and the unifiers for b and b'.

This makes heavy use of ESTAB_TAC and INCONSIST_TAC to try various approaches.

Subgoals: None.

FAILURE

Fails if it cannot solve the goal.

EXAMPLE

SOLVE_TAC solves the following goals directly and automatically.

```
(--`P d`--)

----------------------------
(--`a`--)
(--`a ==> P c`--)
(--`c = d`--)

(--`nonConfidential (getFile SYS_FileSystem SYS_stdout)`--)

----------------------------
(--`inode = SYS_stdout`--)
(--`!inode. (inode = SYS_stdout) ==>
      nonConfidential (getFile SYS_FileSystem inode)`--)
(--`a ==> b`--)
```

SEE ALSO

`ESTAB_TAC, INCONSIST_TAC.`

A.7.2 Tactics to Handle Assumptions

`FILTER_UNDISCH_TAC : ((thm -> bool) -> tactic)`

SYNOPSIS

Undischarge a selected assumption.

KEYWORDS

tactic, assumption, undischarge, filter.

DESCRIPTION

This tactic undischarges one of the assumptions which matches the filter function. Using filter rather than selecting assumptions by order makes proofs slightly less sensitive to change. A program to generate filter functions from an assumption list is given in [6].

Subgoals: None.

FAILURE

```
uncaught exception Empty
```

No assumption matches the filter function.

EXAMPLE

This example comes from a lemma during the proof of information integrity of `logfile()`. The filter function looks for a theorem which is a "for all" term.

```
1 subgoal:
    (--`preFSS inode (getFile SYS_FileSystem' inode)`--)
    ----------------------------
    (--`!inode.
             (inode = inodeOf (deref ^F)) \/
             preFSS inode (getFile SYS_FileSystem' inode)`--)
    (--`~(inode = inodeOf (deref ^F))`--)

- e (FILTER_UNDISCH_TAC (fn t=>is_forall (concl t)));

1 subgoal:
  (--`(!inode.
        (inode = inodeOf (deref ^F)) \/
        preFSS inode (getFile SYS_FileSystem' inode)) ==>
```

```
      preFSS inode (getFile SYS_FileSystem' inode)`--)
  ----------------------------
      (--`~(inode = inodeOf (deref ^F))`--)
```

This is solved by the following tactic.

```
e(LIFT_QUANT_TAC THEN EXISTS_TAC (--`inode:num`--) THEN
        ASM_REWRITE_TAC []);
```

SEE ALSO

`UNDISCH_ALL_TAC`, `UNDISCH_TAC` (HOL).

UNDISCH_ALL_TAC : tactic

SYNOPSIS

Undischarge all assumptions.

KEYWORDS

tactic, assumption, undischarge.

DESCRIPTION

This undischarges all assumptions. It is often easier to use this then to specify exactly which assumption or assumptions to undischarge.

Subgoals: The only subgoal is the original with all assumptions undischarged.

FAILURE

Always succeeds if the goal is the proper form.

EXAMPLE

This comes from the proof of confidentiality of `logfile()`. We are trying to prove that the precondition (from the previous statement's postcondition) implies the function call computed precondition. It shows the use of LIFT_QUANT_TAC afterward.

```
1 subgoal:
  (--`nonConfidential (getFile SYS_FileSystem' SYS_stdout)`--)
  ----------------------------
  (--`F = ^F`--)
  (--`~(inodeOf (deref F) = SYS_stdout)`--)
  (--`!inode.
        (inode = SYS_stdout) ==>
        nonConfidential (getFile SYS_FileSystem inode)`--)
  (--`C_Result16 > 0`--)
  (--`!inode.
        (~(inode = inodeOf (deref F)) \/
         (?prev.
           ((inode = SYS_stdout) ==> nonConfidential prev) /\
           (appendFile (printfSpec "%s %s %s %s %s %d %d " vargs)
            prev = getFile SYS_FileSystem' inode))) /\
        ((inode = inodeOf (deref F)) \/
         ((inode = SYS_stdout) ==>
          nonConfidential (getFile SYS_FileSystem' inode)))`--)
  (--`inode = SYS_stdout`--)

- e (UNDISCH_ALL_TAC);

1 subgoal:
(--`(F = ^F) ==> ~(inodeOf (deref F) = SYS_stdout) ==>
  (!inode. (inode = SYS_stdout) ==>
    nonConfidential (getFile SYS_FileSystem inode)) ==>
  C_Result9 > 0 ==>
  (!inode.
    (~(inode = inodeOf (deref F)) \/
     (?prev.
       ((inode = SYS_stdout) ==> nonConfidential prev) /\
       (appendFile (printfSpec "%s %s %s %s %s %d %d " vargs)
          prev = getFile SYS_FileSystem' inode))) /\
```

```
    ((inode = inodeOf (deref F)) \/
     ((inode = SYS_stdout) ==>
      nonConfidential (getFile SYS_FileSystem' inode)))) ==>
  (inode = SYS_stdout) ==>
  nonConfidential (getFile SYS_FileSystem' SYS_stdout)`--)

- e (LIFT_QUANT_TAC);

1 subgoal:
(--`?inode' inode''.
  !prev. (F = ^F) ==>
    ~(inodeOf (deref F) = SYS_stdout) ==>
    ((inode' = SYS_stdout) ==>
     nonConfidential (getFile SYS_FileSystem inode')) ==>
    C_Result9 > 0 ==>
    (~(inode'' = inodeOf (deref F)) \/
     ((inode'' = SYS_stdout) ==> nonConfidential prev) /\
     (appendFile (printfSpec "%s %s %s %s %s %d %d " vargs)
            prev = getFile SYS_FileSystem' inode'')) /\
    ((inode'' = inodeOf (deref F)) \/
     ((inode'' = SYS_stdout) ==>
      nonConfidential (getFile SYS_FileSystem' inode''))) ==>
    (inode = SYS_stdout) ==>
    nonConfidential (getFile SYS_FileSystem' SYS_stdout)`--)
```

This can be solved with the following tactic

```
REPEAT (EXISTS_TAC (--`inode:num`--)) THEN
       STRIP_THEN_REWRITE_TAC THEN SOLVE_TAC
```

SEE ALSO

FILTER_UNDISCH_TAC, LIFT_QUANT_TAC, UNDISCH_TAC (HOL).

A.7.3 Arithmetic Tactics

ARITH_TAC : tactic

SYNOPSIS

Solve arithmetic goals.

KEYWORDS

tactic, arithmetic.

DESCRIPTION

Apply ARITH_CONV as a tactic. It is simply the following.

```
fun ARITH_TAC = CONV_TAC ARITH_CONV;
```

Subgoals: None.

FAILURE

Fails if it cannot solve the goal.

EXAMPLE

```
1 subgoal:
  (--`1 < 2`--)

- e (ARITH_TAC);

Goal proved.
```

SEE ALSO

`DEPTH_ARITH_TAC`, `ARITH_CONV` (HOL).

DEPTH_ARITH_TAC : tactic

SYNOPSIS

Reduce arithmetic terms in the goal.

KEYWORDS

tactic, arithmetic.

DESCRIPTION

This goes through a goal proving and eliminating as many arithmetic terms as possible. The implementation is

```
val DEPTH_ARITH_TAC = CONV_TAC (ONCE_DEPTH_CONV
        (ARITH_CONV ORELSEC NEGATE_CONV ARITH_CONV));
```

It is inefficient since for every expression it tries to prove it is true (ARITH_CONV) then, if that fails, that it is false (NEGATE_CONV ARITH_CONV).

Subgoals: At most the original goal simplified.

FAILURE

Always succeeds if the goal is the proper form.

EXAMPLE

This goal arose while proving that a piece of code finds the maximum in an array. It says, "if the maximum is greater than zero, the maximum is some element in the array" for the initial maximum, which is zero. DEPTH_ARITH_TAC reduces `0 > 0` to false. The goal may then be proved with REWRITE_TAC.

```
1 subgoal:
  (--`(0 > 0) => (?n. 0 = CA_IDX ar n) | T`--)
  ____________________________
      (--`arSz = CA_SZ ar`--)
      (--`arSz > 0`--)
      (--`j = 0`--)
      (--`max = 0`--)

- e (DEPTH_ARITH_TAC);
```

```
1 subgoal:
  (--`F => (?n. 0 = CA_IDX ar n) | T`--)
  ____________________________
      (--`arSz = CA_SZ ar`--)
      (--`arSz > 0`--)
      (--`j = 0`--)
      (--`max = 0`--)
```

SEE ALSO

ARITH_TAC, TAUT_TAC.

APPENDIX B

OS AND LIBRARY CALL AXIOMS

We distinguish between the types of calls merely to structure our presentation: the distinction is unimportant to the proof itself. Within each section we group the calls by related functionality, for example, higher level I/O commands, lower level I/O commands, time commands, etc. Input/output calls are those which interact with the file system. Miscellaneous operating system calls interact with the program execution environment, such as the process table or memory allocation. Library functions are those which only operate on their parameters.

For each function, we give excerpts from the "man" page [28, 29] covering the important points of the function declaration, behavior or intended use, return values, and notes on errors. Immediately following is our `WF_fnp` or function call axiom (see Inference Rule 6.2, page 62). Note that these are *not* complete descriptions of the functions by any stretch of the imagination. They model enough of the function for our proof.

B.1 Input/Output Calls

fopen

```
SYNOPSIS
    FILE *fopen(const char *pathname, const char *type);

DESCRIPTION
    fopen() opens the file named by pathname and returns a
    pointer to it.

RETURN VALUE
    Upon successful completion, a FILE pointer is returned.
    Otherwise, NULL is returned and errno is set to indicate
    the error.
```

```
|- WF_fnp (T)
    (Func (Var "fopen" 0 (Ptr(Struct "FILE")))
     [Var "pathname" 0 (Ptr Char); Var "type" 0 (Ptr Char)]
     [Var "errno" 0 Int; Var "SYS_cwd" 0 (Ptr Char);
      Var "SYS_root" 0 (Ptr Char)] NOBody)
    ((C_Result = NULL) /\ (?FOPEN_errno.errno=FOPEN_errno) \/
     (?FOPEN_handlefn.C_Result =
                    FOPEN_handlefn (inodeNamed ^pathname) ^type))
```

fclose

```
SYNOPSIS
    int fclose(FILE *stream);

DESCRIPTION
    fclose() causes any buffered data for the named stream to be
    written out, and the stream to be closed.

RETURN VALUE
    Upon successful completion, fclose() returns 0.  Otherwise,
    EOF is returned and errno is set to indicate the error.

|- WF_fnp (T)
    (Func (Var "fclose" 0 Int) [Var "stream" 0(Ptr(Struct "FILE"))]
     [Var "errno" 0 Int] NOBody)
    (T)
```

fprintf

This is one of the most complex axioms because details of its behavior are critical to file integrity and confidentiality. The "varargs" (variable number of arguments to a function) aspect is represented by giving the fixed components, in this case `stream` and `format`, then a special variable, `varargs`. The postcondition comes in three parts corresponding to the three outcomes we must distinguish.

1. The return value is some negative number if there was an error. In some cases, errno is set, too.

2. The return value is zero if nothing happened; the file system is unchanged.

3. The return value is some positive number. The file system is unchanged except that the file pointer passed has something appended.

```
SYNOPSIS
    int fprintf(FILE *stream, const char *format, ...)

DESCRIPTION
    fprintf() writes arguments to the named output stream.

RETURN VALUE
    return the number of bytes written, or a negative value if an
    output error was encountered.  In some cases, errno is set.

|- WF_fnp (SoFS preFSS ^SYS_FileSystem)
  (Func (Var "fprintf" 0 Int)
   (CONS (Var "stream" 0 (Ptr(Struct "FILE")))
    (CONS (Var "format" 0 (Ptr Char)) varargs))
   [Var "SYS_FileSystem" 0
    (Struct "(((permission,fscontents)unixFile,num)prod)CArray");
    Var "errno" 0 Int] NOBody)
  ((?FPRINTF_error.C_Result = int_neg FPRINTF_error)
      /\ (inSomeCasesOf C_Result
                   (?FPRINTF_errno.errno=FPRINTF_errno))
      /\ (SoFS preFSS ^SYS_FileSystem) \/
   (C_Result = 0) /\ (SoFS preFSS ^SYS_FileSystem) \/
   C_Result > 0
      /\ (SoFS (\inode fcontents .
            (inode = inodeOf (deref ^stream)) =>
```

```
(?prev. preFSS inode prev /\
    (appendFile
          (printfSpec ^format varargs)
          prev = fcontents)) |
(preFSS inode fcontents)) ^SYS_FileSystem))
```

There should be an additional precondition that `stream` is valid, or at least not NULL, and is open for write or append.

printf

```
SYNOPSIS
    int printf(const char *format, ...)

DESCRIPTION
    printf() writes arguments to the standard output.

|- WF_fnp (SoFS preFSS ^SYS_FileSystem)
  (Func (Var "printf" 0 Int)
   (CONS (Var "format" 0 (Ptr Char)) varargs)
   [Var "SYS_FileSystem" 0
      (Struct "(((permission,fscontents)unixFile,num)prod)CArray");
    Var "errno" 0 Int] NOBody)
  ((?PRINTF_error.C_Result = int_neg PRINTF_error)
      /\ (inSomeCasesOf C_Result
                          (?PRINTF_errno.errno=PRINTF_errno))
      /\ (SoFS preFSS ^SYS_FileSystem) \/
   (C_Result = 0) /\ (SoFS preFSS ^SYS_FileSystem) \/
   C_Result > 0
      /\ (SoFS (\inode fcontents .
            (inode = SYS_stdout) =>
                (?prev. preFSS inode prev /\
                    (appendFile
                        (printfSpec ^format varargs)
```

```
                        prev = fcontents)) |
                (preFSS inode fcontents)) ^SYS_FileSystem))
```

sscanf

This may change memory indicated by the pointers, but we can't express the "varargs" well enough.

```
SYNOPSIS
    int sscanf(const char *s, const char *format, ...);

DESCRIPTION
    sscanf() reads characters from s, interprets them according to
    the format, and stores any results the pointers.

RETURN VALUE
    If the input ends prematurely, EOF is returned.  Otherwise,
    sscanf() returns the number of successfully assigned input
    items.

|- WF_fnp (T)
    (Func (Var "sscanf" 0 Int)
     (CONS (Var "s" 0 (Ptr Char))
     (CONS (Var "format" 0 (Ptr Char)) varargs))
     [] NOBody)
    ((C_Result = EOF) \/
     (C_Result = sscanfSpecNum(^format)))
```

These are the lower level input/output functions.

open

```
SYNOPSIS
    int open(const char *pathname, int oflag);
```

```
DESCRIPTION
    open() opens the file named by pathname in the given mode.

RETURN VALUE
    Upon successful completion, a file descriptor is returned.
    Otherwise, -1 is returned and errno is set to indicate the
    error.

|- WF_fnp (T)
    (Func (Var "open" 0 Int)
     [Var "pathname" 0 (Ptr Char); Var "oflag" 0 Int]
     [Var "errno" 0 Int;
      Var "SYS_cwd" 0 (Ptr Char); Var "SYS_root" 0 (Ptr Char)]
     NOBody)
    ((C_Result = int_neg 1) /\ (?OPEN_errno.errno=OPEN_errno) \/
      (?OPEN_fildesfn.C_Result =
              OPEN_fildesfn ^pathname ^oflag ^SYS_cwd ^SYS_root))
```

close

```
SYNOPSIS
    int close(int fildes);

DESCRIPTION
    close() closes the file descriptor.

RETURN VALUE
    Upon successful completion, close() returns 0; otherwise, it
    returns -1 and sets errno to indicate the error.

|- WF_fnp (T)
    (Func (Var "close" 0 Int) [Var "fildes" 0 Int]
     [Var "errno" 0 Int] NOBody)
```

```
((C_Result = int_neg 1) /\ (?CLOSE_errno.errno=CLOSE_errno) \/
 (C_Result = 0))
```

read

The last clause in the postcondition says that if the read succeeded, the file descriptor couldn't have been -1.

```
SYNOPSIS
    size_t read(int fildes, void *buf, size_t nbyte);

DESCRIPTION
    read() tries to read nbyte bytes from the file descriptor into
    buf.

RETURN VALUE
    Upon successful completion, read() returns the number of bytes
    actually read and placed in the buffer; this may be less than
    nbyte.  When an end-of-file is reached, a value of 0 is
    returned.  Otherwise, a -1 is returned and errno is set to
    indicate the error.

|- WF_fnp (T)
    (Func (Var "read" 0 Int)
     [Var "fildes" 0 Int; Var "buf" 0 (Ptr Char);Var "nbyte" 0 Int]
     [Var "errno" 0 Int] NOBody)
    ((C_Result = int_neg 1) /\ (?READ_errno.errno=READ_errno) \/
     (C_Result = 0) \/
     ~(^fildes = int_neg 1) /\ (C_Result > 0) /\
                (C_Result = readSpec(^fildes, ^buf, ^nbyte)))
```

write

```
SYNOPSIS
    ssize_t write(int fildes, const void *buf, size_t nbyte);
```

DESCRIPTION

```
    write() tries to write nbyte bytes from buf into the file
    descriptor.
```

RETURN VALUE

```
    Upon successful completion, write() returns the number of
    bytes actually written.  Otherwise, -1 is returned and errno
    is set to indicate the error.
```

```
|- WF_fnp (SoFS preFSS ^SYS_FileSystem)
  (Func (Var "write" 0 Int)
   [Var "fildes" 0 Int; Var "buf" 0 (Ptr Char); Var "nbyte" 0 Int]
   [Var "SYS_FileSystem" 0
      (Struct "(((permission,fscontents)unixFile,num)prod)CArray");
    Var "errno" 0 Int] NOBody)
  ((C_Result = int_neg 1) /\ (?WRITE_errno.errno=WRITE_errno)
      /\ (SoFS preFSS ^SYS_FileSystem) \/
   (?WRITE_res.(C_Result=WRITE_res) /\
                       WRITE_res >= 0 /\ WRITE_res <= nbyte)
      /\ (SoFS (\inode fcontents .
              (inode = inodeOfFileDes ^fildes) =>
                  (?prev. preFSS inode prev /\
                      (appendFile (writeSpec ^buf ^nbyte)
                          prev = fcontents)) |
                  (preFSS inode fcontents)) ^SYS_FileSystem))
```

B.2 Miscellaneous Operating System Calls

chdir

SYNOPSIS

```
    int chdir(const char *path);
```

```
DESCRIPTION
    chdir() causes a directory pointed to by path to become the
    current working directory.

RETURN VALUE
    Upon successful completion, a value of 0 is returned.
    Otherwise, a value of -1 is returned and errno is set to
    indicate the error.

ERRORS
    chdir() fails and the current working directory remains
    unchanged if one or more of the following are true . . .

|- WF_fnp (T)
    (Func (Var "chdir" 0 Int) [Var "path" 0 (Ptr Char)]
     [Var "errno" 0 Int; Var "SYS_cwd" 0 (Ptr Char)] NOBody)
    ((C_Result = 0) /\ (SYS_cwd=^path) \/
     (C_Result = int_neg 1) /\ (SYS_cwd=^SYS_cwd)
                           /\ (?CHDIR_errno.errno=CHDIR_errno))
```

chroot

```
SYNOPSIS
    int chroot(const char *path);

DESCRIPTION
    chroot() causes the named directory to become the root
    directory.

    The effective user ID must be a user having appropriate
    privileges.

RETURN VALUE
```

```
    Upon successful completion, a value of 0 is returned.
    Otherwise, a value of -1 is returned and errno is set to
    indicate the error.

ERRORS
    chroot() fails and the root directory remains unchanged if
    one or more of the following are true . . .

|- WF_fnp (T)
    (Func (Var "chroot" 0 Int) [Var "path" 0 (Ptr Char)]
     [Var "errno" 0 Int; Var "SYS_cwd" 0 (Ptr Char);
      Var "SYS_root" 0 (Ptr Char); Var "SYS_euid" 0 Int] NOBody)
    (((C_Result = 0) /\
         (SYS_root=resolvePath ^SYS_root ^SYS_cwd ^path)
       \/ (C_Result = int_neg 1) /\ (SYS_root=^SYS_root)
                          /\ (?CHROOT_errno.errno=CHROOT_errno))
     /\ (SYS_cwd=^SYS_cwd) /\ (SYS_euid=^SYS_euid))
```

geteuid

The function `geteuid` should return a special system type, `uid_t`. However since we did not implement type casting, we approximate it with `int`. (We discuss type casting in more detail on page 46.) Several other functions have special return types defined, and we approximate those similarly.

```
SYNOPSIS
    uid_t geteuid(void);

DESCRIPTION
    geteuid() returns the effective user ID of the process.

|- WF_fnp (T)
    (Func (Var "geteuid" 0 Int) [] [] NOBody)
    (C_Result = ^SYS_euid)
```

setuid

```
SYNOPSIS
    int setuid(uid_t uid);

DESCRIPTION
    setuid() sets the real user ID, effective user ID, and/or
    saved user ID.

    Summary: if setuid() succeeds, the effective user ID is the
    uid passed.  The real and saved user ID's are set depending
    on complicated conditions.

RETURN VALUE
    Upon successful completion, setuid() returns 0.  Otherwise it
    returns -1 and sets errno.

|- WF_fnp (T)
    (Func (Var "setuid" 0 Int) [Var "uid" 0 Int]
     [Var "errno" 0 Int; Var "SYS_euid" 0 Int;
      Var "SYS_ruid" 0 Int; Var "SYS_suid" 0 Int] NOBody)
    ((C_Result = int_neg 1)
            /\ (?SETEUID_errno.errno=SETEUID_errno) \/
     (C_Result = 0) /\
     (SYS_euid = ^uid) /\
     ((SYS_ruid = ^SYS_ruid) \/ (SYS_ruid = ^uid)) /\
     ((SYS_suid = ^SYS_suid) \/ (SYS_suid = ^uid)))
```

exit

As coded `exit()` produces a totally undefined state. This allows us to prove theorems for "filter" conditionals, such as the following.

$$\vdash \{\mathtt{T}\}\ \mathtt{if\ \ (x < 0)\ \ exit(1);}\ \{x \geq 0\}$$

The postcondition follows if the test fails. To cover the case that the test succeeds, we must prove the following.

$$\vdash \{\mathtt{T} \wedge x < 0\}\ \mathtt{exit(1);}\ \{x \geq 0\}$$

Unfortunately this coding of `exit()` prevents us from analyzing the final state of the program formally. It also allows us to prove the following.

$$\vdash \{\mathtt{T}\}\ \mathtt{exit(1);}\ \ \mathtt{y = x;}\ \{y \geq 0\}$$

Since we can prove this intermediate.

$$\vdash \{\mathtt{T}\}\ \mathtt{exit(1);}\ \{x \geq 0\}$$

Therefore total correctness must be more specific in that the statement not only terminate, but execution reaches the final statement (reachability). This is analogous to preventing flawed reasoning that a nonterminating loop establishes any condition, like below, since $\vdash \{\mathtt{T}\}\ \mathtt{while(1);}\ \{\mathtt{F}\}$.

$$\vdash [\mathtt{T}]\ \mathtt{while(1);}\ \ \mathtt{y = x;}\ [y \geq 0]$$

Instead `exit()` should be coded to express a transfer of control to the end of the program. One possible remedy is to use multiple post conditions, as suggested in [1] and refined for C in [38]. The `exit()` call, and thus `main()` and thttpd itself, would have `F` as the "sequence" postcondition and the constraint as the "exit" postcondition. We would then have to prove that execution actually could reach the points of interest, as opposed to the unreachable `y=x;` above, in addition to partial correctness.

```
SYNOPSIS
    void exit(int status);

DESCRIPTION
    exit() terminates the calling process and passes status to
    the system for inspection, ...
```

```
|- WF_fnp (T)
    (Func (Var "exit" 0 Void) [Var "status" 0 Int] [] NOBody)
    (F)
```

localtime

SYNOPSIS

```
    struct tm *localtime(time_t *time)
```

DESCRIPTION

    Correct for the time zone according to the TZ environment
    variable and convert to (local static?) tm structure.

RETURN VALUE

    return a pointer to the tm structure.

```
|- WF_fnp (T)
    (Func (Var "localtime" 0 (Ptr(Struct "tm")))
     [Var "time" 0 (Ptr Int)]
     [Var "TZ" 0 (Struct "TZstruct")] NOBody)
    (?tsptr.C_Result = tsptr)
```

strftime

SYNOPSIS

```
    size_t strftime(char *s, size_t maxsize, const char *format,
                                        const struct tm *timeptr)
```

DESCRIPTION

    Convert the contents of a tm structure to a formatted date
    and time string.

RETURN VALUE

    return the length of the string put in s. If the string
    exceeds maxsize, return zero; the contents of s are
    indeterminate.

```
|- WF_fnp (T)
```

```
(Func (Var "strftime" 0 Int)
 [Var "s" 0 (Ptr Char); Var "maxsize" 0 Int;
  Var "format" 0 (Ptr Char);
  Var "timeptr" 0 (Ptr (Struct "tm"))]
 [] NOBody)
((strlen(strftimeSpec(format, timeptr)) < maxsize =>
    ((C_Result=strlen(s)) /\
          (strcmp(s,strftimeSpec(format,timeptr))=0))
    | (C_Result = 0))
 /\ (!index.accessed(s, index) ==> index >= 0 /\
                                     index < maxsize))
```

time

SYNOPSIS

```
time_t time(time_t *tloc);
```

DESCRIPTION

time() returns the value of time in seconds since the Epoch.

If tloc is not a null pointer, the return value is also assigned to the object to which it points.

RETURN VALUE

Upon successful completion, time() returns the value of time. Otherwise, a value of (time_t)-1 is returned and errno is set to indicate the error.

```
|- WF_fnp (T)
    (Func (Var "time" 0 Int) [Var "tloc" 0 (Ptr Int)]
     [Var "errno" 0 Int] NOBody)
    ((C_Result = int_neg 1) /\ (?TIME_errno.errno=TIME_errno) \/
      (?some_time.(C_Result = some_time) /\ C_Result > 0 /\
```

```
        (~(^tloc=NULL)==>(deref ^tloc = some_time))))
```

stat

```
SYNOPSIS
    int stat(const char *path, struct stat *buf);

DESCRIPTION
    stat() obtains information about the named file and puts it
    in buf.

RETURN VALUE
    Upon successful completion, 0 is returned.  Otherwise, -1 is
    returned and errno is set to indicate the error.

|- WF_fnp (T)
    (Func (Var "stat" 0 Int)
     [Var "path" 0 (Ptr Char); Var "buf" 0  (Ptr (Struct "stat"))]
     [Var "errno" 0 Int;
      Var "SYS_cwd" 0 (Ptr Char); Var "SYS_root" 0 (Ptr Char)]
     NOBody)
    (((C_Result = 0) /\
                (deref ^buf = statSpec ^path ^SYS_cwd ^SYS_root) \/
      (C_Result = int_neg 1) /\ (?STAT_errno.errno=STAT_errno)) /\
     (SYS_cwd = ^SYS_cwd) /\ (SYS_root = ^SYS_root))
```

S_ISREG

This is implemented as a macro, not a function.

```
SYNOPSIS
    int S_ISREG(ushort st_mode);

DESCRIPTION
    S_ISREG returns true if the file mode indicates a regular
```

```
    file (vs. directory, pipe, or special).

|- WF_fnp (T)
    (Func (Var "S_ISREG" 0 Int) [Var "st_mode" 0 Int] [] NOBody)
    ((C_Result = 0) \/ (C_Result = 1))
```

B.3 Library Calls

The library functions which thttpd uses are limited to those which manipulate strings.

strcat

```
SYNOPSIS
    char *strcat(char *s1, const char *s2)

DESCRIPTION
    Append s2 to the end of s1.

RETURN VALUE
    return a pointer to s1.

|- WF_fnp (T)
    (Func (Var "strcat" 0 (Ptr Char))
     [Var "s1" 0 (Ptr Char); Var "s2" 0 (Ptr Char)]
     [] NOBody)
    ((C_Result = ^s1) /\
     strEq(strderef ^s1,
                (strcatSpec(^s1Contents, ^s2Contents))) /\
     (strderef ^s2 = ^s2Contents))
```

strncasecmp

```
SYNOPSIS
    int strncasecmp(const char *s1, const char *s2, size_t n)
```

```
DESCRIPTION
    return an integer less than, equal to, or greater than zero,
    depending on whether s1 is less than, equal to, or greater than
    s2.  Examine a maximum of n characters.  Null pointers are the
    same as pointers to empty strings.  Characters are folded by
    _tolower() prior to comparison.

|- WF_fnp (T)
    (Func (Var "strncasecmp" 0 Int)
     [Var "s1" 0 (Ptr Char); Var "s2" 0 (Ptr Char); Var "n" 0 Int]
     [] NOBody)
    ((C_Result = strncasecmpSpec((strderef ^s1),
                                 (strderef ^s2), (n))) /\
     (strderef ^s1 = ^s1Contents) /\
     (strderef ^s2 = ^s2Contents))
```

strcpy

```
SYNOPSIS
    char *strcpy(char *s1, const char *s2)

DESCRIPTION
    Copy s2 to s1.

RETURN VALUE
    returns a pointer to s1.

|- WF_fnp (T)
    (Func (Var "strcpy" 0 (Ptr Char))
     [Var "s1" 0 (Ptr Char); Var "s2" 0 (Ptr Char)]
     [] NOBody)
    ((C_Result = ^s1) /\
```

```
 strEq(strderef ^s1,
            (strcpySpec(^s1Contents, ^s2Contents))) /\
 (strderef ^s2 = ^s2Contents))
```

strncpy

```
SYNOPSIS
    char *strncpy(char *s1, const char *s2, size_t n)

DESCRIPTION
    Copy exactly n character of s2 to s1.

RETURN VALUE
    returns a pointer to s1.
```

```
val logn = mk_var{Name="^n", Ty=(==`:num`==)};

|- WF_fnp (T)
    (Func (Var "strncpy" 0 (Ptr Char))
     [Var "s1" 0 (Ptr Char); Var "s2" 0 (Ptr Char); Var "n" 0 Int]
     [] NOBody)
    ((C_Result = ^s1) /\
     strEq(strderef ^s1,
                (strncpySpec(^s2Contents, ^n))) /\
     (strderef ^s2 = ^s2Contents))
```

strlen

```
SYNOPSIS
    size_t strlen(const char *s)

DESCRIPTION
    return the number of characters in s.
```

```
|- WF_fnp (T)
    (Func (Var "strlen" 0 Int)
     [Var "s" 0 (Ptr Char)]
     [] NOBody)
    (C_Result = strlenSpec(strderef ^s))
```

APPENDIX C

THE SECURE WORLD WIDE WEB SERVER CODE

This appendix has the code for the latest version of `thttpd`. It differs slightly from the code which we first verified in that it

1. has `alarm()` calls to limit waits for input and output,
2. defines some values for Unix System V compiling,
3. declares variables globally rather than locally,
4. records more information in log file entries (such as the name and process ID of the logging process),
5. exits with a zero status (rather than a one status) upon error,
6. checks that files are owned by a specific user (WWWAID) rather than the process owner, and
7. checks for "get" requests character by character rather than using `strncasecmp()`.

It took us about four hours to convert the new code to AST using the translator including the manual pre- and postprocessing described in Sect. A.1 and time to extend the C parser. Reproving `logfile()` took about two hours, including the time to write axioms for the three new OS calls it makes.

These program listings have been slightly altered to fit the page width. The current source is available through `http://all.net/` which also demonstrates and uses this server. Licensing information is also available from that site.

```
#ifdef SYSV
#define _IFREG  0100000 /* regular */
#define _IFMT   0170000 /* type of file */
#define S_ISREG(m)      (((m)&_IFMT) == _IFREG)
#endif
#define ERRORLINE "The requested document has moved to
                <A HREF=http://pc31.ca.sandia.gov/>here</A>.<P>\n"
#define REDIRECT "Location: http://pc31.ca.sandia.gov/\n"
#define WWWUID 150
#define WWWAID 501
#define WWWDIR "/u/fc/www"
#define WWWDefaultFile "/top.html"
#define WWWlog "/log"
#define Ktime 20
#define IOtime 120
int CHECKUSER=1;int DOCHROOT=1;

#define BUFSIZE 4096
#define MAXSIZE 2048
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <time.h>
#include <fcntl.h>
#include <syslog.h>
#define LOG2(x,y) {F=fopen(WWWlog,"a+");
        if (F != NULL) {logfile(F);fprintf(F,x,y);} fclose(F);}
#define LOG3(x,y,z) {F=fopen(WWWlog,"a+");
        if (F != NULL) {logfile(F);fprintf(F,x,y,z);} fclose(F);}
#define LOG4(x,y,z,w) {F=fopen(WWWlog,"a+");
        if (F != NULL) {logfile(F);fprintf(F,x,y,z,w);} fclose(F);}
```

```
char line[BUFSIZE],name[BUFSIZE],bs1[BUFSIZE],bs2[BUFSIZE],
   bs3[BUFSIZE],timestamp[64],logline[BUFSIZE],remotehost[BUFSIZE],
   remoteuser[BUFSIZE], daemonname[BUFSIZE],remotedata[BUFSIZE];
struct stat buf; FILE *F;int        i,n,staterr;
time_t *tloc;time_t        t;


void logfile(F)
FILE *F;
{alarm(IOtime);t=time(NULL);
#ifdef SYSV
cftime(timestamp, "%Y/%m/%d %T", localtime(&t));
#else
strftime(timestamp, 20, "%Y/%m/%d %T", localtime(&t));
#endif
fprintf(F,"%s %s %s %s %s %d %d ",remotehost,remoteuser,daemonname,
                remotedata,timestamp,getpid(),getppid());alarm(0);}


error(s)                      /* simulate a 302 - document moved */
char *s;
{alarm(IOtime);printf("HTTP/1.0 302 Found\n");
 printf("Server: ManAl/0.1\n");printf("MIME-version: 1.0\n");
 printf(REDIRECT);printf("Content-type: text/html\n");
 printf("<HEAD><TITLE>Document moved</TITLE></HEAD>\n");
 printf("<BODY><H1>Document moved</H1>\n");printf(ERRORLINE);
 printf("(%s) </BODY>\n",s);
 alarm(IOtime);LOG4("Error:%s - %s %s\n",s,bs1,name);exit(0);}
```

```
void        cat(s)
char        s[];
{alarm(IOtime);i=open(s,0); while ((n=read(i,bs2,MAXSIZE)) > 0)
        {alarm(IOtime);write(1,bs2,n);}close(i);alarm(0);}

fetch()       /* if www owns it, it can be put - else, forget it */
{alarm(IOtime);staterr=stat(name,&(buf));alarm(0);
if (staterr != 0) error("Can't stat file");       /* can't stat the
                                                     file - die */
if (0 == S_ISREG(buf.st_mode)) error("Can't fetch directories");
if (0 != (S_IXOTH & buf.st_mode)) error("Can't fetch executables");
if (CHECKUSER==1) if (buf.st_uid != WWWAID)
          error("Not owned by WWW Admin"); /* don't own it - die */
if (0 != (S_IROTH & buf.st_mode))
    {cat(name); LOG2("cat %s\n",name);exit(0);}       /* Send it*/
error("Access Denied");}


main(argc,argv,envp)
int argc; char *argv[],*envp[];
{alarm(Ktime);if (0 != chdir(WWWDIR))
               error("Cannot change to WWW directory");
if (DOCHROOT == 1) if (chroot(".") != 0)
               error("Cannot change root directory to .");
if (0 != setuid(WWWUID))
               error("setUID failed"); /* become user www or die */
if (argc>1) strncpy(remotehost,argv[1],MAXSIZE);
else strcpy(remotehost,"nowhere");
if (argc>2) strncpy(remoteuser,argv[2],MAXSIZE);
else strcpy(remoteuser,"nobody");
if (argc>3) strncpy(daemonname,argv[4],MAXSIZE);
else strcpy(daemonname,"nodaemon");
if (argc>4) strncpy(remotedata,argv[3],MAXSIZE);
```

```
else strcpy(remotedata,"nodata");
remotehost[MAXSIZE]='\0';remoteuser[MAXSIZE]='\0';
remotedata[MAXSIZE]='\0';daemonname[MAXSIZE]='\0';
alarm(IOtime);read(0,line,MAXSIZE); line[MAXSIZE]='\0';
sscanf(line, "%s %s %s", bs1, name, bs2);
if ((name[0] != '\0') && (name[strlen(name)-1] == '\r'))
                                        name[strlen(name)-1]='\0';
if ((name[0]=='/') && ((name[1]=='\0') || (name[1]==' ')))
                                        strcpy(name,WWWDefaultFile);
if (DOCHROOT!=1)
            {strcpy(bs3,WWWDIR);strcat(bs3,name);strcpy(name,bs3);}
alarm(0);
if (((bs1[0]=='g') || (bs1[0]=='G')) &&
    ((bs1[1]=='e') || (bs1[1]=='E')) &&
    ((bs1[2]=='t') || (bs1[2]=='T'))) fetch();        /* get */
error("Unknown request");        /* all other requests fail */}
```

BIBLIOGRAPHY

[1] Michael A. Arbib and Suad Alagić. Proof rules for gotos. *Acta Informatica*, 11(2):139–148, 1979.

[2] ARIANE 5, Flight 501 Failure, Report by the Inquiry Board. http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html, July 1996. (accessed 16 January 1998).

[3] Ariane 5 report details software design errors. *Aviation Week*, pages 79–81, September 1996.

[4] John Barnes. *High Integrity Ada: The* SPARK *Approach*. Addison-Wesley, 1997.

[5] Paul E. Black, Kelly M. Hall, Michael D. Jones, Trent N. Larson, and Phillip J. Windley. A brief introduction to formal methods. In *Proceedings of the IEEE 1996 Custom Integrated Circuits Conference (CICC '96)*, pages 377–380. IEEE, 1996.

[6] Paul E. Black and Phillip J. Windley. Automatically synthesized term denotation predicates: A proof aid. In E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications (HOL '95)*, volume 971 of *Lecture Notes in Computer Science*, pages 46–57. Springer-Verlag, 1995.

[7] Paul E. Black and Phillip J. Windley. Verifying resilient software. In Ralph H. Sprague, Jr., editor, *Proceedings of the Thirtieth Hawai'i International Conference on System Sciences (HICSS-30)*, volume V, pages 262–266. IEEE Computer Science Press, January 1997.

[8] Paul E. Black and Phillip J. Windley. Formal verification of secure programs in the presence of side effects. In Ralph H. Sprague, Jr., editor, *Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences (HICSS-31)*, volume III, pages 327–334. IEEE Computer Science Press, January 1998.

[9] Hans-Juergen Boehm. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions on Programming Languages and Systems*, 7(4):637–655, October 1985.

[10] J. R. Burch, E. M. Clarke, K. L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the ACM/SIGDA International Workshop in Formal Methods in VLSI Design*. ACM, January 1991.

[11] Frederick B. Cohen. A secure world-wide-web daemon. *Computers & Security*, 15(8):707–724, 1996.

[12] Graham Collins. A proof tool for reasoning about functional programs. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs '96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 109–124. Springer-Verlag, 1996.

[13] R. J. Cunningham and M. E. J. Gilford. A note on the semantic definition of side effects. *Information Processing Letters*, 4(5):118–120, February 1976.

[14] Paul Curzon. Deriving correctness properties of compiled code. In Luc Claesen and Michael Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications*, pages 97–116. IFIP, Elsevier Science Publishers B.V., 1992.

[15] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[16] Ben L. Di Vito and Larry W. Roberts. Using formal methods to assist in the requirements analysis of the space shuttle gps change report. Contractor Report 4752, NASA Langley Research Center, August 1996.

[17] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Inc., 1976.

[18] Michael Dyer. *The Cleanroom Approach to Quality Software Development*. John Wiley & Sons, 1992.

[19] J. Kelly Flanagan. personal communication, February 1998.

[20] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[21] Michael J. C. Gordon. *Programming Language Theory and its Implementation.* Prentice-Hall, Inc., 1988.

[22] Michael J. C. Gordon and Tom F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[23] David Gries and Gary Levin. Assignment and procedure call proof rules. *ACM Transactions on Programming Languages and Systems*, 2(4):564–579, October 1980.

[24] Yuri Gurevich. Evolving algebras: An attempt to discover semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, 1993.

[25] Samuel P. Harbison and Guy L. Steele, Jr. *C, A Reference Manual.* Prentice-Hall, Inc., 1991.

[26] William L. Harrison, Karl N. Levitt, and Myla Archer. A HOL mechanization of the axiomatic semantics of a simple distributed programming language. In Luc Claesen and Michael Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications*, pages 117–126. IFIP, Elsevier Science Publishers B.V., 1992.

[27] I. Hayes. *Specification Case Studies.* Prentice Hall, 1993.

[28] Hewlett-Packard Company. *HP-UX on-line manual, HP-UX Release 9.0*, August 1992.

[29] Hewlett-Packard Company. *HP-UX on-line manual, HP-UX Release 10.10*, November 1995.

[30] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[31] John Hoffman. personal communication, April 1997.

[32] Peter Vincent Homeier. *Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures*. PhD thesis, University of California, Los Angeles, 1995.

[33] Tomasz Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7:357–360, 1977.

[34] Zohar Manna and Richard Waldinger. *Studies in Automatic Programming Logic*. Artificial Intelligence Series. Elsevier North-Holland, Inc., 1977.

[35] Zohar Manna and Richard Waldinger. The logic of computer programming. *IEEE Transactions on Software Engineering*, SE-4(3):199–229, May 1978.

[36] W. Douglas Maurer. A minimization theorem for verification conditions. In *Proc. 8th International Conference on Computing and Automation (ICCI '96)*, 1996.

[37] Michael Norrish. Derivation of verification rules for C from operational definitions. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Supplementary Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '96)*, pages 77–94, 1996.

[38] Michael Norrish. An abstract dynamic semantics for C. Technical Report 421, Computer Laboratory, University of Cambridge, May 1997.

[39] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: a prototype verification system. In Deepkar Kapur, editor, *11th Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer Verlag, June 1992.

[40] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1993.

[41] J. C. Reynolds. *The Craft of Programming.* Prentice Hall, 1981.

[42] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, pages 416–417. Prentice Hall Series in Artificial Intelligence. Prentice-Hall, Inc., 1995.

[43] Phillip J. Windley. Documentation of the `records` library. http://lal.cs.byu.edu/lal/holdoc/library/records/records.html, 1993. (accessed 20 January 1998).

Axiomatic Semantics Verification of a Secure Web Server

Paul E. Black

Department of Computer Science

Ph.D. Degree, February 1998

ABSTRACT

We formally verify that a particular web server written in C is secure, that is, a remote user cannot get files he shouldn't or change the server's files. Although the code was thoroughly reviewed and tested, the verification located some heretofore unknown behavioral weaknesses.

To verify this code, we invented new inference rules for reasoning about expressions with side effects, which occur often in C. We also formalized aspects of Unix file systems and processes, operating system and library calls, parts of the C languages, and security properties.

We propose an architecture for a software verification system which could be widely useful, and argue that our proof demonstrates that real world software written in real world languages can be verified.

COMMITTEE APPROVAL:

Phillip J. Windley
Committee Chairman

Douglas M. Campbell
Committee Member

J. Kelly Flanagan
Committee Member

William A. Barrett
Committee Member

Parris K. Egbert
Committee Member

Scott N. Woodfield
Graduate Coordinator