Inference Rules for Programming Languages with Side Effects in Expressions

Paul E. Black* and Phillip J. Windley

Computer Science Department, Brigham Young University, Provo UT 84602-6576, USA

Abstract. Much of the work on verifying software has been done on simple, often artificial, languages or subsets of existing languages to avoid difficult details. In trying to verify a secure application written in C, we have encountered and overcome some semantically complicated uses of the language.
We present inference rules for assignment statements with pre- and post-evaluation side effects and `while` loops with arbitrary pre-evaluation side effects in the test expression. We also discuss the need to abstract the semantics of program functions and present an inference rule for abstraction.

1 Introduction

In 1969 Hoare published axiomatic semantics for proving programming languages [14] which Gordon implemented in HOL [9]. More recently there has been work such as Hale's Reasoning About Software [11], Agerholm's Mechanizing Program Verification [1], Curzon's Verified Compiler [7], work on distributed systems [13, 20], embedding TLA [4, 18], etc.

Most work has been on relatively simple languages. Verifying programs written in commonly used production languages, such as C, has lagged. Production languages are generally very rich, with many overlapping features instead of a minimal set, to express different kinds of algorithms and data structures succinctly. Models which completely describe the semantics of such production languages are very complicated.

The object of this work is thttpd, a secure http daemon written in C. It is engineered to provide information to the World Wide Web and to be free of security flaws, even in the presence of a few operating system bugs or administrative errors. The code has a five page informal proof of correctness, and has been reviewed and critiqued by dozens of experts. The code for thttpd seems like an ideal candidate for a formal proof, and the proof would add real value.

In order to prove thttpd, we began with code from Harrison [13], but soon extended it with rules from Gordon [9] and wrote goal-directed tactics instead of using Harrison's forward inference functions. We also express code in the rules

* This work was sponsored by the National Science Foundation under NSF grant MIP–9412581

as an abstract syntax tree, rather than HOL strings. A parser [19] converts C code into equivalent abstract syntax trees.

Sect. 2 summarizes the design goals of thttpd. Sect. 3 presents our inference rules for pre-evaluation side effects, post-evaluation side effects, `while` loops with side effect tests, and function calls. We include an example from thttpd in Sect. 4 to demonstrate some of the complexities we encountered. Finally Sect. 5 lists some future work, and Sect. 6 is our conclusions.

2 A Secure HTTP Daemon

In June 1995, Management Analytics wrote a secure http[2] daemon, thttpd. The code is about 100 lines of C. They explain [5, 6] the goals of the daemon and why they believe it to be secure:

> The main risk to providers of [web] services is that someone might be able to fool their server software into doing something it is not supposed to do, thus allowing an attacker to break into their server ...

They continue that their

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

They then list the properties.

> The general properties of interest to us are (in order of highest to lowest priority):
> - Information Integrity - We want to assure that the information residing on the server is not corrupted by the actions of outside users as a result of their use or abuse of the secure service.
> - Availability of Service - We want to assure that outside users of the service cannot make the server unusable for other users as a result of their use or abuse of the secure service.
> - Confidentiality - We want to assure that the service only provides information to outside users that is explicitly authorized for outside access.
>
> We would also like to assure these properties to an even higher degree for information not explicitly designated for outside use than for information that is explicitly designated for outside use.

They verified the properties by argument, but we want to formally verify them. Trying to verify a program which was written in a semantically complex language, C, led us to face problems which are typically avoided such as

- side effects,
- function calls in tests, and
- function semantics abstraction.

[2] HTTP, the HyperText Transfer Protocol, is the most commonly used protocol for the World Wide Web on the Internet. HTTP is operational similar to FTP, but has been enhanced and optimized for Web interaction.

3 Inference Rules for Side Effects

We use Hoare axiomatic semantics to express the correctness of program statements. The representation for partial correctness is

$$\vdash \{\texttt{P}\}\ \texttt{code}\ \{\texttt{Q}\}$$

This means, if predicate `P` is true of the current state, executing `code` results in a state in which predicate `Q` is true.[3]

3.1 A Rule for Pre-evaluation Side Effects

The assignment axiom for `v = expr;` is

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

as long as `expr` doesn't have any side effects ([9], pp. 15-17). That is, if $\texttt{Q}^{\texttt{v}}_{\texttt{expr}}$[4] is true and the assignment is executed, `Q` will be true. Since C statements may have side effects, this does not apply. As a simple example, the semantics of `a = 2 * ++b;` is well defined [12] (it is equivalent to the compound statement `++b; a = 2 * b;`), but the statement modifies the values of `b` as well as `a`.

To reason about it, we introduce a rule to separate pre-evaluation side effects, that is, side effects which take place *before* the expression is evaluated.

$$\frac{\begin{array}{l}\vdash \texttt{SEM\_EQ}\ (\texttt{PreEval}\ \texttt{expr}\ \texttt{s1})\ \texttt{s2} \\ \vdash \{\texttt{pre}\}\ \texttt{expr};\ \{\texttt{interim}\} \\ \vdash \{\texttt{interim}\}\ \texttt{s1}\ \{\texttt{post}\}\end{array}}{\vdash \{\texttt{pre}\}\ \texttt{s2}\ \{\texttt{post}\}} \qquad (1)$$

Informally, if

- `s1` is semantically equivalent[5] to `s2` with `expr` removed and pre-evaluated,
- one can prove {`pre`} `expr`; {`interim`}, and
- one can prove {`interim`} `s1` {`post`}

one can conclude {`pre`} `s2` {`post`}.

Why add another inference rule just to separate side effects? Homeier's language, Sunrise [15], 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.

[3] That is, if `code` terminates. A total correctness theorem $\vdash$ [`P`] `code` [`Q`] (note square brackets) means if `P` is true and `code` executes, `Q` is true and `code` *always* terminates.

[4] The notation $\texttt{Q}^{\texttt{v}}_{\texttt{expr}}$ denotes `Q` with all free occurrences of `v` replaced by `expr`. For instance, $(\texttt{g} * (\texttt{h} + \texttt{1}))^{\texttt{g}}_{(\texttt{i}-\texttt{1})}$ is `((i-1)*(h+1))`.

[5] We have shallowly embedded our logic for now. That is, `SEM_EQ` is defined as $\forall$`s1 s2 . SEM_EQ s1 s2 = T` and an ML function checks equivalence and specializes the definition. Eventually we will prove this from a definitional semantics.

3.2 A Rule for Post-evaluation Side Effects

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

The following rule is similar to the pre-evaluation side effect rule (1).

$$\frac{\begin{array}{l}\vdash \texttt{SEM\_EQ}\,(\texttt{PostEval expr s1})\,\texttt{s2}\\ \vdash \{\texttt{pre}\}\,\texttt{s1}\,\{\texttt{interim}\}\\ \vdash \{\texttt{interim}\}\,\texttt{expr};\{\texttt{post}\}\end{array}}{\vdash \{\texttt{pre}\}\,\texttt{s2}\,\{\texttt{post}\}} \tag{2}$$

Briefly if

- `s1` is semantically equivalent to `s2` with `expr` removed and post-evaluated,
- one can prove {`pre`} `s1` {`interim`}, and
- one can prove {`interim`} `expr`; {`post`}

one can conclude {`pre`} `s2` {`post`}.

3.3 A Rule for `While` Loops with Side Effects

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

$$\frac{\begin{array}{l}\vdash \texttt{IS\_VALUE expr test}\\ \vdash \{\texttt{invariant} \wedge \texttt{test}\}\,\texttt{body}\,\{\texttt{invariant}\}\end{array}}{\vdash \{\texttt{invariant}\}\,\texttt{while expr body}\,\{\texttt{invariant} \wedge \,\tilde{}\,\texttt{test}\}} \tag{3}$$

IS_VALUE means that `test` is the HOL equivalent of `expr`.

When test expressions can have side effects, the rule is more complex. Note that the pre-evaluation side effect rule (1) does *not* apply. Otherwise one could prove

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

by proving

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

but the side effect is not executed every loop! (One purpose of the `SEM_EQ (PreEval ...)` condition in rule 1 is to prevent the rule from being applied incorrectly to `while` loops, `for` loops, etc.) Conceptually a `while` loop with pre-evaluation side effects must be verified as

```
while-begin-tag:
  {interim}
    pre-eval side-effect;
  {invariant}
    if (expr)
      {invariant /\ test}
        body
        goto while-begin-tag
  {invariant /\ ~test}
```

The inference rule for `while` statements is then

$$\frac{\begin{array}{l}\vdash \texttt{SEM\_EQ}\ (\texttt{PreEval sexpr testExpr;})\ \texttt{expr;} \\ \qquad \vee\ (\texttt{testExpr} = \texttt{expr}) \wedge (\texttt{sexpr} = \texttt{expr}) \\ \vdash \{\texttt{interim}\}\ \texttt{sexpr;}\ \{\texttt{invariant}\} \\ \vdash \texttt{IS\_VALUE testExpr test} \\ \vdash \{\texttt{invariant} \wedge \texttt{test}\}\ \texttt{body}\ \{\texttt{interim}\}\end{array}}{\vdash \{\texttt{interim}\}\ \texttt{while expr body}\ \{\texttt{invariant} \wedge \texttt{\~{}test}\}}$$

In other words if

- *Either* `testExpr` is semantically equivalent to `expr` with `sexpr` removed and pre-evaluated *or* `expr` is used for `sexpr` and `testExpr`,
- one can prove {`interim`} `sexpr;` {`invariant`},
- `test` is the HOL equivalent of `testExpr`, and
- one can prove {`invariant` ∧ `test`} `body` {`interim`}

one can conclude {`interim`} `while expr body` {`invariant` ∧ `~test`}.

We allow (`testExpr` = `expr`) ∧ (`sexpr` = `expr`) in the first condition in case `expr` has no side effects. Note that, when `expr` has no side effects, `testExpr` = `sexpr` = `expr` and `interim` = `invariant`, reducing this rule to the standard rule (3).

3.4 Function Call Abstraction

The program thttpd uses 15 library and operating system functions. All of them are important to the correct operation of the program. When the correctness of the program depends on these functions, how can the program be verified?

Often the approach is to essentially include the body of the called function in the code to be verified ([8], page 151). However this won't work for several reasons. First, we don't have access to the operating system code. Even if we did, thttpd should be independent of any one operating system. Second, we do not want to repeat all the work of verifying operating system or library functions every time they are used; verification would never scale up to large projects.

Finally, programmers rarely use all the functionality of an operating system call in one piece of code. So we only need partial semantics to prove the correctness of code. For instance, thttpd calls `time` to log when actions are taken.

Since the properties of interest mentioned in Sect. 2 don't include the log, returning the wrong time does not violate the top-level security policy! That is, the reliable operation of thttpd is not dependent on *which* time is returned, only that some time is. Hence we need some way to abstract the semantics of a function call. Jones [17] treats this problem by using extended type checking, but non-rigorously.

Similarly to Homeier [15] we declare axioms to express the operation of library and system functions. For example, we declare `time` as

$$\begin{array}{l} \vdash \{T\}\ \texttt{time(int}\ \ *\texttt{tloc)} \\ \quad \{(C\_Result = 0) \wedge (\exists\, TIME\_errno.errno = TIME\_errno) \vee \\ \quad\ \exists\, some\_time.(C\_Result = some\_time) \wedge C\_Result > 0 \wedge \\ \qquad {\sim}(tloc = NULL) \longrightarrow (deref(tloc) = some\_time)\} \end{array}$$

That is, either `C_Result`[6] is 0 and `errno` is set, or `C_Result` is set to some non-zero time and that time is also put in `tloc` if `tloc` is not null.

The inference rule to abstract function call semantics from a function declaration is (after [10])

$$\frac{\begin{array}{l} \vdash \texttt{DECLARE type funcName formals body} \\ \vdash \{\texttt{pre}\}\ \texttt{body}\ \{\texttt{post}\} \\ \vdash \texttt{formalsToActuals formals actuals} \\ \vdash \texttt{SUBST pre formals actuals preSub} \\ \vdash \texttt{SUBST post formals actuals postSub} \end{array}}{\vdash \{\texttt{preSub}\}\ \texttt{funcName(actuals)}\ \{\texttt{postSub}\}} \tag{4}$$

That is, if

- `funcName` is declared with `formals` and `body`,
- one can prove {`pre`} `body` {`post`},
- one can prove that the `actuals` are equivalent to the `formals`,
- `preSub` is `pre` with the `formals` replaced by the `actuals`, and
- `postSub` is `post` with the `formals` replaced by the `actuals`

one can conclude that calling `funcName` with the `actuals` in a state satisfying `preSub` results in a state satisfying `postSub`.

4 An Example From thttpd

This section presents one function from thttpd and outlines a verification showing how system calls and side effects are handled.

[6] The special variable `C_Result` is the result or return value of the function call.

4.1 Code and Operation

We present the function `logfile`. This function records an access by a remote user from a remote machine by writing the user's and machine's name along with the time to a log file. Here is the code and applicable declarations:

```
char timestamp[64], remotehost[BUFSIZE], remoteuser[BUFSIZE];

void logfile(F)
FILE *F;
{time_t t;
t=time(NULL);strftime(timestamp, 20, "%Y/%m/%d %T",localtime(&t));
fprintf(F,"%s %s %s ",remotehost,remoteuser,timestamp);}
```

In detail, the function `logfile` gets the current "time in seconds since the Epoch" [16] with the call of `time`. The call of `localtime` returns the time converted to the local time zone, and `strftime` formats the time as "`yyyy/mm/dd hh:mm:ss`" and saves it in `timestamp`. Finally `logfile` writes the `timestamp` to the file `F` along with the requesting user's name and the name of the user's computer by calling `fprintf`.

Two of the system calls used in thttpd have complicated semantics: `time`, whose semantics we gave in Sect. 3.4, and `strftime`, which is

$$\begin{aligned}
&\vdash \{T\}\ \texttt{strftime}(\texttt{char}\ *\texttt{s},\ \texttt{int maxsize},\ \texttt{char}\ *\texttt{format},\ \texttt{tm}\ *\texttt{timeptr})\\
&\quad \{(strlen(strftimeSpec(format, timeptr)) < maxsize \Rightarrow\\
&\qquad (C_Result = strlen(s) \wedge strcmp(s, strftimeSpec(format, timeptr)) = 0)\\
&\qquad \mid (C_Result = 0))\\
&\quad \wedge(\forall index.accessed(s, index) \longrightarrow index \geq 0 \wedge index < maxsize)\}
\end{aligned}$$

The function `strftime` returns the length of the string which it placed in `s`. If the string exceeds `maxsize`, zero is returned and the contents of `s` are indeterminate. Note that this is not the full semantics of `strftime`; no mention is made, for instance, of mapping from the month number and locale into a full month name for the `%B` format or the hundreds of other details of `strftime`. But this level of detail is sufficient for thttpd.

4.2 Verifying System Calls and Side Effects

We begin verifying `logfile` by using the function rule (4) with the assumption

```
!s. strlen(strftimeSpec(‘%Y/%m/%d %T‘,s)) < 20
```

(informally: any time formatted with `%Y/%m/%d %T` yields a string less than 20 characters long) and take care of the call of `time`. Then we use an inference rule for sequential statements to separate the `strftime` statement from the `fprintf`. The condition after `strftime` must be that `timestamp` has a formatted time string:

```
"strcmp(timestamp, strftimeSpec(‘%Y/%m/%d %T‘,tsptr))=0"
```

Notice that the call of `localtime` is embedded in the call of `strftime`. We separate it with the pre-evaluation side effect rule (1) and prove it with an axiom for its semantics.

Now we prove the call of `strftime`. In the following, `CALL_TAC` sets a goal to weaken the postcondition from the that given in the axiom. We rewrite with the definition of conditional ($\Rightarrow$) from `COND_CLAUSES` to simplify it. Next a selector function, `chooseStrcmp` (generated by `find_filter` [3]), selects the initial assumption (that the output is always less than the maximum size) and a rewrite solves the subgoal.

```
e(CALL_TAC SYS_strftime THEN
    ASM_REWRITE_TAC [COND_CLAUSES] THEN
    REPEAT STRIP_TAC THEN
    let chooseStrcmp (t:term) =
       (fst o dest_var o rator o rand o rator)t=`strcmp`?false in
     ASSUM_LIST (\thl .
       UNDISCH_TAC (find chooseStrcmp (map (snd o dest_thm) thl)))
    THEN ASM_REWRITE_TAC []);;
```

The proof of the final statement follows quickly and this verification is done.

5 Future Work

We plan to prove our inference rules and predicates, such as SEM_EQ, from a denotational semantic of C. We also plan to rework from the current HOL88 to HOL90.

For post-hoc verification of C programs to be practical, we must improve the various tactics so they will prove more subgoals automatically. Even better would be to change to a verification condition generator style.

We need to add inference rules to handle straight-forward array accesses. Notice that thttpd uses arrays very conservatively. We believe an approach such as [9] (page 31) will suffice.

Currently Hoare style axiomatic semantics only allow for a single postcondition. Since we are mostly concerned with partial correctness, both `return` and `exit` are modeled with a postcondition of F. That is, since control never flows from a `return` or `exit` to an immediately subsequent statements, any condition is allowed. However verifying total correctness with these jumps may be difficult. Therefore we plan to introduce multiple exit conditions as suggested in [2]. This will allow us to reason about `continue` and `break` statements in loops as well as `return` statements.

Much work is needed on function calls. How can we supply different levels of abstraction of operating system and library function calls for different needs?

Finally, we have the skeleton of the proof for thttpd almost done. With arrays we can finish the proof, but the properties proved are quite weak. We will go back through the proof with formalizations of the properties of interest and flesh out the proof. Then we believe that we can fairly claim to have formally verified the program. Management Analytics has several other servers which should be simple to prove once the infrastructure is done.

6 Conclusions

Our work on a well-engineered production program written in a complex language exposed a number of technically interesting problems, such as side effects and abstracting function semantics. We propose a number of new inference rules to handle

- pre-evaluation side effects more generally than before,
- post-evaluation side effects, and
- pre-evaluation side effects in loop tests.

We also point out the importance of abstracting the semantics of functions in doing large proofs or proofs involving operating system or library calls. Work on simple or primarily academic languages is fruitful, but verifying "industrial" languages is possible and useful. With the computer performance and theorem provers available today, the complexities found in industrial programs need not be a barrier to real verifications.

Acknowledgements

We are grateful to Dr. Frederick B. Cohen of Management Analytics for allowing and encouraging us to use thttpd as a test case for verification. We thank William L. Harrison whose code we used to begin our implementation. We also thank the reviewers that pointed out errors in our understanding of C semantics and a fatal flaw in an earlier version of the post-evaluation rule.

References

1. Sten Agerholm, "Mechanizing Program Verification in HOL," in *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and Its Applications (HOL '91)*, edited by Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, IEEE Computer Society Press, Los Alamitos, California, 1991, pp. 208–222.
2. Michael A. Arbib and Suad Alagić, "Proof Rules for Gotos," *Acta Informatica*, Vol. 11, No. 2, 1979, pp. 139–148.
3. Paul E. Black and Phillip J. Windley, "Automatically Synthesized Term Denotation Predicates: A Proof Aid," in *Higher Order Logic Theorem Proving and Its Applications (HOL '95)*, edited by E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, Springer-Verlag, Berlin, Germany, 1995, pp. 46–57.
4. Holger Busch, "A Practical Method for Reasoning about Distributed Systems in a Theorem Prover," in *Higher Order Logic Theorem Proving and Its Applications (HOL '95)*, edited by E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, Springer-Verlag, Berlin, Germany, 1995, pp. 106–121.

5. Frederick B. Cohen, "Why is thttpd Secure?" http://all.net/ManAl/white/whitepaper.html or http://all.net/ → *Products* → *Secure http and gopher daemons* (14 March 1996).
6. Frederick B. Cohen, "A Secure World Wide Web Daemon," *Computers and Security*, submitted, 1995.
7. Paul Curzon, "Deriving Correctness Properties of Compiled Code," in *Higher Order Logic Theorem Proving and Its Applications (HOL '92)*, edited by Luc Claesen and Michael Gordon, Elsevier Science Publishers, 1992, pp. 97–116.
8. Nissim Francez, *Program Verification.* Addison-Wesley, 1992.
9. Michael J. C. Gordon, *Programming Language Theory and its Implementation.* Prentice-Hall, Inc., New Jersey, 1988.
10. David Gries & Gary Levin, "Assignment and Procedure Call Proof Rules," *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 4, Oct 1980, pp. 564–579.
11. Roger Hale, "Reasoning About Software," in *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and Its Applications (HOL '91)*, edited by Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, IEEE Computer Society Press, Los Alamitos, California, 1991, pp. 52–58.
12. Samuel P. Harbison and Guy L. Steele Jr., *C, A Reference Manual.* Prentice-Hall, Inc., 1991.
13. William L. Harrison, Karl N. Levitt, Myla Archer, "A HOL Mechanization of The Axiomatic Semantics of a Simple Distributed Programming Language," in *Higher Order Logic Theorem Proving and Its Applications (HOL '92)*, edited by Luc Claesen and Michael Gordon, Elsevier Science Publishers B.V., 1992, pp. 117–126.
14. C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, October 1969, pp. 576–583.
15. Peter Vincent Homeier, *Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures*, Ph.D. Dissertation, University of California, Los Angeles, 1995.
16. HP-UX on-line manual, HP-UX Release 9.0: August 1992, Hewlett-Packard Company, Palo Alto, California.
17. Derek Jones, "Checking an Application's use of API's," http://www.knosof.co.uk/apichk.html (14 March 1996).
18. Sara Kalvala, "A Formulation of TLA in Isabelle," in *Higher Order Logic Theorem Proving and Its Applications (HOL '95)*, edited by E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, Springer-Verlag, Berlin, Germany, 1995, pp. 214–228.
19. John P. Van Tassel, "The HOL parser Library," http://lal.cs.byu.edu/lal/holdoc/library.html (13 June 1996).
20. Cui Zhang, Brian R. Becker, Mark R. Heckman, Karl Levitt, and Ron A. Olsson, "A Hierarchical Method for Reasoning about Distributed Programming Languages," in *Higher Order Logic Theorem Proving and Its Applications (HOL '95)*, edited by E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, Springer-Verlag, Berlin, Germany, 1995, pp. 385–400.

This article was processed using the LaTeX macro package with LLNCS style