Provably Sound and Secure Automatic Proving and Generation of Verification Conditions

DIDRIK LUNDBERG

Master in Computer Science
Date: December 6, 2018
Supervisor: Roberto Guanciale
Examiner: Johan Håstad
Swedish title: Tillförlitligt sund och säker automatisk generering och bevisning av verifieringsvillkor
School of Electrical Engineering and Computer Science

Abstract

Formal verification of programs can be done with the aid of an interactive theorem prover. The program to be verified is represented in an intermediate language representation inside the interactive theorem prover, after which statements and their proofs can be constructed. This is a process that can be automated to a high degree. This thesis presents a proof procedure to efficiently generate a theorem stating the weakest precondition for a program to terminate successfully in a state upon which a certain postcondition is placed. Specifically, the Poly/ML implementation of SML is used to generate a theorem in the HOL4 interactive theorem prover regarding the properties of a program written in BIR, an abstract intermediate representation of machine code used in the PROSPER project.

Sammanfattning

Bevis av säkerhetsegenskaper hos program genom formell verifiering kan göras med hjälp av interaktiva teorembevisare. Det program som skall verifieras representeras i en mellanliggande språkrepresentation inuti den interaktiva teorembevisaren, varefter påståenden kan konstrueras, som sedan bevisas. Detta är en process som kan automatiseras i hög grad. Här presenterar vi en metod för att effektivt skapa och bevisa ett teorem som visar sundheten hos den svagaste förutsättningen för att ett program avslutas framgångsrikt under ett givet postvillkor. Specifikt använder vi Poly/ML-implementationen av SML för att generera ett teorem i den interaktiva teorembevisaren HOL4 som beskriver egenskaper hos ett program i BIR, en abstrakt mellanrepresentation av maskinkod som används i PROSPER-projektet.

Acknowledgements

First, I would like to thank Mads Dam and Roberto Guanciale for introducing me to the research in the field of formal verification at KTH Royal Institute of Technology. This was the avenue which led me to write this thesis. Roberto then also took the time to supervise my work, which I am very grateful for.

I would also like to acknowledge the assistance given from Johan Håstad, Roberto Guanciale, Cathrine Bergh and Claudius Sundlöf in the form of remarks on how to improve the writing in the thesis.

Finally, special thanks to Cathrine Bergh and my parents Lars-Johan Lundberg and Lena Lundberg.

Contents

Chapter 1

Introduction

Ever since the first computer program was written, answering questions about the effect of execution of programs has been an important topic. While this can be determined through symbolic execution assuming some specific initial state, the main subject in research is more general properties. Consider the questions of whether (and how) a program can enter exceptional states, or if there are memory leaks, a particularly awful special case being leaks of confidential data in cryptographic algorithms.

Specifically, what is needed for verification are predicates on which initial states can lead to certain final states after execution. For an arbitrary program this is of course undecidable, since it is impossible to say if it will ever halt [49]. Indeed, even determining whether an arbitrary program has decidable behaviour is also an undecidable problem. Accordingly, either only specific comparatively simple cases are considered, or heuristics are used to obtain acceptable running times and memory consumption with the trade-off that false negatives and/or false positives might be obtained. In this thesis, only loop-free unstructured[1] passified programs in dynamic single assignment form which terminate after executing a finite sequence of statements will be treated (explained further in Sections 2.2.4 and 2.2.6). However, previous results have extended methods similar to the ones used here to also include loops [3][1].

[1]The structured programming paradigm forbids usage of jump statements such as `goto`, while using syntactic constructs to achieve selection (`if` statements), iteration (`while` statements) and procedure calling. While this eventually compiles to unstructured binary code, readability is improved for the programmer.

The sub-field of logic as applied to computer science dealing with the study of execution of programs is called *formal verification*, since formal methods are used to verify that code fulfils contracts which encapsulate the intent of the programmer. However, it is impossible to do any form of meaningful analysis without a representation of the object of analysis which preferably also is tailored to the properties to be proved. *Formal specification* aims to create representations of programs and systems that are amenable to analysis.

In order to ensure the actual behaviour of the program under analysis is being verified, it is necessary to start out from the binary code, the compiled version of the program. This in contrast to analyzing a program in a higher-level language, which will be compiled to binary code before execution. The compiler might change the inner workings of the program in subtle ways which might be utilized by a potential adversary. In the worst case, the compiler has bugs or has even been intentionally tampered with, as was the case with the famous XCodeGhost malware which infected Objective-C compilers. This resulted in malicious apps in Apple Store [47]. These infected apps had the capability to open phishing websites, steal device information, read from and write to the clipboard. Apart from being an embarrassment to companies developing these infected apps, this breach had both potential economic and privacy repercussions for the user. Verification of the high-level language code could not have prevented this, but verification on the binary level could have. The wider field of study of binary programs is called *binary analysis*, and includes the formal methods described above, but also numerous heuristic approaches to achieve the same results.

Today, operating advanced industrial processes is typically reliant on software. This means that any potential vulnerabilities in software could be used to alter the function of the process, triggering a meltdown of a nuclear reactor or releasing toxic compounds from chemical plants. The most well-known example of such industrial sabotage was the Stuxnet malware, which targeted and destroyed equipment in Iranian uranium enrichment facilities [32]. Raising the investment threshold for performing such an attack by using more rigorous software verification means restricting usage of these methods to nations with well-funded and highly specialized intelligence agencies. Removing the ability of rogue actors to perform industrial sabotage, potentially leading to ecological disasters, could be considered a way of contribut-

ing to a more robust and sustainable society.

Consider the program *prog*. The first requirement of formal verification is making, or obtaining, the syntax and semantics of the language *prog* is written in. Secondly, a *contract* is defined - a formal specification of the property to be examined. It is then possible to prove whether *prog* satisfies the contract or not. However, study of *prog* exactly as-is might require taking into account a large syntax, which is only there to provide syntactic sugar in the form of additional options and abbreviations for the programmer. Therefore, the standard procedure is to transpile *prog* to a simplified (in terms of the size of the syntax) *intermediate language*, preserving the properties of interest. This transpilation can assure correctness by supplying a machine-checkable proof for each transpiled output, as in the work by Metere, Lindner and Guanciale [39].

The programs treated in this thesis are written in BIR - a language defined inside the theorem prover HOL4. BIR is shorthand for "BAP Intermediate Representation", a language defined inside the ITP (interactive theorem prover) HOL4 inspired by the internal representation of programs of the BAP (Binary Analysis Platform) toolkit [7]. In Chapter 3, a method to automatically generate proven Hoare triples using HOL4 for programs written in HOL4 BIR is described. This enables verification by checking if the program fulfils contracts by comparing a generated Hoare triple to a contractual Hoare triple, using HOL4 and possibly also an SMT (satisfiability modulo theories) solver like Z3 [10]. These concepts will be described in more detail in the following section.

Chapter 2

Background

2.1 Related Work

In 1910, the first part of the *Principia Mathematica* of Bertrand Russell and Alfred North Whitehead was published [51]. This was a bold attempt at describing a minimal set of axioms and inference rules, and then deriving all of mathematics from them. Sadly, because of the unwieldiness of the work and the pace of development in the field of logic at the time (in particular formal grammar, as noted by Kurt Gödel [20]), it quickly grew out of relevance other than as a historical milestone. It could be said the advent of the interactive theorem prover is the next step on the quest to formalize all mathematical knowledge. There are several important advancements: most importantly, the storage of theorems in structured, digital formats allow for proof automation using the metalanguage of the ITP. In this thesis, the main contribution to the sum of formalized knowledge stored for usage in ITPs is a theory of weakest preconditions of BIR statements formulated as Hoare triples.

Interactive theorem provers, also known as proof-checking programs and theorem proving assistants, have been in development since Robin Milner developed the first version of the LCF (Logic for Computable Functions) prover in 1972 [40] for the Stanford Artificial Intelligence Project conducted under the auspices of NASA. As the LCF prover quickly rose from verifying small hardware components to become powerful enough to prove properties of complex algorithms, the tool for assisting in single proofs evolved slowly over time to instead systematically save and categorize theorems for later use [22]. In this

way, modern theorem proving assistants have accumulated a wealth of theories, all of which are built from definitions and axioms of the foundational logic. The successor to LCF is known as HOL, and is competing with a dozen other theorem proving assistants, of which perhaps Coq (launched in 1989) [13], Isabelle (launched in 1986) [44] and ACL2 (launched in 1996) [6] are the most well-known alternatives.

Theorem proving assistants have both been used to construct completely new and surprising proofs and to poke holes in existing ones. In the 1930s, E.V. Huntington and Herbert Robbins suggested two different bases for Boolean algebra: while both included commutativity and associativity they differed on a third equation describing the action of the complement operation. The conjecture that Huntington's third equation could be derived from Robbins' three became known as the Robbins conjecture. Despite efforts by Robbins, Huntington and Alfred Tarski, the Robbins conjecture was not proved until 1996 by William McCune, using the EQP automated theorem prover [38]. The original proof which was presented in a very human-unfriendly form has since been summarized in a more readable format by other authors [15][37], showing that even inscrutable computer-assisted proofs can be arranged in a form understandable to humans with some effort.

In 1998, Thomas Hales announced a proof of the Kepler conjecture [25] - a conjecture stating that the face-centred cubic and hexagonally close-packed lattices maximize the percentage of space filled by equally-sized spheres. This proof relied on exhaustive computation performed by programs he had written in Java. Sadly, this proof was not reasonable to verify for humans due to the massive amount of output - the proof submitted for publication consisted of 250 pages which summarized 3 GB of programs and results. The 12 referees appointed by the Annals of Mathematics could not in the space of four years arrive at a certain conclusion with regard to the veracity of the proof, but decided to allow it to be published anyway. In response to this, Hales would depart on an even lengthier journey to formalize his proof inside the Isabelle and HOL Light ITPs. This would yield a proof checkable to agree with a small set of trusted axioms, inference rules and definitions, removing the need for human referees. In 2014, the formalized proof was complete [26]. However, in the process of formalization Hales had discovered several errors in the reasoning of his earlier proof, justifying the scepticism of the referees (but the general proof strategy, originally suggested by Fejes Tóth in 1953 [48],

still proved sound). As increasing power of computers enables mathematical proofs reliant on extensive combinatorial checking of cases, ITPs have a larger role in safeguarding against errors where humans cannot.

Due to their history of development by computer scientists, the use of theorem proving assistants in areas of computer science such as hardware and software verification is widespread: ITPs have been used to verify aspects of commercial hardware as well as critical software such as ECC algorithms since the early 1990s [28]. Usage of ITPs in other areas heavy in mathematics, such as theoretical physics, is unknown to the author.

The concept of the weakest precondition was first introduced by Edsger Dijkstra in 1975 [12]. The most notable extensions to his simple predicate transformer semantics are extensions to loops using loop invariants [23] and to concurrent programming [31].

The paper which perhaps was the foremost inspiration when writing this thesis describes a similar project using the theorem proving assistant Coq [50]. The main difference lies in the two languages used to represent the program being verified. The intermediate language of Vogels et al. (here denoted simply by IL to preserve the terminology in their paper) is on the extreme structured end of languages: it does not feature any type of analogue to `goto` statements pointing back in the program other than pure loops, in contrast to BIR with its `Jmp` and `CJmp` statements. Languages like Java completely without `goto` statements instead rely on function calls and other constructs to mimic `goto` usage, something which is also absent in IL. This simplification means that weakest preconditions generated by branching statements become simpler (as do the proofs of their correctness). If the intent would instead be to create an intermediate representation for concrete, practical programs, an unstructured approach would prevent the code duplication required to present arbitrary programs in a structured form.

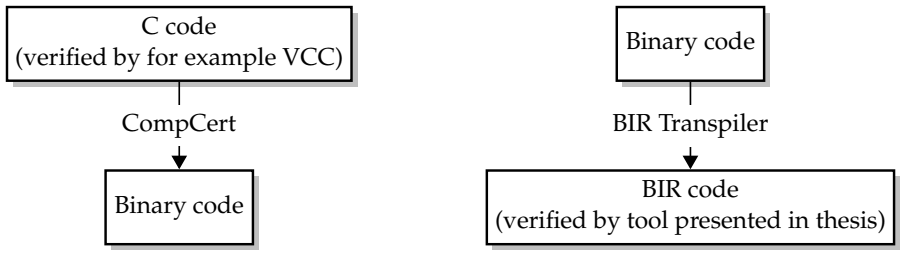
The tools for verification described in this thesis forms the final step in a verification workflow together with the ARMv8 to BIR transpiler of Metere, Lindner and Guanciale [39]. This means that this is demonstrably useful for verification of actual ARMv8 binary code. In contrast, the work of Vogels et al. is not connected to any particular transpiler and could be used for proofs related to abstract algorithms rather than concrete programs.

There is also a fundamental difference in computation as the method of Vogels et al. is *verified* - a function inside the proof assistant logic - the method presented here is *verifying*, using the metalanguage to compute the weakest precondition. While the end result is the equivalent theorems, using hand-tailored metalanguage for computation may increase efficiency, in particular with regards to time consumption. However, Vogels et al. do not provide any quantitative hints to the performance of their algorithms.

There also exist methods for program verification which use an entirely different workflow compared to the one described in this thesis, something which can be very advantageous in certain situations. Consider the case of designing a program using some high-level language. The program is required to have certain properties, but it is not critical exactly how the final binary looks as long as these properties can be guaranteed. Then, verifying these properties using the compiled binary might not be an ideal approach, since it can be difficult to understand what changes to make in the high-level language code based on an analysis of the machine code.

A solution to this is to verify the properties in the high-level language before it is compiled. Then, use a compiler which preserves these properties. As a consequence of this workflow, it is not necessary to bother with understanding how the source code relates to the binary and trying to fine-tune the former using the latter. The drawback of this method is that it is not possible to use it for binaries written in several languages - in particular, the combination of C and assembly which is found in many drivers and hypervisors. In any case, when trying to create a tool to verify C with in-line assembly the advantage of working with the potentially well-structured high-level language would be nullified, since all the intricacies of the assembly code would have to be taken into account.

Figure 2.1 is a comparison between the BIR workflow and the one described in the two paragraphs above, concretely exemplified by VCC and CompCert. VCC is a tool for verification of C code launched by Microsoft Research in 2009 [9]. VCC works by transpiling annotated C code to the BoogiePL intermediate language and then letting the program verifier Boogie [2] generate verification conditions which are handed to the Z3 SMT solver. It has been used in practice to verify aspects of the Hyper-V hypervisor [33] as well as parts of the PikeOS kernel [4].



(a) Source code verification, then certified compilation.

(b) Lifting, then verification of intermediate representation.

Figure 2.1: Comparison between different workflows when verifying binary code.

CompCert is a certified compiler for C to PowerPC, ARM, RISC-V and x86 assembly. It is written inside Coq, using Coq proofs of the semantic equivalence between C code and the resulting assembly code. Currently, CompCert is compatible with almost all of ISO C99, and extending the amount of C syntax supported by CompCert is an ongoing effort led by Xavier Leroy at INRIA [35].

There are also more heuristically-oriented tools which aim to solve similar problems but do so without producing exportable, independently verifiable proofs. Valgrind [41] is a tool which can help detect issues with memory safety. Angr is a newly-developed tool which uses fuzzing and dynamic symbolic execution to detect actionable exploits of programs [47]. Choosing heuristics over a theorem prover means lowering the least needed time for detecting errors at the expense of completeness. The trust base of a theorem prover with exportable, independently verifiable proofs is the axioms and inference rules (which must be consistent), while the trust base of a heuristic tool is the programmer of the tool. This means that while descriptions of specific exploits are just as useful when they come from a heuristic, the absence of exploits in the result of a heuristic cannot be considered reliable. The approach suggested in this thesis is, therefore, to prefer over heuristics when analysing critical but comparatively small code segments, such as cryptographic algorithms, while tools such as Angr and Valgrind is to prefer for obtaining concrete actionable exploits from large binaries, rather than proofs of their absence.

2.2 Preliminaries

Here, descriptions are given of a few central concepts this thesis revolves around.

2.2.1 Interactive Theorem Prover

Also known as a *proof assistant*, the *interactive theorem prover* (ITP) is a program which assists with formal proofs. An *automated theorem prover* is slightly different since it allows no guidance from the user during the construction of the proof, but the end product is the same. At the core of an interactive theorem prover is a formal system; it includes a set of symbols, a grammar to decide whether formulae of these symbols are well-formed or not, a set of axioms (and/or axiom schemata), and inference rules which map premises onto conclusions. Building on this very small core (around ten axioms and about as many primitive inference rules), it is possible to make additional definitions and prove properties of formulae constructed from these definitions. The resulting theorems are then stored in theories, which are used in turn to build new theories.

A theorem is a proven proposition. The well-formedness of propositions is guaranteed: propositions can only be formed according to grammatic rules. Every proposition can be viewed as a syntax tree consisting of variables and symbols as leaves, and every branching a grammatically valid application of some operator.

A theorem is a similar derivation tree which can only have axioms and definitions as leaves, and branches are *inference rules* instead of grammar rules. An inference rule maps axioms, definitions and theorems onto new theorems. When constructing a proof, inference rules are successively applied until the resulting theorem is the one to be proved. Of course, when doing so, the rich flora of already existing theorems is utilized, minimizing the time it would take to prove everything from the raw foundations. An abstract example of a theorem is shown in Figure 2.2. Actual theorems are far more complex, but the figure conveys the basic notion of the theorem as a tree-like structure with axioms and definitions as leaves.

Since different theorem provers use different formal systems, theorems are not trivially portable between them. However, recently developed tools like OpenTheory allow porting theorems between ITPs,

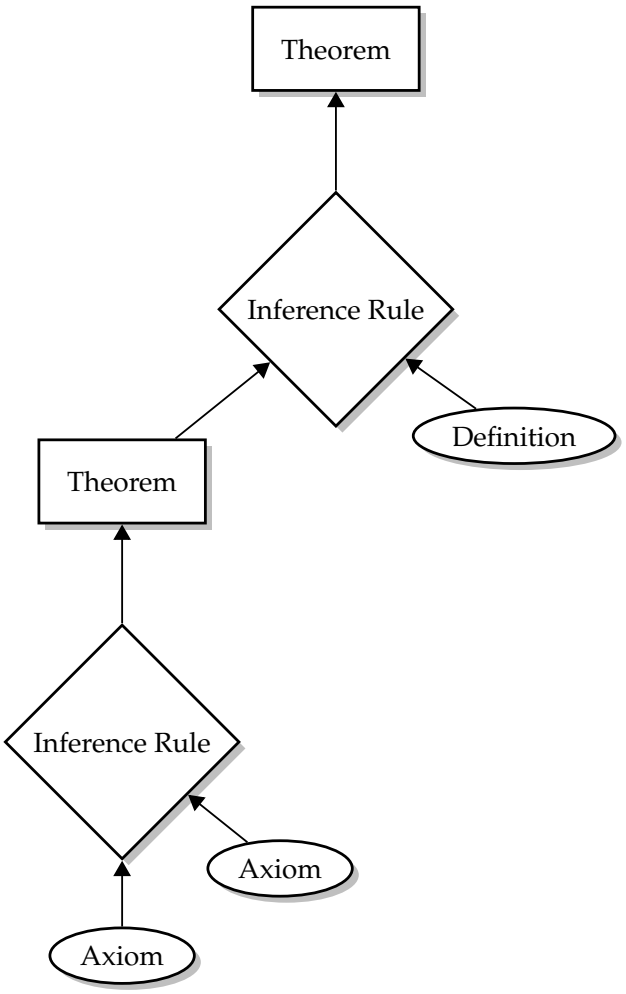


Figure 2.2: Example structure of a theorem.

something which can increase the reliability of the usage of ITPs in general, eliminating the reliance on a single ITP [29].

This thesis has used the ITP HOL4. The choice of HOL4 was solely based on the fact that it is used in the local research group, and in the group of Anthony Fox in Cambridge working in related areas. However, HOL4 is quite widespread and provided as free software, which will enable others to easily build upon the work described in this thesis. The choice of ITP is mostly a practical matter. Throughout this text, HOL4 will be referred to simply as "HOL". HOL is an abbreviation standing for "Higher-Order Logic", and using the same name for the ITP and the type of logic it uses is a point of confusion inherited from the predecessor of HOL, LCF (Logic of Computable Functions).

Other logics, such as Zermelo-Fraenkel set theory, can be used with other ITPs. A particularly important clarification is that if the reader should explore any of the articles referenced in this thesis "HOL/Isabelle" does not mean "the proof assistant HOL or the proof assistant Isabelle" but rather "the proof assistant Isabelle, using higher-order logic".

2.2.2 Hoare triples and verification

A *Hoare triple* (HT) is a logical abbreviation which is helpful when reasoning about the effect of execution of programs [17][27]. Consider some formal language L with an accompanying small-step semantics defining the effects of execution. Out of this language, construct a program C: a list of statements in L. Furthermore consider two runtime states S_1 and S_2, structures holding all information about the current execution - most importantly, the status (is the state currently running, or has it somehow terminated?), the program counter and the values of initialized variables.

A Hoare triple can be written as

$$\{P\}\, C\, \{Q\}\ , \tag{2.1}$$

meaning that execution of C from any state S_1 for which P holds will always end up in some state S_2 for which Q holds. In addition, if P is not only a precondition but the *weakest precondition*[1] of C and Q, then no state not fulfilling P can lead to S_2 fulfilling Q by execution of C.

In program verification by means of the weakest precondition, the programmer will formulate a contract Hoare triple

$$\{P_p\}\, C\, \{Q\}\ , \tag{2.2}$$

which the goal is to prove. Given C and Q, the weakest precondition WP is generated and

$$\{WP\}\, C\, \{Q\} \tag{2.3}$$

is proved to hold. Then, the theorem stating the consequence rule is used:

$$\begin{aligned} &\{WP\}\, C\, \{Q\} \implies \\ &(P_p \implies WP) \implies \\ &\{P_p\}\, C\, \{Q\}\ , \end{aligned} \tag{2.4}$$

[1] If $A \Rightarrow B$, then A is a *stronger* predicate implying the *weaker* predicate B.

observing that as the first antecedent is already proved, it is left to prove $(P_p \implies WP)$, which would give the result that $\{P_p\}\, C\, \{Q\}$ holds.

Detailing how WP is computed, let the weakest-precondition predicate transformer be a function wp mapping a statement $stmt$ and one postcondition Q for every statement execution of $stmt$ can lead to onto a weakest precondition WP.

To compute wp in general, weakest preconditions of the different types of statements in L must be known, and also a rule of sequential composition.

Commonly, a distinction is made between *partial correctness* and *total correctness*: total correctness is described in the Hoare triple definition above, and the weakest precondition is known as *weakest conservative precondition* (same as the wp in this thesis). Partial correctness only requires that Q holds in S_2 if execution of C terminates at all, and the weakest precondition under that criterion is known as the *weakest liberal precondition*. Since only loop-free programs are considered in this thesis (and termination does not become an issue for computability), this distinction is not relevant for the final resulting Hoare triple produced by the proof procedure, which technically will be referring to total correctness. However, there are usages for the wlp even in loop-free programs, such as in more efficiently generating weakest preconditions for programs with conditional jumps [16]. There is a lot more to say about how termination is treated in the HOL Hoare triple definitions later on in Section 2.2.4.

Note that since the starting point is a known postcondition from which it is possible to reason about the potential states before execution, the above is a method which goes through the program backwards, in the opposite direction of execution. It is of course also possible to start with a known precondition, and reason about what is the *strongest postcondition* after execution - the most specific, limiting constraint on the final state S_2 [11]. This is an equally legitimate way of program verification, however using the weakest preconditions is more common. Which one is preferred might depend on whether the weakest precondition or strongest postcondition predicate transformer semantics is the easiest to state formally.

2.2.3 Proof procedure

A *proof procedure* is a function which yields a theorem. Specifically, this thesis will discuss *verifying procedures*, which are opposed to *verified procedures* in the sense that the former use the metalanguage of an ITP, while the latter are functions inside the logic of the ITP itself.

Using a verifying procedure instead of a verified procedure has the advantage of being able to design the exact computation in detail, potentially achieving greater efficiency and greater ease of integration with other tools. However, also note that many ITPs can generate code in functional languages from verified procedures inside the theorem prover logic, a contemporary summary of which is given in [24].

2.2.4 BIR

BIR is shorthand for *BAP Intermediate Representation*[2] and is an abstract representation of programming language meant to be amenable to formal analysis. In practice, no programs are originally written in BIR, but rather programs written in other programming languages are compiled and then transpiled from machine code into BIR. The syntax and semantics of BIR are defined in HOL as part of earlier work by Roberto Metere and Thomas Türk. This means that it is possible to formally describe properties regarding BIR programs in HOL, prove them and treat them as any other HOL theorems.

A BIR program consists of a list of BIR blocks, where every block consists of one BIR label (a block name in string format), one list of basic BIR statements and one BIR end statement. Please note that in general, a BIR block may have either a label with a string name or an address consisting of an immediate value and any of these may be used to jump to the block. In this thesis, only labels with block names in string format will be considered.

The effects of execution are kept track of inside the BIR state, which consists of a program counter, a variable environment with information about declared variables and the current status (for example, whether an assumption has been violated, if execution has failed or execution is running along normally). The program counter points to the statement about to be executed using a block label and a statement index. Upon execution, if termination does not occur, the statement index of

[2] *Binary Analysis Platform* is a toolkit with heuristics for program verification [7].

the program counter is incremented, or the label changed and the index set to 0 if a jump from the current block has occurred (for example, while executing `Jmp` or `CJmp`).

The BIR variable environment is a map between string names of variables to tuples of type and value. Values can be either memories or immediate values.

All statuses apart from Running imply termination, which means that further execution will have no effect on the state. These other statuses are used frequently in the report, so the prefixes used in their formal definitions will be left out and they will instead be colloquially referred to as just Failed, Halted, JumpOutside, and AssumptionViolated. Their descriptions are given in Table 2.1.

Execution proceeds by applying the effects on the BIR state of the basic BIR statements (`Assume` and `Assert` - there are more, but not in the passified version of the language presented here) sequentially, transitioning between blocks by means of the end statements (`Halt`, `Jmp` and `CJmp`). Note that inside HOL (and inside HOL theorems recounted verbatim later on) these statements will be preceded by prefixes, which are discarded in the main body of text of this thesis for brevity. A BIR program might have several entry points, in the sense that there are several blocks not pointed to by any other block, and it might also have several endpoints (blocks ending with `Halt`).

Additionally, the `Observe`, `Assign` and `Declare` statements are part of the BIR syntax but will not be treated here, since they are assumed to have been removed through the passification process (described in Section 2.2.6) before analysis of the BIR program starts.

`Assert` and `Assume` both take BIR expressions as arguments. A BIR expression can be a range of typical arithmetic expressions with values of varying types, as well as casts between types. These will be glossed over in this report since they are only of indirect importance when computing the WP, but two values require a special mention: BIR True and BIR False, which are represented inside BIR as binary 1 and 0, respectively (henceforth only referred to as "True" and "False" for brevity).

`Assert` will evaluate a BIR expression and set the state status to Failed if the asserted BIR expression does not evaluate to True. Similarly, `Assume` will evaluate a BIR expression and set the state status to AssumptionViolated if the assumed BIR expression evaluates to False (if it is neither True nor False, the status will be set to Failed).

Status	Description
Running (`BST_Running`)	When in this status, execution will proceed as normal.
Failed (`BST_Failed`)	Will occur when the argument to `Assert` does not evaluate to True, or when any BIR expression provided as argument to any statement is not well-typed (not counting `Halt`).
Halted (`BST_Halted`)	Will occur when executing the `Halt` statement.
JumpOutside (`BST_JumpOutside`)	Will occur when executing `Jmp` or `CJmp` and being instructed to jump to a label not found in the program.
AssumptionViolated (`BST_AssumptionViolated`)	Will occur when argument to `Assume` evaluates to False.

Table 2.1: BIR statuses

There are four other ways evaluation of a BIR expression can change state status when executing `Assume` or `Assert`: firstly, if the expression is not well-typed (types of the parameters of a binary operator do not match as needed, for example, addition of a 1-bit number to an 8-bit number) the resulting value will be BIR Unknown (henceforth referred to as Unknown), and state status will always be set to Failed. The same will happen if any variable is not correctly typed (the value stored has a different type from the variable type) or if it has not yet been initialized or declared. Declaration of variables normally happens through the `Declare` statement - in this thesis the precondition of the Hoare triple will instead pass along a constraint that the encountered variables have already been declared and initialized earlier on in the program.

2.2.5 The WP Predicate Transformer Semantics of BIR

Consider a BIR statement s. For every such statement, a function $wp(s, Q)$ can be defined which maps a statement s and a postcondi-

tion Q onto the weakest precondition of the state satisfying Q under the execution of s. The resulting weakest precondition P is, just like Q, a condition on a state (specifically, a BIR expression which should evaluate to True in the state): accordingly, it can be passed on in recursion. As a result, for sequential concatenation of the statements s_1 and s_2 the following must hold for wp:

$$wp(s_1; s_2, Q) = wp(s_1, (wp(s_2, Q))) \ . \tag{2.5}$$

To resolve this recursion, it remains to discuss the WPs of the statements themselves. These WPs are BIR expressions.

Assert

The `Assert (`e`)` basic statement in BIR has only one parameter, which is the BIR expression e. In executing the `Assert` statement, e will be evaluated at the current state. If e evaluates to True, the program execution will continue without having affected the state other than incrementing the program counter. Otherwise, execution of the program will immediately terminate with the state status set to Failed.

Consider the Hoare triple $\{P\}\, C\, \{Q\}$. Depending on which Hoare triple variant is used, the postcondition will either say that the final state S_2 has status Halted, or that the program counter of S_2 is pointing to the statement after C (in the order of symbolic execution). Clearly, if the code C inside the Hoare triple is `Assert (`e`)`, it is a necessary precondition that e evaluated to True, or else the program would have terminated with a failure, setting state status to Failed and never reaching the state fulfilling Q (which says that S_2 must either be at the statement after C, or be Halted). The only other requirement of the postcondition is that Q itself must hold. Since `Assert` does not change the variables in S_1 in any way, if Q must hold in S_2, then Q must also hold in S_1. The preconditions presented are not only necessary but also sufficient since they will always lead to the postcondition being fulfilled in S_2. Thus, it must hold that

$$wp(\texttt{Assert (}e\texttt{)}, Q) = e \wedge Q \ , \tag{2.6}$$

where the $\wedge$ represents a BIR conjunction, making the right-hand side a BIR expression.

Assume

The `Assume(`e`)` basic statement is superficially similar to `Assert`. Semantically, the difference between the two is that upon evaluation of the BIR expression e, the status will be set to AssumptionViolated (not Failed) if e evaluates to False. Accordingly, from the perspective of symbolic execution, AssumptionViolated seems rather equivalent to Failed - both will terminate if their argument does not evaluate to True (with different state statuses). However, due to the way Hoare triples are defined (in Section 3.1) the statuses are quite different from the perspective of formal verification. Since the Hoare triples allow beginning or ending in AssumptionViolated, this status effectively becomes a pseudo-Running state, where every assertion holds: if status is AssumptionViolated in S_1, that status will remain after execution of any amount of steps and so remains in S_2, which means the postcondition will be fulfilled - there is never any need to consider any assumptions or assertions in between, something which could be said to imply the statement "every assertion holds".

The `Assume` statement is nothing found in any actual programming language - it is only used in formal analysis. Roughly speaking, what the statement and the Hoare triple logic regarding it is conveying is that e is introduced as an assumption for the rest of the current execution path. Accordingly, if e evaluates to False in the state where `Assume(`e`)` is executed, a contradiction has been introduced: it has been assumed that False equals True. Since anything can be derived from a contradiction, this implies that whether the postcondition Q holds does not matter. In other words, only if e evaluates to true must Q hold afterwards.

$$wp(\texttt{Assume}\,(e)\,, Q) = (e \Rightarrow Q)\ , \tag{2.7}$$

Halt

The `Halt(`e`)` end statement is perhaps the simplest one to treat. Execution of `Halt(`e`)` will just evaluate the expression e and then terminate execution of the program by setting the status of the state to Halted with the evaluation of e as an accompanying termination code. The `Halt` statement can also never fail to terminate execution and end up in a state with any status other than Halted. Accordingly, in the related proofs there is no need to account for whether or not the vari-

ables of e are initialized, or even if e is not well-typed. It follows that the WP of the `Halt` statement is simply the postcondition Q.

$$wp(\texttt{Halt}\,(e)\,, Q) = Q\ . \tag{2.8}$$

Jump

The `Jmp` (l) end statement in BIR has as sole argument a BIR label l of a BIR block. Upon execution of the `Jmp` statement, the program counter will move to the first basic instruction of that block - if it exists, otherwise it will terminate with the status "JumpOutside l". The weakest precondition of `Jmp` is, like the `Halt` statement, trivial under the assumption that there are no jumps to labels not in the program.

$$wp(\texttt{Jmp}\,(l)\,, Q) = Q\ . \tag{2.9}$$

Looking back at Equation 2.2.5, note that it is implicit that the next statement after the `Jmp` is the first statement of the block with the label l. Only programs with unique block labels will be treated in this thesis. In practice, if there would exist multiple blocks with identical labels, the BIR semantics will always just choose the first it finds.

Conditional Jump

The `CJmp` (e, l_1, l_2) end statement has three parameters: the expression e and the two labels l_1 and l_2. Conditional jumps work in much the same way as ordinary jumps, with the slight difference that the conditional jump has different targets depending on the evaluation of e. If e evaluates to True, then the target is l_1, if False it is l_2, and if Unknown then status will be set to Failed and execution will terminate.

Intuitively, because of the two different execution branches proceeding from a `CJmp` statement, the WP actually must have *two* postconditions Q_1 and Q_2, corresponding to jumping to l_1 and l_2. Accordingly the WP becomes

$$wp(\texttt{CJmp}\,(e, l_1, l_2)\,, Q_1, Q_2) = ((e \implies Q_1) \wedge (\neg e \implies Q_2))\ . \tag{2.10}$$

The fact that this has a different number of arguments from the other WPs is no issue since any explicit WP function is never defined in the formal proofs, nor in the proof procedures - if it eases the reader's mind the wp above can be thought of as a new function.

$$\texttt{Assign}\ (y, x + x)$$
$$\texttt{Assign}\ (z, y + y)$$

Figure 2.3: A BIR code snippet suitable for passification.

Typically, in other predicate transformer semantics, the equivalent of the `CJmp` statement is illustrated by a "choice" statement which takes two sub-programs as parameters and executes both. The logic of the condition can then be modelled with assumptions of e and $\neg e$ in the two branches, respectively, which is logically equivalent to a conditional jump.

2.2.6 Passification and single assignment forms

Consider the WP predicate transformer semantics described in Section 2.2.5. The semantics is missing any means to set the value of a variable, or to declare the type of a variable - the `Declare` and `Assign` statements exist in the full BIR, but not in the passive subset treated in this thesis. *Passive* statements are the subset of statements which do not affect the variable environment. *Passification* is the process in which a program is rewritten to a passive form which is equivalent from the perspective of Hoare triples. To motivate why passification is useful, it is first required to understand the weakest precondition of `Assign`.

Since `Assign` (var, exp) changes the value of the variable var to the evaluation of exp in the current state, when generating the weakest precondition for an `Assign` statement all instances of evaluations of var in the WP must be replaced by exp. Now consider the code snippet in Figure 2.3 with the postcondition z. Generating the weakest precondition would yield $y + y$ in the first step, and then $x + x + x + x$ - an exponential increase in size.

The solution to this issue is to change all `Assign` statements to equivalent `Assume` statements. It is perhaps closest at hand to compare `Assign` (var, exp) to `Assume` $var = exp$. Using the example code snippet in Figure 2.3, the weakest precondition would instead be $y = x + x \implies (z = y + y \implies z)$, adding the now passified assignments as assumptions to the weakest precondition in accordance with the WP predicate transformer semantics outlined in Section 2.2.5.

$$\texttt{Assign}\ (z, 2)$$
$$\texttt{Assign}\ (z, z + 2)$$

Figure 2.4: A BIR code snippet not suitable for passification.

If these assumptions are used to substitute their left-hand side in the postcondition directly, the same WP is obtained as for `Assign`. Suddenly there is now only a linear growth of the generated WP.

As it turns out, passification is not so easy. Consider the example in Figure 2.4. Again starting out from the postcondition $z = 4$, the generated weakest precondition can be described as $WP_{\neg pass} \equiv 4 = 4 \equiv \mathrm{True}$ for the non-passified program, but after passification, instead it is obtained that $WP_{pass} \equiv z = 2 \implies (z = z + 2 \implies z = 4)$, which is a nonsensical precondition. Since $z = z + 2$ is a contradiction, it is possible to show that $WP_{pass} \equiv \mathrm{True}$ regardless of the other components. This would yield the WP True for all programs with incrementation of variables. How is it possible to deal with this issue?

The solution is called *single-assignment form*: if every variable is assigned to only once, there will be no contradictions like $z = z+2$ in the weakest precondition, and also never different choices for which value to substitute for a variable. There are several types of single assignment form, the simplest of which is *static single assignment* (SSA) [45] - meaning that every variable is assigned to at most once anywhere in the code. In *dynamic single assignment* (DSA) [14], every variable is assigned to at most once on each execution path. Transformations to single assignment forms were first introduced in the field of compiler optimization around 1990 and found their way to formal verification of software a decade later. The most recent single-assignment technique was introduced in 2005 [1], which further helped reduce the number of variable versions.

What are the other issues with regard to computation? None of the expressions takes a lot of time to compute, so any improvements to reduce time complexity seems unlikely, however, the size of the generated weakest precondition WP may also grow exponentially for nested `CJmp` statements. This is the worst-case situation for the size of WP: none of the other passive statements can produce exponential growth of WP, so by reducing the growth of WP due to `CJmp` to linear,

it would be possible to make the size complexity of WP linear overall. As shown by Flanagan and Saxe in 2001 [16], this is actually possible, if the program for which the WP is generated is passified.

Assume a program to verify in passified form. Every symbolic execution step can either end up in an exceptional state[3], at which point regular execution further on is not guaranteed, or in a non-exceptional state. At this point, it is necessary to introduce the different concepts of *weakest conservative precondition* (wp) and *weakest liberal precondition* (wlp). The $wlp(C, Q)$ is a predicate on a state preceding execution of C which describes exactly all states which upon execution ends up obeying Q, possibly becoming exceptional along the way. The weakest conservative precondition does not allow for exceptional states; for example, a program which always enters an exceptional state will simply have the weakest conservative precondition of False. Now, it clearly holds that

$$wp(s, Q) = wp(s, \mathrm{True}) \wedge wlp(s, Q) \ , \tag{2.11}$$

since $wp(s, \mathrm{True})$ evaluates to True or False depending on whether or not s can enter an exceptional state, and $wlp(s, Q)$ encapsulates the predicate on the preceding states regardless of exceptionality. This is the first step, but obviously not enough, since the exponential increase will remain in the wlp.

Consider the equation

$$wlp(s, Q) = wlp(s, \mathrm{False}) \vee Q \ . \tag{2.12}$$

If this held, in combination with Equation 2.11, it is obtained that

$$wp(s, Q) = wp(s, \mathrm{False}) \wedge (wlp(s, \mathrm{False}) \vee Q) \tag{2.13}$$

which would be very useful, since Q is not duplicated when computing the weakest precondition of statements which can be preceded by several statements. However, this is not true for all statements s. Those statements which modify the state upon which Q is a predicate do not obey this equation: the declarations and assignments, which modify the value of a state variable. Accordingly, this simplification can be used in passified programs [16] [34].

[3]For example, through the assignment of malformed expressions. This is however dependent on the semantics of the language in question, which will not be examined in greater detail here.

Chapter 3

Method

This chapter describes the specific implementation of the theoretical concepts outlined previously, and how this is used to verify actual programs. Sections 3.1 and 3.2 describe the implementation of the Hoare triple first introduced in Section 2.2.2.

First, weakest precondition soundness theorems for all individual statements are proved (Section 3.3). They relate an arbitrary postcondition Q to the corresponding weakest precondition WP for the statement. Note that only a soundness theorem and not a completeness theorem is provided. Thus, it is proved that what is claimed as a weakest precondition WP is a sound precondition, but it is not proved formally in HOL that WP is also the actual weakest precondition - the theoretical arguments are however given in Section 2.2.5.

When the WP soundness theorems (WPSTs) of all statements are proved, the next step is to prove composition theorems for the different Hoare triple variants (Section 3.4). Assume the program $C_1; C_2$ and two Hoare triples $\{P\}\, C_1\, \{Q\}$ and $\{Q\}\, C_2\, \{R\}$. These Hoare triples are called *adjacent*, since the first describes execution ending in a state where the second begins execution. The composition theorems show how to obtain a single HT $\{P\}\, C_1; C_2\, \{R\}$ which describes the joint execution of the code in both previous HTs. As a technical aside, note that the WPST for the `CJmp` statement is actually modelled as a conjunction of two `Jmp` WPSTs, thus also making the corresponding composition theorem somewhat of an exception to the others.

After these theorems have been proved, there remains to construct a function which takes a concrete BIR program, a concrete postcondition in the form of a BIR expression, and then computes a concrete

HT for the entire program. In fact, several HTs might be computed - one for each block - since the entry point of a BIR program is an arbitrary block. This function is the proof procedure `get_ht_of_program` (Section 3.5).

The last part is the actual verification step: at this point, the goal is to prove that the contractual Hoare triple provided by the verifier holds, given the HT proved by `get_ht_of_program`. This is done in the same fashion as described previously in Section 2.2.2. The HOL theorem used for this purpose is described in Section 3.6.

3.1 The Hoare triple

As explained in Section 2.2.2, a Hoare triple $\{P\}\, C\, \{Q\}$ is a predicate on the effect of execution of the code segment C, and can be translated into natural language as "if the *precondition* P holds before execution of C, then the *postcondition* Q holds after execution of C". When translating the Hoare triple to HOL, the natural first step is to formulate the requirement of the precondition holding, as seen in Figure 3.2.

Since this is the first HOL definition shown, some explanation regarding how they are presented might be necessary. HOL definitions are, in general, stated in a form where arguments to the term to be defined are universally quantified, then the term (given the universally quantified arguments) is stated to be equivalent to its definition - an expression which can feature additional quantifiers. While the dot after universal and existential quantification is just a separator, a dot after variables followed by an entry name means that the variable is in fact a record (a tuple with named fields), with the dot picking out the record entry with the name on its right-hand side. For the BIR state described in Section 2.2.4, bst_pc is the program counter, bst_status is the status and bst_environ the variable environment. The program counter in turn has the entries bpc_label for the block label and bpc_index for the index.

First, it is required by `bir_ht_precond_holds` that the precondition P holds in the initial state S_1 of the Hoare triple. This is captured with the statement

$$\texttt{bir\_prop\_true}\ P\ S_1.\text{bst\_environ}\ . \tag{3.1}$$

There is actually one other case which should be accepted: the state where an assumption has been violated, the "top" exceptional state $\top$.

In a state with status AssumptionViolated, every proposition can be considered to hold.

Then, any other status than Running or AssumptionViolated (Failed, JumpOutside or Halted) is explicitly forbidden by `bir_is_valid_status` (shown in Figure 3.1), since this would make the program terminate. It is also required that variables be initialized (through `bir_env_vars_are_initialised`) and that the environment is well-typed (through `bir_is_well_typed_env`). The use of passing along initialized variables will first be made clear in Sections 3.3.3 and 3.6, but note already that there is a separate set of variables for those from the initial postcondition.

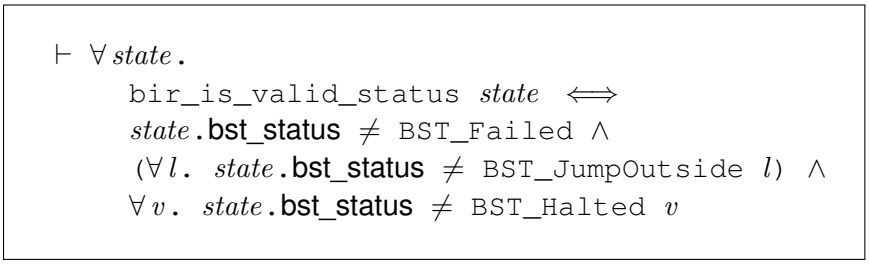


Figure 3.1: `bir_is_valid_status`

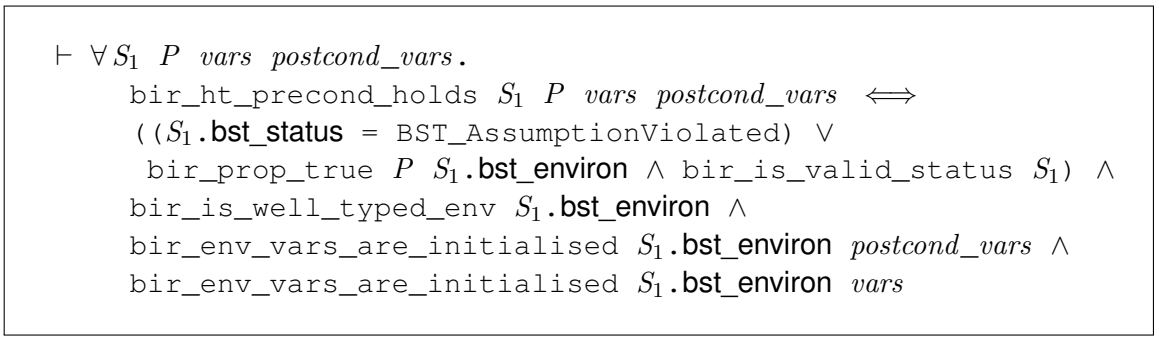


Figure 3.2: `bir_ht_precond_holds`

The postcondition is handled similarly to the precondition, as seen in Figure 3.3.

For the purposes of proving the HTs of individual statements, and for composing them, it is not necessary to keep track of variable initialization. However, to perform the verification step - to prove the middle antecedent in the theorem shown in Figure 3.14 on page 46 the

```
⊢ ∀ S2 Q vars postcond_vars.
    bir_ht_halt_postcond_holds S2 Q vars postcond_vars ⟺
    ((S2.bst_status = BST_AssumptionViolated) ∨
     bir_prop_true Q S2.bst_environ ∧
     ∃ hcode. S2.bst_status = BST_Halted hcode) ∧
    bir_is_well_typed_env S2.bst_environ ∧
    bir_env_vars_are_initialised S2.bst_environ postcond_vars ∧
    bir_env_vars_are_initialised S2.bst_environ vars
```

Figure 3.3: `bir_ht_halt_postcond_holds`

set of initialised variables is used to translate BIR disjunctions to HOL disjunctions.

One could question the requirement of variables to be initialized in the postcondition. Note that BIR does not contain any mechanism to un-initialize variables - the requirement is therefore always fulfilled. Variables being initialized after execution is used in the proofs of the theorems on combination of adjacent Hoare triples - the choice between having a lemma on impossibility of un-initialization and keeping a predicate on variable initialization in the definitions is cosmetic, but the current approach is more robust with regards to future inclusion of a `Havoc` statement undoing initialization of variables.

Another approach which could be used in place of variable initialisation would be to keep track of which variables must not evaluate to Unknown inside the `Assume` and `CJmp` expressions. This would require a theorem stating that if a variable does not evaluate to Unknown, it also would not evaluate to Unknown in a previous execution state - a property which clearly holds for a passified program.

Using the definitions in Figure 3.2 and 3.3, the regular Hoare triple - here named `bir_ht_halt_n_holds` - is defined as seen in Figure 3.4 for the BIR program $prog$, the precondition p, the postcondition q, the program counter pc and the sets of variables *vars* and *postcond_vars*. The pre- and postconditions p and q are both BIR expressions[1]. The

[1]Note that it is not explicitly required that p and q be Boolean expressions at this point: If they are not, then the definition would hold just as well, since anything can be derived from the contradiction resulting from the assertion inside `bir_ht_precond_holds` that they must be True. The restriction to Boolean expressions will

```
⊢ ∀prog P Q pc vars postcond_vars.
    bir_ht_halt_n_holds prog P Q pc vars postcond_vars ⟺
    ∀S1 S2.
      (S1.bst_pc = pc) ⇒
      bir_ht_precond_holds S1 P vars postcond_vars ⇒
      ∃n.
        (S2 = bir_exec_step_n_state prog S1 n) ⇒
        bir_ht_halt_postcond_holds S2 Q vars postcond_vars
```

Figure 3.4: `bir_ht_halt_n_holds`

program counter points to where in *prog* C from the Hoare triple starts - C then ends wherever execution reaches a Halted state (which can only happen through termination with the Halt statement). Note that this allows for execution to end at multiple different points, which can be the case after a conditional jump.

In summary, `bir_ht_precond_holds` summarizes the pre-condition holding and `bir_ht_halt_postcond_holds` the post-condition holding, while S_1.bst_pc $=$ pc and the application of `bir_exec_step_n_state` determine the execution of C.

3.2 The n-step Hoare triple

Other than execution which ends whenever a Halted state is reached, it is of interest to consider the execution of specific numbers of steps inside blocks and out from blocks for use in the proof procedures. The Hoare triple shown in Figure 3.5 represents execution of exactly n steps from the statement with index i_1 in block l_1 to the statement with index i_2 in block l_2. `bir_ht_postcond_holds` is defined in Figure 3.6 and differs from the halt Hoare triple in Section 3.1 firstly in that it does not require non-failing execution to end in termination by Halt (in fact, halting is explicitly forbidden by `bir_is_valid_status`), secondly by the constraint on the program counter which must end up on the position determined by l_2 and i_1 after n execution steps.

only first come into play inside the weakest precondition theorems for the individual statements.

```
⊢ ∀prog P Q l1 i1 l2 i2 vars postcond_vars n.
    bir_ht_n_holds prog P Q l1 i1 l2 i2 vars postcond_vars n ⟺
    ∀S1 S2.
      (S1.bst_pc = <|bpc_label := l1; bpc_index := i1|>) ⇒
      bir_ht_precond_holds S1 P vars postcond_vars ⇒
      (S2 = bir_exec_step_n_state prog S1 n) ⇒
      bir_ht_postcond_holds S2 Q l2 i2 vars postcond_vars
```

Figure 3.5: `bir_ht_n_holds`

```
⊢ ∀S2 Q l2 i2 vars postcond_vars.
    bir_ht_postcond_holds S2 Q l2 i2 vars postcond_vars ⟺
    ((S2.bst_status = BST_AssumptionViolated) ∨
     bir_prop_true Q S2.bst_environ ∧ bir_is_valid_status S2 ∧
     (S2.bst_pc.bpc_index = i2) ∧ (S2.bst_pc.bpc_label = l2)) ∧
    bir_is_well_typed_env S2.bst_environ ∧
    bir_env_vars_are_initialised S2.bst_environ postcond_vars ∧
    bir_env_vars_are_initialised S2.bst_environ vars
```

Figure 3.6: `bir_ht_postcond_holds`

3.3 Weakest precondition soundness theorems

After the previous section has provided the tools to formulate Hoare triples, the next step is to relate concrete post- and preconditions to specific BIR statements. This section will examine all the statements in a passified BIR program and formulate theorems stating the soundness of their weakest preconditions as presented in Section 2.2.5.

A theorem stating *soundness* of a precondition expresses that the precondition holding in the initial state implies the postcondition holding in the final state, after execution. A *completeness* theorem, on the other hand, would state that the precondition implies all other sound preconditions. This terminology is borrowed from terms applied to proof or type systems in logic. In more general terms, sound-

ness can be thought of as the property guaranteeing no false positives (no contradiction presented as truth), while the completeness property guarantees no false negatives (no truth presented as a contradiction).

Consider the effects of these properties on the verification step: without soundness, the generated WP in the HT would allow for initial states from which execution would not lead to states fulfilling the postcondition. If the contractual precondition is sound, this would not make any difference. However, it would be possible to also supply unsound preconditions and prove contract HTs which in fact do not hold. For example, the result that some program is functionally correct when it is not, or that forbidden areas of memory are untouched when they are not. Lack of completeness would yield the opposite problem: some contracts would be impossible to prove when they in fact hold. This would at worst cause delays (until the error is found) or cancellations of work, while the former situation might lead to the deployment of unsafe systems. In addition, the weakest precondition soundness theorems of the individual statements are used by the proof procedures when constructing Hoare triples for entire programs.

The proofs have mostly been split up on the basis of properties, meaning they are easier to follow for humans but might require more lines in HOL compared to a proof which directly uses the totality of the effect of execution.

Although the proofs are different for each statement, this is the general outline: first rely on showing that the postcondition of the HT holds using the preconditions as assumptions, and then use the semantics of the statements themselves to see how properties of the initial state carry over to the final state. Figure 3.3 and 3.6 give an idea of what is needed to prove.

Two of the properties necessary to prove are required by all postcondition variants and true under execution for all statements in BIR: that well-typedness is kept over execution, and that initialization of variables is kept as well. This means that in BIR, there is no way to uninitialize variables, nor to make the variable environment badly typed through execution. These theorems are found in Figure A.9 (on page 69).

Then there is one property which is required by all postcondition variants, but which holds only under execution of any passive statements: that all expressions evaluating to True needs to keep holding under execution. This property holds for all passive statements since

they cannot change the variable environment, which could make an expression previously evaluating to True evaluate to False. This theorem is shown in Figure A.10 (on page 70).

Having accounted for those properties, the proofs start to diverge. For the WP soundness theorem of `Halt` it is required that the final state always has status `Halted`. For the others a valid status is required (as defined by `bir_is_valid_status_def`) - Running or AssumptionViolated. In addition, `Assume` requires explicitly that status of final state be set to AssumptionViolated if `Assume` argument evaluates to False.

`Assert` and `Assume` require the program counter to move to the next statement in the block if the status of the initial state was Running and the argument of the statement evaluates to True. `Jmp` requires the program counter to point to the start of the target block of the jump if initial state status was Running, and `CJmp` similarly requires the program counter to point to two different blocks depending on whether the condition evaluates to True or False.

3.3.1 Trivial WP soundness theorem

This theorem describes execution of zero steps, and is used when generating Hoare triples for empty blocks.

Theorem 3.3.1 (`bir_0_step_wp`)**.** *The Hoare triple* `bir_ht_n_holds` *is always true for* 0 *execution steps, unaltered program counter and precondition equal to postcondition:*

$$\vdash \forall\, prog\ \ P\ \ l_1\ \ i_1\ \ vars\ \ postcond\_vars\,.$$
$$\texttt{bir\_ht\_n\_holds}\ \ prog\ \ P\ \ P\ \ l_1\ \ i_1\ \ l_1\ \ (i_1\ +\ 0)\ \ vars$$
$$postcond\_vars\ \ 0$$

Figure 3.7: `bir_0_step_wp`

Proof. The Hoare triple definition `bir_ht_n_holds` is expanded. Execution of zero steps yields a final state equal to the initial state. Then, for the same state (and corresponding program counter), condition, sets of initialized variables, and making zero execution steps, the precondition holding implies the postcondition holding (as stated

by bir_ht_pre_impl_post in Figure A.1 on page 65), which completes the proof. □

3.3.2 Assert WP soundness theorem

This theorem is used when generating Hoare triples for Assert statements. The proof is structured around lemmata of required properties of the Assert statement.

Theorem 3.3.2 (bir_assert_wp)**.** *If the current statement is* Assert *exp and the postcondition is Q, then the one-step Hoare triple holds with the precondition $exp \wedge Q$:*

```
⊢ ∀Q exp l1 i1 vars postcond_vars prog.
    (bir_get_current_statement prog
       <|bpc_label := l1; bpc_index := i1|> =
     SOME (BStmtB (BStmt_Assert exp))) ⇒
    bir_ht_n_holds prog (BExp_BinExp BIExp_And exp Q) Q l1 i1
      l1 (i1 + 1) vars postcond_vars 1
```

Figure 3.8: bir_assert_wp

Proof. The Hoare triple and Hoare precondition definitions is expanded, and bir_and_equiv (Figure A.6, page 67) is used for equivalence between the BIR conjunction and HOL conjunction.

Then, consider the cases of two statuses of the initial state S_1: AssumptionViolated and not AssumptionViolated. In case the status is AssumptionViolated, then by bir_exec_step_n_state_unchanged (in Figure A.4 on page 66) it remains so in S_2, and so the conclusion is proved by expanding the Hoare triple postcondition and seeing that the AssumptionViolated clause is fulfilled.

If status of S_1 is not AssumptionViolated, then by the principle of exclusion it must be Running, since the current assumption that the status of S_1 is valid forbids all other cases. Refer to the HT postcondition (as defined in Figure 3.6) to see exactly what properties must be proved.

1. **Variables remain initialized**: Given by bir_varinit_invar_n in Figure A.8 on page 69.

2. **Environment remains well-typed**: Given by `bir_welltypedness_invar_n` in Figure A.9 on page 69.
3. Q **keeps holding**: Given by `bir_prop_true_invar_pass_n1` in Figure A.11 on page 70.
4. **Valid status is preserved**: Given by `bir_assert_valid_status` in Figure A.12 on page 71.
5. pc **is incremented by one, and stays in the same block**: Given by `bir_assert_pc` in Figure A.13 on page 71.

With these properties proved, the postcondition in the goal holds, and the proof is complete. □

3.3.3 Assume WP soundness theorem

This theorem is used when generating Hoare triples for `Assert` statements. The proof structure is similar to that of `Assert`, apart from the consequences of there being three possibilities for next status when evaluating *exp*.

Theorem 3.3.3 (`bir_assume_wp`)**.** *If the current statement is* `Assume` *exp and the postcondition is* Q*, and if exp is a Boolean expression, then the one-step Hoare triple holds with the precondition* $exp \implies Q$*:*

```
⊢ ∀Q exp l1 i1 vars postcond_vars prog.
    (bir_get_current_statement prog
       <|bpc_label := l1; bpc_index := i1|> =
     SOME (BStmtB (BStmt_Assume exp))) ⇒
    bir_is_bool_exp exp ⇒
    bir_ht_n_holds prog
      (BExp_BinExp BIExp_Or (BExp_UnaryExp BIExp_Not exp) Q) Q
      l1 i1 l1 (i1 + 1) vars postcond_vars 1
```

Figure 3.9: `bir_assume_wp`

Proof. First, expand the Hoare triple and Hoare triple precondition definitions.

In case the status of the initial state is AssumptionViolated, the proof follows along the same lines as the `Assert` WP soundness theorem in Section 3.3.2.

If status of S_1 is not AssumptionViolated, then by the principle of exclusion it must be Running, since the current assumption that the status of S_1 is valid forbids all other cases. Refer to the HT postcondition (as defined in Figure 3.6) to see what properties must be proved.

Three of these properties can be proved directly:

1. **Variables remain initialized**: Given by `bir_varinit_invar_n` in Figure A.8 on page 69.

2. **Environment remains well-typed**: Given by `bir_welltypedness_invar_n` in Figure A.9 on page A.9.

3. **Valid status is preserved**: Using the fact that all subsets of the set of initialized variables are also initialized, the proof is given by `bir_assume_valid_status` in Figure A.14 on page 72.

It remains to show that for all possible cases, either execution ends up in AssumptionViolated, or Q keeps holding and the program counter is incremented by one. To proceed, consider the different values of exp in the initial state as they relate to execution and the WP:

1. $\neg exp$: status will be set to AssumptionViolated by execution, and proof follows by `bir_assume_violated` (Figure A.16, page 74).

2. exp: since $exp \implies Q$ also holds in the initial state, Q must hold there as well. Then, proof follows by `bir_prop_true_invar_pass_n1` (Figure A.11) and `bir_assume_pc` (Figure A.15, page 73).

Note that considering only the values True and False of exp and not Unknown is possible since the antecedents in `bir_assume_wp` state that exp is a Boolean expression, and furthermore the precondition states that a disjunction of $\neg exp$ and Q holds, which together with the assumption of well-typedness of the environment implies that exp cannot evaluate to Unknown. □

3.3.4 Halt WP soundness theorem

This theorem is used when generating Hoare triples for `Halt` statements. The proof differs significantly from the previous ones since it states that the `bir_ht_halt_n_holds` HT holds, and not the `bir_ht_n_holds` HT as before.

Theorem 3.3.4 (`bir_halt_wp`)**.** *If the current statement is `Halt`, then the Hoare triple `bir_ht_halt_n_holds` holds for the pre- and postcondition Q:*

```
⊢ ∀Q exp pc vars postcond_vars prog.
    (bir_get_current_statement prog pc =
     SOME (BStmtE (BStmt_Halt exp))) ⇒
    bir_ht_halt_n_holds prog Q Q pc vars postcond_vars
```

Figure 3.10: `bir_halt_wp`

Proof. Expand the Hoare triple and Hoare triple precondition definitions, and specify the existentially quantified step number in the definition of `bir_ht_halt_postcond_holds` (Figure 3.3) as 1 - with the current statement being `Halt`, only one step is needed to fulfil the postcondition.

In case the status of the initial state is AssumptionViolated, the proof follows along the same lines as the `Assert` WP soundness theorem in Section 3.3.2.

If status of S_1 is not AssumptionViolated, then by the principle of exclusion it must be Running, since the current assumption that the status of S_1 is valid forbids all other cases. Refer to the HT postcondition (as defined in Figure 3.3) to see what properties must be proved.

1. **Variables remain initialized**: Given by `bir_varinit_invar_n` in Figure A.8 on page 69.
2. **Environment remains well-typed**: Given by `bir_welltypedness_invar_n` in Figure A.9 on page 69.
3. Q **keeps holding**: Given by `bir_prop_true_invar_pass_n1` in Figure A.11 on page 70.

4. **`Halt` always halts**: Given by `bir_halt_halts` in Figure A.17 on page 74.

□

3.3.5 Jump WP soundness theorem

This theorem is used when generating Hoare triples for `Jmp` statements. The proof differs from the previous ones since it involves transitions between blocks. It is also important to note that while the argument to `Jmp` can in general be either a block label or an expression evaluating to a block address, only the former is considered in this thesis.

Theorem 3.3.5 (`bir_jmp_wp`)**.** *If the current statement is `Jmp` pointing to a block with label $label$, and if there is a block in the program with label $label$, then the one-step Hoare triple holds for the pre- and postcondition Q with the final program counter pointing to the first statement in the block with label $label$:*

```
⊢ ∀Q l1 i1 vars postcond_vars prog label.
    (bir_get_current_statement prog
       <|bpc_label := l1; bpc_index := i1|> =
     SOME (BStmtE (BStmt_Jmp (BLE_Label label)))) ⇒
    MEM label (bir_labels_of_program prog) ⇒
    bir_ht_n_holds prog Q Q l1 i1 label 0 vars postcond_vars 1
```

Figure 3.11: `bir_jmp_wp`

Proof. Expand the Hoare triple (Figure 3.5) and Hoare triple precondition definitions.

In case the status of the initial state is AssumptionViolated, the proof follows along the same lines as the `Assert` WP soundness theorem in Section 3.3.2.

If status of S_1 is not AssumptionViolated, then by the principle of exclusion it must be Running, since the current assumption that the status of S_1 is valid forbids all other cases. Refer to the HT postcondition (as defined in Figure 3.6) to see what properties must be proved.

1. **Variables remain initialized**: Given by `bir_varinit_invar_n` in Figure A.8 on page 69.
2. **Environment remains well-typed**: Given by `bir_welltypedness_invar_n` in Figure A.9 on page 69.
3. Q **keeps holding**: Given by `bir_prop_true_invar_pass_n1` in Figure A.11 on page 70.
4. **Valid status is preserved**: Given by `bir_jmp_valid_status` in Figure A.18 on page 75.
5. **Jump reaches target**: Given by `bir_jmp_target` in Figure A.19 on page 75.

□

3.3.6 Conditional jump WP soundness theorem

This theorem is used when generating Hoare triples for `CJmp` statements. The proof differs significantly from the previous ones since the conclusion states that a conjunction of `bir_ht_n_holds` HTs hold, one for each target block of the `CJmp` statement.

Consider the different forms the weakest precondition of a `CJmp` statement might take. Since for Boolean values A, B and C it holds that

$$(A \wedge B) \vee (\neg A \wedge C) \equiv (\neg A \vee B) \wedge (A \vee C) \ ,$$

it would be possible to write the `CJmp` statement weakest precondition (with C as the jump condition, and B and C as the preconditions of the Hoare triples whose execution starts at the jump targets) as either the right-hand or left-hand side of the above equation. Now, the proofs of soundness for these weakest preconditions would actually be similar in difficulty. If a case-split is performed on A, it is left to prove B with A as an assumption and C with $\neg B$ as an assumption in either case. In the theorem shown in Figure 3.12, a disjunction of conjunctions is the chosen form of writing the weakest precondition of a `CJmp` statement.

Theorem 3.3.6 (`bir_cjmp_wps`)**.** *If the current statement is a* `CJmp` *with Boolean condition* $cond$*, first label* $label_1$ *and second label* $label_2$*, and if* $cond$ *is a Boolean expression, and if both labels are are in the program* $prog$*, then*

both the 1-step Hoare triple `bir_ht_n_holds` *with a precondition consisting of a BIR conjunction of* $cond$ *and* Q_1*, the postcondition* Q_1 *and the jump target label* $label_1$ *and the 1-step Hoare triple* `bir_ht_n_holds` *with the precondition consisting of a BIR conjunction of* $\neg cond$ *and* Q_2*, the postcondition* Q_2*, and the jump target label* $label_2$ *hold:*

```
⊢ ∀vars postcond_vars Q1 Q2 l1 i1 prog cond label1 label2.
    (bir_get_current_statement prog
       <|bpc_label := l1; bpc_index := i1|> =
     SOME
       (BStmtE
          (BStmt_CJmp cond (BLE_Label label1)
             (BLE_Label label2)))) ⇒
    bir_is_bool_exp cond ⇒
    MEM label1 (bir_labels_of_program prog) ⇒
    MEM label2 (bir_labels_of_program prog) ⇒
    bir_ht_n_holds prog (BExp_BinExp BIExp_And cond Q1) Q1 l1
      i1 label1 0 vars postcond_vars 1 ∧
    bir_ht_n_holds prog
      (BExp_BinExp BIExp_And (BExp_UnaryExp BIExp_Not cond) Q2)
      Q2 l1 i1 label2 0 vars postcond_vars 1
```

Figure 3.12: `bir_cjmp_wps`

Proof. Expand the Hoare triple (Figure 3.5) and Hoare triple precondition definitions. Then, use `bir_and_equiv` to convert the BIR conjunctions to HOL conjunctions.

Since the conclusion is a conjunction, it is necessary to prove that both of the HTs hold. The first part of the proofs for the two HTs is similar:

In case the status of the initial state is AssumptionViolated, the proof follows along the same lines as the `Assert` WP soundness theorem in Section 3.3.2.

If status of S_1 is not AssumptionViolated, then by the principle of exclusion it must be Running, since the current assumption that the status of S_1 is valid forbids all other cases. Refer to the HT postcondition (as defined in Figure 3.6) to see what properties must be proved.

1. **Variables remain initialized**: Given by `bir_varinit_invar_n` in Figure A.8 on page 69.
2. **Environment remains well-typed**: Given by `bir_welltypedness_invar_n` in Figure A.9 on page 69.
3. **Valid status is preserved**: Given by `bir_cjmp_valid_status` in Figure A.20 on page 76.

From this point on, the proofs diverge. For the first HT, it is required that Q_1 keeps holding and that `CJmp` always jumps to the block with label $label_1$ if the condition holds. For the second one, it is required that Q_2 keeps holding and that `CJmp` always jumps to the block with label $label_1$ if the condition does not hold:

1. Q_1 **or** Q_2 **keeps holding**: Given by `bir_prop_true_invar_pass_n1` in Figure A.11 on page 70.
2. **Conditional jump always hits correct target**: Given by `bir_cjmp_target1` (in Figure A.21, page 77) and `bir_cjmp_target2` (in Figure A.22, page 78).

□

3.4 Hoare triple composition theorems

In the previous section, proof sketches have been given - tracing the lines of the proofs in HOL - for theorems stating the soundness of weakest preconditions of individual statements. However, the topic of interest is entire programs, and so the following theorems, which prove the soundness of Hoare triples composed out of combinations of already proven adjacent Hoare triples, are needed. The notion of adjacency here means that the program counter in one of the HTs points to the place where it ends up after the execution described in the other one. The composition theorems are written so that the pre-existing Hoare triples to be composed are simply antecedents with the resulting Hoare triple being the conclusion, making for simple usage later on in the proof procedures.

First, recall the two different Hoare triple variants: n-step Hoare triples describing n execution steps and halt Hoare triples describing

execution until `Halt` is reached. The end product of the proof procedure proving HTs will be one halt HT for every block in the program, which describes execution from the start of that block until termination by `Halt`. The following composition theorems are used in the process:

1. n**-step and halt Hoare triple composition theorem** (described in Appendix A.6.1): Result of composition is one halt HT. Used when building the halt HT for a block ending in `Halt`, or when generating a HT for the complete block after one of the following two composition theorems for end statements of a block has been used.

2. **Jump and halt Hoare triple composition theorem** (described in Appendix A.6.2): Result of composition is one halt HT. Used together with `Jmp` WPST and halt HT of `Jmp` target block to start generating HT for jump blocks.

3. **Conditional jump and halt Hoare triple composition theorem** (described in Appendix A.6.3): Result of composition is one halt HT. Used together with `CJmp` WPST and halt HTs of `CJmp` target blocks to start generating HT for conditional jump blocks. Note that since the `CJmp` WP is a conjunction of two HTs, this theorem technically composes two HTs with one halt HT.

3.5 Proof procedures

All of the previous theorems can be thought of as building blocks. There is still to describe the procedures which obtain the Hoare triples for actual programs and postconditions, using these building blocks. These procedures are written in SML - the metalanguage of HOL. "Metalanguage" here means that while HOL is the language the theorems are stated in, SML is the language used to combine and manipulate the theorems with axioms and definitions to prove further results, based on the primitive inference rules of HOL.

3.5.1 Proving the HT of a BIR program

Recall that a BIR program is a list of BIR blocks, which in turn consist of a list of basic statements and one ending statement. To prove HTs

for an entire program, the proof procedure get_ht_of_program is used.

To prove a Hoare triple for execution of a block b, a postcondition is also needed. Specifically, to construct a halt HT of the type seen in Figure 3.4, it is also required that b ends with a `Halt` statement. Alternatively, a HT stating the effect of execution from the start of b until termination could be proved. This would require another HT stating the effect of execution from after b until `Halt` is reached for all execution paths, in which case the precondition of this HT becomes the postcondition supplied when proving the HT for b, after which the two are combined using a composition theorem.

Just like a BIR program can have multiple endpoints where termination with `Halt` occurs, it can also have multiple *entry points*. Strictly speaking, any block whatsoever may be the entry point of the program: it is only dependent on where the initial program counter points. BIR programs are represented as a list of blocks, whose ordering is entirely arbitrary[2].

It is therefore ambiguous what is meant by "getting the HT of a program". In this context, it signifies proving the halt HTs of all blocks. get_ht_of_program returns a dictionary between block labels and corresponding HTs. Then, the halt HT corresponding to the entry point of interest can be obtained by looking up the label of the entry point block.

get_ht_of_program first puts all the block labels in a list l_1: for each block label bl, the function will check if a Hoare triple has been generated for the block labels jumped to from the block bl. If not, bl is put in another list l_2 with block labels to check later on and go to the next block label in l_1. If yes, a new HT is generated for the bl block from the one already describing execution to the end of the program from immediately after the block bl, and l_2 is concatenated to l_1 forming the new l_1, after which l_1 is searched again from the start in the same fashion. Note that if a block ending in `Halt` is encountered, no other previously obtained HTs are required to generate the block HT. `Jmp` blocks require HT of one subsequent block, while `CJmp` often require two HTs.

The generation of WPs and proof of HTs for different block types is outsourced to functions get_ht_of_halt_block, get_ht_of_jmp_block and get_ht_of_cjmp_block.

[2]Note that since the ordering of the list does not matter, and since programs with duplicate BIR block labels are not considered, the list is functionally a set.

The dictionary returned by get_ht_of_program (as well as the set representations used internally) uses applicative maps implemented using Okasaki-style Red-Black trees [43], courtesy of Ken Friis Larsen.

3.5.2 Proving the HT of a BIR program: Example

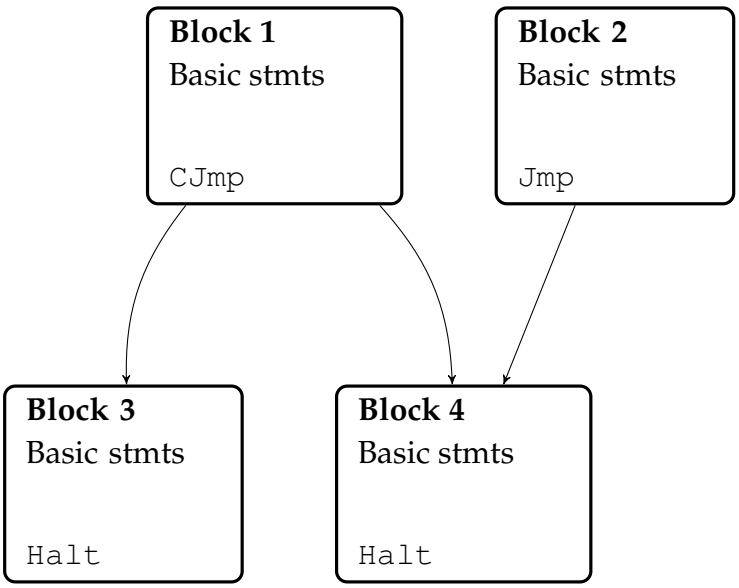


Figure 3.13: Example BIR program

For a concrete example of get_ht_of_program showcasing usage of all three composition theorems, consider applying get_ht_of_program on the program in Figure 3.13 and some postcondition Q. Assume that in the program - the list of blocks - the blocks are sorted by their numbers in ascending order. As a consequence of this, first a Hoare triple for block 3 would be generated since this is the first block in this order which does not require any HTs to have been generated before.

Since block 3 is a block ending with `Halt`, get_ht_of_halt_block (described in Section 3.5.3) is called. The WPST of `Halt` (shown in Figure 3.10) is specialized and becomes the initial HT of only the ending statement. After this, the n-step + halt HT composition theorem (shown in Figure A.23 on page 79) is used to consecutively add execution of the basic statements in block 3 to the initial Hoare triple, one by one. This results in HT_3, a Hoare triple describing execution from the start of block 3 until termination by `Halt`.

Next, all the blocks would be searched again. Note that with only HT_3 it is not yet possible to prove halt HTs for blocks 1 and 2, after which block 4 remains. Block 4 has no prerequisites for proving a halt HT (apart from Q, of course) and so a Hoare triple HT_4 is proved in the exact same fashion as for block 3, using get_ht_of_halt_block.

Now looking at the blocks again, it becomes possible to prove a halt HT for block 1, since HTs have been proved for every block reachable after execution of block 1. This HT is generated by get_ht_of_cjmp_-block (described in Section 3.5.3) first using the WPST of `CJmp` (shown in Figure 3.12) to obtain a conjunction of two HTs, after which the conditional jump + halt HT composition theorem (shown in Figure A.25 on page 82) is used together with HT_3 and HT_4 to obtain a halt HT onto which the basic statements in block 1 are subsequently added, in the same fashion as for blocks 3 and 4.

Now only block 2 remains, and the prerequisite for proving a halt HT of block 2 is the halt HT of block 4, which is already computed. get_ht_of_jmp_block (described in Section 3.5.3) is called, and the WPST of `Jmp` is used to obtain a HT of the ending statement. This is then composed with the halt HT of block 2 using the jump + halt HT composition theorem (shown in Figure A.24 on page 81). The HTs of the basic statements of the block are then consecutively added onto this in the same fashion as for the other blocks.

At this point, computation is finished and the dictionary mapping block labels to the halt HTs HT_1, HT_2, HT_3 and HT_4 is the return value of get_ht_of_program.

3.5.3 Proving the HT of a BIR block

Here, the proof procedures for obtaining halt HTs of individual blocks are described.

Note that while HTs for blocks ending in `Halt` can be proved using only a postcondition Q, those ending in jumps require HTs to have been proved already for every block the ending statement can jump to.

get_ht_of_halt_block

This proof procedure returns a theorem stating the halt Hoare triple of a block ending in `Halt`. It takes as arguments a postcondition $postcond$ (a BIR expression), a program $prog$ and a block label bl. The weakest

precondition is generated along the way. The structure is roughly the following:

1. Prove the halt Hoare triple $halt\_stmt\_ht$ of the `Halt` statement at the end of the block bl in $prog$ (recall that the WP of `Halt` is simply the postcondition).

2. Starting with the last basic statement, successively generate WPs for and prove corresponding HTs for the basic statements of the block bl. As the HTs are proved, compose them together with $halt\_stmt\_ht$, using the composition theorem for n-step and halt Hoare triples to ultimately form one halt HT describing execution of the entire block.

get_ht_of_jmp_block

This proof procedure returns a theorem stating the halt Hoare triple HT_{Jmp} of a block ending in `Jmp`. Note that this means that HT_{Jmp} describes execution from the start of the block ending with a `Jmp` statement until termination by `Halt`, which may involve an arbitrary number of blocks after the current one in order of execution. Accordingly, the arguments of get_ht_of_jmp_block are a BIR program $prog$, the label of the jump block bl as well as a tuple of a halt HT $halt\_ht$ and a variable set $varset$. The structure is roughly the following:

1. Prove the Hoare triple jmp_stmt_ht of the `Jmp` statement at the end of the block bl in $prog$ (recall that the WP of `Jmp` is simply the postcondition). This is done with the aid of the `bir_jmp_wp` theorem, and any of its antecedents which cannot be obtained trivially are proved by the proof procedures is_label_member_eval and bir_get_current_statement_eval.

2. Compose jmp_stmt_ht and $halt\_ht$ using the `bir_jmp_halt_comp_wp` theorem into a new halt HT new_halt_ht, describing execution from the `Jmp` statement until termination by `Halt`.

3. Starting with the last basic statement, successively generate WPs for and prove corresponding HTs for the basic statements of the block bl. As the HTs are proved, compose them together with $halt\_stmt\_ht$, using the composition theorem for n-step and halt Hoare triples to ultimately form one halt HT describing execution from the start of the block until termination by `Halt`.

get_ht_of_cjmp_block

This proof procedure returns a theorem stating the halt Hoare triple HT_{CJmp} of a block ending in `CJmp`. The arguments are a BIR program *prog*, the block label *bl* of the conditional jump block and two tuples of the halt HTs *halt_ht1* (describing execution from the first jump target) and *halt_ht2* (describing execution from the second jump target) with their variable sets *varset1* and *varset2*. Note that the variable sets are not HOL sets but set representations in SML, for more efficient set operations. The structure is roughly the following:

1. Prove the Hoare triple *cjmp_stmt_ht* of the `CJmp` statement at the end of the block in *prog* with the block label *bl*.
   - Generate *curr_stmt_thm*, a theorem stating the effect of bir_get_current_statement applied on *prog* and a program counter *pc* pointing to the `CJmp` statement (by using bir_get_current_statement_eval).
   - Construct the `CJmp` statement HT *cjmp_hts* by means of modus ponens of the `bir_cjmp_wps` theorem with *curr_stmt_thm*, *is_bool_exp_cond_thm* (a theorem stating that the loop condition is a Boolean expression generated by bir_is_bool_exp_pp) and theorems stating that the first and second target labels of the `CJmp` statement exist in the program.
2. Combine *cjmp_hts*, *halt_ht1* and *halt_ht2* through modus ponens with the `bir_cjmp_halts_comp_wp` theorem into a new halt HT *new_halt_ht*, describing execution from the `CJmp` statement until termination by `Halt`.
   - Generate theorems stating the variables in the constraints on initialized variables in *halt_ht1* and *halt_ht2* are both subsets of the variables in the variable constraint of *cjmp_hts*.
   - Finally, supply all of the arguments and generated theorems to modus ponens of `bir_cjmp_halts_comp_wp`.
3. Generate HTs for all basic statements in the block, and successively combine them into *new_halt_ht*.

- This is accomplished by using the get_ht_of_bstmt_list proof procedure with the precondition of new_halt_ht, $prog$, bl and a tuple of new_halt_ht and the corresponding varset as arguments.

expand_bstmt_ht

expand_bstmt_ht looks at the block with label bl in the program $prog$. There, it successively generates HTs for statements from the end of the block to the start of the block and continuously merges them into $halt\_ht$.

When generating a Hoare triple for a list of basic BIR statements, it is first necessary to check for the special case of an empty list of basic BIR statements, which is not just a theoretical possibility but a typical situation for BIR blocks representing branch instructions (since the `Jmp` and `CJmp` statements are end statements, they cannot syntactically be in the list of basic statements). This is represented by the `bir_0_step_wp` theorem and the composition is achieved by modus ponens with the `bir_n_step_halt_comp_wp` theorem.

In case of a non-empty list of basic statements, for every statement the variables which at that point need to be initialized are obtained as an SML red-black set of HOL terms, then the subset relation theorem needed for the merge is constructed using is_subset_pp. The HTs for the individual statements are obtained in separate functions, from the theorems `bir_assert_wp` and `bir_assume_wp`.

Note that the `Assume` statement requires more computation than the `Assert` statement since the `Assume` HT (`bir_assume_wp`) contains an antecedent which must be proved by means of a proof procedure: the booleanity of the assumed expression is proved via bir_is_bool_exp_pp.

The merge is finalized by obtaining the consequent of `bir_n_step_halt_comp_wp` through modus ponens with the antecedents computed before.

Various other proof procedures

The previous section is not a full account of all the proof procedures, but describes the most central ones to the results in this thesis. A few more have been written to compare with the performance of the evaluation tools of HOL:

- bir_vars_of_exp_pp obtains the result of bir_vars_of_exp applied to the expression *exp* as a theorem,
- bir_is_bool_exp_pp decides whether an expression is Boolean or not,
- is_member_set_pp obtains the theorem stating that the set consisting of the element *elem* is a subset of the set $elem \cup superset$ (and simplifies the union),
- is_subset_pp and is_subset_tm_pp (using union_subset_pp) obtain the theorem stating that the set *subset* is the subset of the set *superset*,
- and is_label_member_pp obtains the theorem stating that the block label *label* is among the list of block labels in the program *prog*

3.6 Verification step

The theorem in Figure 3.14 is used for the purposes of verification, as described in Section 2.2.2. It is also known as the consequence rule for precondition strengthening.

Theorem 3.6.1 (`bir_verification`)**.** *If the halt HT of program* $prog$*, precondition* WP*, postcondition* Q*, program counter* pc *and sets of variables* $vars$ *and* $postcond\_vars$ *holds, then if for all well-typed states* s *where the variables in* $vars$ *and* $vars\_postcond$ *are initialized and* P *holds* WP *holds, then the halt HT of program* $prog$*, precondition* P*, postcondition* Q*, program counter* pc *and sets of variables* $vars$ *and* $postcond\_vars$ *holds:*

Proof. Expand the halt HT definition and specialize the results with the correct states. Move expressions from the goal to the assumptions, and when the definition of the HT precondition is expanded, the conclusion of the middle antecedent of the original theorem can then be obtained, which completes the proof. □

```
⊢ ∀prog P WP Q pc vars postcond_vars.
    bir_ht_halt_n_holds prog WP Q pc vars postcond_vars ⇒
    (∀s.
       bir_is_well_typed_env s.bst_environ ⇒
       bir_env_vars_are_initialised s.bst_environ vars ⇒
       bir_env_vars_are_initialised s.bst_environ
         postcond_vars ⇒
       bir_prop_true P s.bst_environ ⇒
       bir_prop_true WP s.bst_environ) ⇒
    bir_ht_halt_n_holds prog P Q pc vars postcond_vars
```

Figure 3.14: `bir_verification`

3.7 Supporting tools

3.7.1 Tactics

A few tactics were also devised in order to automate the verification step (proving that the contractual precondition implies the computed weakest precondition) as much as possible.

First, the antecedents on initialized variables are merged with the help of BIR_FIX_VARSETS_TAC. Then, the computed weakest precondition WP is reduced to a list of HOL assumptions. For this purpose, the following tactics were created:

- BIR_ASSUME_TAC simplifies the effects of an `Assume` statement on the postcondition,
- BIR_CJMP_DISJ_TAC simplifies the effects of a `CJmp` statement on the postcondition,
- and BIR_DISJ_TAC simplifies BIR Or in goal.

At this point, the various expressions assumed throughout the program are in the list of assumptions of HOL. The proof strategy is to translate these from BIR to something usable by a library for bitblasting. This is not a matter of trivial translation, but rather requires some manual trickery. Still, the following tactics pull the most of the weight:

- BIR_NOT_EQUIV_TAC simplifies BIR Not in goal and assumptions,
- BIR_EQ_EQUIV_TAC simplifies BIR Equal and MemEq in goal and assumptions,
- BIR_LT_32_EQUIV_TAC simplifies BIR LessThan (between 32-bit immediate values) in goal and assumptions,
- BIR_SLT_32_EQUIV_TAC simplifies BIR SignedLessThan (between 32-bit immediate values) in goal and assumptions,
- BIR_LOE_32_EQUIV_TAC simplifies BIR LessOrEqual (between 32-bit immediate values) in goal and assumptions,
- BIR_SLOE_32_EQUIV_TAC simplifies BIR SignedLessOrEqual (between 32-bit immediate values) in goal and assumptions,
- BIR_ASM_DISJ_EQUIV_TAC simplifies BIR Or in assumptions (note that this gives more subgoals),
- BIR_REWR_IMMEXP_TAC simplifies evaluation of BIR Den immediate expressions to BIR Imm values using free variables given tuples of names of variables and their sizes,
- BIR_REWR_IMMEXPS_TAC does the same as the above but for lists of arguments,
- BIR_REWR_MEMEXP_TAC simplifies evaluation of BIR Den memory expressions to BIR Mem values using free variables given tuples of names of variables and their address and value types,
- and BIR_REWR_MEMEXPS_TAC does the same as the above but for lists of arguments.

after which it is possible to rewrite Load and Store expressions, and translate the result to bit-blastable form.

Since the theorem stating equivalence between BIR and HOL conjunctions does not have any antecedents, it does not really require any corresponding tactic. The above tactics can only be guaranteed to work if the expressions to be simplified are at the top level of goal and assumptions - although they do a rudimentary search through consecutive conjunctions and equalities.

Chapter 4

Results

4.1 Example application: Verifying GCD

For evaluating the newly designed verification procedure, the loop content of a common algorithm to compute the greatest common divisor was chosen. The C code is shown in Figure 4.1. To ensure a practical, realistic example, actual ARM assembly code of the loop content - shown in Figure 4.2 - of the GCD algorithm was translated into BIR in passified DSA (dynamic single assignment) form. The 13 assembly instructions translate into equally many BIR blocks, typically with one basic statement and one end statement each, and finally, a block with `Halt` was appended.

```
int gcd(int x, int y) {
  while ((x>0) && (y>0) && (x != y)) {
    if (x > y)
      x = x-y;
    else
      y = y-x;
  }
  return x;
}
```

Figure 4.1: GCD function in C

In Figure 4.3, the individual boxes are different BIR blocks with their names (corresponding to the ARM assembly program addresses)

```
400580:    ldr    w1, [sp,#12]
400584:    ldr    w0, [sp,#8]
400588:    cmp    w1, w0
40058c:    ble    4005a4
400590:    ldr    w1, [sp,#12]
400594:    ldr    w0, [sp,#8]
400598:    sub    w0, w1, w0
40059c:    str    w0, [sp,#12]
4005a0:    b      4005b4
4005a4:    ldr    w1, [sp,#8]
4005a8:    ldr    w0, [sp,#12]
4005ac:    sub    w0, w1, w0
4005b0:    str    w0, [sp,#8]
4005b4:    .end
```

Figure 4.2: GCD loop content in ARM assembly language

in bold typeface. Below the name is a (possibly empty) list of basic statements and at the bottom of the box an end statement. Arrows correspond to possible jumps between the blocks triggered by the end statements. $[MEM]_{SP}$ is a memory load operation, returning the immediate value stored in the memory MEM at the address SP. $[MEM]_{SP}(i)$ is a memory store operation, returning the memory obtained by storing the immediate value i in the memory MEM at the address SP.

Two properties were evaluated: memory safety and functional correctness, both with separate proofs for signed and unsigned integers. The performance was evaluated on a 2010 laptop with a 2.4 GHz Intel Core i3-M370 CPU.

4.1.1 Memory Safety

At the end of execution, it is usually desirable that memory stays unchanged apart from a certain region that the program is allowed to write to, for all possible executions. The contract (a Hoare triple) cor-

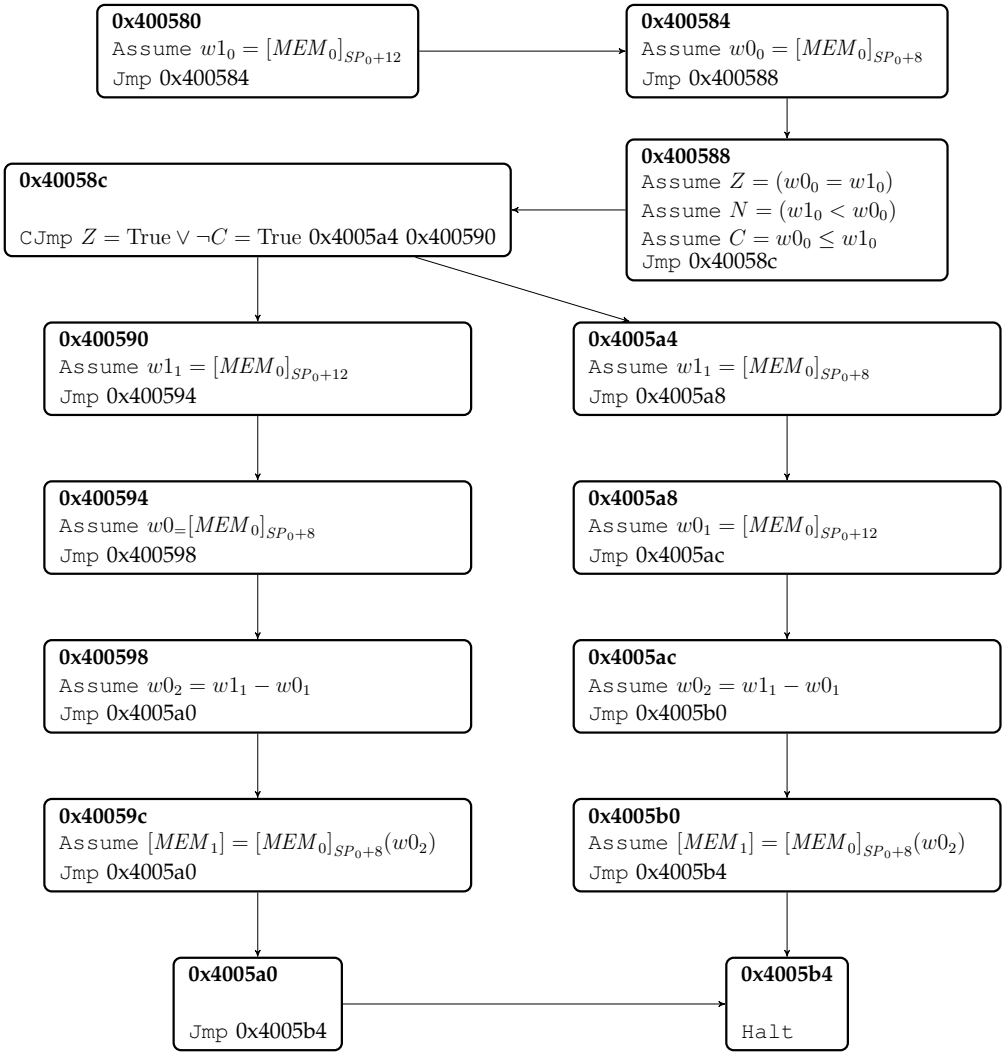


Figure 4.3: GCD loop content in BIR

responding to this property, which the goal then is to prove, is

$$\begin{aligned} \{\text{True}\}\ gcd\_prog\ \{((sp < uint\_max(32) - 15) \wedge \\ (a < sp + 8 \vee a > sp + 15)) \implies \\ (mem'(a) = mem(a))\}\ , \end{aligned} \tag{4.1}$$

where SP is the stack pointer, $UINT\_MAX(32)$ is the maximum value of an unsigned 32-bit word, and $mem(a)$ means reading memory at the address a. mem is the version of the memory at the start of the program, and mem' the version at the end.

The first step is now to use get_ht_of_program on gcd_prog and the postcondition in Equation 4.1. The resulting dictionary d is obtained in four seconds, and in d the label of the entry point of gcd_prog is looked up to obtain a HT $computed\_ht$ stating the effect of execution from the start of gcd_prog until termination by `Halt`.

Then, modus ponens is used on $bir\_verification$ (shown in Figure 3.14) with $computed\_ht$, the universally quantified variable p specialized as True. All that remains is now to prove the last antecedent of $bir\_verification$ stating that the precondition provided in the contract (True) implies the weakest precondition computed in $computed\_ht$. Unlike the generation of $bir\_verification$ this is not a step that is entirely automated, the tactics introduced in Section 3.7 were created to translate BIR expressions to equivalent word operations or generic Boolean expressions.

The final parts of the proof consist of using rewrite theorems to compute the effect of Load and Store expressions. At this point, it is also necessary to do some renaming of variables automatically generated by HOL as well as final technical touching-up of the assumptions. After this, it is possible to proceed to bit-blasting[1].

The proof of the verification step is computed in 16 seconds.

4.1.2 Functional Correctness

Functional correctness means that the code computes what it is supposed to. For this small example, functional correctness might seem redundant, but for some larger programs, the functional correctness criterion can be expressed much more succinctly compared to the code. The contract is

$$\begin{aligned}\{\mathrm{True}\}\ gcd\_prog\ \{((x_0 > y_0\ \Longrightarrow\ & \\ (x_1 = x_0 - y_0) \wedge y_1 = y_0) \wedge\ & \\ (x_0 \leq y_0\ \Longrightarrow\ & \\ (y_1 = y_0 - x_0 \wedge x_1 = x_0))\}\ ,\ & \end{aligned} \tag{4.2}$$

where x and y are two memory addresses and the indexes their version. In practice, they are offsets from the stack pointer in a versioned BIR memory. Since DSA is used, the same second version can be used for both if-cases.

[1]Bit-blasting means transforming a bit-vector formula to an equivalent propositional formula, trying to prove it using tools specialized for propositional formulae.

The first parts of the proof, where the computed weakest precondition is transformed into HOL assumptions, is identical to the proof for memory safety since the proof is related to the same program. The last part, related to fitting the parts into place for bit-blasting is entirely different but only from a technical point of view, which does not merit further description here.

4.2 Performance evaluation

For purposes of judging the efficacy of the approach using proof procedures, their performance with regard to computation time was evaluated. This was done by generating random BIR expressions and random sets of variables as arguments to the proof procedures and taking averages of computation times over 15 runs with newly randomized arguments. In Figure 4.4, the performance evaluation of the bir_vars_-

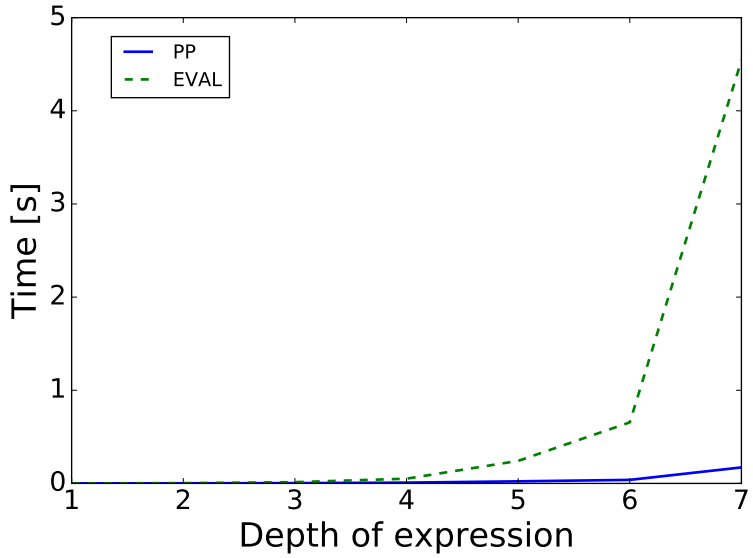


Figure 4.4: Performance evaluation of bir_vars_of_exp_pp

of_exp_pp proof procedure is shown. This procedure provides a theorem stating the set of variables in an expression. It is used in the verification step, when proving equivalence between BIR and HOL expressions. This computation becomes a bottleneck as expressions get very large. The computation time of the proof procedure (PP) is compared

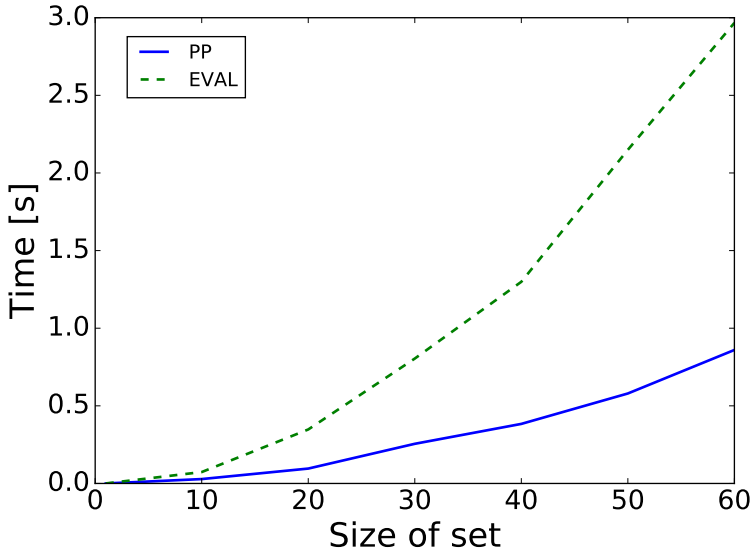


Figure 4.5: Performance evaluation of is_subset_pp

with that of the in-built HOL tool for evaluating terms, EVAL. Figure 4.5 shows the performance evaluation of the is_subset_pp proof procedure. This is used once for `Assume` and two times for `CJmp`. The time of computation increases as the constraint on initialized variables grows larger. It is important to note that since is_subset_pp is only ever used to prove the positive case (it is never asked whether A is a subset of B, only for a proof of that being true), testing with completely random variable sets might skew the comparison. Therefore, all comparisons are made when computing the positive case.

As for the other auxiliary proof procedures used in proving concrete HTs, only small gains were achieved over comparable solutions which used the simplifier of HOL. However, these also have computation times around $1/100$ of a second for feasible values while being called about as often as the above, and so are less critical for computational time overall.

The performance of get_ht_of_program was also evaluated, which is shown in Figure 4.6. This was done by generating random programs without branching statements: a sequence of Jump blocks ending in a Halt block, all with one Assume statement each assuming a Boolean BIR expression of depth 3. The provided postcondition had depth 2. Variables were created using random 5-letter names, meaning they

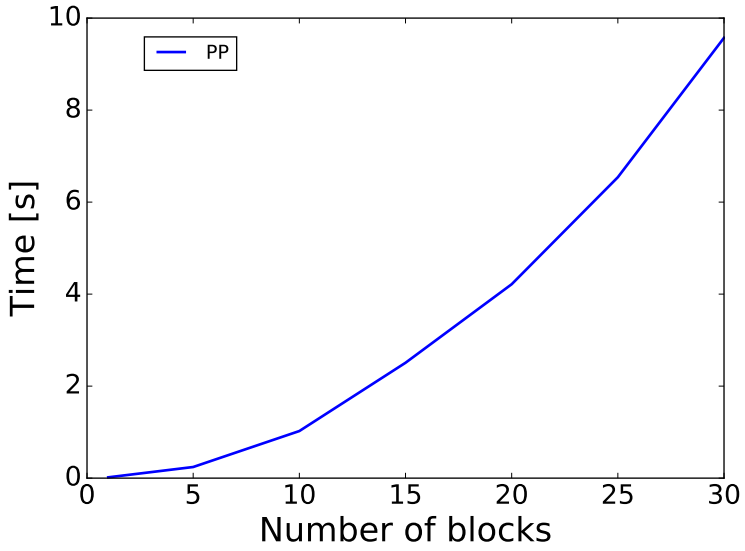


Figure 4.6: Performance evaluation of get_ht_of_program

rarely occur multiple times. These conditions are likely more severe than typical conditions of usage, apart of course from the absence of conditional jumps.

The performance bottleneck when proving Hoare triples of programs is the `Assume` and `CJmp` statements. These involve the most usage of proof procedures for antecedents.

When treating programs of the small sizes similar to the example application in this chapter, memory is not an issue. An experiment with calling get_ht_of_program on longer linear programs of length 100 (constructed as described in the previous paragraph) reveals that maximum memory usage (according to casual inspection of `htop`) will fluctuate up to 200 MiB above the maximum usage for programs of length 10. Then, from length 100 to 200 another 500 MiB is added. For this type of program, the feasible sizes treatable by a cheap laptop likely range in the hundreds. For comparison, AES-128 can be implemented in 560 assembly instructions [46].

Chapter 5

Conclusions

In this thesis, a new verifying procedure generating verification conditions (VCs) for programs, as well as rudimentary tactics to automate the proof of these VCs, have been presented. The resulting proofs are exportable and independently verifiable. The new approach proved feasible with regard to computation time. Even for examples more advanced than GCD, like cryptographic protocols, both the step obtaining the HTs and the verification step should be possible to do almost in real-time (even with the addition of loop invariants).

Relying on speed-up due to writing custom proof procedures instead of relying on functions inside the theorem prover logic trades off programmer time in order to gain computational efficiency. The biggest gains can be found where the HOL representation of the objects involved in the computation is not performance-oriented, the best example of which is the manipulation of sets. Where there is no advantage to be made by writing more efficient data structures in the metalanguage, writing proof procedures by hand might even risk decreased performance.

The entire project - not counting signature files, Python scripts, data et cetera - uses $12\,\mathrm{K}$ physical lines of code. Of those, $3\,\mathrm{K}$ are the manual HOL proofs of GCD properties, which contain several similar parts and comments. $4,5\,\mathrm{K}$ are all the proof procedures, while $4,5\,\mathrm{K}$ are the theories themselves, half of which concern Hoare triples and weakest preconditions, and half of which are lemmata used by the proof procedures and tactics. In particular, the proofs of the WPSTs and composition theorems could have been written in a much more compact form, but are instead optimized for human-readability and robustness

to changes in the underlying theories.

There exist two main performance limitations for extending this work: firstly, the potentially exponential growth of the generated WP. While Flanagan and Saxe [16] were able to reduce the worst-case size complexity of the generated WP from exponential to polynomial, the same is not as easy to do when generating WPs for BIR programs, due to differences in the syntax of BIR to that of the language of Flanagan and Saxe. While one could very well make WP size improvements for branching segments in if-then-else format, the unstructured nature of BIR makes it impossible to guarantee branching only takes this shape, while this is implicit in the language of Flanagan and Saxe.

Secondly, bit-blasting expressions containing multiplication of two variables can be very computationally costful: general multiplication is limited to at most eight-bit words when using the bit-blasting library of HOL4 [18]. The very last part of the proofs described in Chapter 4 is typically completed by bit-blasting. If that should fail for similar proofs, the problem could be exported to an external SMT solver like Z3 and imported back to HOL through an oracle, thus giving the resulting theorem a tag showing it is obtained through a potentially unsound external tool. As this would increase the trust base of the result, it would slightly defeat the purpose of the approach suggested in this thesis. However, it might suffice as a temporary solution when HOL4 bit-blasting does not do the job. Another approach would be to write a procedure to automatically reconstruct the proofs given by Z3 in HOL4, an approach which was developed by Böhme et al. in 2011 [5]. This would seem like the best of both worlds, however, the drawback is that the status of Z3 as a closed-source project makes automatic reconstruction a very difficult endeavour - when the proofs given by Z3 lack detail, the user has no way to obtain more information.

Mitigating the difficulty of bit-blasting general multiplication is a current field of research, which the method in this thesis is dependent upon for increasing the scope of application. Bryant at al. has suggested a decision procedure which uses successive under- and over-approximation of bit-vector formulae to demonstrably shorten computation times for problems involving multiplication [8]. In 2015, Fröhlich et al. presented a stochastic local search (SLS) method operating directly on the bit vectors as an alternative to bit-blasting [19]. This has later been integrated with bit-blasting into a sequential portfolio, as well as extended with propagation-based move selection utilizing

encountered formulae to decrease reliance on brute force [42]. Neither of these approaches is in any way integrated with HOL4, other than in the sense that they might be used as external oracles.

One bottleneck when using this type of method is programmer time used in specification and proving steps. For this thesis, a series of rudimentary re-usable tools were developed (as described in Section 3.7), which still leaves the proof requiring some human guidance. In theory, the proving step could be automated to the ability of the best SMT solvers and decision procedures, either by reconstructing proofs inside HOL4, or by re-implementing them inside HOL4 logic or as proof procedures. As for specification, there appears to be no obvious way to save programmer time other than perhaps translating the BIR specifications from some higher-level language or tool which is faster to use. However, note that this translation would require additional trust unless it is also proven correct in HOL4.

5.1 Future work

The inquiring reader might, of course, argue that it is useless to verify programs as long as the hardware running the programs is not verified. Luckily, the work on verifying commercial microprocessors (to the degree that is possible due to intellectual property restrictions) and on top of that verifying the implementation of machine code instructions has been steadily going on since the 2000s [21]. The work by Anthony Fox et al. at Cambridge in these areas is connected to this thesis in several ways, in particular since they originally developed some HOL theories used. In order for this work to continue to have relevance, reliance is placed also on verification of new hardware. The extension of the main proof procedure to correctly handle the `Declare` statement if encountered should be trivial - it should allow removing the variable declared from the set of variables required to be initialized getting passed along the WP-generating procedure if this variable is also assigned to before being used in any expressions.

Similarly, the `Observe` statement is even simpler to handle. Since only passified programs are handled (where the the `Assign` statements have been exchanged for the `Assume` statements via the passification procedure), the `Assign` statement is not necessary to handle.

The antecedent in the verification theorem stating that the con-

tractual precondition implies the computed weakest precondition is currently proved manually in HOL. This proof could instead be automated by means of a proof procedure, further decreasing reliance on a user fluent in HOL. This has already been automated to a large degree, but there are still improvements that could be made which would save time for any user of the methods presented in this thesis.

The functionality of the tool could be extended by introducing support for loop invariants, which would enable dropping the restriction on loop-free programs.

Leslie Lamport provided a predicate transformer semantics for concurrent programs in 1990 [31]. As far as the author is aware, this has yet to be implemented in a fully verified verification condition generator (although concurrent program verification tools such as JPF which do not provide independently verifiable proofs are widespread).

A Chinese group implemented a weakest-precondition-style predicate transformer semantics for quantum computation in Isabelle/HOL based on the quantum Hoare logic defined in 2011 by Mingsheng Ying [52] [36]. Adding this to the current tool might prove an interesting extension, which then could be used to verify properties of quantum algorithms.

A weakest-precondition-style predicate transformer semantics for probabilistic programs was used in 2016 to reason about bounds on running times of probabilistic programs [30]. This might be a possible future addition if one should be interested in analyzing probabilistic programs and providing proofs for bounds of their running times.

Bibliography

[1] Mike Barnett and K Rustan M Leino. "Weakest-precondition of unstructured programs". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 31. 1. ACM. 2005, pp. 82–87.

[2] Mike Barnett et al. "Boogie: A modular reusable verifier for object-oriented programs". In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2005, pp. 364–387.

[3] S. K. Basu and R. T. Yeh. "Strong verification of programs". In: *IEEE Transactions on Software Engineering* 1 (Sept. 1975), pp. 339–346. ISSN: 0098-5589. DOI: 10.1109/TSE.1975.6312858. URL: doi.ieeecomputersociety.org/10.1109/TSE.1975.6312858.

[4] Christoph Baumann et al. "Lessons learned from microkernel verification–specification is the new bottleneck". In: *arXiv preprint arXiv:1211.6186* (2012).

[5] Sascha Böhme et al. "Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL". In: *International Conference on Certified Programs and Proofs*. Springer. 2011, pp. 183–198.

[6] Bishop Brock, Matt Kaufmann, and J Strother Moore. "ACL2 theorems about commercial microprocessors". In: *International Conference on Formal Methods in Computer-Aided Design*. Springer. 1996, pp. 275–293.

[7] David Brumley et al. "BAP: A binary analysis platform". In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 463–469.

[8] Randal E Bryant et al. "An abstraction-based decision procedure for bit-vector arithmetic". In: *International journal on software tools for technology transfer* 11.2 (2009), pp. 95–104.

[9] Ernie Cohen et al. "VCC: A practical system for verifying concurrent C". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 23–42.

[10] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[11] L. Peter Deutsch. *An interactive program verifier*. Tech. rep. 1973.

[12] Edsger W Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs". In: *Communications of the ACM* 18.8 (1975), pp. 453–457.

[13] Gilles Dowek et al. *The COQ Proof Assistant: User's Guide: Version 5.6*. INRIA, 1992.

[14] Paul Feautrier. "Dataflow analysis of array and scalar references". In: *International Journal of Parallel Programming* 20.1 (1991), pp. 23–53.

[15] Branden Fitelson. "Using Mathematica to understand the computer proof of the Robbins Conjecture". In: *Mathematica in Education and Research* 7 (1998), pp. 17–26.

[16] Cormac Flanagan and James B Saxe. "Avoiding exponential explosion: Generating compact verification conditions". In: *ACM SIGPLAN Notices*. Vol. 36. 3. ACM. 2001, pp. 193–205.

[17] Robert W Floyd. "Assigning meanings to programs". In: *Mathematical aspects of computer science* 19.19-32 (1967), p. 1.

[18] Anthony CJ Fox. "LCF-style bit-blasting in HOL4". In: *International Conference on Interactive Theorem Proving*. Springer. 2011, pp. 357–362.

[19] Andreas Fröhlich et al. "Stochastic Local Search for Satisfiability Modulo Theories." In: 2015.

[20] Kurt Gödel. "Russell's mathematical logic". In: *1944* (1944), pp. 123–153.

[21] MJC Gordon. "Formal Specification and Verification of ARM6". In: ().

[22] Mike Gordon. "From LCF to HOL: a short history." In: *Proof, Language, and Interaction*. 2000, pp. 169–186.

[23] David Gries. “A note on a standard strategy for developing loop invariants and loops”. In: *Science of Computer Programming* 2.3 (1982), pp. 207–214.

[24] Florian Haftmann. “Code generation from specifications in higher-order logic”. PhD thesis. München, Techn. Univ., Dissertation, 2009.

[25] Thomas C Hales. “A proof of the Kepler conjecture”. In: *Annals of mathematics* (2005), pp. 1065–1185.

[26] Thomas Hales et al. “A formal proof of the Kepler conjecture”. In: *Forum of Mathematics, Pi*. Vol. 5. Cambridge University Press. 2017.

[27] Charles Antony Richard Hoare. “An axiomatic basis for computer programming”. In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[28] Warren A Hunt et al. “Industrial hardware and software verification with ACL2”. In: *Phil. Trans. R. Soc. A* 375.2104 (2017), p. 20150399.

[29] Joe Hurd. “The OpenTheory standard theory library”. In: *NASA Formal Methods Symposium*. Springer. 2011, pp. 177–191.

[30] Benjamin Lucien Kaminski et al. “Weakest precondition reasoning for expected run–times of probabilistic programs”. In: *European Symposium on Programming Languages and Systems*. Springer. 2016, pp. 364–389.

[31] Leslie Lamport. “win and sin: Predicate transformers for concurrency”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 396–428.

[32] Ralph Langner. “Stuxnet: Dissecting a cyberwarfare weapon”. In: *IEEE Security & Privacy* 9.3 (2011), pp. 49–51.

[33] Dirk Leinenbach and Thomas Santen. “Verifying the Microsoft Hyper-V hypervisor with VCC”. In: *International Symposium on Formal Methods*. Springer. 2009, pp. 806–809.

[34] K Rustan M Leino. “Efficient weakest preconditions”. In: *Information Processing Letters* 93.6 (2005), pp. 281–288.

[35] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Commun. ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: http://doi.acm.org/10.1145/1538788.1538814.

[36] Tao Liu et al. "A theorem prover for quantum Hoare logic and its applications". In: *arXiv preprint arXiv:1601.03835* (2016).

[37] Allen L Mann. "A complete proof of the Robbins conjecture". In: (2003).

[38] William McCune. "Solution of the Robbins problem". In: *Journal of Automated Reasoning* 19.3 (1997), pp. 263–276.

[39] Roberto Metere, Andreas Lindner, and Roberto Guanciale. "Sound Transpilation from Binary to Machine-Independent Code". In: *Brazilian Symposium on Formal Methods*. Springer. 2017, pp. 197–214.

[40] Robin Milner. *Logic for computable functions description of a machine implementation*. Tech. rep. DTIC Document, 1972.

[41] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices*. Vol. 42. 6. ACM. 2007, pp. 89–100.

[42] Aina Niemetz, Mathias Preiner, and Armin Biere. "Propagation based local search for bit-precise reasoning". In: *Formal Methods in System Design* 51.3 (2017), pp. 608–636.

[43] Chris Okasaki. "Red-black trees in a functional setting". In: *Journal of functional programming* 9.4 (1999), pp. 471–477.

[44] Lawrence C Paulson. "Natural deduction as higher-order resolution". In: *The Journal of Logic Programming* 3.3 (1986), pp. 237–258.

[45] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. "Global value numbers and redundant computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1988, pp. 12–27.

[46] Peter Schwabe and Ko Stoffelen. "All the AES you need on Cortex-M3 and M4". In: *International Conference on Selected Areas in Cryptography*. Springer. 2016, pp. 180–194.

[47] Yan Shoshitaishvili et al. “SOK:(State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. 2016, pp. 138–157.

[48] L Fejes Tóth. “On close-packings of spheres in spaces of constant curvature”. In: *Publ. Math, Debrecen* 3 (1953), pp. 158–167.

[49] Alan Mathison Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London mathematical society* 2.1 (1937), pp. 230–265.

[50] Frédéric Vogels, Bart Jacobs, and Frank Piessens. “A machine-checked soundness proof for an efficient verification condition generator”. In: *Proceedings of the 2010 ACM symposium on Applied Computing*. ACM. 2010, pp. 2517–2522.

[51] A. N. Whitehead and B. Russell. *Principia Mathematica. Vol. I.* English. Cambridge: University Press. xv, 666 S. 4° (1910). 1910.

[52] Mingsheng Ying. “Floyd–hoare logic for quantum programs”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33.6 (2011), p. 19.

Appendix A

Lemmata

Here, some of the less central theoretical results are shown. While the proofs in the thesis cannot be understood without the below, they have been moved out of the way into an appendix to allow the reader to more quickly access the more central results while still keeping the details for reference.

Any theorems referenced in the below sections which cannot be found in the theories written by the author can be found in the generic BIR theories written by Roberto Metere and Thomas Türk.

A.1 Lemmata on pre- and postconditions

This theorem facilitates proving the trivial HT.

Theorem A.1.1 (`bir_ht_pre_impl_post`)**.** *If the Hoare triple precondition holds in state S for the expression P and the sets of variables $vars$ and $postcond\_vars$, the the Hoare triple postcondition holds in the same state for P, the program counter of S, zero execution steps, and the sets of variables $vars$ and $postcond\_vars$:*

Proof. The two definitions are identical apart from the predicate on the program counter in `bir_ht_postcond_holds_def`, but since the number of steps is zero, it still holds. □

This theorem helps merge two adjacent Hoare triples into one.

Theorem A.1.2 (`bir_ht_post_impl_pre`)**.** *If the Hoare triple postcondition holds in the state S for the expression Q, the program counter pc, the number of steps n and the sets of variables $vars$ and $postcond\_vars$, and if*

```
⊢ ∀S P vars postcond_vars.
    bir_ht_precond_holds S P vars postcond_vars ⇒
    bir_ht_postcond_holds S P S.bst_pc.bpc_label
      S.bst_pc.bpc_index vars postcond_vars
```

Figure A.1: `bir_ht_pre_impl_post`

$vars'$ is a subset of $vars$, then the Hoare triple precondition holds in S for Q, $vars'$ and $postcond\_vars$:

```
⊢ ∀S P l i n vars vars' postcond_vars.
    bir_ht_postcond_holds S P l i vars postcond_vars ⇒
    vars' ⊆ vars ⇒
    bir_ht_precond_holds S P vars' postcond_vars
```

Figure A.2: `bir_ht_post_impl_pre`

Proof. The definition of `bir_ht_postcond_holds` contains everything needed from `bir_ht_precond_holds` apart from the constraint on initialized variables $vars'$. `bir_ht_postcond_holds` also says that the set of variables $vars$ are initialized in S. With this, and the antecedent saying that $vars' \subset vars$, it is possible to obtain that $vars$ must also be initialized in S, which completes the proof. □

A.2 Some lemmata on termination

Theorem A.2.1 (`bir_exec_step_n_terminated_unchanged`)**.** *If the BIR state S has terminated, then S is unchanged under further execution using `bir_exec_step_n`:*

Proof. Proof follows directly through the semantics of `bir_exec_step_n_state`. □

Theorem A.2.2 (`bir_exec_step_n_state_unchanged`)**.** *If the status of BIR state S is AssumptionViolated, then S is unchanged under further execution using `bir_exec_step_n`:*

```
⊢ ∀prog S1 n.
    bir_state_is_terminated S1 ⇒
    (bir_exec_step_n_state prog S1 n = S1)
```

Figure A.3: `bir_exec_step_n_terminated_unchanged`

```
⊢ ∀prog S1 n.
    (S1.bst_status = BST_AssumptionViolated) ∨
    (S1.bst_status = BST_Failed) ∨
    (∃v. S1.bst_status = BST_Halted v) ∨
    (∃l. S1.bst_status = BST_JumpOutside l) ⇒
    (bir_exec_step_n_state prog S1 n = S1)
```

Figure A.4: `bir_exec_step_n_state_unchanged`

Proof. Proof follows modus ponens with the antecedent on `bir_exec_step_n_terminated_unchanged`, after expanding the definition of `bir_state_is_terminated`. □

Theorem A.2.3 (`bir_running_by_exclusion`)**.** *If the status of BIR state S is valid (not JumpOutside, Failed or Halted), then if status of S is also not AssumptionViolated, it must be Running:*

```
⊢ ∀S1.
    bir_is_valid_status S1 ⇒
    S1.bst_status ≠ BST_AssumptionViolated ⇒
    (S1.bst_status = BST_Running)
```

Figure A.5: `bir_running_by_exclusion`

Proof. Proof follows from the principle of exclusion: simply perform a case split on the possible statuses of S and resolve all cases with the given antecedents, giving either contradictions or the conclusion. □

A.3 Lemmata on Boolean BIR values

These theorems can be found in bir_bool_equivTheory, and relate to translation between BIR and HOL logical connectives.

Theorem A.3.1 (`bir_and_equiv`)**.** *The BIR conjunction of two expressions is True iff the HOL conjunction of the values of the two expressions is True:*

$$\vdash \forall\, env_1\ exp\ Q\,.\ \texttt{bir\_prop\_true}\ exp\ env_1 \wedge \texttt{bir\_prop\_true}\ Q\ env_1 \iff \texttt{bir\_prop\_true}\ (\texttt{BExp\_BinExp}\ \texttt{BIExp\_And}\ Q\ exp)\ env_1$$

Figure A.6: `bir_and_equiv`

Proof. Consider the two directions of the equivalence: first, starting from the assumption that the HOL conjunction holds. It follows that the BIR conjunction holds by evaluation of the expression, using the True values of the individual subexpressions.

If instead the starting point is that the BIR conjunction holds, then the goal becomes the symmetrical task of proving individually that this implies that both conjuncts hold. If all the possible values of the subexpressions are considered, the only value which does not contradict the BIR conjunction holding is True. □

Theorem A.3.2 (`bir_disj_impl`)**.** *The BIR disjunction of two expressions is True if the HOL disjunction of the values of the two expressions is True:*

$$\vdash \forall\, env\ exp_1\ exp_2\,.\ \texttt{bir\_prop\_true}\ (\texttt{BExp\_BinExp}\ \texttt{BIExp\_Or}\ exp_1\ exp_2)\ env \Rightarrow \texttt{bir\_prop\_true}\ exp_1\ env \vee \texttt{bir\_prop\_true}\ exp_2\ env$$

Figure A.7: `bir_disj_impl`

Proof. The proof follows along similar lines to that of the second direction in the proof of `bir_and_equiv`. Considering the different cases for the values of the subexpressions leads either to type contradictions among assumptions or to a word expressions from which one can obtain the goal through bitblasting. □

A.4 Lemmata on BIR variables

In BIR, initializing a variable means declaring it (using `Declare`) and then assigning a value to it.

`bir_env_order` is a preorder on the variable environments of BIR states based on declared and initialized variables. There are three criteria for the environment env_1 to relate to env_2:

1. Both environments need to be well-typed.
2. Every declared variable in env_1 also needs to be declared in env_2.
3. Every initialized variable in env_1 also needs to be initialized in env_2.

`bir_exec_step_ENV_ORDER` is a lemma from the generic BIR theories stating that the environment $S_1.bst\_environ$ is related to `bir_exec_step_state` *prog* S_1 (`bir_exec_step_ENV_ORDER` says the same for `bir_exec_step_n_state`), and `bir_env_vars_are_initialised_ORDER` states that if env_1 is related to env_2, then if the set of variables $vars$ are initialized in env_1, they are also initialized in env_2.

The below lemmata relate this concretely to variable initialization and well-typedness.

Theorem A.4.1 (`bir_varinit_invar_n`)**.** *Variable initialization is invariant under n execution steps:*

Proof. Similar to the above, but using `bir_exec_step_n_ENV_ORDER` and a technical lemma explicitly stating that `bir_exec_step_n` has a return value. □

Theorem A.4.2 (`bir_welltypedness_invar_n`)**.** *Well-typedness of variable environment is invariant under n execution steps:*

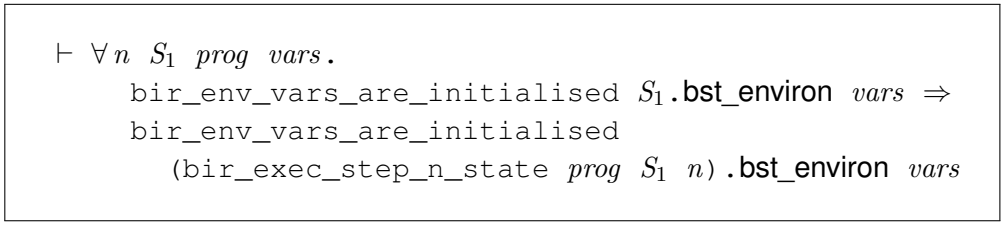


Figure A.8: `bir_varinit_invar`

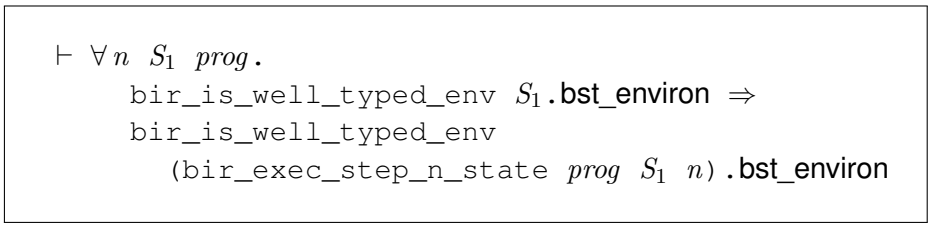


Figure A.9: `bir_welltypedness_invar_n`

Proof. Similar to the proof of `bir_varinit_invar_n` (in Figure A.8), but using `bir_env_order_well_typed` instead of `bir_env_vars_are_initialised_ORDER`. □

The lemmata shown in Figure A.10 and A.11 summarize the notion that since execution of passive statements do not change the variable environment, they also do not affect the values resulting from evaluating expressions. From the generic BIR theories, `bir_exec_stmt_assert_SAME_ENV`, `bir_exec_stmt_assume_SAME_ENV` and `bir_exec_stmt_observe_SAME_ENV` say that the corresponding basic statements leave the variable environment of the state unchanged. All BIR ending statements are also passive.

Theorem A.4.3 (`bir_prop_true_invar_pass`)**.** *Properties keep holding under an execution step, if the executed statement is passive:*

Proof. Either S_1 is terminated or not: if it is, then the conclusion holds since execution will not change the state in any way. If it is not, then consider the different possible statements: among the basic statements, `Assert`, `Assume` and `Observe` are passive. `bir_exec_stmt_assert_SAME_ENV`, `bir_exec_stmt_assume_SAME_ENV` and `bir_exec_stmt_`

```
⊢ ∀ Q S1 prog stmt.
    (bir_get_current_statement prog S1.bst_pc = SOME stmt) ⇒
    bir_is_passive_stmt stmt ⇒
    bir_prop_true Q S1.bst_environ ⇒
    bir_prop_true Q (bir_exec_step_state prog S1).bst_environ
```

Figure A.10: `bir_prop_true_invar_pass`

`observe_SAME_ENV` state this. Finally, consider the end statements, all of which are passive (`bir_exec_stmtE_env_unchanged`). □

Theorem A.4.4 (`bir_prop_true_invar_pass_n1`)**.** *Properties keep holding under one execution step, if the executed statement is passive:*

```
⊢ ∀ n Q S1 prog stmt.
    (bir_get_current_statement prog S1.bst_pc = SOME stmt) ⇒
    bir_is_passive_stmt stmt ⇒
    bir_prop_true Q S1.bst_environ ⇒
    bir_prop_true Q
      (bir_exec_step_n_state prog S1 1).bst_environ
```

Figure A.11: `bir_prop_true_invar_pass_n1`

Proof. Here, the goal is instead a theorem saying that one execution step by `bir_exec_step_n_state` does not impact evaluation of expressions. To solve this, the theorem stating equivalence between one step of `bir_exec_step_n_state` and `bir_exec_step_state` (`bir_exec_step_1_gen`) is used together with `bir_prop_true_invar_pass`, which suffices to prove the theorem. □

A.5 Lemmata on WPs of statements

A.5.1 Assert

Theorem A.5.1 (`bir_assert_valid_status`)**.** *If the current statement in state S_1 is `Assert` exp, exp holds in S_1 and status of S_1 is Running,*

then the state resulting from execution of one step from S_1 has valid status:

```
⊢ ∀ exp S1 prog.
    (bir_get_current_statement prog S1.bst_pc =
     SOME (BStmtB (BStmt_Assert exp))) ⇒
    bir_prop_true exp S1.bst_environ ⇒
    (S1.bst_status = BST_Running) ⇒
    bir_is_valid_status (bir_exec_step_n_state prog S1 1)
```

Figure A.12: `bir_assert_valid_status`

Proof. When executing an `Assert` statement, only the evaluation of *exp* can change the current status: if it is not True, status is set to Failed. Since an assumption is that initial status is Running, and since another assumption is that *exp* evaluates to True, the status will remain Running, which is a valid status. Thus, the proof follows by evaluating the execution. □

Theorem A.5.2 (`bir_assert_pc`)**.** *If the current statement in state S_1 is `Assert` exp, exp holds in S_1 and status of S_1 is Running, then the state resulting from execution of one step from S_1 has its program counter incremented by one:*

```
⊢ ∀ exp S1 prog.
    (bir_get_current_statement prog S1.bst_pc =
     SOME (BStmtB (BStmt_Assert exp))) ⇒
    bir_prop_true exp S1.bst_environ ⇒
    (S1.bst_status = BST_Running) ⇒
    ((bir_exec_step_n_state prog S1 1).bst_pc.bpc_index =
     S1.bst_pc.bpc_index + 1) ∧
    ((bir_exec_step_n_state prog S1 1).bst_pc.bpc_label =
     S1.bst_pc.bpc_label)
```

Figure A.13: `bir_assert_pc`

Proof. Firstly, looking at the semantics there is no way that execution of `Assert` can change the pc label, since only end statements can jump to a different block. Secondly, the index of pc is always incremented by one as long as the initial state is Running and execution of `Assert` does not cause termination, which cannot happen if exp evaluates to True in the current state. Thus, the proof follows by evaluating the execution. □

A.5.2 Assume

Theorem A.5.3 (`bir_assume_valid_status`)**.** *If the current statement is* `Assume` exp*,* exp *is a Boolean expression, the BIR disjunction of the negation of* exp *and* Q *holds, the environment of* S_1 *is well-typed and status of* S_1 *is Running, then the state resulting from execution of one step from* S_1 *has valid status:*

```
⊢ ∀ exp Q S1 prog.
    (bir_get_current_statement prog S1.bst_pc =
     SOME (BStmtB (BStmt_Assume exp))) ⇒
    bir_prop_true
      (BExp_BinExp BIExp_Or (BExp_UnaryExp BIExp_Not exp) Q)
      S1.bst_environ ⇒
    bir_is_bool_exp exp ⇒
    bir_is_well_typed_env S1.bst_environ ⇒
    (S1.bst_status = BST_Running) ⇒
    bir_is_valid_status (bir_exec_step_n_state prog S1 1)
```

Figure A.14: `bir_assume_valid_status`

Proof. Given the disjunction among the antecedents, it follows that neither $\neg exp$ nor Q can be Unknown. Since the environment is well-typed and exp is a Boolean expression, exp also cannot be Unknown. Then, it is possible to make a case split on the value of exp (True or False) after which execution gives that the status can only be Running or AssumptionViolated, both of which are valid, which completes the proof. □

Theorem A.5.4 (`bir_assume_pc`)**.** *If the current statement in state* S_1 *is* `Assume` exp*,* exp *holds in* S_1 *and status of* S_1 *is Running, then the state*

resulting from execution of one step from S_1 has its program counter incremented by one:

```
⊢ ∀ S1 exp prog.
    (bir_get_current_statement prog S1.bst_pc =
     SOME (BStmtB (BStmt_Assume exp))) ⇒
    bir_prop_true exp S1.bst_environ ⇒
    (S1.bst_status = BST_Running) ⇒
    ((bir_exec_step_n_state prog S1 1).bst_pc.bpc_index =
     S1.bst_pc.bpc_index + 1) ∧
    ((bir_exec_step_n_state prog S1 1).bst_pc.bpc_label =
     S1.bst_pc.bpc_label)
```

Figure A.15: `bir_assert_pc`

Proof. Firstly, there is no way that execution of `Assume` can change the pc label, since only end statements can jump to a different block. Secondly, the index of pc is always incremented by one as long as the initial state is Running and execution of `Assume` does not cause termination, which cannot happen if exp evaluates to True in the current state. Thus, the proof follows by evaluating the execution. □

Theorem A.5.5 (`bir_assume_violated`)**.** *If the current statement in state S_1 is `Assume` exp, the BIR negation of exp holds in S_1 and status of S_1 is Running, then the state resulting from execution of one step from S_1 has status AssumptionViolated:*

Proof. When executing `Assume`, only evaluation of exp can change the state status. If exp evaluates to False, status is set to AssumptionViolated. Thus, the proof follows by evaluating the execution. □

A.5.3 Halt

Theorem A.5.6 (`bir_halt_halts`)**.** *If the current statement in state S_1 is `Halt` and status of S_1 is Running, then status will be Halted in the state resulting from execution of one step from S_1:*

Proof. When executing `Halt` from an initial state with status Running, the status of the resulting state will always be Halted. Thus, the proof follows by evaluating the execution. □

```
⊢ ∀ S1 exp prog.
    (bir_get_current_statement prog S1.bst_pc =
     SOME (BStmtB (BStmt_Assume exp))) ⇒
    bir_prop_true (BExp_UnaryExp BIExp_Not exp)
      S1.bst_environ ⇒
    (S1.bst_status = BST_Running) ⇒
    ((bir_exec_step_n_state prog S1 1).bst_status =
     BST_AssumptionViolated)
```

Figure A.16: `bir_assert_pc`

```
⊢ ∀ S1 exp prog.
    (bir_get_current_statement prog S1.bst_pc =
     SOME (BStmtE (BStmt_Halt exp))) ⇒
    (S1.bst_status = BST_Running) ⇒
    ∃ hcode.
      (bir_exec_step_n_state prog S1 1).bst_status =
      BST_Halted hcode
```

Figure A.17: `bir_halt_halts`

A.5.4 Jump

Theorem A.5.7 (`bir_jmp_valid_status`)**.** *If the current statement in state S_1 is `Jmp` label, label is a block label of the program prog and status of S_1 is Running, then the state resulting from execution of one step from S_1 has valid status:*

Proof. When executing a `Jmp` statement with a block label as argument, the current status can only be changed if *label* is not a label in the program, in which case status is set to JumpOutside. Since an assumption is that initial status is Running, and since another assumption is that *label* can be found in the program *prog*, status will remain Running, which is a valid status. Thus, the proof follows by evaluating the execution. □

Theorem A.5.8 (`bir_jmp_target`)**.** *If the current statement in state S_1 is `Jmp` label, label is a block label of the program prog and status of S_1 is*

```
⊢ ∀ S1 prog label.
    (bir_get_current_statement prog S1.bst_pc =
     SOME (BStmtE (BStmt_Jmp (BLE_Label label)))) ⇒
    MEM label (bir_labels_of_program prog) ⇒
    (S1.bst_status = BST_Running) ⇒
    bir_is_valid_status (bir_exec_step_n_state prog S1 1)
```

Figure A.18: `bir_jmp_valid_status`

Running, then the program counter of the state resulting from execution of one step from S_1 points to the first statement in the block with label label:

```
⊢ ∀ S1 prog label.
    (bir_get_current_statement prog S1.bst_pc =
     SOME (BStmtE (BStmt_Jmp (BLE_Label label)))) ⇒
    MEM label (bir_labels_of_program prog) ⇒
    (S1.bst_status = BST_Running) ⇒
    ((bir_exec_step_n_state prog S1 1).bst_pc =
     <|bpc_label := label; bpc_index := 0|>)
```

Figure A.19: `bir_jmp_target`

Proof. When executing `Jmp` with a block label argument *label* from an initial state with status Running, the existence of *label* in *prog* will be evaluated. If *label* is found in *prog* - which in this case is given by an assumption - the program counter of the next state will always point to the first statement of the block labelled *label*. Thus, the proof follows by evaluating the execution. □

A.5.5 Conditional Jump

Theorem A.5.9 (`bir_cjmp_valid_status`)**.** *If the current statement is `CJmp`, and if both jump target labels exist in the program prog, and if either cond holds the the negation of cond, and if the status of S_1 is Running, then the state resulting from execution of one step from S_1 has valid status:*

```
⊢ ∀ S1 cond prog label1 label2 .
    (bir_get_current_statement prog S1.bst_pc =
     SOME
       (BStmtE
          (BStmt_CJmp cond (BLE_Label label1)
             (BLE_Label label2)))) ⇒
    bir_prop_true cond S1.bst_environ ∨
    bir_prop_true (BExp_UnaryExp BIExp_Not cond)
      S1.bst_environ ⇒
    MEM label1 (bir_labels_of_program prog) ⇒
    MEM label2 (bir_labels_of_program prog) ⇒
    (S1.bst_status = BST_Running) ⇒
    bir_is_valid_status (bir_exec_step_n_state prog S1 1)
```

Figure A.20: `bir_cjmp_valid_status`

Proof. When executing a `CJmp` statement, the current status can only be changed if any of the target labels is not a label in the program, in which case status is set to JumpOutside, or if the jump condition is Unknown, in case status is set to Failed. Since an assumption is that initial status is Running, and since another assumption rules out $cond$ evaluating to Unknown, and since it is also assumed that the jump target labels can be found in the program $prog$, status will remain Running, which is a valid status. Thus, the proof follows by evaluating the execution. □

Theorem A.5.10 (`bir_cjmp_target1`)**.** *If the current statement is* `CJmp`*, and if the first jump target label* $label_1$ *exists in the program* $prog$*, and if* $cond$ *holds, and if the status of* S_1 *is Running, then the program counter resulting from execution of one step from* S_1 *points to the first instruction in the block with label* $label_1$*:*

Proof. When executing `CJmp` with a jump condition that holds and first jump target label $label_1$ from an initial state with status Running, the existence of $label_1$ in $prog$ will be evaluated. If $label_1$ is found in $prog$ - which in this case is given by an assumption - the program counter of the next state will always point to the first statement of the block labelled $label_1$. Thus, the proof follows by evaluating the execution. □

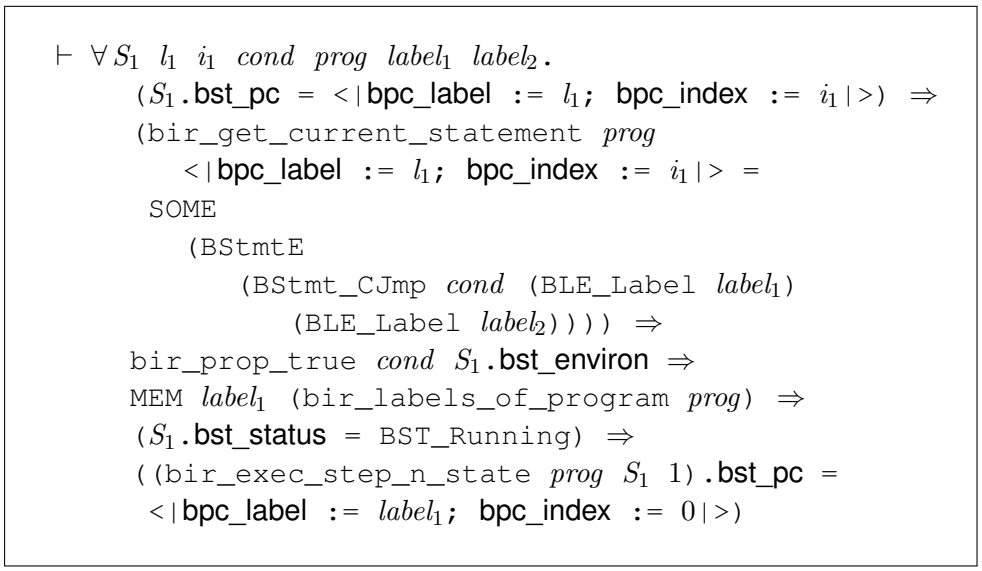


Figure A.21: `bir_cjmp_target1`

Theorem A.5.11 (`bir_cjmp_target2`)**.** *If the current statement is* `CJmp`*, and if the second jump target label* $label_2$ *exists in the program* $prog$*, and if the negation of* $cond$ *holds, and if the status of* S_1 *is Running, then the program counter resulting from execution of one step from* S_1 *points to the first instruction in the block with label* $label_2$*:*

Proof. When executing `CJmp` with a jump condition whose negation holds and first jump target label $label_2$ from an initial state with status Running, the existence of $label_2$ in $prog$ will be evaluated. If $label_2$ is found in $prog$ - which in this case is given by an assumption - the program counter of the next state will always point to the first statement of the block labelled $label_2$. Thus, the proof follows by evaluating the execution. □

A.6 Hoare triple composition theorems

These theorems describe the rules for combining two adjacent - meaning execution of one ends where execution in the other begins - Hoare triples into one. These theorems typically consist of two antecedent HTs, the latter of which is a halt HT, and a consequent halt HT. With the WPSTs for the individual statements, these provide the requisite tools to prove HTs for entire programs.

```
⊢ ∀ S1 l1 i1 cond prog label1 label2.
    (S1.bst_pc = <|bpc_label := l1; bpc_index := i1|>) ⇒
    (bir_get_current_statement prog
       <|bpc_label := l1; bpc_index := i1|> =
     SOME
       (BStmtE
          (BStmt_CJmp cond (BLE_Label label1)
             (BLE_Label label2)))) ⇒
    bir_prop_true (BExp_UnaryExp BIExp_Not cond)
      S1.bst_environ ⇒
    MEM label2 (bir_labels_of_program prog) ⇒
    (S1.bst_status = BST_Running) ⇒
    ((bir_exec_step_n_state prog S1 1).bst_pc =
     <|bpc_label := label2; bpc_index := 0|>)
```

Figure A.22: `bir_cjmp_target2`

The general proof strategy is to first use the precondition and program counter of the consequent HT to obtain the postcondition of the first antecedent HT. Then, use a theorem giving the precondition this postcondition implies for the same state. This, as well as splitting the execution described in the consequent HT into two parts, gives the postcondition of the antecedent halt HT which is identical to the postcondition of the consequent HT apart from possibly some set of initialized variables (which then must be the ones initialized in the first antecedent HT). The proof is then finally complete by the property of variables to never be uninitialized.

A.6.1 n-step and halt Hoare triple composition theorem

Theorem A.6.1 (`bir_n_step_halt_comp_wp`)**.** *The n-step Hoare triple stated for n execution steps starting at the position in $prog$ determined by the program counter pc holds for the precondition p and postcondition q and for the sets of initialized variables $vars$ and $postcond\_vars$, then if the halt Hoare triple stated for execution starting at pc with index incremented by n holds for the precondition q and postcondition r and the sets of variables*

$vars'$ and $postcond\_vars$, then if $vars'$ is a subset of $vars$, the halt Hoare triple stated for execution starting at the program counter pc holds for the precondition p and postcondition r and the sets of variables $vars$ and $postcond\_vars$.

```
⊢ ∀prog p q r l1 i1 vars vars' postcond_vars n.
    bir_ht_n_holds prog p q l1 i1 l1 (i1 + n) vars
      postcond_vars n ⇒
    bir_ht_halt_n_holds prog q r
      <|bpc_label := l1; bpc_index := i1 + n|> vars'
      postcond_vars ⇒
    vars' ⊆ vars ⇒
    bir_ht_halt_n_holds prog p r
      <|bpc_label := l1; bpc_index := i1|> vars postcond_vars
```

Figure A.23: `bir_n_step_halt_comp_wp`

Proof. The definitions of `bir_ht_n_holds` (Figure 3.5) and `bir_ht_halt_n_holds` (Figure 3.4) are expanded. The consequent halt HT is instantiated with the states S_1 and S_3, S_2 is introduced via an abbreviation as the result of executing n steps from S_1, the n-step HT is instantiated with S_1 and S_2, and the antecedent halt HT with S_2 and S_3.

With this, the postcondition of the HT describing execution from S_1 to S_2 can be obtained by modus ponens with existing assumptions from the consequent HT. This postcondition and `bir_ht_post_impl_pre` (Figure A.2) - the usage of which requires the antecedent with the subset relation - directly provide the precondition of the HT describing execution from S_2 to S_3. Now expanding the postcondition, the program counter of the S_2 to S_3 HT is obtained (the disjunction in the postcondition also gives the case status of S_2 is AssumptionViolated, which proves the goal directly).

The existentially quantified number of execution steps in the S_2 to S_3 HT can now be written as a new free variable n', and following this $n' + n$ is given as witness to the existentially quantified number of execution steps in the consequent HT. Splitting the resulting execution up in two parts of n' and n steps finally yields the postcondition of

the S_2 to S_3 HT. This is exactly to goal to prove, apart from the set of initialized variables $vars'$ (and not $vars$). However, `bir_varinit_invar_n` (Figure A.8) gives that $vars$, which were initialized in S_1 according to the HT stating execution from S_1 to S_3 is still initialized, which completes the proof. □

A.6.2 Block transition and halt Hoare triple composition theorem

Note that this theorem only features one step of execution. Since `Jmp` does not involve evaluating any expression (and potentially setting the state status to Failed if any variables are not initialized), it is never needed to add new variables to the set of initialized variables when computing HTs for `Jmp` statements. This means that for the special case of one step, it is possible to have the same set of initialized variables for both HT and eliminate the computation of one subset relation in the proof procedures for HTs.

Theorem A.6.2 (`bir_jmp_halt_comp_wp`)**.** *If the Hoare triple stated for one execution step starting at the position in $prog$ determined by the program counter pc and ending at the first position in the bloc with label $label$ holds for the precondition p and postcondition q and for the sets of initialized variables $vars$ and $postcond\_vars$, then if the halt Hoare triple stated for execution starting at the first position in the block with label $label$ holds for the precondition q and postcondition r and the sets of variables $vars$ and $postcond\_vars$, the halt Hoare triple stated for execution starting at the program counter pc holds for the precondition p and postcondition r and the sets of variables $vars$ and $postcond\_vars$.*

Proof. This proof is very similar to the proof of `bir_n_step_halt_comp_wp`. The difference is that there is now a HT with execution starting in one block and ending in another, and there is no $vars'$ involved. This means that there is no need to use `bir_varinit_invar_n`. □

A.6.3 Conditional Jump and two halt Hoare triples composition theorem

Theorem A.6.3 (`bir_cjmp_halts_comp_wp`)**.** *If the Hoare triple stated for one execution step starting at pc and ending at the beginning of the block with label $label_1$ holds for the precondition $cond \wedge q_1$, postcondition q_1 and the*

```
⊢ ∀prog p q r l1 i1 vars postcond_vars label.
    bir_ht_n_holds prog p q l1 i1 label 0 vars postcond_vars
      1 ⇒
    bir_ht_halt_n_holds prog q r
      <|bpc_label := label; bpc_index := 0|> vars
      postcond_vars ⇒
    bir_ht_halt_n_holds prog p r
      <|bpc_label := l1; bpc_index := i1|> vars postcond_vars
```

Figure A.24: `bir_jmp_halt_comp_wp`

sets of initialized variables $vars$ and $postcond\_vars$, then if the Hoare triple stated for one execution step starting at pc and ending at the beginning of the block with label $label_2$ holds for the precondition $\neg cond \wedge q_2$, postcondition q_2 and the sets of variables $vars$ and $postcond\_vars$, then if the halt Hoare triple stated for execution starting at the beginning of the block with label $label_1$ holds for the precondition q_1, the postcondition r and the sets of initialized variables $vars1'$ and $postcond\_vars$, then if the halt Hoare triple stated for execution starting at the beginning of the block with label $label_2$ holds for the precondition q_2, the postcondition r and the sets of initialized variables $vars2'$ and $postcond\_vars$, then if both $vars1'$ and $vars2'$ are subsets of $vars$, then the halt Hoare triple stated for execution starting at pc holds for the precondition $(cond \wedge q_1) \vee (\neg cond \wedge q_2)$, the postcondition r and the sets of initialized variables $vars$ and $postcond\_vars$.

Proof. This proof follows along similar lines to that of `bir_jmp_halt_comp_wp`. The difference is that the BIR disjunction in the consequent HT must be translated to a HOL one (using `bir_disj_impl` shown in Figure A.7), yielding two symmetrical cases to prove. A subset relation between the variable sets in the different HTs is also needed, since `CJmp` might introduce new variables via the condition, something which is treated the same way as in the proof of `bir_n_step_halt_comp_wp`. □

```
⊢ ∀prog q1 q2 r vars l1 i1 vars1' vars2' postcond_vars label1
    label2 cond.
  bir_ht_n_holds prog (BExp_BinExp BIExp_And cond q1) q1 l1
    i1 label1 0 vars postcond_vars 1 ∧
  bir_ht_n_holds prog
    (BExp_BinExp BIExp_And (BExp_UnaryExp BIExp_Not cond) q2)
    q2 l1 i1 label2 0 vars postcond_vars 1 ⇒
  bir_ht_halt_n_holds prog q1 r
    <|bpc_label := label1; bpc_index := 0|> vars1'
    postcond_vars ⇒
  bir_ht_halt_n_holds prog q2 r
    <|bpc_label := label2; bpc_index := 0|> vars2'
    postcond_vars ⇒
  vars1' ⊆ vars ⇒
  vars2' ⊆ vars ⇒
  bir_ht_halt_n_holds prog
    (BExp_BinExp BIExp_Or (BExp_BinExp BIExp_And cond q1)
       (BExp_BinExp BIExp_And (BExp_UnaryExp BIExp_Not cond)
          q2)) r <|bpc_label := l1; bpc_index := i1|> vars
    postcond_vars
```

Figure A.25: `bir_cjmp_halts_comp_wp`