Encoding Hoare Logic in Typed Certified Code[⋆]

Nikolaos S. Papaspyrou, Michalis A. Papakyriakou, and Angelos Manousaridis

National Technical University of Athens
School of Electrical and Computer Engineering, Division of Computer Science
Software Engineering Laboratory, Polytechnioupoli, 15780 Zografou, Athens, Greece
{nickie, mpapakyr, amanous}@softlab.ntua.gr

Abstract. Hoare logic and proof-carrying code are two independent frameworks for reasoning that programs meet their specifications. In this paper, we merge the two approaches by embedding axiomatic specifications in a type system for foundational proof-carrying code. By annotating programs with proof hints, proof checking of Hoare triples becomes decidable and as efficient as type checking.

1 Introduction

Since it was first proposed [1], the axiomatic approach to proving program correctness which is commonly referred to as *Hoare Logic* has greatly influenced the methods for verifying and designing programs [2]. In brief, Hoare logic introduced the strength of formal logic in computer programming, not only as a tool to reason about program properties but also to derive programs from their specifications and to define the semantics of programming languages [3, 4].

Modern approaches to building programs that certifiably meet their specifications combine a formal logic with the programming language in which the programs are written. *Proof-carrying code* [5] and *foundational proof-carrying code* [6] are general frameworks expressing this philosophy: certified programs are annotated with proofs that their specifications are met. More recently, type-theoretic frameworks for constructing, composing and reasoning about certified software have been proposed [7, 8], based on the "formulae as types" principle [9]. The type-theoretic approach provides an embedding of logic in the type system of the programming language: program properties are encoded in types and proof checking is reduced to type checking.

In this paper, we propose an encoding of Hoare logic in the type-theoretic approach. Specifications of WHILE programs are represented as Hoare triples, encoded in a type language that is a variation of the *Calculus of Inductive Constructions* (CIC) [10–13]. We annotate WHILE programs with all that is necessary to make proof checking decidable. In this way, we obtain a formal system that automatically checks the correctness of programs w.r.t. their specifications; this

[⋆] This paper is based on work within the research project "Theory of Algorithms and Logic: Applications in Computer Science", partially funded by the European Social Fund (75%) and the Greek Ministry of Education (25%). ΕΠΕΑΕΚ ΙΙ: *Πυθαγόρας*.

system is consistent with classic Hoare logic for WHILE programs and builds on a large body of scientific knowledge in the area of program specifications.

Hamid and Shao have also proposed an interface between Hoare logic and a syntactic type system [14]. They use preconditions to guarantee that low-level typed assembly programs do not violate a given safety policy. Our approach differs in using a high-level programming language and arbitrary Hoare triples as specifications. The work of Franssen and de Swart [15] is very similar to ours. They propose an embedding of many-sorted first-order logic in a pure type system with constants, capable of encoding Hoare logic for WHILE programs. Our work differs in the expressiveness of the type system which reflects on the expressiveness of logic (CIC can encode higher-order predicate logic). Moreover, it does not use constants but is based on foundational mathematical logic.

2 The type language

We split the programming language in two: the *type language* and the *computation language* [7, 16]. The former is the language in which logic is encoded: propositions and predicates, used in program specifications, as well as proofs are written in it. The latter, in which programs are written, is WHILE. The abstract syntax of the type language is given by the following grammar:

$$\begin{array}{lll} A, B & ::= & \mathsf{Set} \mid \mathsf{Type} \mid \mathsf{Ext} \mid X \mid \Pi X{:}A.\,B \mid \lambda X{:}A.\,B \mid A\,B \\ & \mid & \mathsf{Ind}(X{:}A)\{\boldsymbol{A}\} \mid \mathsf{Constr}(n, A) \mid \mathsf{Elim}[A'](A{:}B\,\boldsymbol{B})\{\boldsymbol{A}\} \end{array}$$

where X denotes a variable, A and B denote terms, n denotes a natural number and $\boldsymbol{A}$ denotes a sequence of terms. The *sorts* Set, Type and Ext are the constants of the type language. Apart from sorts and variables, a term can be a product $\Pi X{:}A.\,B$, an abstraction $\lambda X{:}A.\,B$, an application $A\,B$, an inductive type $\mathsf{Ind}(X{:}A)\{\boldsymbol{A}\}$, a constructor of an inductive type $\mathsf{Constr}(n, A)$ or an elimination of an inductive type $\mathsf{Elim}[A'](A{:}B\,\boldsymbol{B})\{\boldsymbol{A}\}$.

The *typing relation* in the type language determines the semantic validity of terms. For any pair of terms, $A : B$ is read “A has type B”. Among sorts, we have $\mathsf{Set} : \mathsf{Type}$ and $\mathsf{Type} : \mathsf{Ext}$. Products are essentially dependent function types: an abstraction $\lambda X{:}A.\,B$ has type $\Pi X{:}A.\,B'$ provided that $B : B'$ and, if it is applied to a term of type A, it produces a term of type B'. We write $A \to B$ instead of $\Pi X{:}A.\,B$ if X does not occur free in B.

Inductive types can be defined by using Ind. As an example, the types Bool of Boolean values and Nat of natural numbers can be defined as follows:

$$\begin{array}{ll} \mathsf{Bool} \equiv \mathsf{Ind}(X{:}\mathsf{Set})\{X; X\} & : \mathsf{Set} \\ \mathsf{Nat} \equiv \mathsf{Ind}(X{:}\mathsf{Set})\{X; X \to X\} & : \mathsf{Set} \end{array}$$

Inside Ind, X is a synonym for the defined type. Nat and Bool have two constructors, the types of which are given inside the braces. $\mathsf{Constr}(n, A)$ provides access to the n-th constructor of A but it is convenient to give constructors descriptive names like true and succ. Most interesting operations on elements of inductive types require the use of Elim, whose rôle is twofold. First of all, it destructs elements of inductive types and provides access to the components of which they were built. Moreover, it allows primitive recursion on inductive types.

The type language defines four types of *reductions*: α, β, η and ι. The first three are well-known from the study of λ-calculus; the fourth is used in the elimination of inductive types.

Based on the "formulae as types" principle [9], it is possible to encode propositions and proofs in the type language. Propositions are terms of type Set. A proof of a proposition P is a term p such that $p : P$. Functions $P \rightarrow Q$ correspond to logical implication, whereas products $\mathit{\Pi}\, X : A.\, P$ correspond to universal quantification. Inductive types can be used to define specific propositions and operators; their constructors correspond to logical axioms. For example:

$$\begin{array}{lll}
\mathsf{True} & \equiv \mathsf{Ind}(X\!:\!\mathsf{Set})\{X\} & : \mathsf{Set} \\
\mathsf{False} & \equiv \mathsf{Ind}(X\!:\!\mathsf{Set})\{\} & : \mathsf{Set} \\
\mathsf{propNot} & \equiv \lambda P\!:\!\mathsf{Set}.\, P \rightarrow \mathsf{False} & : \mathsf{Set} \rightarrow \mathsf{Set} \\
\mathsf{propAnd} & \equiv \lambda P\!:\!\mathsf{Set}.\, \lambda Q\!:\!\mathsf{Set}.\, \mathsf{Ind}(X\!:\!\mathsf{Set})\{P \rightarrow Q \rightarrow X\} & : \mathsf{Set} \rightarrow \mathsf{Set} \rightarrow \mathsf{Set}
\end{array}$$

The constructor of True is an axiom stating that True is a valid proposition. In contrast, False has no constructors. Negation and conjunction are also naturally defined. Similarly, we can define properties of mathematical objects, e.g. equality, and then prove theorems about them, e.g. transitivity and symmetry. The theories built in this way are the tools that help us in reasoning about programs.

3 The computation language

Let $n : \mathsf{Int}$ be any integer number, $b : \mathsf{Bool}$ be any boolean value and $x : \mathsf{Var}$ be any variable. We define the simple imperative language WHILE as follows.

$$\begin{array}{lll}
e : \mathsf{Expr} & ::= & n \mid b \mid x \mid \ \diamond e \mid e \star e \\
c : \mathsf{Comm} & ::= & \mathtt{skip} \mid x := e \mid c ; c \mid \mathtt{if}\ e\ \mathtt{then}\ c\ \mathtt{else}\ c \mid \mathtt{while}\ e\ \mathtt{do}\ c \\
\diamond : \mathsf{UnOp} & ::= & - \mid \neg \\
\star : \mathsf{BinOp} & ::= & + \mid - \mid * \mid \mathtt{div} \mid \mathtt{mod} \mid = \mid \neq \mid < \mid > \mid \leq \mid \geq \mid \mathtt{and} \mid \mathtt{or}
\end{array}$$

A type τ is either int or bool. The set of types is denoted by Ω. A type environment $\mathit{\Gamma}$ is a function mapping variables to types, i.e. an element of $\mathsf{Env} = \mathsf{Var} \rightarrow \Omega$. For each unary operator $\diamond$, we denote by $\mathcal{U}_1(\diamond)$ the type of its operand and by $\mathcal{U}_R(\diamond)$ the type of the result. Similarly, for each binary operator $\star$ we denote by $\mathcal{B}_1(\star)$ and $\mathcal{B}_2(\star)$ the types of the two operands and by $\mathcal{B}_R(\star)$ the type of the result. The typing relations $\mathit{\Gamma} \vdash e : \tau$ and $\mathit{\Gamma} \vdash c$ are easy to define.

The semantic domain corresponding to type τ is denoted by $[\![\tau]\!]$. We take $[\![\mathsf{int}]\!] = \mathsf{Int}$ and $[\![\mathsf{bool}]\!] = \mathsf{Bool}$. We also define the semantics of operators as functions $[\![\diamond]\!] : [\![\mathcal{U}_1(\diamond)]\!] \rightarrow [\![\mathcal{U}_R(\diamond)]\!]$ and $[\![\star]\!] : [\![\mathcal{B}_1(\star)]\!] \rightarrow [\![\mathcal{B}_2(\star)]\!] \rightarrow [\![\mathcal{B}_R(\star)]\!]$, e.g. $[\![*]\!]$ is the multiplication function on elements of Int. Given a type environment $\mathit{\Gamma}$, we define the set of *stores* satisfying $\mathit{\Gamma}$ as $\mathsf{Store}\, \mathit{\Gamma} = \mathit{\Pi}\, x\!:\!\mathsf{Var}.\, [\![\mathit{\Gamma}\, x]\!]$. Such a store s is a function that maps variables to elements of the semantic domains corresponding to their types. We denote by $s\{x \mapsto v\}$ the store that results from

Table 1. Axioms and rules for expression specifications.

$$\{F\,n\}\ n\ \{F\} \qquad \{F\,b\}\ b\ \{F\} \qquad \{\lambda s.\, F\,(s\,x)\,s\}\ x\ \{F\}$$

$$\frac{\{P\}\ e\ \{\lambda v.\, F\,(\llbracket\diamond\rrbracket\,v)\}}{\{P\}\ \diamond\ e\ \{F\}} \qquad \frac{\{P\}\ e_1\ \{G\} \quad \{G\,v_1\}\ e_2\ \{\lambda v_2.\, F\,(\llbracket\star\rrbracket\,v_1\,v_2)\}}{\{P\}\ e_1 \star e_2\ \{F\}}$$

Table 2. Axioms and rules for command specifications.

$$\{P\}\ \texttt{skip}\ \{P\} \qquad \frac{\{P\}\ e\ \{\lambda v.\,\lambda s.\, Q\,s\{x \mapsto v\}\}}{\{P\}\ x := e\ \{Q\}} \qquad \frac{\{P\}\ c_1\ \{R\} \quad \{R\}\ c_2\ \{Q\}}{\{P\}\ c_1; c_2\ \{Q\}}$$

$$\frac{\{P\}\ e\ \{F\} \quad \{F\,\mathsf{true}\}\ c_1\ \{Q\} \quad \{F\,\mathsf{false}\}\ c_2\ \{Q\}}{\{P\}\ \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{Q\}} \qquad \frac{\{P\}\ e\ \{F\} \quad \{F\,\mathsf{true}\}\ c\ \{P\}}{\{P\}\ \texttt{while}\ e\ \texttt{do}\ c\ \{F\,\mathsf{false}\}}$$

Table 3. Consequence rules for specifications.

$$\frac{P \Rightarrow P' \quad \{P'\}\ e\ \{F\}}{\{P\}\ e\ \{F\}} \qquad \frac{P \Rightarrow P' \quad \{P'\}\ c\ \{Q\}}{\{P\}\ c\ \{Q\}} \qquad \frac{\{P\}\ c\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ c\ \{Q\}}$$

mapping x to the value v in s. A standard large-step operational semantics can then be defined: $\llbracket e\rrbracket\, s \Downarrow v$ denotes that evaluation of e in s results in $v : \llbracket\tau\rrbracket$ (where τ is the expression's type), and $\llbracket c\rrbracket\, s \Downarrow s'$ denotes that execution of c in s results in a new store s'.

A *predicate* $P : \mathsf{Pred}\,\Gamma$ is defined as a function that takes a store and returns a proposition, i.e. $\mathsf{Pred}\Gamma = \mathsf{Store}\,\Gamma \to \mathsf{Set}$. A specification for commands is a Hoare triple $\{P\}\ c\ \{Q\}$, where P and Q are predicates; $\{P\}\ c\ \{Q\}$ is valid if for all initial stores $s : \mathsf{Store}\,\Gamma$, if $P\,s$ and $\llbracket c\rrbracket\, s \Downarrow s'$, for some final state $s' : \mathsf{Store}\,\Gamma$, then $Q\,s'$. Instead of the classical approach, which embeds expressions in predicates, we use specifications for expressions of the form $\{P\}\ e\ \{F\}$, where $\Gamma \vdash e : \tau$ and $F : \llbracket\tau\rrbracket \to \mathsf{Pred}\,\Gamma$. $\{P\}\ e\ \{F\}$ is valid if for all $s : \mathsf{Store}\,\Gamma$, if $P\,s$ and $\llbracket e\rrbracket\, s \Downarrow v$ for some value $v : \llbracket\tau\rrbracket$, then $F\,v\,s$. This approach facilitates the encoding of Hoare Logic in our type language and allows extensions of WHILE where expressions can have side-effects, e.g. supporting Pascal-like functions.

The axioms and inference rules for deriving specifications are given in Tables 1 and 2. Table 3 provides the inference rules for strengthening preconditions and weakening postconditions. $P \Rightarrow Q$ denotes that the proposition $\Pi\, s{:}\mathsf{Store}\,\Gamma.\, P\,s \to Q\,s$ is provable in the type language. Soundness can easily be proved: if a specification is derivable then it is valid.

In this formulation of Hoare logic, proof of specifications is undecidable: some inference rules contain unknown predicates in their premises. In the case of binary operators, sequential composition and `if`, weakest preconditions [3] solve the problem. For example, in the case of sequential composition $\{P\}\ c_1; c_2\ \{Q\}$, the unknown predicate R that is required in Table 2 as the assertion between c_1 and c_2 can be calculated as the weakest precondition $wp[c_2](Q)$ (see Table 5 on the next page). However, in the case of `while` and strengthening/weakening, decidability can only be achieved by annotating the program with predicates and proofs. We therefore define an *annotated* WHILE language. Annotations provide

Table 4. Extra rules for specifications in the annotated language.

$$\{P\}\ \texttt{assert}\,[p : P \Rightarrow Q]\ \{Q\}$$

$$\frac{\{P\}\ e\ \{F\} \quad \{F\,\mathsf{true}\}\ c\ \{P\}}{\{P\}\ \texttt{inv}\,[P]\ \texttt{while}\ e\ \texttt{do}\ c\ \{F\,\mathsf{false}\}} \qquad \frac{\{Q\}\ e\ \{F\}}{\{P\}\ \texttt{assert}\,[p : P \Rightarrow Q], e\ \{F\}}$$

Table 5. Weakest preconditions for expressions and commands.

$$\begin{array}{ll}
\mathit{wp}[n](F) \ = \ F\,n & \mathit{wp}[\texttt{skip}](Q) \ = \ Q \\
\mathit{wp}[b](F) \ = \ F\,b & \mathit{wp}[x := e](Q) \ = \ \mathit{wp}[e](\lambda v.\,\lambda s.\,Q\,s\{x \mapsto v\}) \\
\mathit{wp}[x](F) \ = \ \lambda s.\,F\,(s\,x)\,s & \mathit{wp}[c_1; c_2](Q) \ = \ \mathit{wp}[c_1](\mathit{wp}[c_2](Q)) \\
\mathit{wp}[\diamond\ e](F) \ = \ \mathit{wp}[e](\lambda v.\,F\,(\llbracket\diamond\rrbracket\,v)) & \mathit{wp}[\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2](Q) \ = \\
\mathit{wp}[e_1 \star e_2](F) \ = & \quad \mathit{wp}[e](\lambda b.\,\mathsf{if}\ b\ \mathsf{then}\ \mathit{wp}[c_1](Q)\ \mathsf{else}\ \mathit{wp}[c_2](Q)) \\
\quad \mathit{wp}[e_1](\lambda v_1.\,\mathit{wp}[e_2](\lambda v_2.\,F\,(\llbracket\star\rrbracket\,v_1\,v_2))) & \mathit{wp}[\texttt{inv}\,[P]\ \texttt{while}\ e\ \texttt{do}\ c](Q) \ = \ P \\
\mathit{wp}[\texttt{assert}\,[p : P \Rightarrow Q], e](F) \ = \ P & \mathit{wp}[\texttt{assert}\,[p : P \Rightarrow Q]](Q') \ = \ P
\end{array}$$

Table 6. Well definedness of weakest preconditions for expressions and commands.

$$\vdash \mathit{wp}[n](F) \qquad \vdash \mathit{wp}[b](F) \qquad \vdash \mathit{wp}[x](F) \qquad \frac{\vdash \mathit{wp}[e](\lambda v.\,F\,(\llbracket\diamond\rrbracket\,v))}{\vdash \mathit{wp}[\diamond\ e](F)}$$

$$\frac{\begin{array}{c}\vdash \mathit{wp}[e_2](\lambda v_2.\,F\,(\llbracket\star\rrbracket\,v_1\,v_2)) \\ \vdash \mathit{wp}[e_1](\lambda v_1.\,\mathit{wp}[e_2](\lambda v_2.\,F\,(\llbracket\star\rrbracket\,v_1\,v_2)))\end{array}}{\vdash \mathit{wp}[e_1 \star e_2](F)} \qquad \frac{\mathit{wp}[e](F) \Leftrightarrow Q}{\vdash \mathit{wp}[\texttt{assert}\,[p : P \Rightarrow Q], e](F)}$$

$$\vdash \mathit{wp}[\texttt{skip}](Q) \qquad \frac{\vdash \mathit{wp}[e](\lambda v.\,\lambda s.\,Q\,s\{x \mapsto v\})}{\vdash \mathit{wp}[x := e](Q)} \qquad \frac{\vdash \mathit{wp}[c_2](Q) \quad \vdash \mathit{wp}[c_1](\mathit{wp}[c_2](Q))}{\vdash \mathit{wp}[c_1; c_2](Q)}$$

$$\frac{\vdash \mathit{wp}[c_1](Q) \quad \vdash \mathit{wp}[c_2](Q) \quad \vdash \mathit{wp}[e](\lambda b.\,\mathsf{if}\ b\ \mathsf{then}\ \mathit{wp}[c_1](Q)\ \mathsf{else}\ \mathit{wp}[c_2](Q))}{\vdash \mathit{wp}[\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2](Q)}$$

$$\frac{\vdash \mathit{wp}[c](P) \quad \vdash \mathit{wp}[e](G) \quad P \Leftrightarrow \mathit{wp}[e](\lambda b.\,\mathsf{if}\ b\ \mathsf{then}\ \mathit{wp}[c](P)\ \mathsf{else}\ Q)}{\vdash \mathit{wp}[\texttt{inv}\,[P]\ \texttt{while}\ e\ \texttt{do}\ c](Q)}$$

$$\frac{Q \Leftrightarrow Q'}{\vdash \mathit{wp}[\texttt{assert}\,[p : P \Rightarrow Q]](Q')}$$

invariants for `while` statements. Moreover, all uses of strengthening/weakening are made explicit and require the use of `assert`. The new syntax is:

$$\begin{array}{lcl}
e : \mathsf{AExpr}_\Gamma & ::= & n \mid b \mid x \mid \ \diamond e \mid e \star e \mid \texttt{assert}\,[p : P \Rightarrow Q], e \\
c : \mathsf{AComm}_\Gamma & ::= & \texttt{skip} \mid x := e \mid c; c \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{inv}\,[P]\ \texttt{while}\ e\ \texttt{do}\ c \\
 & \mid & \texttt{assert}\,[p : P \Rightarrow Q]
\end{array}$$

where $P, Q : \mathsf{Pred}\,\Gamma$ and p is a proof of $P \Rightarrow Q$. The additional inference rules for the annotated language are given in Table 4.

In the annotated language, we use weakest preconditions to make proof of derivations decidable. Weakest preconditions are defined in Table 5. Nevertheless these equations do not guarantee the well definedness of weakest preconditions, because of the presence of annotations. The rules in Table 6 are required for this purpose. The judgement $P \Leftrightarrow Q$ denotes a decidable notion of equivalence between predicates, which allows substitution of equals for equals in specifications.

Syntactic or $\alpha\beta\eta\iota$-equality are strong equivalence relations for predicates, but sufficient for this purpose. A weaker notion of equivalence would result in fewer explicit assertions, as well as more freedom in formulating predicates.

Our results can be summarized in the three theorems that follow. It can be proved that the logic of the annotated language is compatible with standard Hoare logic for WHILE, i.e. by removing annotations one preserves typing, operational semantics and derivation of specifications (Theorem 1). Furthermore, weakest preconditions are not only *correct* but also *exact*, for the annotated language (Theorem 2). Proving specifications is now decidable: given a decidable equivalence relation between predicates, it is possible to decide the well definedness of a weakest precondition and, thus, to decide the derivability of specifications (Theorem 3).

Theorem 1 (annotations preserve typing, semantics and logic)
Let $e : \mathsf{AExpr}_\Gamma$ *and* $c : \mathsf{AComm}_\Gamma$*. Let* $e' : \mathsf{Expr}$ *and* $c' : \mathsf{Comm}$ *be the results of removing all annotations from* e *and* c*. Then:*

1. *If* $\Gamma \vdash e : \tau$ *then* $\Gamma \vdash e' : \tau$*. If* $\Gamma \vdash c$ *then* $\Gamma \vdash c'$*.*
2. *If* $[\![e]\!]\, s \Downarrow v$ *then* $[\![e']\!]\, s \Downarrow v$*. If* $[\![c]\!]\, s \Downarrow s'$ *then* $[\![c']\!]\, s \Downarrow s'$*.*
3. *If* $\{P\}\ e\ \{F\}$ *is derivable then* $\{P\}\ e'\ \{F\}$ *is derivable. If* $\{P\}\ c\ \{Q\}$ *is derivable then* $\{P\}\ c'\ \{Q\}$ *is derivable.*

Proof sketch. All parts are proved in a straightforward way, by induction on the derivation of the annotated version. □

Theorem 2 (weakest preconditions are correct and exact)

1. *If* $wp[e](F)$ *is defined, then* $\{wp[e](F)\}\ e\ \{F\}$ *is derivable. Conversely, if* $\{P\}\ e\ \{F\}$ *is derivable, then* $wp[e](F)$ *is defined and* $P \Leftrightarrow wp[e](F)$*.*
2. *If* $wp[c](Q)$ *is defined, then* $\{wp[c](Q)\}\ c\ \{Q\}$ *is derivable. Conversely, if* $\{P\}\ c\ \{Q\}$ *is derivable, then* $wp[c](Q)$ *is defined and* $P \Leftrightarrow wp[c](Q)$*.*

Proof sketch. Part (1): The direct is proved by induction on the derivation that $wp[e](F)$ is defined. For the converse, both subparts are proved simultaneously by induction on the derivation $\{P\}\ e\ \{F\}$. Part (2): The direct is proved by induction on the derivation that $wp[c](Q)$ is defined, using part (1). For the converse, both subparts are proved simultaneously by induction on the derivation $\{P\}\ c\ \{Q\}$, using part (1). □

Theorem 3 (decidability of proving specifications)
Proving specifications $\{P\}\ e\ \{F\}$ *and* $\{P\}\ c\ \{Q\}$ *is decidable.*

Proof sketch. It is trivial to check that the well definedness of weakest preconditions is decidable, according to Table 6. Consider the algorithm which answers "yes" if and only if $wp[e](F)$ is defined and $P \Leftrightarrow wp[e](F)$. If the algorithm answers "yes", then the direct of part (1) of Theorem 2 guarantees that $\{P\}\ e\ \{F\}$ is derivable. Conversely, if $\{P\}\ e\ \{F\}$ is derivable, then the converse of part (1) of Theorem 2 guarantees that the algorithm will answer "yes". Similarly for commands. □

4 Example

Let us consider the following simple WHILE program p. Assuming that the initial value of n is positive, this program computes in m the integer part of $\log_2(n)$.

$$m := 0;\ \texttt{while}\ n > 1\ \texttt{do}\ (n := n\,\texttt{div}\,2;\ m := m + 1)$$

An appropriate specification for p, provable in Hoare logic, is the following:

$$\{\lambda s.\, s\,n \geq 1 \wedge s\,n = X\}\ p\ \{\lambda s.\, 2^{s\,m} \leq X < 2^{s\,m+1}\}$$

However, the proof of this specification in standard Hoare logic requires an appropriate invariant for the loop to be found. Such an invariant is:

$$\lambda s.\, X/2^{s\,m} = s\,n \wedge s\,n \geq 1 \wedge s\,m \geq 0$$

If this invariant is given by the programmer as an annotation, the rest of the proof is relatively simple and may be conducted automatically, provided that the proof checker has access to a theory for integer arithmetic and limited theorem-proving capabilities. In our prototype implementation, this is not the case. With an equivalence relation for predicates as strong as $\alpha\beta\eta\iota$-equality, several annotations and proofs must be included before the proof checker can automatically deduce the specification's proof. The full annotated program is:

```
assert [$p_1 : (\lambda s.\, s\,n \geq 1 \wedge s\,n = X) \Rightarrow (\lambda s.\, X/2^0 = s\,n \wedge s\,n \geq 1 \wedge s\,n = X)$];
$m := 0$;
inv [$\lambda s.\, X/2^{s\,m} = s\,n \wedge s\,n \geq 1 \wedge s\,m \geq 0$]
while assert [$p_2 : (\lambda s.\, X/2^{s\,m} = s\,n \wedge s\,n \geq 1 \wedge s\,m \geq 0) \Rightarrow$
                  ($\lambda s.$ (if gt ($s\,n$) 1 then
                            ($\lambda s.\, X/2^{s\,m} = s\,n \wedge s\,n \geq 1 \wedge s\,m \geq 0 \wedge s\,n > 1$)
                        else
                            ($\lambda s.\, X/2^{s\,m} = s\,n \wedge s\,n \geq 1 \wedge s\,m \geq 0 \wedge s\,n \leq 1$)) $s$)],
      $n > 1$
do assert [$p_3 : (\lambda s.\, X/2^{s\,m} = s\,n \wedge s\,n \geq 1 \wedge s\,m \geq 0 \wedge s\,n \leq 1) \Rightarrow$
                ($\lambda s.\, X/2^{s\,m+1} = s\,n/2 \wedge s\,n/2 \geq 1 \wedge s\,m + 1 \geq 0$)];
    $n := n$ div 2;
    $m := m + 1$;
assert [$p_4 : (\lambda s.\, X/2^{s\,m} = s\,n \wedge s\,n \geq 1 \wedge s\,m \geq 0 \wedge s\,n \leq 1) \Rightarrow$
              ($\lambda s.\, 2^{s\,m} \leq X < 2^{s\,m+1}$)]
```

where p_1, p_2, p_3 and p_4 stand for the proofs of four implications that the proof checker cannot automatically deduce. The reader can easily verify that they are simple proofs (p_4 is the hardest). If a weaker equivalence relation was chosen, these proofs and the corresponding `assert` annotations would not be needed.

5 Conclusion

We have shown how certified programs can be represented in a high-level imperative language with proof hints as annotations. Specifications are written in the form of Hoare triples and proof checking is decidable and efficient. The annotated language is consistent to the original in terms of typing, operational semantics and validity of specifications. In this way, we combine the benefits of using Hoare logic, a well studied formal system for program verification, in a type-theoretic foundational proof-carrying code setting. For further details and examples, the reader is referred to the companion technical report [17].

References

1. C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–585, 1969.
2. K. R. Apt, "Ten years of Hoare's logic: A survey, part I," *ACM Transactions on Programming Languages and Systems*, vol. 3, no. 4, pp. 431–483, 1981.
3. E. W. Dijkstra, *A Discipline of Programming.* Prentice Hall, 1976.
4. D. Gries, *The Science of Programming.* Springer-Verlag, 1981.
5. G. Necula, "Proof-carrying code," in *Proceedings of the 24th ACM Symposium on the Principles of Programming Languages*, pp. 106–119, 1997.
6. A. W. Appel, "Foundational proof-carrying code," in *Proceedings 16th IEEE Symposium on Logic in Computer Science*, pp. 247–258, 2001.
7. Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou, "A type system for certified binaries," *ACM Transactions on Programming Languages and Systems*, vol. 27, no. 1, pp. 1–45, 2005.
8. K. Crary and J. C. Vanderwaart, "An expressive, scalable type theory for certified code," in *Proceedings of the 7th ACM International Conference on Functional Programming*, pp. 191–205, 2002.
9. W. A. Howard, "The formulae-as-types notion of constructions," in *To H. B. Curry: Essays on Computation Logic, Lambda Calculus and Formalism*, Academic Press, 1980.
10. T. Coquand and C. Paulin-Mohring, "Inductively defined types," in *Proceedings of the International Conference on Computer Logic (COLOG'88)* (P. Martin-Löf and G. Mints, eds.), vol. 417 of *Lecture Notes in Computer Science*, pp. 50–66, Springer-Verlag, 1990.
11. B. Werner, *Une Théorie des Constructions Inductives.* Thèse de doctorat, Université Paris VII, 1994.
12. The Coq Proof Assistant Reference Manual, URL: `http://coq.inria.fr/`.
13. Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions.* Springer-Verlag, 2004.
14. N. A. Hamid and Z. Shao, "Interfacing Hoare logic and type systems for foundational proof-carrying code," in *Proceedings of the 17th International Conference on the Applications of Higher Order Logic Theorem Proving*, vol. 3223 of *LNCS*, pp. 118–135, Springer-Verlag, 2004.
15. M. Franssen and H. de Swart, "Cocktail: A tool for deriving correct programs," *Revista de la Real Academia de Ciencias, Ser. A*, vol. 98, no. 1, pp. 95–111, 2004.
16. N. S. Papaspyrou, D. C. Vytiniotis, and V. M. Koutavas, "Logic-enhanced type systems: Programming language support for reasoning about security and other program properties," in *Proceedings of the 4th Panhellenic Logic Symposium*, pp. 141–145, 2003.
17. N. S. Papaspyrou, M. A. Papakyriakou, and A. Manousaridis, "Encoding Hoare logic in typed certified code," Tech. Rep. CSD-SW-TR-1-05, Software Engineering Laboratory, National Technical University of Athens, 2005.