A Hoare Logic Contract Theory: An Exercise in Denotational Semantics

Dilian Gurov and Jonas Westman

Abstract We sketch a simple theory of Hoare logic contracts for programs with procedures, presented in denotational semantics. In particular, we give a simple semantic justification of the usual procedure-modular treatment of such programs. The justification is given by means of a proof of soundness of a *contract-relative* denotational semantics against the standard denotational semantics of procedures in the context of procedure declarations. The suggested formal development can be used as an inspiration for more ambitious contract theories.

1 Introduction

Hoare logic [2, 5] is a well-established logic for reasoning about programs. Its judgments express what a program is intended to compute by relating the values of its variables before and after executing the program. This is adequate when the program is not interacting with its environment (i.e., when it has no side-effects), and when it is expected to terminate; mathematically speaking, from a user's view what such a program does is completely captured by a binary relation on states, initial and final ones, respectively. In Hoare logic, such a binary relation is expressed by means of a pair of logical assertions, called precondition and postcondition, respectively. These assertions are essentially first-order logic formulas over program variables and so-called logical variables; the latter are used to relate values before and after execution. The program and the two assertions form a so-called Hoare triple. What is referred to as Hoare logic is essentially a deductive proof system over Hoare triples.

D. Gurov (✉)
KTH Royal Institute of Technology, Stockholm, Sweden
e-mail: dilian@kth.se

J. Westman
KTH Royal Institute of Technology, Stockholm, Sweden

Systems Development Division, Scania AB, Södertälje, Sweden

P. Müller, I. Schaefer (eds.), *Principled Software Development*,
https://doi.org/10.1007/978-3-319-98047-8_8

Meyer advocated in [6] the use of Hoare-style preconditions and postconditions as *contracts* for programs under development, and a design methodology called *design-by-contract*. Contracts support *modular* development of software: if one program relies on another one, the two can be decoupled by means of a contract for the latter program; the contract is what the second program is obliged to fulfill toward the first. The first program can then be developed relying on this contract, without requiring access to, or knowledge of, the implementation of the second program (which may or may not be available). Furthermore, the first program can be *verified* to meet its own contract, under the assumption that the second program meets its contract. Besides the methodological advantages of using contracts, this has the effect that verification also becomes modular and therefore scales well with the number of modules. Many well-known tools for deductive verification, such as OpenJML [4] and VCC [3], are in fact *procedure-modular*: they expect every procedure to be accompanied by a contract, and verify each procedure in isolation.

Procedures can be mutually recursive. For instance, consider the Java program shown in Fig. 1. It consists of two procedures (or "methods"), `even` and `odd`, which call each other in order to determine whether their argument is an even number or an odd one, respectively. Both procedures are equipped with a (JML-like) contract, consisting of a precondition, expressed in a `requires` annotation, and a postcondition, expressed in an `ensures` annotation. In the latter, `\result` refers to the value returned by the respective procedure, while `n` refers to the value of the formal parameter at the time of invoking the procedure (but in classical Hoare logic this would be expressed by means of logical variables).

We can verify each procedure against its own contract in isolation, by just assuming that the procedures it calls meet their respective contracts. For instance, we can infer from the precondition of procedure `even` and its statements, that at the control point where it calls procedure `odd`, the value of variable `n` must be

```
public class EvenOdd {

   //@ requires n >= 0;
   //@ ensures \result == (\exists int k; n == 2 * k);
   public boolean even(int n) {
      if (n == 0) return true;
      else return odd(n-1);
   }

   //@ requires n >= 0;
   //@ ensures \result == (\exists int k; n == 2 * k + 1);
   public boolean odd(int n) {
      if (n == 0) return false;
      else return even(n-1);
   }
}
```

Fig. 1 A Java program with mutually recursive procedures

positive (since it is assumed to be non-negative in the precondition, and must be different from zero in the `else`-branch), and hence `n-1` must be non-negative. The precondition of `odd` is thus met, and we can therefore assume that upon return from this procedure call, by virtue of its postcondition, the value returned by `even` will be `true` for `n` if the value of `n-1` is odd, and *vice versa*, thus entailing the postcondition of `even`.

But is such such a circular, assume-guarantee style reasoning *sound*? It is well-known from the literature that for *partial correctness*, where termination is not required, this indeed is the case (see e.g. [8]). The usual way of showing this is based on a natural (or "big-step") operational semantics of the programming language, consisting of a set of derivation rules, and a proof system in the form of a sequent calculus over Hoare triples, also presented as a set of derivation rules. Proving soundness thus typically requires proofs by induction on the height of derivation trees, which can be rather cumbersome. We believe that an elegant, and more economic, alternative way of showing soundness could be based on *denotational semantics* and fixed-point theory. In this paper, we sketch our idea on a toy programming language with procedures, and argue that more ambitious contract theories can be developed following the suggested scheme.

Structure of the Paper We start by recalling in Sect. 2 the denotational semantics of a toy programming language without procedures. We then define, also in denotational semantics, the semantics of Hoare logic contracts. Next, in Sect. 3, we extend the programming language and its semantics with procedures. In addition to the usual denotational treatment of procedures, we propose an alternative, contract-relative semantics, and show it to be sound with respect to the former one. We conclude in Sect. 4.

2 Programs Without Procedures

We start with a toy programming language, which is imperative, has no procedures, and is sequential and deterministic.

2.1 Syntax and Denotational Semantics

The typical toy imperative programming language considered in the literature is generated by the following BNF grammar, where x ranges over (integer) program variables, a over arithmetic expressions, b over Boolean expressions, and S over statements:

$$S \ ::= \ \mathsf{skip} \mid \mathsf{x} := \mathsf{a} \mid S_1; S_2 \mid \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 \mid \mathsf{while}\ b\ \mathsf{do}\ S$$

the meaning of which is well-understood.

A program *state* s is defined as a mapping from the variables of the program to their respective domains of values (in this case the integers). The set of all states shall be denoted by **State**.

The "direct style" denotation $[\![S]\!]$ of a statement S is traditionally defined as a single mathematical object, namely as a partial function on **State**, with $[\![S]\!](s) = s'$ meaning that the execution of statement S from state s terminates in the state s', and $[\![S]\!](s)$ being undefined meaning that the execution of statement S from state s does not terminate. A more general approach, which also encompasses non-determinism, is to define $[\![S]\!]$ as a binary relation on **State**, with $(s, s') \in [\![S]\!]$ meaning that there is an execution of statement S from state s that terminates in the state s'. To be able to unify the denotation of statements with that of Hoare logic contracts, we shall adopt the latter approach here.

The meaning of the program constructs is given by induction on the structure of statements, via defining equations. For instance, the meaning of sequential composition is given by the equation:

$$[\![S_1; S_2]\!] \stackrel{\text{def}}{=} [\![S_1]\!] \circ [\![S_2]\!]$$

i.e., as relation composition. The most involved case is the defining equation for the while loop: it is defined as the least solution to a semantic equation derived from the (intended) equality of the denotation of a loop and its unfolding. For details, the interested reader is referred to standard textbooks such as [7, 12].

2.2 Hoare Logic and Contracts

Hoare logic is based on so-called *Hoare triples* of the shape $\{P\}S\{Q\}$, where S is a statement, and where P and Q are logical formulas called *assertions*, which are interpreted over states. We denote by $s \models P$ the fact that the assertion P is true in state s.

For Hoare logic without logical variables, a Hoare triple $\{P\}S\{Q\}$ is defined to be *valid* w.r.t. partial correctness, denoted $\models_{par} \{P\}S\{Q\}$, if for every state s such that $s \models P$, if execution of S from s terminates in a state s', then $s' \models Q$. The addition of logical variables requires the semantics to be relativised on interpretations $\mathscr{I}$ that map logical variables to values; the Hoare triple is then valid if for every state s and interpretation $\mathscr{I}$ such that $s \models_{\mathscr{I}} P$, if execution of S from s terminates in a state s', then $s' \models_{\mathscr{I}} Q$. For the details of the formalization, we refer again to [12].

We refer to the pair $C = (P, Q)$ as a *Hoare logic contract*. We say that a particular implementation S *meets* the contract C w.r.t. partial correctness, denoted $S \models_{par} C$, if and only if the corresponding Hoare triple is valid, i.e., if $\models_{par} \{P\}S\{Q\}$.

2.3 The Denotational Semantics of Contracts

A natural way to define formally the semantics of a contract is to define it as the set of denotations of the programs that satisfy it. This is for instance the approach taken in [9]. In the present case, however, we can take a simpler approach and define the denotation $[\![C]\!]$ of a contract $C = (P, Q)$ in the *same domain* as the denotation of programs, namely as a binary relation on **State**. This will allow us later to give a simple definition of a contract-relative semantics for programs with procedures.

Definition 1 (Denotation of Contracts) Let $C = (P, Q)$ be a Hoare logic contract. The *denotation* of the contract is defined as follows:

$$[\![C]\!] \stackrel{\text{def}}{=} \left\{(s, s') \mid \forall \mathscr{I}.\ (s \models_{\mathscr{I}} P \Rightarrow s' \models_{\mathscr{I}} Q)\right\}$$

We then obtain the following simple set-theoretic characterization of a program meeting a contract:

$$S \models_{par} C \quad \text{iff} \quad [\![S]\!] \subseteq [\![C]\!]$$

which could also be considered as an alternative definition of this notion.

Many formal frameworks express satisfaction of a contract (or specification) through set inclusion, and the theoretical implications of this are well understood. For instance, we obtain a natural notion of *precision* in the lattice of denotations.

3 Programs with Procedures

We now extend the above treatment to programs with procedures. For brevity, we shall assume that procedures do not take parameters or return values, and that they do not declare local variables; all these features can be encoded using dedicated (global) variables.

3.1 Syntax and Denotational Semantics

Consider that we extend our toy programming language with the statement call p, where p ranges over a set P of names of procedures *declared* using the syntax $p = S_p$. A *program with procedures* is a statement S in the context of a set of procedure declarations.

To give a denotational semantics of such programs, the denotation of call p needs to be defined. Considering non-recursive programs, a straightforward approach is to define $[\![\text{call } p]\!]$ to be equal to the denotation $[\![S_p]\!]$ of S_p, i.e., of the *body* of procedure p, and rely on the structural induction approach of Sect. 2.1 to give the

program a meaning. However, this will not work for the general case with recursive programs, since the structural induction will become circular.

To handle the general case, a more indirect approach is taken, as in [12], by introducing: for each $p \in P$, a variable X_p ranging over **State** $\times$ **State**, and a function ρ, called *environment*, which maps each variable X_p to a binary relation on states, i.e., $\rho(X_p) \subseteq$ **State** $\times$ **State**. The denotation of statements S is relativized in terms of this environment as $[\![S]\!]_\rho$; in particular, $\left[\!\left[\mathsf{call}\ p\right]\!\right]_\rho \stackrel{\mathsf{def}}{=} \rho(X_p)$, while the defining equations for the other language constructs remain unchanged.

Notably, this approach gives rise to a system of equations:

$$\left\{X_p = [\![S_p]\!]_\rho\right\}_{p \in P}$$

Every solution of this system can be seen as an environment itself. Viewing ρ as a variable, we can see the whole system of equations as one equation, but over environments. Thus, semantically, the system induces a function ξ : **Env** $\rightarrow$ **Env** over environments, defined by $\xi(\rho)(X_p) \stackrel{\mathsf{def}}{=} [\![S_p]\!]_\rho$.

Let $\rho \sqsubseteq \rho'$ denote point-wise set inclusion over environments, and let $\rho_\perp$ be the environment mapping every variable to the empty relation. (**Env**, $\sqsubseteq$, $\rho_\perp$) can be shown to be a chain-complete partial order with bottom, in which the function ξ is continuous (i.e., ξ is monotone and respects least upper bounds of ω-chains). By the well-known Knaster-Tarski fixed-point theorem, ξ has a least and a greatest fixed point. We take the least fixed point ρ_0, which constitutes the least solution to the equation system, and define the semantics of a program with procedures S relative to this particular solution, i.e.:

$$[\![S]\!] \stackrel{\mathsf{def}}{=} [\![S]\!]_{\rho_0}$$

3.2 A Contract-Relative Denotational Semantics

For a program with procedures, S, consider a contract C for the statement S and a set of contracts $\{C_p\}_{p \in P}$, one for each declared procedure p.

The set of contracts $\{C_p\}_{p \in P}$ gives rise to an alternative, *contract-relative* notion $[\![S]\!]^{cr}$ of the denotational semantics of programs with procedures. To this end, we introduce the special *contract environment* ρ_c defined by $\rho_c(X_p) \stackrel{\mathsf{def}}{=} [\![C_p]\!]$, for each $p \in P$. Notice that this is only possible because of our careful choice of denotational semantics of contracts, made in Sect. 2.3. We base the semantics of programs with procedures on this environment, instead of on ρ_0, and define:

$$[\![S]\!]^{cr} \stackrel{\mathsf{def}}{=} [\![S]\!]_{\rho_c}$$

Notice that this semantics is *not* recursive and does not involve fixed points: the contract-relative denotation $[\![S_p]\!]^{cr}$ of every procedure body can be computed *procedure-modularly*, i.e., independently of each other. We thus obtain an alternative, contract-relative notion of a statement meeting a contract:

$$S \models^{cr}_{par} C \quad \text{iff} \quad [\![S]\!]^{cr} \subseteq [\![C]\!]$$

The contract-relative semantics embodies an *assume-guarantee* style treatment of programs with procedures: the contract-relative denotation of a statement provides a set-theoretic "upper bound" on its standard denotation, assuming upper bounds on the denotations of the called procedures as specified by their contracts. Procedure-modular verification as used in practice essentially follows this treatment.

3.3 Soundness of the Contract-Relative Semantics

In fixed-point theory, a common technique to prove that a point x is greater than the least fixed point y of a continuous function f is to show that x is a pre-fixed point of f, i.e., that $f(x) \sqsubseteq x$. Since the least fixed point of a continuous function is also its least pre-fixed point, it follows that $y \sqsubseteq x$.

We have a similar situation here. Intuitively, the environment ρ_0, which by definition is the least fixed point of ξ, embodies the denotation (i.e., meaning) of the procedure declarations. The contract environment ρ_c embodies the denotation of the contracts of the declared procedures, while $\xi(\rho_c)$ embodies the contract-relative denotation of the procedure declarations. To show that the procedure declarations meet their contracts, we just show that their contract-relative denotation meets the contract (as we essentially do when we apply procedure-modular verification); in this way we establish that the denotation of the contracts ρ_c is a pre-fixed point of ξ.

Thus, our contract-relative semantics is *sound* w.r.t. the standard one, whenever all declared procedures meet their contracts.

Theorem 1 (Soundness) *Let S be a program with procedures as described above, and let for all $p \in P$, $S_p \models^{cr}_{par} C_p$. We have:*

$$S \models^{cr}_{par} C \quad \textit{implies} \quad S \models_{par} C$$

Proof We show that $[\![S]\!] \subseteq [\![S]\!]^{cr}$, from which soundness follows. Since by assumption $S_p \models^{cr}_{par} C_p$ for all $p \in P$, we have $[\![S_p]\!]^{cr} \subseteq [\![C_p]\!]$ and hence $[\![S]\!]_{\rho_c} \subseteq [\![C_p]\!]$ for all $p \in P$. Therefore, by the definitions of ξ and ρ_c, we have $\xi(\rho_c)(X_p) \subseteq \rho_c(X_p)$ for all $p \in P$. Thus, $\xi(\rho_c) \sqsubseteq \rho_c$, i.e., ρ_c is a pre-fixed point of ξ. Since ρ_0 is the least fixed point of ξ, ρ_0 is also its least pre-fixed point, and therefore $\rho_0 \sqsubseteq \rho_c$. By monotonicity of ξ, we obtain $[\![S]\!]_{\rho_0} \subseteq [\![S]\!]_{\rho_c}$, and therefore $[\![S]\!] \subseteq [\![S]\!]^{cr}$. □

4 Conclusion

We presented a simple treatment of Hoare logic contracts in denotational semantics. It gives rise to a procedure-modular, contract-relative semantics of statements in the context of a set of procedure declarations. We showed this semantics to be sound w.r.t. the standard denotational semantics of procedural languages, which is not procedure-modular. The proof of soundness is simple, utilizing the fact that the denotations of the Hoare logic contracts of the declared procedures constitute a pre-fixed point of the semantic function used to define the standard denotational semantics, as long as all procedures meet their contracts procedure-modularly.

One concrete application of the above framework that we are currently developing is a formal justification of a technique to automatically compute procedure contracts. What the framework gives us is a notion of precision of contracts w.r.t. the (standard) denotation of their respective procedure bodies. We will use this notion to characterize the technique.

We presented here the treatment of Hoare logic contracts in the context of a toy programming language. Building on previous work [10, 11], we plan to extend our contract theory to the far more ambitious case of embedded C code. Because of the interactive nature of such code, the denotation of statements cannot be adequately defined as a binary relation on states; instead, we plan to develop the framework around the theory of nested words presented in [1].

Acknowledgement We thank Wolfgang Ahrendt for valuable comments on an earlier draft of the paper.

References

1. Rajeev Alur and Swarat Chaudhuri. “Temporal Reasoning for Procedural Programs”. In: *Verification, Model Checking and Abstract Interpretation (VMCAI 2010)*. Vol. 5944. Lecture Notes in Computer Science. Springer, 2010, pp. 45–60.
2. Krzysztof R. Apt. “Ten Years of Hoare’s Logic: A Survey Part 1”. In: *ACM Transactions on Programming Languages and Systems* 3.4 (1981), pp. 431–483. https://doi.org/10.1145/357146.357150. URL: http://doi.acm.org/10.1145/357146.357150
3. E. Cohen et al. “VCC: A Practical System for Verifying Concurrent C”. In: *Theorem Proving in Higher Order Logics (TPHOLs 2009)*. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 23–42.
4. David R. Cok. “OpenJML: JML for Java 7 by Extending OpenJDK”. In: *NASA Formal Methods (NFM 2011)*. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 472–479.
5. C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (1969), pp. 576–580.
6. Bertrand Meyer. “Applying “Design by Contract””. In: *IEEE Computer* 25.10 (1992), pp. 40–51.
7. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer* Berlin, Heidelberg: SpringerVerlag, 2007. ISBN: 1846286913.

8. David von Oheimb “Hoare Logic for Mutual Recursion and Local Variables”. In: *Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1999)*. Vol. 1738. Lecture Notes in Computer Science. Springer, 1999, pp. 168–180.
9. Dimitrios Vytiniotis et al. “HALO: Haskell to logic through denotational semantics”. In: *Proceedings of POPL 2013*. ACM, 2013, pp. 431–442.
10. Jonas Westman and Mattias Nyberg. “Conditions of contracts for separating responsibilities in heterogeneous systems”. In: *Formal Methods in System Design* 52.2 (2018), pp. 147–192.
11. Jonas Westman et al. “Formal architecture modeling of sequential non-recursive C programs”. In: *Science of Computer Programming* 146 (2017), pp. 2–27.
12. Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA, USA: MIT Press, 1993. ISBN: 0-262-23169-7.