

Towards Mechanized Program Verification with Separation Logic^{*}

Tjark Weber

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany
`webertj@in.tum.de`

Abstract. Using separation logic, this paper presents three Hoare logics (corresponding to different notions of correctness) for the simple While language extended with commands for heap access and modification. Properties of separating conjunction and separating implication are mechanically verified and used to prove soundness and relative completeness of all three Hoare logics. The whole development, including a formal proof of the Frame Rule, is carried out in the theorem prover Isabelle/HOL.

Keywords. Separation Logic, Formal Program Verification, Interactive Theorem Proving

1 Introduction

Since C. A. R. Hoare’s seminal work in 1969 [9], extensions of his logic have been developed for a multitude of language constructs [1, 2], including recursive procedures, nondeterminism, and even object-oriented languages. Extending Hoare logic to pointer programs however is not without difficulties. Recently separation logic was proposed by O’Hearn, Reynolds et al. [15, 19, 16] to overcome the local reasoning problem that is raised by the treatment of record components as arrays [6, 5].

Machine support is indispensable for formal program verification. Manual proofs are error-prone, and the verification of medium-sized programs has become feasible only because systems like SVC [3] can automatically discharge many proof obligations. Separation logic, although its usability has been demonstrated in several case studies [18, 4], currently lacks such support. In this paper we show how separation logic can be embedded into the theorem prover Isabelle/HOL [14]. We thereby lay the foundations for the use of separation logic in a semi-automatic verification tool. Our work is based on a previous formalization of a simple imperative language [12] which however did not consider pointers or separation logic. The current focus is on fundamental semantic properties of the resulting Hoare logics.

This paper is organized as follows. In Section 2 we define the programming language, together with its operational and denotational semantics. Section 3

^{*} Research supported by *Graduiertenkolleg Logik in der Informatik* (PhD Program Logic in Computer Science) of the Deutsche Forschungsgemeinschaft (DFG).

introduces separation logic. In Section 4 we present three Hoare logics for our language, all of which are proved to be sound and relative complete. Also the Frame Rule is addressed, and its soundness is proved for one of the Hoare logics. We discuss the mechanical verification of a simple pointer algorithm, in-place list reversal, in Section 5.

2 The Language

2.1 Semantic Domains

We use an unspecified type *var* of variables. Addresses are elements of a numerical type, namely naturals (*nat*), to permit address arithmetic. For simplicity, the same type is used for values. Thereby the value of a variable can immediately be used as an address, with no need for a conversion function (cf. [16]).

Stores map variables to values. Heaps are modelled as partial functions from addresses to values. Other possibilities would be to define heaps as subsets of $addr \times val$ (with functionality constraints), or as $(addr \times val)$ *list* (again with functionality constraints, and modulo order). However, our current definition is much easier to state and work with in Isabelle/HOL since it can make use of readily available function types and does not require subtyping. On the other hand it also permits infinite heaps. This seemingly minor difference will become important again in Section 4.4, when we consider the Frame Rule.

A program state is either a pair consisting of a store and a heap, or *None*. The latter value will be used in the semantics of the language to indicate that a memory error occurred during program execution. Arithmetic and boolean expressions are only modelled semantically: they are just functions on stores (and hence independent of the heap).

Most of Isabelle's syntax used in this paper is close to standard mathematical notation and should not require further explanation. Both \implies and \longrightarrow mean implication. $\llbracket P_1 ; \dots ; P_n \rrbracket \implies Q$ is an abbreviation for $P_1 \implies \dots \implies P_n \implies Q$. We use $'a \Rightarrow 'b$ for the type of total functions from $'a$ to $'b$. Likewise, infix \multimap is used to denote the type of partial functions. Other type constructors, e.g. *list*, are written postfix. Thus the abovementioned semantic domains can be formalized as follows:

```
types addr = nat
      val   = nat
      store = var  $\Rightarrow$  val
      heap  = addr  $\multimap$  val
      state = (store  $\times$  heap) option
      aexp  = store  $\Rightarrow$  val
      bexp  = store  $\Rightarrow$  bool
```

2.2 Syntax

We consider an extension of the simple While language [9, 12] with new commands for memory allocation (*list*, *alloc*), heap lookup, heap mutation, and memory deallocation (*dispose*).