A Three-Valued Hoare Calculus

Viviana Bono
Dipartimento di Informatica
Università di Torino
Corso Svizzera 185
I-10149 Torino, Italy
`www.di.unito.it/~bono`
`bono@di.unito.it`

Manfred Kerber
School of Computer Science
The University of Birmingham
Birmingham B15 2TT
England
`www.cs.bham.ac.uk/~mmk`
`M.Kerber@cs.bham.ac.uk`

If we consider programs on concrete computers then physical limitations have to be taken into account. The implemented integers, for instance, can form only a subset of all (mathematical) integers. This can be done (and is done in many programming languages) in the form that two integers MIN and MAX are defined and any integer computation that leads outside this range of integers leads to an error state. We will introduce a Hoare calculus that can deal with this phenomenon in an adequate way.

As soon as we consider concrete computers, functions are typically partial. Even addition is no longer total, since in a finite subset of $\mathbb{Z}$, the sum of two big enough numbers will exceed any given bound (assumed addition in the subset is a restriction of standard addition). Let us assume that we have in our programming language integers which are defined as $\mathbb{Z}_f = \mathbb{Z} \cap [\mathrm{MIN}, \mathrm{MAX}]$, where $\mathbb{Z}$ is the mathematical (infinite) set of all integers. Let us define $\mathrm{MAX} = 32768$ and $\mathrm{MIN} = -32767$, for instance. With this setting $+$ is partial, since $\mathrm{MAX} + 1$ is not defined and leads to a crash.

If we want to reason about crash as well, we have to consider how a Hoare triple

$$\{precondition\}\ \mathtt{P}\ \{postcondition\}$$

should read. We keep the original reading: *If the precondition holds and the program terminates then the postcondition holds.* We could also have given a Hoare triple a weaker meaning, namely: "*If the precondition holds, the program does not crash and terminates, then the postcondition holds.*" We don't follow this possible reading, since we want to be able to prove that the program does not crash and don't want to assume it.

Our programming language is a minimal language that has two types, the usual constructs of a Turing complete language, and the possibility of crash.

- The program that does not do anything: `skip`.
- The program that just crashes: `crash`.
- Assignment: `x` := `t` assigns the term `t` of type `int` to the variable `x`.
- Sequencing: `P;Q` means execute first `P` then `Q`.
- Conditional: `IF C THEN P ELSE Q FI` means that if the boolean expression `C` (of type `bool`) is true then `P` will be executed. If `C` is false then `Q` will be executed. To build conditional expressions we make use of the primitive binary predicates $=$, $<$, and $<=$, which result with terms `s` and `t` of type `int` in boolean expressions of type `bool`. Note that a conditional expression may crash, e.g., for $\mathtt{MAX} < \mathtt{MAX} + \mathtt{1}$.
- Loop: `WHILE C DO P OD` means that if the boolean expression `C` (of type `bool`) is true the program fragment `P` will repeatedly be executed until either `C` is false or crashes, or `P` itself crashes.
- The last construct is a finite version for a loop traversed at most k times (for a natural number k with a new counter nc, which must not occur in the program `P`). We write: `WHILEF (nc k) C DO P OD`. We are not interested in programs which make use of this construct, but use it only as a vehicle to reason about crashes of while loops.

An example program `P` is:

```
WHILE i < n DO
  i   := i + 1;
  sum := sum + i
OD
```

which assumes `i`, `n`, and `sum` to be defined and to contain values of type `int`. With `sum` considered to be 0 initially, the program adds up the first `n` natural numbers and the result will be `sum`, if `sum` is smaller than or equal to MAX; otherwise the program will crash.

Logic

In order to speak about crash, we need to know when an expression is defined and when not. For instance, if we assume integer division with rest, $\frac{x}{y}$ is defined for any x and any $y \neq 0$. That is, the expression is defined for

$\{(x,y)\,|\,x \in \mathbb{Z} \wedge y \in \mathbb{Z} \wedge y \neq 0\}$. We write $\mathbf{D_{term}}(\frac{x}{y}) \leftrightarrow x \in \mathbb{Z} \wedge y \in \mathbb{Z} \wedge y \neq 0$. $\mathbf{D_{term}}$ is a predicate symbol which expects a *term* (indicated by the index), and gives back a truth value true or false (that is, we assume that $\mathbf{D_{term}}$ itself never crashes). Likewise we will have a construct which tells us whether a *formula* is undefined, e.g., whether $\frac{x}{y} \doteq 0$ is defined or not. We write in this case $\mathbf{D}(\frac{x}{y} \doteq 0)$. $\mathbf{D}$ is a connective which is true iff the formula is true or false, and false otherwise. Together with the traditional connectives $\neg$, $\wedge$, $\vee$, and $\rightarrow$, this constitutes the propositional logic fragment of the three-valued language. Terms are variables, constant symbols, or applications of function symbols to terms. They are interpreted in a standard non-empty domain D, or as an error element error. We assume all functions to be strict, that is, if a subterm of a term is mapped to the error element, then the whole term will be mapped to the error element.

We define the semantics of the connectives as follows:

$\neg$	
false	true
crash	crash
true	false

$\mathbf{D}$	
false	true
crash	false
true	true

$\wedge$	false	crash	true
false	false	false	false
crash	crash	crash	crash
true	false	crash	true

Note that the connective $\wedge$ is not symmetric (but associative). This is made in order to mirror the behaviour of the lazy boolean function `and` in the programming language (and the ones that can be defined like `or`). If the `A` in `A and B` is false the whole expression is false indiscriminately of the `B`. Likewise, if the evaluation of `A` crashes, the whole expression crashes. Only if the `A` evaluates to true, the second, the `B`, is evaluated and its value will be the value of the conjunction.

Quantification will always be guarded by a type T (or sort), $\forall x_T\, A$ and $\exists x_T\, A$. We assume a simple type system of (non-empty) disjoint types (we consider only $T = \mathbb{Z}$).

We define $I_\phi(\mathsf{Q} x_S A) := \widetilde{\mathsf{Q}}(\{I_{\phi,[a/x]}(A)\,|\,a \in A_S\})$ where $\mathsf{Q} \in \{\forall, \exists\}$ and furthermore

$$\widetilde{\forall}(T) := \begin{cases} \mathsf{true} & \text{for } T = \{\mathsf{true}\} \\ \mathsf{crash} & \text{for } T = \{\mathsf{true}, \mathsf{crash}\} \text{ or } \{\mathsf{crash}\} \\ \mathsf{false} & \text{for } \mathsf{false} \in T \end{cases}$$

From the semantics we can directly construct a tableau calculus analogously to the work in Kerber and Kohlhase (1996).

Hoare calculus

The Hoare calculus that can deal with crash makes use of the assignment schema:

$\{\mathbf{D_{term}}(t) \wedge A^t_x\}\ \mathtt{x := t}\ \{\mathbf{D_{term}}(x) \wedge A\}$ with $A^t_x = A[x/t]$ replace all free x in A by t.

and an axiom schema which says that we can't recover from crash: $\{\mathsf{CRASH}\}\,\mathtt{P}\,\{\mathsf{CRASH}\}$

The rules consider the possibilities how a crash can be avoided or obtained, for instance, for an implication this means:

$$\frac{A \rightarrow \mathbf{D}(C) \qquad \{A \wedge C\}\,\mathtt{P}\,\{B\} \qquad \{A \wedge \neg C\}\,\mathtt{Q}\,\{B\}}{\{A\}\ \mathtt{IF\ C\ THEN\ P\ ELSE\ Q\ FI}\ \{B\}}$$

$$\frac{A \rightarrow \neg\mathbf{D}(C)}{\{A\}\ \mathtt{IF\ C\ THEN\ P\ ELSE\ Q\ FI}\ \{\mathsf{CRASH}\}}$$

The second rule reads: if the precondition A of a rule implies that the condition C will crash, then the whole if-then-else expression will crash. The first rule is almost the traditional rule with the extension, however, that from the precondition A follows that the condition C will not crash. Note that for the first rule the postcondition B can also be CRASH.

Likewise we can build rules which characterise the behaviour of loops.

We have proved the soundness of the rules relative to an operational semantics following the approach outlined in Mitchell (1996). Proving a relative completeness property is left as future work.

With the calculus it is possible to establish:

$$\left\{\begin{array}{l}\mathbf{D_{term}}(n) \wedge \mathbf{D_{term}}(i) \wedge \mathbf{D_{term}}(sum) \\ \wedge\, 0 \leq n \leq 255 \wedge i \doteq 0 \wedge sum \doteq 0\end{array}\right\} \mathtt{P}$$
$$\left\{\begin{array}{l}\mathbf{D_{term}}(n) \wedge \mathbf{D_{term}}(i) \wedge \mathbf{D_{term}}(sum) \\ \wedge\, sum \doteq \frac{1}{2}n(n+1)\end{array}\right\}$$

and

$$\left\{\begin{array}{l}\mathbf{D_{term}}(n) \wedge \mathbf{D_{term}}(i) \wedge \mathbf{D_{term}}(sum) \\ \wedge\, n \doteq 256 \wedge i \doteq 0 \wedge sum \doteq 0\end{array}\right\} \mathtt{P} \left\{\mathsf{CRASH}\right\}$$

Summary

We have extended the traditional Hoare calculus to a three-valued setting to deal with crash. The meaning of the Hoare triple is that if the precondition holds before the execution of `P` and `P` terminates, then after the execution of `P` the postcondition will hold. If the postcondition says that the variables are in a particular state, then the program will not crash, if the postcondition says CRASH then the program will crash.

References

Manfred Kerber and Michael Kohlhase. A tableau calculus for partial functions. *Collegium Logicum – Annals of the Kurt-Gödel-Society*, **2**:21–49, 1996.

John C. Mitchell. *Foundations of Programming Languages*. MIT Press, Cambridge, Massachusetts, USA, 1996.