Hoare Calculation and its Application

Group 2: From Models to Software
Robert Lang TUM

Abstract. Modern software uses lots of new and groundbreaking algorithms to handle with more and more complicated tasks in less and less time. These algorithms are high efficient, but also hard to understand in their stucture. Most of the time you are interested in ensuring absolutely the correctness of the program and its algorithms. Especially for safety critical application as you can find in medical science, aerospace industry or military, this insurance is one of the most important aspects in developing software solutions. The Hoare Calculation allows to meet that challenge and to prove the correctness of algorithms or to find errors in software and to fix them.

1 Hoare Rules

1.1 C.A.R. Hoare[1]

Sir Charles Anthony Hoare was born on 11. January 1934 Colombo, Sri Lanka. He studied at Oxford and at Moscow State University computer translation of human languages. Hoare is known for the development of Quicksort (Hoaresort) in 1960 and the Hoare Logic (or Hoare Calculation). His programming model Communicating Sequential Processes (CSP), a formal language, affected significantly the engine of the languages Ada and Occam. Today Hoare is Emeritus Professor at Oxford and senior researcher at Microsoft Resarch in Cambridge. The picture[2] was made in 2005.

"I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies." Tony Hoare

[1] Reference [W1]
[2] Reference [W2]

1.2 Hoare Triple[3]

A central concept in Hoare Calculation is the Hoare Triple

$$\{\texttt{P}\}\ \mathbf{S}\ \{\texttt{Q}\}.$$

`P` and `Q` are predicates[4] with values `true` or `false`. **S** is a statement, a program with correct syntax. Hoare Triples are binary expressions with values `true` or `false`.

$\{\texttt{P}\}\ \mathbf{S}\ \{\texttt{Q}\} = \texttt{true} :\Leftrightarrow$ If the predicate $\{\texttt{P}\}$ is true immediately before execution of **S**, then immediately **S** has terminated, the predicate $\{\texttt{Q}\}$ is true.

For an efficient way of notation the the following symbol is introduced: $\frac{\text{X}}{\text{Y}} :\Leftrightarrow \text{X} \Rightarrow \text{Y}$

1.3 The six Hoare Rules[5]

In the following the Hoare Rules are listed as a Hoare Triple. If there is `true` in the nominator, then there are no preconditions to the assertion. Mind that sometimes in literature the order of the Hoare Rules can differ.

1. Skip-Axiom:

$$\frac{\texttt{true}}{\{\texttt{A}\}\ \mathbf{skip}\ \{\texttt{A}\}}$$

Here **skip** means the program with no commands.

2. Axiom of Assignment:

$$\frac{\texttt{true}}{\{\texttt{A}_{\beta/x}\}\ x := \beta\ \{\texttt{A}\}}$$

$\texttt{A}_{\beta/x}$ is predicate `A`, but x instead of β.

3. Rule of Composition:

$$\frac{\{\texttt{A}\}\ \mathbf{S1}\ \{\texttt{B}\}\ \wedge\ \{\texttt{B}\}\ \mathbf{S2}\ \{\texttt{C}\}}{\{\texttt{A}\}\ \mathbf{S1}, \mathbf{S2}\ \{\texttt{C}\}}$$

4. Rule of Conditional Branching:

$$\frac{\{\texttt{A}\ \wedge\ \texttt{B}\}\ \mathbf{S1}\ \{\texttt{Q}\}\ \wedge\ \{\texttt{A}\ \wedge\ \neg\texttt{B}\}\ \mathbf{S2}\ \{\texttt{Q}\}}{\{\texttt{A}\}\ \mathbf{if}\ \texttt{B}\ \mathbf{then\ S1\ else\ S2\ end\ if}\ \{\texttt{Q}\}}$$

5. Rule of Iteration:

$$\frac{\{\texttt{I}\ \wedge\ \texttt{B}\}\ \mathbf{S}\ \{\texttt{I}\}}{\{\texttt{I}\}\ \mathbf{while}\ \texttt{B}\ \mathbf{loop\ S\ end\ loop}\ \{\texttt{I}\ \wedge\ \neg\texttt{B}\}}$$

Such an `I` is called loop-invariant.

6. Rule of Consequence:

$$\frac{\texttt{A}\ \Rightarrow\ \texttt{A'}\ \ \wedge\ \{\texttt{A'}\}\ \texttt{S}\ \{\texttt{B'}\}\ \wedge\ \ \texttt{B'}\ \Rightarrow\ \texttt{B}}{\{\texttt{A}\}\ \mathbf{S}\ \{\texttt{B}\}}$$

[3] Reference [Cl]
[4] Reference [Gel]
[5] Reference [Hei]

2 Verifying the correctness of software

A very simple problem in software design is to add two non-negative integers x and y. There are several programs to solve this problem. In the following two different algorithms are discussed.

2.1 A first example

```
function result(x,y)
  if x == 0
    return (y);
  else
    result(x-1,y+1);
  end
```

The program should answer the sum of two non-negative integers x and y. In the case of `result(x,y)` the proof of its correctness is quite easy: Proof of the assertion by induction on x.

Induction top: $x = 0 \Rightarrow$ `result(0,y)` $\underbrace{=}_{\texttt{x==0}} y$, and $y = y + x$ ✓

Let the assumption be proved for some $x \in \mathbb{N}_0$ and all $y \in \mathbb{N}_0$.

$\Rightarrow$ `result(x+1,y)` $\underbrace{=}_{\texttt{else}}$ `result(x,y+1)` $\underbrace{=}_{\text{induction hypothese}}$ $x + (y + 1)$

$\Rightarrow$ `result(x+1,y)` $= x + (y + 1) = (x + 1) + y$ □

It is interesting that in the proof the fact that y is a non-negative integer isn't used at all. So the proof shows that for all non-negative integers x and also for all real numbers y the program `result(x,y)` answers the sum of x and y; of course considering the computational accuracy. In the case of `result(x,y)` the correctness could be proved without Hoare Calculation but only with a proof method which is well-known for all mathematicians. This is because of the recursive structure of the algorithm. For this and similar cases the method of induction is a very useful tool for verification.

2.2 A second example

```
function result_2(x,y)
  while x > 0
    x = x-1;
    y = y+1;
  end
  return y;
```

The second program uses a `while`-loop instead of recursive runs. This program is not result of an functional program, but of an iterative one. It is not possible to prove the correctness of `result_2(x,y)` using induction on x like in the proof before, because there's no way to use the induction hypothese. Here the Hoare Rules must be applied. In the following the Rule of Composition is used without saying it explicitly.

2.2.1 Additional lines

The Hoare Rules are listed as Hoare Triples. For their use lines must be added to the programming code to check the preconditions of the Hoare Triples.

```
function result_2(x,y)
```
$\{\text{P: } x \geq 0 \wedge y \geq 0,\ r := x + y\}$
$\{\text{I}\}$
```
while x > 0
```
$\{\text{I} \wedge \text{B}\}$
$\{\text{Item 1}\}$
```
x = x-1;
```
$\{\text{Item 2}\}$
```
y = y+1;
```
$\{\text{I}\}$
```
end
```
$\{\text{I} \wedge \neg\ \text{B}\}$
$\{\text{Q: } y = r\}$
```
return y;
```

Of course the predicate B is given by the condition of the `while`-loop: $x > 0$. As long as this predicate is true, the programming code in the `while`-loop is running. The loop-invariant I is given by the predicate

$$\text{I} : r = x + y.$$

At the beginning I is true by definition, but during the execution x and y change, so this statement is not a trivial statement and must be proved.

2.2.2 The `while`-loop

In the `while`-loop Item 2 is, using the Rule of Assignment, given by

$$\text{Item2} : \{x + (y + 1), x \geq 0\}.$$

Again using the Rule of Assignment, the Item 1 is given by

$$\text{Item1} : \{(x - 1) + (y + 1), x - 1 \geq 0\}.$$

The question is if Item 1 is the same predicate as the first predicate in the `while`-loop

$$\{\text{I} : x + y \wedge \text{B} : x > 0\}.$$

In fact they are the same, because

$$x - 1 \geq 0 \Leftrightarrow x > 0$$

for a non-negative integer x. The down-to-up direction of assignment is a typical way in Hoare Calculation. All in all a proof using Hoare Calculation must work from up to down, because this is the direction of program execution.

2.2.3 The answer

After the `while`-loop, the Rule of Iteration says, that the predicate $\{I \wedge \neg B\}$ is true. So the last task is to show that $\{\text{Q} : \text{y} = \text{r}\}$ is true. This is clear, because $\{\text{P} : x \geq 0 \wedge y \geq 0\}$ and $\neg(x > 0)$, so $x = 0$. But this means that $\{\text{Q} : y = y + x = r\}$, because of the loop-invariant $\{\text{I} : r = x + y\}$. Then the program returns the latest value of y , so `result_2(x,y)` answers the sum of x and y. □

3 Further application in modern software[6]

One big task for numerical mathematics is numerical quadrature. That means the approximation of the functional

$$F(f, a, b) := \int_a^b f(x)\, dx$$

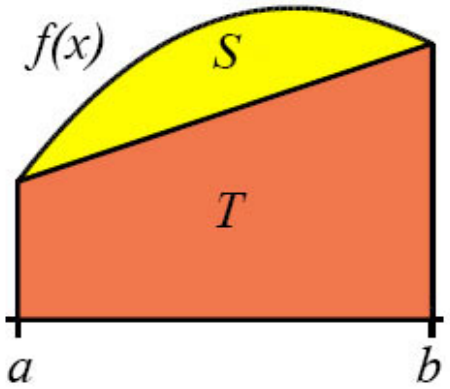
for a sufficiently smooth function $f : [a, b] \to \mathbb{R}$ (e.g. $f \in C^2$). There are many different ways to calculate an approximation, for example using the Newton-Cotes formulas[7]. The 2-point closed Newton-Cotes formula is known as the Trapezodial-Rule. If the interval $[a, b]$ is divided by n grid points into $n + 1$ equidistant intervals, then the Trapezodial-Rule in sum is given by

$$F \approx TS := \frac{b-a}{n+1} \cdot \left[\frac{f(a)}{2} + \sum_{k=1}^{n} f(x_k) + \frac{f(b)}{2} \right].$$

For high-dimensional functions f, the use of the generalized Trapezodial-Rule is quite hard to calculate and not efficient at all. To handle with those functions the method of Hierachical Decomposition is a good approach.

3.1 Hierachical Decomposition

The first step in Hierachical Decomposition is again the Trapezodial-Rule with $n = 0$.



$$F(f, a, b) \approx T(f, a, b) = (b - a) \cdot \frac{f(a) + f(b)}{2}$$

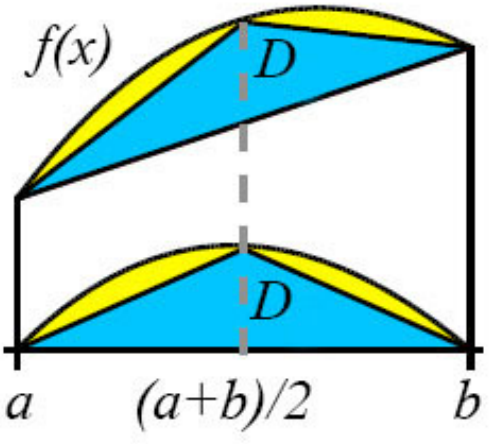
[6] Reference [BaZi], [Ker]
[7] Reference [MW]

So the functional F is approximated by the area of a trapezoid and there is a residuum S:

$$F(f,a,b) = T(f,a,b) + S(f,a,b)$$

The idea now is to decompose the residuum S into a triangle D with projected high

$$h = f\left(\frac{a+b}{2}\right) - \frac{f(a)+f(b)}{2}.$$



It is easy to calculate the area of D using the projected high. So the residuum S can be approximated, but there is a new residuum given by $S'(f,a,b) = S(f,a,b) - D(f,a,b)$. This new residuum can be approximated by using that idea of decomposition recursively. Of course the arguments of the residuum function S change:

$$S(f,a,b) = D(f,a,b) + S(f,a,\frac{a+b}{2}) + S(f,\frac{a+b}{2},b)$$

3.2 Approximation via basis functions

3.2.1 Basis functions

If $u : [a,b] \to \mathbb{R}$ is an approximation to f, then

$$F(f,a,b) \approx F(u,a,b).$$

Let $u(x)$ be a linear combination of basis functions $\Phi_k(x)$:

$$u(x) = \sum_{k=1}^{N} \alpha_k \Phi_k(x)$$

Then it is clear that the functional F is approximated by a linear combination of integrals:

$$F(f,a,b) \approx \sum_{k=1}^{N} \alpha_k \int_a^b \Phi_k(x)$$

This is only helpful for computing if the integrals can be calculated very easily. To ensure this the $\Phi_k(x)$ are defined as “hat functions“. Consider that the index k is a double index n,i:

$$\Phi_{n,i} = \Phi\left(\frac{x - x_{n,i}}{h_n}\right)$$

In this definition the function $\Phi(x)$ is given by

$$\Phi(x) := \max\{1 - |x|, 0\}.$$

h_n mean the mesh size $h_n := 2^{-n}$ and $x_{n,i}$ mean the grid points $x_{n,i} = i \cdot h_n$.

3.2.2 Generator system and basis

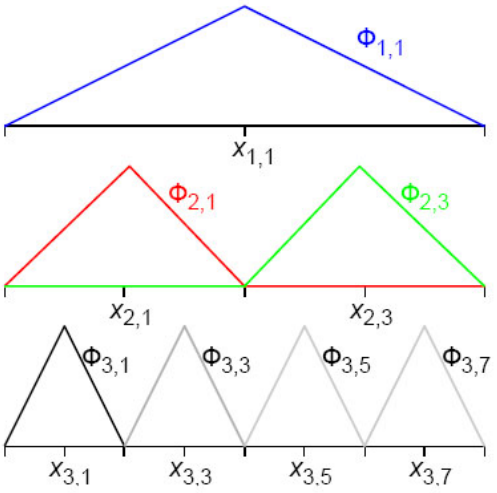
Let V_N be the space of the continuous, on grid h_n piecewise linear functions $u : [0,1] \to \mathbb{R}$ with $u(0) = u(1) = 0$. Then

$$\Psi_N := \bigcup_{n=1}^{N} \{\Phi_{n,i} : 1 \leq i < 2^n\}$$

is a generator system for V_N (but unfortunately for $N > 1$ not a basis). Hence, instead of Ψ_N it is better to use the hierachical basis

$$\Psi_N^H := \bigcup_{n=1}^{N} \{\Phi_{n,i} : 1 \leq i < 2^n \mid i \text{ odd}\}.$$

The picture shows all seven elements of the hierachical basis of V_3.



The task of software using the hierachical basis is to convert the approximating vector $v \in V_N$ with

$$v(x) = \sum_{n=1}^{N} \sum_{i=1}^{2^n - 1} \alpha_{n,i} \Phi_{n,i}(x)$$

into the hierachical basis

$$v(x) = \sum_{n=1}^{N} \sum_{i=1}^{2^n - 1} \alpha'_{n,i} \Phi_{n,i}(x)$$

with $\alpha'_{n,i} = 0$ for all even i.

3.3 Why using the hierachical basis?

The use of the hierachical basis has some advantages in comparison to other methods: During the running time an adaptive stop criterion via the projected high is calculated instantly: If the coefficient $\alpha_{n,i}$ is smaller than a given ϵ, then the deepest iteration of the hierachical basis is reached. It is interesting that then the global error ΔF can be estimated by $\Delta F \leq \epsilon(b - a)$. Furthermore the conversion of the approximation into the hierachical basis leads to a high efficient algorithm for numerical quadrature.

Particulary for functions in high dimensions this is a very good approach. For a d-dimensional function the memory requirements $M \propto N^d$ ($N = \dim V_N$). In comparison with the hierachical basis it can be showed that $M \propto N \cdot (\ln N)^{d-1}$. Getting such an effectiveness must be paid with a very complicated and opaque programming code. Here the methods of the Hoare Calculation can be used to verify the correctness of the coefficients' conversion.

4 References

[BaZi] Michael Bader, Stefan Zimmer: Hierarchische Zerlegung (eindimensional)

[Cl] Volker Claus: Einführung in die Informatik 2005/06 - Kapitel 7: Semantik von Programmen

[Gel] Michael Gellner: Der Umgang mit dem Hoare-Kalkül zur Programmverikfikation

[Hei] Peter Heinig: Program Verification using Hoare Logic - An Introduction

[Ker] Samuel Kerschbaumer: The Hoare Logic - Providing Numerical Algorithms (2006)

[MW] Wolfram MathWorld: http://mathworld.wolfram.com/Newton-CotesFormulas.html

[W1] Wikipedia: http://en.wikipedia.org/wiki/C._A._R._Hoare

[W2] Wikipedia, the free encyclopidea: http://en.wikipedia.org/wiki/Image:CAR_Hoare.jpg