Reasoning About Functions

Jan van Eijck
CWI & ILLC, Amsterdam

August 7, 2012

Module Declaration

```
module Reasoning

where

import Data.List
import System.Random
import System.IO.Unsafe
```

Partiality in Functional Programming

One way to think about a function is as an assignment of values to some objects of a certain type.

In mathematics, $f : A \to B$ expresses that f is a total function from A to B, i.e., for every $x \in A$ there is an $fx \in B$.

This does not quite match the situation in functional programming. `f :: a -> b` does declare `f` as a function from objects of type `a` to objects of type `b`, but there is no assumption that this function is total.

The following is well-defined:

```
bot :: a -> b
bot _ = undefined
```

This defines `bot` as the partial function that is everywhere undefined.

Information Order

Thinking about the class of partial functions from objects of type `a` to objects of type `b`, there is an obvious information order on them. Let us use $fx = \bot$ for 'fx is undefined'. It follows that $fx \neq \bot$ expresses that fx is defined.

We start with a partial order on basic types. For any basic type a, we introduce an object $\bot :: a$ for the undefined object of that type, and we define an ordering on the type by means of

$$x \sqsubseteq y \text{ if } x = \bot \vee x = y.$$

Thus, if u and v are two objects of type a that are different from $\bot$ and from each other, then this ordering is given by:

$$\sqsubseteq = \{(\bot, \bot), (\bot, u), (\bot, v), (u, u), (v, v)\}.$$

Information Order on the Booleans

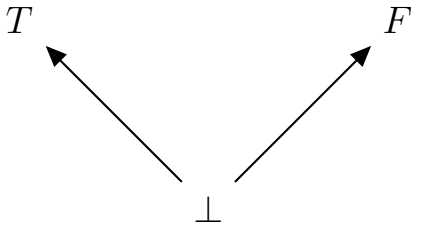
For the particular case of the Booleans, we have:

$$\sqsubseteq = \{(\bot, \bot), (\bot, T), (\bot, F), (T, T), (F, F)\}.$$

The corresponding strict partial order $\sqsubset$ on the Booleans is given by:

$$\sqsubset = \{(\bot, T), (\bot, F)\}.$$

In a picture:



Information Order on Functions

Next, we lift this ordering to functional types, by stipulating:

$$f \sqsubseteq g \text{ iff } \forall x : \text{ if } x \neq \bot \wedge fx \neq \bot \text{ then } fx = gx.$$

Note that $fx \neq \bot$ expresses that f is defined on x. We exclude the situation where $x = \bot$ and fx is defined. In other words, we expect our partial functions to be monotonic for the information order $\sqsubseteq$.

A function f is monotonic for $\sqsubseteq$ if $x \sqsubseteq y$ holds iff $fx \sqsubseteq fy$ holds.

$\sqcap$

Use $f \sqcap g$ for the function h given by:

$$h(x) = \begin{cases} f(x) & \text{if } f(x) = g(x) \\ \bot & \text{otherwise} \end{cases}$$

Partial Functions as a Meet Semilattice

We say that partial functions in $A \to B$ are a meet semilattice for the partial order $\sqsubseteq$.

For every pair of functions f and g with domain A and codomain B, their meet $f \sqcap g$ is defined.

Moreover, $f \sqcap g$ is the largest function h in the $\sqsubseteq$ order that satisfies $h \sqsubseteq f$ and $h \sqsubseteq g$.

Action in Functional Programming

At an abstract level, action is change of state. In imperative programming, it is clear that programming statements denote actions, for the execution of an imperative program has as its effect that the memory state of the machine that runs the program gets modified.

Is there also a notion of action for functional programming? Yes, there is, for instead of change in a memory state we can look at change in a function space. Take a function f, and an operation $[x := d]$ that changes f into f', where f' is given by:

$$f'(x) = d, \text{ and for } y \neq x, f'(y) = f(y).$$

The operation $[x := d]$ has changed the definition of the function.

This kind of **update of a function** is a fundamental operation in many algorithms.

Assigmnent Update

In the semantics of First Order Logic, a variable assignment is a map from the variables of a first order language to the elements of a domain of discourse. Assume V is the list of variables and D is a domain of discourse. Then a variable assignment is a function from a subset of V to D. The notion of truth of a formula in a model is defined in terms of truth of a formula in a model, given a variable assignment. The variable assignment is needed to interpret the free variables that occur in a formula.

Consider the following example. The formula is $\forall x \exists y Rxy$, the domain of the model consists of the first 10 natural numbers $\{0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 9\}$, and the relation symbol R is interpreted as the following set of pairs:

$$\{(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9)\}.$$

Truth of $\forall x \exists y Rxy$

To check whether $\forall x \exists y Rxy$ is true on this domain, under this interpretation for R, check whether $\exists y Rxy$ is true for every variable assignment for x. There are ten such variable assignments:

$$\{(x,0)\}, \{(x,1)\}, \{(x,2)\}, \{(x,3)\}, \{(x,4)\}, \{(x,5)\}, \{(x,6)\}, \{(x,7)\}, \{(x,8)\}, \{(x,9)\}.$$

To check whether $\exists y Rxy$ is true for all of these, we take any one of them, and check whether we can find an appropriate extension with a value for y, to see whether $R(x,y)$ is true for this new assignment. It then turns out that this works for all of them, except for $\{(x,9)\}$. There is no element d in the domain for which $\{(x,9),(y,d)\}$ makes $R(x,y)$ true.

To state the truth definition in full generality, one also needs to cater for the possibility that an assignment function gets changed or updated rather than extended.

Assignment Update, Ctd

The notion of an assignment update also works for assignment functions of type $V \to D$, i.e., for functions that are defined for every variable in the set V. Logic textbooks have notations like

$$g[v/d]$$

or

$$g^v_d$$

or

$$g[v := d]$$

for the assignment that is like g except for the fact that it maps v to d.

Function Update, General Definition

We call this an update of a function. Here is the general definition of the update of a function for a single argument-value pair:

```
update :: Eq a => (a -> b) -> (a,b) -> a -> b
update f (y,z) x = if x == y then z else f x
```

And here is the update of a function for a list of argument-value pairs:

```
updates :: Eq a => (a -> b) -> [(a,b)] -> a -> b
updates = foldl update
```

These function updates will be a key ingredient in some of the algorithms in later chapters.

Updates and Partiality

We can use updates to change a function from undefined to defined at some particular argument(s).

```
partialSuccessor =
   updates bot [(n,n+1)| n <- [0..100] ]
```

Function Update as a Basic Action

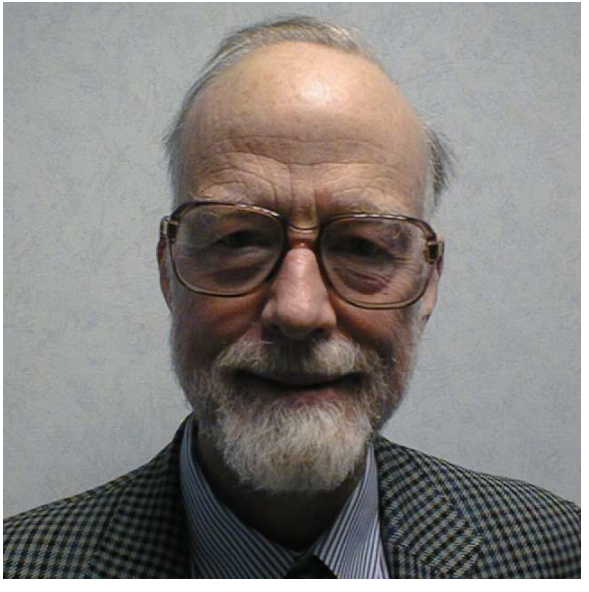
We can look at **function update** as a basic action, and ask ourselves how basic actions are combined, and how the results of performing such combined actions can be specified by means of appropriate properties.

Algorithms are formal descriptions of combinations of basic actions.

Formal specifications of algorithms are formal descriptions of what the algorithms are expected to do. A standard tool for formal specification of algorithms is dynamic logic, a logical tool that allows us to describe the interplay between action and state. Action changes state, and state is what is changed by action.

Functional programming does not focus on machine state, but we will see that talking about state change still makes a lot of sense when analyzing functional programs.

Tony Hoare



About Tony Hoare

- Inventor of the QuickSort Algorithm http://en.wikipedia.org/wiki/Quicksort
- Inventor of Hoare Logic http://en.wikipedia.org/wiki/Hoare_logic
- Inventor of CSP (specification language for concurrent processes) and moving force behind Occam.
- Winner of the 1980 ACM Turing Award
- Main Focus of Research: make software more reliable, explain how software gets more reliable in practice.
- See http://en.wikipedia.org/wiki/Tony_Hoare

Tony Hoare on the Purpose of Testing

> Philosophers of science have pointed out that no series of experiments, however long and however favourable can ever prove a theory correct; but even only a single contrary experiment will certainly falsify it. And it is a basic slogan of quality assurance that ”you cannot test quality into a product”. How then can testing contribute to reliability of programs, theories and products? Is the confidence it gives illusory?
>
> Tony Hoare, How did software get so reliable without proof? [3]

The Purpose of Testing is to Test a Method

> The resolution of the paradox is well known in the theory of quality control. It is to ensure that a test made on a product is not a test of the product itself, but rather of the methods that have been used to produce it — the processes, the production lines, the machine tools, their parameter settings and operating disciplines. If a test fails, it is not enough to mend the faulty product. It is not enough just to throw it away, or even to reject the whole batch of products in which a defective one is found. The first principle is that the whole production line must be re-examined, inspected, adjusted or even closed until the root cause of the defect has been found and eliminated.
>
> Tony Hoare, How did software get so reliable without proof? [3]

Hoare on the Value of Testing

> The real value of tests is not that they detect bugs in the code, but that they detect inadequacy in the methods, concentration and skills of those who design and produce the code. Programmers who consistently fail to meet their test- ing schedules are quickly isolated, and assigned to less intellectually demanding tasks. The most reliable code is produced by teams of programmers who have survived the rigours of testing and delivery to deadline over a period of ten years or more. By experience, intuition, and a sense of personal responsibility they are well qualified to continue to meet the highest standards of quality and reliability. But don't stop the tests: they are still essential to counteract the distracting effects and the perpetual pressure of close deadlines, even on the most meticulous programmers.
>
> Tony Hoare, How did software get so reliable without proof? [3]

Formal Specification With Hoare Triples

In general a triple

initial state – statement – final state

$$\{P\}\ S\ \{Q\}$$

has the following operational meaning:

> If execution of S in a state that satisfies P terminates, then the termination state is guaranteed to satisfy Q.

Such triples $\{P\}\ S\ \{Q\}$ are called Hoare triples after Tony Hoare.

The predicate for the initial state is called the precondition, and the predicate for the final state is called the postcondition.

Key Importance of Hoare Triples for Testing

- Consider a Hoare triple $\{P\}\, S\, \{Q\}$
- Such a triple can be viewed as a specification for S.
- It provides the means to test whether S fits its specification.
- The predicate P specifies which values for the parameters mentioned in P are relevant.
- The predicate Q specifies what the test outcome should be. Here is how:
- Randomly generate values for the parameters mentioned in P.
- Run S in a state where the parameters have these values.
- Check whether the program ends in a state that satisfies Q.

assignment	$\dfrac{}{\{\varphi_a^v\}\ v := a\ \{\varphi\}}$
skip	$\dfrac{}{\{\varphi\}\ \text{SKIP}\ \{\varphi\}}$
sequence	$\dfrac{\{\varphi\}\ C_1\ \{\psi\} \qquad \{\psi\}\ C_2\ \{\chi\}}{\{\varphi\}\ C_1; C_2\ \{\chi\}}$
conditional choice	$\dfrac{\{\varphi \wedge B\}\ C_1\ \{\psi\} \qquad \{\varphi \wedge \neg B\}\ C_2\ \{\psi\}}{\{\varphi\}\ \text{ if } B \text{ then } C_1 \text{ else } C_2\ \{\psi\}}$
guarded iteration	$\dfrac{\{\varphi \wedge B\}\ C\ \{\varphi\}}{\{\varphi\}\ \text{ while } B \text{ do } C\ \{\varphi \wedge \neg B\}}$
precondition strengthening	$\dfrac{\mathbb{N} \models \varphi' \to \varphi \qquad \{\varphi\}\ C\ \{\psi\}}{\{\varphi'\}\ C\ \{\psi\}}$
postcondition weakening	$\dfrac{\{\varphi\}\ C\ \{\psi\} \qquad \mathbb{N} \models \psi \to \psi'}{\{\varphi\}\ C\ \{\psi'\}}$

Example of Non-deterministic Processing

```
randomNat :: Int -> Int
randomNat n = unsafePerformIO $
              getStdRandom (randomR (0,n-1))
```

The function `randomNat 4` generates a random integer number in the range $[0..3]$.

A Random Tuple

Consider the following tuple:

```
tuple :: (Int,Int)
tuple = let x = randomNat 4 in (x, 3-x)
```

The result is that both elements of the tuple get assigned a number from the set $\{0, 1, 2, 3\}$, and that the following property holds:

```
tupleProp :: (Int,Int) -> Bool
tupleProp = \tuple ->
    fst tuple >= 2 || snd tuple >= 2
```

Clearly we cannot derive from this that `fst tuple >= 2` nor that `snd tuple >= 2`.

Invalid Inference

The example shows that from

$$\{P\}\, S\, \{Q \vee R\}$$

one cannot infer that

$$(\{P\}\, S\, \{Q\}) \vee (\{P\}\, S\, \{R\})$$

Take `True` for P, `fst tuple >= 2` for Q, and `snd tuple >= 2` for R.

Hoare Rule for Repetition

$$\frac{\{P \wedge C\}\ S\ \{P\}}{\{P\}\ \text{while } C \text{ do } S\ \{P \wedge \neg C\}}$$

The key element is finding an appropriate loop invariant P: a predicate with the property that if it holds immediately before the step S, it also holds immediately after the step.

$$\{P\}\ S\ \{P\}.$$

To make this fly, we also need the other rules of the calculus.

Hoare Rules for Functions

Recall that, from the functional imperative perspective, a step S is simply a function. In the simplest possible case S has type `a -> a`.

The rule for the function $\perp$ is given by:

$$\{P\} \perp \{P\}.$$

Explanation: $\perp$ never terminates with success, so the Hoare assertion is trivially true.

Hoare rule for function update

$$\{(u = x \wedge gu = y \wedge P(x, y)) \vee P(u, fu)\}\ g = \text{update}\ f(x, y)\ \{P(u, gu)\}.$$

Hoare rule for function composition

$$\frac{\{P\}\ g\ \{R\} \quad \{R\}\ f\ \{Q\}}{\{P\}\ f.g\ \{Q\}}$$

Hoare rule for choice (function definition by cases)

$$\frac{\{Q \wedge P\}\ f\ \{R\} \quad \{Q \wedge \neg P\}\ g\ \{R\}}{\{Q\}\ \text{if } P \text{ then } f \text{ else } g\ \{R\}}$$

This can be extended for other ways of defining functions in Haskell.

Precondition Strengthening, Postcondition Weakening

Precondition Strengthening:

$$\frac{P' \Rightarrow P \quad \{P\}\ f\ \{Q\}}{\{P'\}\ f\ \{Q\}}$$

Postcondition Weakening:

$$\frac{\{P\}\ f\ \{Q\} \quad Q \Rightarrow Q'}{\{P\}\ f\ \{Q'\}}$$

Hoare Logic as a Fragment of Dynamic Logic

Hoare logic is a fragment of the following more general system of (propositional) dynamic logic:

The language of propositional dynamic logic was defined by Pratt in [5, 6] as a generic language for reasoning about computation. Axiomatisations were given independently by Segerberg [7], Fisher/Ladner [2], and Parikh [4]. These axiomatisations make the connection between propositional dynamic logic and modal logic very clear.

PDL Language

Let p range over the set of basic propositions P, and let a range over a set of basic actions A. Then the formulae φ and programs α of propositional dynamic logic are given by:

$$\begin{aligned} \varphi &::= \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \langle\alpha\rangle\varphi \\ \alpha &::= a \mid ?\varphi \mid \alpha_1; \alpha_2 \mid \alpha_1 \cup \alpha_2 \mid \alpha^* \end{aligned}$$

Abbreviation:

$[\alpha]\varphi$ abbreviates$\neg\langle\alpha\rangle\neg\varphi$.

Expressing Hoare Triples in PDL

Floyd-Hoare correctness assertions are expressible in PDL, as follows. If φ, ψ are PDL formulae and α is a PDL program, then

$$\{\varphi\}\ \alpha\ \{\psi\}$$

translates into

$$\varphi \rightarrow [\alpha]\psi.$$

Clearly, $\{\varphi\}\ \alpha\ \{\psi\}$ holds in a state in a model iff $\varphi \rightarrow [\alpha]\psi$ is true in that state in that model.

Deriving Hoare Rules in PDL

The Floyd-Hoare inference rules can now be derived in PDL. As an example we derive the rule for guarded iteration:

$$\frac{\{\varphi \wedge \psi\}\ \alpha\ \{\psi\}}{\{\psi\}\ \text{WHILE}\ \varphi\ \text{DO}\ \alpha\ \{\neg\varphi \wedge \psi\}}$$

Let the premise $\{\varphi \wedge \psi\}\ \alpha\ \{\psi\}$ be given, i.e. assume (1).

$$\vdash (\varphi \wedge \psi) \to [\alpha]\psi. \tag{1}$$

We wish to derive the conclusion

$$\vdash \{\psi\}\ \text{WHILE}\ \varphi\ \text{DO}\ \alpha\ \{\neg\varphi \wedge \psi\},$$

i.e. we wish to derive (2).

$$\vdash \psi \to [(?\varphi; \alpha)^*; ?\neg\varphi](\neg\varphi \wedge \psi). \tag{2}$$

From (1) by means of propositional reasoning:

$$\vdash \psi \to (\varphi \to [\alpha]\psi).$$

From this, by means of the test and sequence axioms:

$$\vdash \psi \to [\varphi;\alpha]\psi.$$

Applying the loop invariance rule gives:

$$\vdash \psi \to [(\varphi;\alpha)^*]\psi.$$

Since ψ is propositionally equivalent with $\neg\varphi \to (\neg\varphi \wedge \psi)$, we get from this by propositional reasoning:

$$\vdash \psi \to [(\varphi;\alpha)^*](\neg\varphi \to (\neg\varphi \wedge \psi)).$$

The test axiom and the sequencing axiom yield the desired result (2).

Summary, and Further Reading

The chapter on action logic in [1] develops a general perspective on the logical analysis of actions.

References

[1] Johan van Benthem, Hans van Ditmarsch, Jan van Eijck, and Jan Jaspars. Logic in Action. Internet, 2012. electronic book, available from www.logicinaction.org.

[2] M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. Journal of Computer and System Sciences, 18(2):194–211, 1979.

[3] C.A.R. Hoare. How did software get so reliable without proof? In FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods, pages 1–17, London, UK, 1996. Springer-Verlag. Keynote Address.

[4] Rohit Parikh. The completeness of propositional dynamic logic. In Mathematical Foundations of Computer Science 1978, pages 403–415. Springer, 1978.

[5] V. Pratt. Semantical considerations on Floyd–Hoare logic. Proceedings 17th IEEE Symposium on Foundations of Computer Science, pages 109–121, 1976.

[6] V. Pratt. Application of modal logic to programming. Studia Logica, 39:257–274, 1980.

[7] K. Segerberg. A completeness theorem in the modal logic of programs. In T. Traczyck, editor, Universal Algebra and Applications, pages 36–46. Polish Science Publications, 1982.