Equational Reasoning and Intended Semantics in Functional Programming

or

How to think about functional programs

Lee Naish
Bernard Pope
Harald Søndergaard

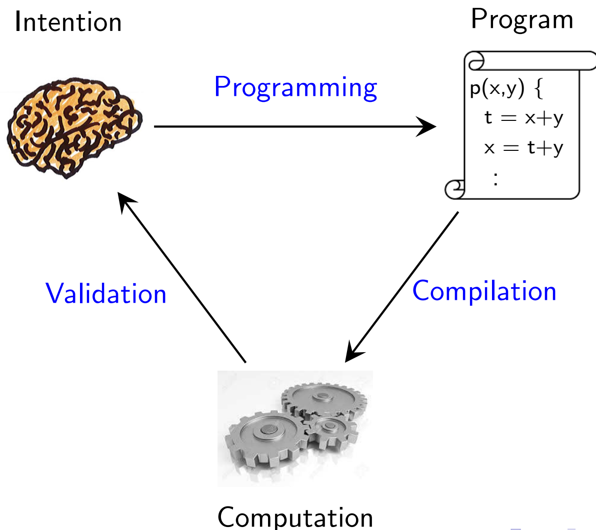
Computing and Information Systems
The University of Melbourne

Outline

- Our intentions, our programs and what they compute
- Simple equational reasoning
- Problem: undefinedness; "fast and loose" equational reasoning
- Problem: underspecification
- Solution: tweaking equational reasoning
- Conclusion

Aside: behind the scenes there are a whole lot of theoretical definitions and theorems, some of which are works in progress

Our intentions, programs and what they compute



Equational reasoning, evaluation and transformation

A simple functional program:

```
>length [] = 0
>length (x:xs) = 1 + length xs
```

View it as two mathematical equations describing an intended function

View evaluation/computation as "replacing equals by equals":

```
length (1:2:[]) →
1 + length (2:[]) →
1 + 1 + length [] →
1 + 1 + 0 →
2
```

We can use laws for transformation/optimization, eg `map f (map g ys)` `= map (f.g) ys` (saves creating and traversing an intermediate list)

Equational reasoning and declarative debugging

If every equation is correct the result (if any) is correct

So if an incorrect result is computed there must be an incorrect equation

And a declarative debugger can find it!

The debugger needs 1) the program, 2) the expression/computation that gives the wrong result and 3) the programmer intentions (an "oracle")

Suppose we use 1 instead of 0 in the base case for `length`:

```
Does length (1:2:[]) equal 3? n
Does length (2:[]) equal 2? n
Does length [] equal 1? n
Incorrect equation:
length [] = 1
```

Problem: undefinedness

Programming language functions are not quite mathematical functions: they can have "infinite loops" (or pattern match failure, or other errors)

"Fast and loose reasoning is morally correct" points out that this results in various intuitive laws being wrong, but we can get away with using them

```
>foo x = bar x
>bar x = ...
```

Naive equational reasoning says we can replace "`bar x`" by "`foo x`"

```
>foo x = foo x
```

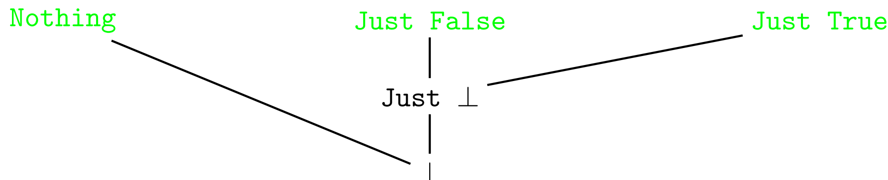
In FP semantics infinite loops etc are modelled by a special "undefined" value, $\perp$ (the "bottom" value of a partial ordering)

The equation `foo x = foo x` is consistent with any intended function but this definition gives no information to compute with

The information ordering

The result of computing something of type `Maybe Bool` has the following structure

Lines indicate the higher value is more defined or has more information



Problem: underspecification

Our intention for a program function is not always a mathematical function

Consider the `merge` operation from mergesort

If the arguments are sorted lists the intended result is clear, otherwise we typically don't know or care what the result is

For "garbage in" all results are considered correct

What if a debugger asks:
`Does merge [3,1] [3,2] equal [3,1,3,2]?`

The best debuggers support **y**/**n** answers and also "inadmissibility"

Forms of underspecification

One form of underspecification is "preconditions" that we expect to hold for all calls (part of the "contract")

We (try to) write code that ensures preconditions are always satisfied; if not, the code that did the call is considered buggy

Another is having multiple representations for one (abstract) value

For example, we may choose to represent sets of integers as sorted lists but allow duplicates: `insert 1 [1,2,2]` could return `[1,1,2,2]` or `[1,2,2]` or `[1,2]`; all these would be considered correct

Both have implications for designing declarative debugger oracles

Underspecification and equational reasoning

Consider the code below with the precondition that sets are represented as sorted lists

```
>list_last = last          -- Equation 1 (OK)
>set_max = maximum         -- Equation 2 (OK)
>set_max' = list_last      -- Equation 3 (OK)
>list_last' = set_max      -- Equation 4 (Dubious)
```

Equations 3 and 4 are the same from the equational reasoning perspective but (only) Equation 4 can cause a precondition violation:

```
list_last' [2,1] →
set_max [2,1] →
maximum [2,1] → ...
2
```

Tweaking equational reasoning

Our intentions for `set_max` are similar to those for `list_last` but more flexible because we only require `set_max` to work when the list is sorted

The set of (mathematical) functions that are considered correct implementations of `set_max` is a superset of those for `list_last`

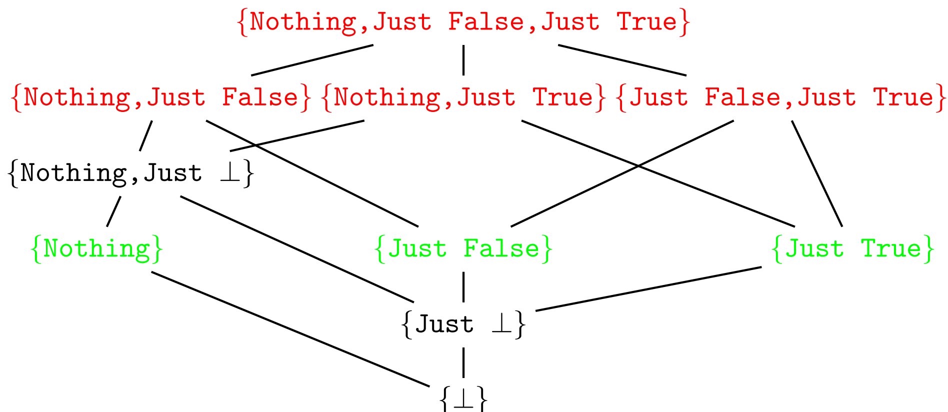
If we view the function names as sets of correct functions we have

`list_last` $\supseteq$ `last`
`set_max` $\supseteq$ `maximum`
`set_max'` $\supseteq$ `list_last`
`list_last'` $\not\supseteq$ `set_max`

If the left side of each equation in the program is *greater or* equal to the right side (according to our intentions and the set-based ordering), partial correctness is guaranteed!

An ordering for computed and intended values

Sets include all sets below them (⊥ and Just ⊥ are implicit in most sets)



Computation within this ordering

Most computation steps replace equals by equals and stay at the same point in the ordering

Going lower in the ordering is also OK (under-specification)

Going “sideways” is wrong: it typically means returning a wrong result

Going higher is also wrong (violating a precondition gets you to the top)

If you go up, you can then go down a different path to get a wrong result

Conclusion

Understanding the relationships between our intentions, programs and computations helps with software development (including tools)

Functional programming languages provide a good foundation

Viewing function definitions as equations, and both our intentions and what is computed as simple mathematical functions, is not quite right

Viewing definitions as inequations allows greater flexibility and various pitfalls can be avoided

We can have an elegant theoretical framework: a partial order, where "undefined" (nothing computed) and "don't know/care" (nothing intended) are at opposite ends

Further reading

- John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. CACM 21(8): 613-641, 1978
- Bernard Pope and Lee Naish. Practical aspects of declarative debugging in Haskell 98. Proc. Fifth Int. Conf. Principles and Practice of Declarative Programming, pp230-240, 2003
- Nils Danielsson et al. Fast and loose reasoning is morally correct. Proc. 33rd Ann. Symp. Principles of Programming Languages, pp206-217, 2006
- Franklyn Turban, David Gifford and Mark Sheldon. Design Concepts in Programming Languages, Chapter 4, MIT press, 2008
- Lee Naish and Harald Søndergaard. Truth versus information in logic programming. Theory and Practice of Logic Programming 14(6): 803-840, 2014