Formalizing Dijkstra

John Harrison

University of Cambridge

I've been playing around recently formalizing Dijkstra's "A Discipline of Programming". This talk is about a few aspects of the work.

- A Discipline of Programming
- Mechanizing programming logics
- Relational semantics
- Weakest preconditions
- Theorems about loops

A Discipline of Programming

This classic monograph by Dijkstra has several interesting features.

- Stress on programs as primarily mathematical formalisms, whose runnability of a machine is, so to speak, a lucky accident.
- Systematic use of the (then new) method of weakest preconditions to give semantics to programs.
- Formal treatment of a number of attractive algorithms, several of which have subsequently become classics, e.g. Hamming's problem and the Dutch National Flag.

It's surely Dijkstra's best book. In fact, the people who buy books for Cambridge University's libraries seem to think it's his only good book.

Why formalize it?

It seemed that it might be fun to formalize ADOP, for several reasons:

- Formalization tends to inspire a close reading, which this book probably deserves.
- Dijkstra is very pro-correctness proofs, but very anti-computer checking. It seemed interesting to see how his arguments stand up to formalization.
- This sort of formalization is generally pretty easy compared with floating point verification, so it provides light relief and the feeling of making rapid progress.
- "None of the programs in this monograph, needless to say, has been tested on a machine." [p. xvi]

This isn't new

Mike Gordon showed in 1988 how to formalize programming logics in higher order logic theorem provers. It would also work fine in set theory or any suitable general mathematical formalism.

He and Tom Melham actually used a tactic to do verification condition generation, which works very nicely. (I've used this approach in floating point verification.)

Since then there's been a slew of work formalizing programming languages based on the same ideas, e.g. Agerholm, Grundy, Homeier, Nipkow, Tredoux and von Wright, to name just a few.

As well as programming languages, there have been formalizations of hardware description languages and other CS formalisms, e.g. CCS, CSP, ELLA, π-calculus, TLA, UNITY, Verilog and VHDL.

Formalizing states

Following von Wright, we have a sort of “shallow embedding” of states, where the state is represented as a tuple of variables. Commands are implicitly abstracted over these variables, e.g. if we have three variables `x`,`y` and `z`, the assignment `x := y + z` would be:

```
Assign (\(x,y,z). (y + z,y,z))
```

All this is dealt with by parsing and printing, so the surface syntax is generally acceptable.

The problem with a more explicit representation of the environment is that one ends up fixing the possible types for variables in advance. In set theory, this is not a problem, as Mark Staples will show in his thesis.

Logical operators

Most of Dijsktra's use of logical operators is implicitly at the predicate level, so it's handy to define various liftings of logical operators, e.g.

```
|- p And q = \x. p x /\ q x

|- Forall P l = \x. FORALL (\a. P a x) l
```

In fact, I wondered if his use of 'non' for negation is a sort of pun (e.g. 'x is non empty if not (x is empty)'.

Sometimes Dijkstra is pretty vague here about where he implicitly means 'for all states'. I believe he nowadays writes things in square brackets to indicate quantification over all free variables. We have two separate forms of implication, again following von Wright:

```
|- p Imp q = \x. p x ==> q x

|- p Implies q = !x. p x ==> q x
```

Relational semantics

Dijsktra actually defines commands via their weakest proconditions. This was also done in HOL by von Wright et al.

> We take the point of view that we know the possible performance of the mechanism S sufficiently well, provided that we can derive for any postcondition R the corresponding weakest precondition $wp(S, R)$, because then we have captured what the mechanism can do for us; and in the jargon the latter is called "its semantics". [p17]

To us it seems more satisfactory to start with a more intuitive and operational view of programs and derive weakest preconditions afterwards. Dijkstra doesn't manage to escape from operational thinking completely, however hard he tries.

Nondeterminism

Using relations $\Sigma \to \Sigma \to bool$ or $\Sigma \times \Sigma \to bool$ has the defect, as noted in Gordon's original paper, that we can't really treat nondeterminism properly. We want to be able to distinguish possible and certain termination.

Jim Grundy shows in his thesis (also the proceedings of a conference in Novosibirsk, LNCS 735) that all ways of interpreting relations of this form lead to problems treating nondeterminism.

Instead, we use $\Sigma \to \Sigma_\perp \to bool$, i.e. introduce a separate type of 'outcomes' $\Sigma_\perp$. In HOL:

```
(A)outcome = Loops | Terminates A
```

We basically follow Hesselink's CUP book on weakest preconditions; some of the later theorems are also taken from his book, supplementing those given by Dijkstra.

Weakest preconditions

It's now straightforward to define weakest preconditions and weakest liberal preconditions:

```
|- terminates c s = ~c s Loops

|- wlp c q s =
     (!s'. c s (Terminates s') ==> q s')

|- wp c q s = terminates c s /\ wlp c q s
```

Note that our semantics allows non-total commands, i.e. ones with no final outcome. According to the above definition these satisfy every postcondition!

Hesselink uses them to interpret *guards* relationally. Anyway, all the actual commands we use are total.

Healthiness conditions

Dikstra gives some healthiness conditions that predicate transformers of the form `wp c` should obey. With a proviso about total commands, these are all trivial to prove in HOL (call `MESON_TAC` with some relevant facts).

```
|- (wp c False = False) = total c
|- q Implies r ==> wp c q Implies wp c r
|- wp c q And wp c r = wp c (q And r)
|- wp c q Or wp c r Implies wp c (q Or r)
|- deterministic c
     ==> (wp c p Or wp c q = wp c (p Or q))
```

where:

```
|- deterministic c =
     (!s t1 t2. c s t1 /\ c s t2
                ==> (t1 = t2))

|- !c. total c = (!s. ?t. c s t)
```

Other theorems

We also prove various other assertions by Dijkstra in the same chapter, and some more from Hesselink, e.g.

```
|- wp c r = wlp c r And wp c True

|- total c =
     !p. wp c p Implies Not(wlp c (Not p))

|- deterministic c =
     !p. Not(wlp c (Not p)) Implies wp c p
```

They're all pretty easy, except for the case where Dijkstra gets it wrong. Once `MESON_TAC` had taken 10 seconds I knew either Dijkstra or I must have made a mistake.

Dijkstra [pp. 21-2] enumerates the 7 'mutually exclusive' possibilities when a nondeterministic command c is started in a given state with a postcondition r in mind:

Dijkstra's error (1)

1. c will terminate and establish r
2. c will terminate and establish $\overline{r}$
3. c will not terminate
4. c will terminate and may or may not satisfy r
5. c may or may not terminate, but if it does will satisfy r
6. c may or may not terminate, but if it does will satisfy $\overline{r}$
7. c may or may not terminate, and if it does may or may not satisfy r

This is quite right. But his rendering of these in terms of weakest preconditions is wrong.

Dijkstra's error (2)

In the precise terms of Dijkstra's description, far from all being mutually exclusive, area (c) is contained in areas (ac) and (bc).

Dijkstra uses `Not (wp c True)` to indicate possible nontermination, but this wrongly includes the third case of *certain* nontermination.

We replace this with `Not (wp c True Or wlp c False)`, and with this change all the cases are indeed distict.

His error is basically a confusion of two different notions of doubt or certainty. Perhaps there's something unintuitive about nondeterministic machines, despite his confident pronouncements:

> Once the mathematical equipment needed for the design of nondeterministic mechanisms achieving a purpose has been developed, the nondeterministic machine is no longer frightening. On the contrary!

Guarded commands

Dijkstra's actual commands are a bit eccentric, making up the 'guarded command language'. Essentially:

$$
\begin{array}{rcl}
\mathit{command} & \longrightarrow & \texttt{skip} \\
 & \longrightarrow & \texttt{abort} \\
 & \longrightarrow & x_1, \ldots, x_n \texttt{:=}\ E_1, \ldots, E_n \\
 & \longrightarrow & \mathit{command}\texttt{;}\ \mathit{command} \\
 & \longrightarrow & \texttt{if}\ \mathit{gc} \,\square\, \cdots \,\square\, \mathit{gc}\ \texttt{fi} \\
 & \longrightarrow & \texttt{do}\ \mathit{gc} \,\square\, \cdots \,\square\, \mathit{gc}\ \texttt{od} \\
\mathit{gc} & \longrightarrow & \mathit{expression} \rightarrow \mathit{command}
\end{array}
$$

Semantics of loops

It's trivial to derive the weakest preconditions for most of the commands. The more interesting ones are for loops.

Dijkstra gives a definition of a semantics for loops on pp. 35-6. But this is completely bogus, sneaking in the assumption that a loop will terminate iff there is an upper bound on the number of iterations.

This requires an assumption of bounded nondeterminacy (and an appeal to König's lemma). Dijkstra eventually discusses this in chapter 9.

We define the semantics of loops at a relational level in a fairly obvious way, sticking to the spirit of Dijkstra's definition, i.e. talking about some number of iterations. Dijkstra prefers this to inductive or recursive definitions.

Theorems for loops

Dijkstra gives several theorems for loops, which we can prove relatively easily in HOL. His most 'basic' theorem is:

```
|- p And Exists (\(g,c). g) gcs Implies wp(If gcs) p
   ==> p And wp (Do gcs) True
       Implies wp (Do gcs)
                  (p And Not(Exists (\(g,c). g) gcs))
```

This has just `wp (Do gcs) True` as the hypothesis that the loop terminates. Of course in practice, one wants to show this using some reduction in the state w.r.t. a wellfounded ordering round each iteration of the loop. So we also derive:

```
|- WF(<<) /\
   (!X. p And Exists (\(g,c). g) gcs And (\s. s = X)
        Implies wp (If gcs) (p And (\s:S. s << X)))
   ==> p Implies wp (Do gcs)
                    (p And Not(Exists (\(g,c). g) gcs))
```

We get from this the exact theorems Dijkstra gives.

Reflections on loops

One can derive the 'less basic' theorem that is actually used in practice purely from a fixpoint assertion about the weakest precondition:

```
|- wp (Do gcs) (q:S->bool) =
      q And Not (Exists (\(g,c). g) gcs) Or
      wp (If gcs) (wp (Do gcs) q)
```

For the more basic theorem with `wp (Do gcs) True` as the hypothesis this isn't true — we need leastness. For example this loop has `x := 0` as a fixpoint:

```
do x /= 0 -> x := x + 1 od
```

We think this point is worth mentioning. Even if, like Dijkstra, you hate recursion and induction, that kind of loop unrolling is intuitive.

It's nice that we don't need any more precise fixing of the semantics of loops if we are merely interested in proving total correctness of programs in the usual way.

Future work

Most of Dijkstra's language is formalized; we just need to deal with variable declarations and array variables.

The main idea is to formalize the proofs he gives for the correctness of algorithms, and see how well this goes.

I've already learned quite a bit about semantics and in particular weakest preconditions doing this work. I reckon so far it has been worthwhile.