Formalizing Dijkstra

John Harrison

Intel Corporation, EY2-03
5200 NE Elam Young Parkway, Hillsboro, OR 97124, USA
`johnh@ichips.intel.com`

Abstract. We present a HOL formalization of the foundational parts of Dijkstra's classic monograph "A Discipline of Programming". While embedding programming language semantics in theorem provers is hardly new, this particular undertaking raises several interesting questions, and perhaps makes an interesting supplement to the monograph. Moreover, the failure of HOL's first order proof tactic to prove one 'theorem' indicates a technical error in the book.

0 A Discipline of Programming

Dijkstra's "A Discipline of Programming" [4] is widely, and we think rightly, regarded as a classic. As he describes it, the original intention was to present some algorithms, emphasizing the process of discovery leading to them rather than giving them as cut-and-dried results. However, Dijkstra also wished to present the programs using more mathematical rigour than is the norm. The book emphasizes a view of a program as an abstract mathematical object, whose runnability on a machine is, so to speak, a fortunate accident:

> Historically speaking . . . the fact that programming languages could be used as a vehicle for instructing existing automatic computers . . . has for a long time been regarded as their most important property. . . . I view a programming language primarily as a vehicle for the description of (potentially highly sophisticated) abstract mechanisms. [pp. 8–9]

Dijkstra's main technical innovation, covered in depth for the first time in this book, is the use of predicate transformers to give the semantics of programs. Predicate transformer semantics is quite convenient for formal correctness proofs, since it has a direct relationship with the satisfaction of appropriate input-output conditions. Moreover, it turned out [1] that one could introduce predicate transformers not implementable as code, and use these as stepping stones in formal program derivations, giving a natural formalization of informal top-down design methods.

Dijkstra was one of the earliest and strongest advocates of formal correctness proofs of programs rather than extensive testing. Nowadays this point of view is increasingly having a practical impact, with major hardware companies pursuing formal verification. But for a long time Dijkstra must have felt like a prophet crying in the wilderness.

> As I have now said many times and written in many places: program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence. [p. 20]

These points of view must lie behind flourishes such as:

> None of the programs in this monograph, needless to say, has been tested on a machine. [p. xvi]

In the light of this comment, it seemed interesting to check his proofs by machine! While Dijkstra [7] attacked the anti-verification polemic of DeMillo, Lipton, and Perlis [2] as a 'political pamphlet from the Middle Ages', he accepted that long tedious proofs are inadequate, and that 'communication between mathematicians is an essential part of our culture'. Moreover Dijkstra [5] elsewhere seems to oppose the idea of checking proofs by computer:

> To the idea that proofs are so boring that we cannot rely upon them unless they are checked mechanically I have philosophical objections, for I consider mathematical proofs as a reflection of my understanding and 'understanding' is something we cannot delegate, either to another person or to a machine.

Formalizing programming languages inside theorem provers has become a major research topic. Our work largely follows the classic paper by Gordon [9], and doesn't pretend to offer any major technical advances, but we think that in combination with an analysis of Dijkstra's book it raises a few interesting issues.

1 Formalization of States

A fundamental concept throughout the book, and imperative programming generally, is the notion of a *state*. Dijkstra devotes all of Chap. 2 to a gentle and rather non-operational introduction to the concept. To fall short of his ideal somewhat, we may briefly describe the state as a mapping that given a particular point during execution returns the values of all the program variables at that point.

For the moment, we will not concern ourselves with how states are represented and how variables as rvalues or lvalues consult or modify the state, nor how variables are declared or scoped – this is discussed much later, as in Dijkstra's monograph where it is delayed until Chap. 10. For all the basic semantics and program command definitions, we can think of the state as simply some arbitrary type, and we normally use the HOL type variable `:S`.

In what follows, predicates over states, or equivalently sets of states, are used incessantly.[1] One often wants to say that for example 'P and Q both hold in state

[1] Dijkstra [p14] talks about predicates 'corresponding' to sets; in the HOL formalization they actually are sets.

s'. This isn't the same as $P \wedge Q$, but rather $P(s) \wedge Q(s)$. It's often attractive – and in any case Dijkstra does it this way – to 'hide' the state in such assertions. The easiest way, already used in many programming language embeddings, is to define analogs of all the logical operations but lifted up to the level of predicates:

```
|- False = (λx. F)

|- True = (λx. T)

|- Not p = (λx. ¬p x)

|- p And q = (λx. p x ∧ q x)

|- p Or q = (λx. p x ∨ q x)

|- p Imp q = (λx. p x ⇒ q x)

|- (!!) q = (λx. ∀k. q k x)

|- (??) q = (λx. ∃k. q k x)
```

These correspond to Dijkstra's `F`, `T`, `non`, `and`, `or`, $\Rightarrow$, `A` and `E` respectively. (It would be possible, and consistent with our later approach to operators in the programming language, to overload the standard logical symbols for this level too, but on balance that is probably too confusing.) We also use the following variant, which doesn't give a function on states, but rather says that the implication holds for all states:

```
|- p Implies q = ∀x p x ⇒ q x
```

Dijkstra doesn't define this explicitly, but rather says sometimes in words 'for all states'. Some writers use a special triple-lined implication sign for this purpose. Dijkstra and his followers also sometimes enclose an expression in square brackets to indicate universal quantification over all free variables, though that isn't quite the same thing.

2 The Characterization of Semantics

Chapter 3 of Dijkstra's book discusses the behaviour of 'mechanisms', viewing them as systems that when started in an initial state, will end up in a final state (or else fail to terminate). Dijkstra distinguishes between deterministic and nondeterministic machines (in the former, 'the happening that will take place upon activation of the mechanism is fully determined by its initial state' [p. 15]), but doesn't describe mechanisms with great formality. There is not much controversy over how to formalize this in HOL: essentially, a mechanism is formalized as a relation between possible initial and final states.

Gordon [9] actually used relations $\Sigma \times \Sigma \rightarrow bool$. But as he pointed out, this formalization can only indicate the nontermination of R on a state s by the absence of any state s' with $R(s, s')$. So while it allows us to consider nondeterministic machines, there is no obvious way to indicate *possible nontermination* rather than *certain nontermination*. Indeed, Grundy [10] shows how there is no really satisfactory way of doing so based on this formalization. Instead, therefore, we consider relations $\Sigma \rightarrow \Sigma_\perp \rightarrow bool$, where $\Sigma_\perp$ augments the state space with an additional element denoting nontermination. We refer to $\Sigma_\perp$ as the set of *outcomes*; it is defined in HOL as a type:

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

However, our formalization still has one peculiar feature that should be commented on. It does not automatically follow that the relation associates each state s with some outcome. How are we to interpret a relation where this is not the case – some abnormal condition such as arithmetic overflow or division by zero? We don't do this here; as we shall see below, its interpretation in terms of program correctness is quite the reverse! In any case we define a notion of *totality* and include it as a condition in theorems where needed.

Dijkstra introduces in Chap. 3 the key notions of the weakest precondition and weakest liberal precondition. The weakest precondition of a command[2] c with respect to a postcondition q is the set of initial states such that c, when started in one of those states, is guaranteed to terminate in a state satisfying q. The weakest *liberal* precondition is the set of initial states such that *if* the command terminates it does so in a state satisfying q, but nontermination is allowed as an alternative. The HOL formalization follows Dijkstra except that we use a curried rather than paired *wp* function. This is mainly a matter of taste, but as we will see shortly, our version makes sequencing of commands correspond exactly to function composition of the weakest 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
```

A feature of non-total commands is that they trivially satisfy every pre/post-condition relationship! Hesselink [11] regards this as a virtue, using them as an 'miracles'. We, however, regard non-total commands as a blemish, and rule them out where needed.

Dijkstra then gives informal derivations of some important conditions that the predicate transformer must obey if it is to arise as *wp c* for some 'mechanism' c. In our formalization, the first of these only follows from an assumption of totality, and is in fact equivalent to it:

[2] Dijkstra unusually bows to the masses and refers to *statements*, but we will stick to *commands*. Dijkstra admits himself [p. 25] that this is better.

```
|- total c = ∀s. ∃t. c s t

|- (wp c False = False) = total c
```

Dually, we have:

```
|- terminating c = ∀s. terminates c s

|- (wp c True = True) = terminating c
```

The other ‘healthiness conditions’ (2–4) are rendered in HOL very easily, and the proofs are essentially automatic using a tactic for first order reasoning by model elimination:

```
|- 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)
```

A stronger form of the last is predicated on an assumption of determinacy:

```
|- deterministic c = ∀s t1 t2. c s t1 ∧ c s t2 ⇒ (t1 = t2)

|- deterministic c ⇒ (wp c p Or wp c q = wp c (p Or q))
```

Conversely, it’s straightforward to recover (the relational semantics of) a command *c* from *wp c* and *wlp c*. This topic is not discussed explicitly by Dijkstra, who keeps operational details informal, though Hesselink [11] does mention it [p. 105]. Once again, the HOL proof is essentially automatic:

```
|- (c s Loops = ¬wp c True s) ∧
   (c s (Terminates s’) = Not (wlp c (λx. ¬(x = s’))) s)
```

For a deterministic command, *wp c* alone suffices:

```
|- deterministic c
   ⇒ (c s (Terminates s’) = ¬wp c False s ∧ wp c (λx. x = s’) s) ∧
      (c s Loops = ¬wp c True s)
```

Indeed, on the assumption of totality, determinism implies a simple relation between *wp c* and *wlp c*; as Hesselink [11] mentions [p. 111] we can split this up into two strong equivalences:

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

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

|- ∀c. total c ∧ deterministic c = ∀p. wp c p = Not (wlp c (Not p))
```

However, for nondeterministic commands, *wp c* alone isn't enough – as with a relation on $\Sigma \times \Sigma$, the weakest precondition semantics cannot distinguish between possible and certain nontermination. Dijkstra only introduces *wlp c* on p. 21, some time after *wp c*, probably precisely because it is necessary to give a satisfactory account in predicate transformer terms of the behaviour of a nondeterministic machine. On pp. 21–2 Dijkstra enumerates 7 'mutually exclusive' possibilities when a nondeterministic command c is started in a given state with a postcondition r in mind:

- (a) c will terminate and establish r
- (b) c will terminate and establish $\overline{r}$
- (c) c will not terminate
- (ab) c will terminate and may or may not satisfy r
- (ac) c may or may not terminate, but if it does will satisfy r
- (bc) c may or may not terminate, but if it does will satisfy $\overline{r}$
- (abc) c may or may not terminate, and if it does may or may not satisfy r

Unfortunately, Dijkstra's rendering of some of these in formal terms is wrong, a fact we noticed only when one of HOL's automatic tactics failed to prove 3 out of the 15 mutual exclusions between the above. (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. There is a confusion here of levels of certainty: we need to be *uncertain whether we are certain* that a command will not terminate. We can express this correctly by saying we are not certain it will terminate, and not certain that it will fail to terminate. Using instead `Not (wp c True Or wlp c False)`, we find that all the cases are indeed distinct:

```
|- total c
   ⇒ (wp c r And wp c (Not r) = False) ∧
      (wp c r And wlp c False = False) ∧
       ....
```

and still enumerate all the possibilities:

```
|- total c
   ⇒ (wp c r Or
       wp c (Not r) Or
       wlp c False Or
       wp c True And Not (wlp c r) And Not (wlp c (Not r)) Or
       wlp c r And Not (wp c True Or wlp c False) Or
       wlp c (Not r) And Not (wp c True Or wlp c False) Or
       Not (wlp c r Or wlp c (Not r) Or wp c True)
      = True)
```

That Dijkstra should make such an elementary error is perhaps indicative of something slightly unintuitive about nondeterministic machines, despite his confident pronouncements:

> Eventually I came to regard nondeterminacy as the normal situation, determinacy being reduced to a –not even very interesting – special case. [p. xv]
> 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! We shall learn to appreciate it, even as a valuable stepping stone in the design of an ultimately fully deterministic mechanism. [p. 20]

3 The Semantic Characterization of a Programming Language

Up to now we have considered a fairly abstract notion of mechanism; now we specialize this by considering how to build them up from a fixed repertoire of constructs. It must be made clear that Dijkstra defines the weakest preconditions axiomatically, and often stresses the primacy of this view:

> 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 post-condition 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". [p. 17]

We sometimes have freedom to choose a particular operational definition that yields the same notion of weakest precondition; see for example the discussion of `abort` below. The commands or command-building constructs are all defined as HOL constants. We don't give any HOL version of the concrete syntax at this stage, but we indicate the concrete syntax for the sake of familiarity, and to allow easy comparison with Dijkstra's book.

The simplest command is `skip` which 'does nothing', rather like a no-op in machine codes. This is defined in HOL as the identity relation between initial states and final outcomes.

```
|- Skip s z = (z = Terminates s)
```

It's easy to see that this gives the identity as its weakest precondition:

```
|- ∀q. wp Skip q = q
```

More interesting is the `abort` command, which always fails to establish any postcondition. Our operational definition is that it always loops indefinitely:

```
|- Abort s z = z = Loops
```

This gives the appropriate weakest precondition:

```
  |- ∀q. wp Abort q = False
```

On the other hand, we have the following, for which Dijkstra offers no particular support:

```
  |- ∀q. wlp Abort q = True
```

Obeying Dijkstra's mantra that the weakest precondition is all we are interested in, we need not consider whether the operational definition is reasonable. It has some support in the literature, e.g. in Hesselink [11] [p. 17]. But the name rather suggests the immediate erroneous termination of the computation, and this conception is borne out by some of Dijkstra's later comments. For example at the end of Chap. 7 [p. 50], he comments on a program's "pleasant property that attempted activation outside its domain will lead to immediate abortion", something that can hardly be called pleasant if abortion is an infinite loop. It's interesting that occasionally Dijkstra's mask slips and operational thinking can be glimpsed.

Next comes the assignment statement. This is written concretely using the assignment symbol `:=`, but at this level, we abstract away from variables and so on, treating an assignment simply as a functional state transition:

```
  |- Assign f s z = (z = Terminates (f s))
```

and we find simply:

```
  |- ∀f q. wp (Assign f) q = q o f
```

These are all the 'atomic' commands, and next come the ways in which compound commands can be built up from other commands. The simplest and most conventional is sequencing, where two commands are executed one after the other. This is defined by an infix constant `Seq`, corresponding to a semicolon in the concrete syntax:

```
  |- (c1 Seq c2) s z =
        c1 s Loops ∧ (z = Loops) ∨
        (∃s'. c1 s (Terminates s') ∧ c2 s' z)
```

This operational definition is a bit involved, because we need to consider separately whether the first command loops or not. However the weakest precondition version could hardly be simpler. Two equivalent forms of it are:

```
  |- ∀c1 c2 q. wp (c1 Seq c2) q = wp c1 (wp c2 q)

  |- ∀c1 c2. wp (c1 Seq c2) = wp c1 o wp c2
```

Dijkstra's other composite constructs involve 'guarded commands', and are more complicated than the usual if-then-else and while-do forms. It's clear he considers them a significant innovation, for he starts Chap. 15 [p. 117] with 'When the guarded commands had emerged and the word got around . . . '. He doesn't really offer any detailed justification for not taking conventional forms, and we can scarcely dare to ask for some when we read elsewhere [3]:

> I do not know whether . . . it is a Swiss national trait to be "solid" first and only "adventurous" as far as then allowed (and that is not very far). Part of my talk dealt with guarded commands. Now, for anyone with some understanding it is clear that as sequencing tools they are much more attractive to use than the traditional while-do and if-then-else, and if, fifteen years ago, someone had thought of them, while-do and if-then-else would perhaps never have become established the way they are now. While at other places – Albuquerque and Toronto, for instance – it sufficed to show the difference, I felt this time more or less pressed to quantify the improvement.

We start with the notion of a guarded command; this is simply a pair of a predicate (the 'guard') and a command, written $b \longrightarrow c$, with the approximate meaning 'only execute the command c if the guard b is true'. However this doesn't tell us what to do if the guard is false, and in fact it depends on context, so we can't really consider these as independent commands (and Dijkstra doesn't try to). Rather, they are building-blocks for the alternative and repetitive constructs, each of which takes a finite number of guarded commands. We take as the HOL formalization of a such a set of guarded commands a list of predicate-command pairs. Using lists means that first, there can be zero guarded commands; Dijkstra doesn't rule this out, but remarks [p. 34, p. 36] that in this case the `if` and `do` constructs reduce to `abort` and `skip` respectively. Also, we are introducing an order, but this seems quite reasonable at the level of abstract syntax: the semantics is, as we shall see shortly, independent of this order.

Using lists means that many of the theorems require quantification over lists, which we do both at the boolean and state level:

```
|- (EX P [] = F) ∧ (EX P (CONS h t) = P h ∨ EX P t)

|- (FORALL P [] = T) ∧ (FORALL P (CONS h t) = P h ∧ FORALL P t)

|- (Exists P [] = False) ∧ (Exists P (CONS h t) = P h Or Exists P t)

|- (Forall P [] = True) ∧ (Forall P (CONS h t) = P h And Forall P t)
```

Dijkstra instead uses an indexing function, so it could be argued that we would stay closer to his treatment by doing the same, but then we would need to include an indication of the domain, i.e. the number of guarded commands.

Dijkstra's conditional statement is written as follows:

```
if  $g_0 \longrightarrow c_0$
 ▯  $g_1 \longrightarrow c_1$
...
 ▯  $g_n \longrightarrow c_n$
fi
```

The intuitive meaning is: if one of the guards is true, execute one of the commands with a true guard; otherwise abort. This permits nondeterminism since more than one guard can be true. The HOL translation of the intuitive semantics is as follows:

```
|- If gcs s t =
     EX (λ(g,c). g s ∧ c s t) gcs ∨
     ¬EX (λ(g,c). g s) gcs ∧ (t = Loops)
```

We can derive the weakest precondition effectively as Dijkstra gives it:

```
|- ∀gcs q.
     wp (If gcs) q =
        Exists (λ(g,c). g) gcs And Forall (λ(g,c). g Imp wp c q) gcs
```

The repetitive construct is constructed syntactically just like the conditional:

```
do  $g_0 \longrightarrow c_0$
 ▯  $g_1 \longrightarrow c_1$
...
 ▯  $g_n \longrightarrow c_n$
od
```

The intended semantics is: while some guard is true, execute one of the commands with a true guard then repeat. If no guard is true, terminate immediately. Dijkstra pointedly defines the semantics at the level of weakest preconditions in terms of k-fold iteration. Although our definitions are at the operational level, we try to follow his style, rather than use an inductive definition. First, we define a relation between initial and final states meaning that this input-output relation can hold after executing the loop a given number of times.

```
|- (Do_step 0 gcs s s' = s' = s) ∧
   (Do_step (SUC k) gcs s s' =
    (∃s''. If gcs s (Terminates s'') ∧ Do_step k gcs s'' s'))
```

This is then used to define the semantics of the do-loop as a whole. Note that we need to ensure that looping is possible if the body can be executed indefinitely.

```
|- (Do gcs s Loops =
      (∃k s'. Do_step k gcs s s' ∧
              EX (λ(g,c). g s' ∧ c s' Loops) gcs) ∨
      (∃ss. (ss 0 = s) ∧
            (∀k. EX (λ(g,c). g (ss k) ∧
                 c (ss k) (Terminates (ss (SUC k))))
                 gcs))) ∧
   (Do gcs s (Terminates s') =
      (∃k. Do_step k gcs s s' ∧ ¬EX (λ(g,c). g s') gcs))
```

This definition is rather messy, but it's easy to get fixpoint characterizations, which are useful later:

```
|- Do gcs s Loops =
   EX (λ(g,c). g s ∧ c s Loops) gcs ∨
   (∃s'. EX (λ(g,c). g s ∧ c s (Terminates s')) gcs ∧
         Do gcs s' Loops)

|- Do gcs s (Terminates s') =
   ¬EX (λ(g,c). g s) gcs ∧ (s' = s) ∨
   (∃s''. EX (λ(g,c). g s ∧ c s (Terminates s'')) gcs ∧
          Do gcs s'' (Terminates s'))
```

We can't actually derive Dijkstra's (axiomatic) weakest precondition semantics for loops, which somewhat hampers our ability to copy his later proofs. The reason is that his axiomatic definition is based on the assumption that if a loop is guaranteed to terminate, there is some maximum number of iterations after which it is guaranteed to terminate. This follows from an assumption of 'bounded nondeterminacy', i.e. that a command guaranteed to terminate cannot have infinitely many possible successor states. Presumably Dijkstra hoped to sneak this assumption past his readers till he was ready to discuss it.

It is in fact the case that all commands constructed so far have only bounded nondeterminism, as Dijkstra proves in Chap. 9, where he finally discusses the notion. Here he belatedly admits [p. 77] that in the presence of unbounded nondeterminacy, the 'semantics of the repetitive construct would have been subject to doubt, to say the least'. Actually, the idea of bounded nondeterminacy is only meaningful from an operational point of view, and he proves a property of continuity that is the appropriate concept at the level of weakest preconditions. Of course Dijkstra's proof presupposes the semantics of do-loops, so while perfectly sound from his axiomatic viewpoint, it is useless (circular) as a justification of the semantics. To be fair to Dijkstra, he doesn't explicitly claim otherwise, though this has often been misunderstood [6]. This impression is heightened by the fact that he remarks in this chapter [p. 77] that unbounded nondeterminism cannot be implemented, but doesn't give a serious discussion of why it isn't a useful abstraction any more than the assumption of unbounded execution time or unlimited storage.

One thing we can easily prove without further assumptions is a fixpoint equation for the weakest preconditions of do-loops:

```
|- wp (Do gcs) q =
     q And Not (Exists (λ(g,c). g) gcs) Or wp (If gcs) (wp (Do gcs) q)
```

We could in fact prove that `Do gcs` is the *least* fixpoint, i.e. the smallest (w.r.t. inclusion) solution of the above equation. It is customary in programming language semantics to define the semantics of loops as least fixpoints. Dijkstra seems to dislike this trend [p. xvii], and in some ways we agree that an iterative version is more intuitive. However fixpoints are very nice to work with, and as we shall see, the leastness is not normally needed.

4 Theorems about Commands

Dijkstra devotes Chap. 5 to proving some useful theorems about `if` and `do` commands, that are more useful than the raw weakest preconditions for performing correctness proofs of programs. First of all, we have the following theorem for the conditional:

```
|- (∀s. (q Imp Exists (λ(g,c). g) gcs) s) ∧
   (∀s. Forall (λ(g,c). q And g Imp wp c r) gcs s)
   ⇒ (∀s. (q Imp wp (If gcs) r) s)
```

This is more or less a direct translation of Dijkstra's statement. In fact, we can strengthen it to hold pointwise. (Dijkstra normally specifies 'for all states', though occasionally forgets to even when it is clearly intended.)

```
|- (q Imp Exists (λ(g,c). g) gcs) And
   Forall (λ(g,c). q And g Imp wp c r) gcs Implies
   q Imp wp (If gcs) r
```

These are really versions of the traditional Hoare rule: if `q` implies that one of the guards holds, and if `q` together with the i^{th} guard is enough to ensure that the i^{th} command terminates in a state satisfying `r`, then the whole conditional always leads from a state satisfying `q` to one satisfying `r`.

Next we have the theorem for the do-loop. Once again, this is close to the traditional Hoare rule, with `p` acting as a loop invariant, and the assumption `wp (Do gcs) True` included because we want to guarantee termination, not just partial correctness.

```
|- p And Exists (λ(g,c). g) gcs Implies wp (If gcs) p
   ⇒ p And wp (Do gcs) True Implies
       wp (Do gcs) (p And Not (Exists (λ(g,c). g) gcs))
```

Because our semantics of loops is different from Dijkstra's, we can't use his proof, but the above is easy to prove by induction on the number of steps as seen by `Do_step`.

The above theorem simply includes an assumption of termination, with no indication as to how it might be proved. In Chap. 6, this difficulty is addressed, and versions of the above theorems are proved with the assumption of a non-negative 'variant' that decreases with each iteration of the loop. We actually generalize Dijkstra's version somewhat by allowing any wellfounded ordering on the state, not just those defined by measure functions:

```
|- WF (<<) ∧
   (∀X. p And Exists (λ(g,c). g) gcs And (λs. s = X) Implies
        wp (If gcs) (p And (λs. s << X)))
   ⇒ p Implies wp (Do gcs) (p And Not (Exists (λ(g,c). g) gcs))
```

where wellfoundedness is defined in the usual way:

```
|- WF (<<) = ∀P. (∃x. P x) ⇒ (∃x. P x ∧ (∀y. y << x ⇒ ¬P y))
```

We then specialize this general version to Dijkstra's theorem based on an integer (sic) measure function f. This just uses the fact that the order x<<$y =_{def} 0 < f(x) \wedge f(x) < f(y)$ is wellfounded, where $f : \Sigma \rightarrow \mathbb{Z}$. We use the auxiliary notion:

```
|- wdec c t s = wp c (λs'. t(s') < t(s)) s
```

and so derive the exact theorem he gives:

```
|- ∀t. p And Exists (λ(g,c). g) gcs Implies (λs. t s > & 0) ∧
       FORALL (λ(g,c). p And g Implies wp c p And wdec c t) gcs
       ⇒ p Implies wp (Do gcs) (p And Not (Exists (λ(g,c). g) gcs))
```

There is an interesting feature of these theorems using a wellfounded ordering or measure function. The proofs only rely on the fixpoint property of the do-loop, not leastness or any equivalent property. Even if one dislikes inductive definitions, the fixpoint is highly intuitive, for in the context of an ordinary while-loop it just amounts to the admissibility of a one-step loop unrolling of '`while e do c`' to '`if e then (c; while e do c) else skip`'. That the above theorems – the ones actually used in proving programs – should only depend on this seems worth noting.

Note that the first theorem about do-loops, with a raw assumption of termination, doesn't follow from the fixpoint property alone. For example

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

has a fixpoint of guaranteed termination with x = 0, which plainly doesn't satisfy the constraints of that theorem. We actually prove this in HOL:

```
|- ¬(∀D gcs p.
        (∀q. D q = q And Not (Exists (λ(g,c). g) gcs) Or
                  wp (If gcs) (D q)) ∧
        p And Exists (λ(g,c). g) gcs Implies wp (If gcs) p
        ⇒ p And D True
           Implies D (p And Not (Exists (λ(g,c). g) gcs)))
```

5 Program Variables

So far, we have used a completely indeterminate type for the state. Though this is fine for the general theory, particular programs manipulate the state by referencing and assigning to program variables. We now need to decide how to represent program variables in the HOL formalization. Perhaps the simplest approach, used by von Wright, Hekanaho, Luostarinen, and Långbacka [15] for example, is to regard the state space as a large tuple, and implicitly abstract expressions involving variables over this tuple. For example, if the state consists of variables x, y and z, the expression $x + y$ is translated by the parser into $\lambda(x, y, z).x+y$, while the assignment $z := x+y$ is translated into a state mapping $\lambda(x, y, z).\ (x, y, x + y)$. With this approach, operations on the data values can be inherited directly from HOL's theories and used in programs, and there is no problem using arbitrary different types for the variables. Moreover, HOL's typechecking and type inference apply automatically. Some parsing and printing support is needed to maintain these transformations, but it isn't really difficult. The main defect is, however, that variable names have no first-class existence. From a HOL point of view $\lambda(x, y, z).\ x + y$ is the same as $\lambda(y, x, z).\ y + x$; this means that the intuitive meaning of programs is sensitive to the choice of bound variable names. Compositionality is poor: one can only plug together program fragments in the obvious way if they have exactly the same state, even with the variables in the same order.

A simple alternative, originally used by Gordon [9], that gives variables a first-class existence is to represent the state as a function from names to values. In a simply typed system like HOL, however, this presents some problems because the state function can have only a single type `string -> X` for some particular `X`. If used in a straightforward way, this would require all the variables in a program to have the same type, rather an irksome restriction. A first way of avoiding this is to declare an enumerated type containing all the possible types one might want to use. However even in simple imperative languages the range of types is potentially unlimited, e.g. assuming one allows arrays of arrays of arrays . . . . But there is a reasonable alternative, which we adopt: we can use instead of an enumerated type a properly *recursive* type.[3] This has already been used [13, 14] for languages with rather richer type systems. The idea is that we can allow in this recursive type certain ways of constructing new types, e.g. 'array of', 'pointer to' etc:

[3] We are grateful to Tanja Vos for pointing out this possibility.

```
  value = Bool bool
        | Int int
        | Array ((value)array)
        | Pointer value
        | ...
```

Then we still have an unlimited range of types constructed using this fixed repertoire of constructors. At present, we only allow booleans, integers and arrays. Arrays, with various operations as defined in Chap. 11 of Dijkstra's book, are represented as a pair consisting of a starting index and a list of elements, the size and upper bound of the array being calculated from these. HOL's nested type definition package is able to define this type automatically. We can immediately define appropriate type discrimination and destructor functions, e.g.

```
 |- Intval (Int x) = x
```

We take the view that program expressions are functions from state to values. This means that various transformations are required by the parser and printer. Actually, we shift a lot of the transformations away from this level and down to the logical level, by defining new operations on program expressions, e.g.

```
 |- x + y = λs. Int (Intval (x s) + Intval (y s))
```

Thanks to operator overloading (available in the HOL Light version of HOL), we can use the standard addition symbol here, and similarly for all the other operators. (Excluding completely polymorphic ones such as equality; at present we use a C-like operator `==`.) This is done for addition by the following directive, which tells HOL Light that instances of `+` with the appropriate type map down to the constant `value_add`.

```
 overload_interface("+",
   `(value_add):((string->value)->value)->((string->value)->value)
               ->((string->value)->value)`);;
```

Of course, this requires us to set up versions of every operation we want to use. This seems acceptable since typical programming languages only offer a fairly small range. However it stands in sharp contrast to von Wright's approach, where all the standard operators can be used 'as is'. At present it should be regarded as experimental, and we will see how it performs when we consider more particular programs. We still need to perform parser transformations for constants and variables. Variables are translated using the function that looks up the appropriate name (a HOL `string`) in the state:

```
 |- lookup x s = s x
```

Assignments are built up in several stages. The modification of the state `s` caused by assignment of a single variable `x` the value `a` is as follows:

```
|- update (x,a) s = λy. if y = x then a else s(y)
```

However, following Dijkstra, we allow concurrent assignments of the form `x1,...,xn := E1,...,En`, where the `xi` can be assumed distinct. To implement this we need to evaluate all the `Ei` in the starting state, then iteratively apply the function `update`. This is done by the following function:

```
|- Assignment asl s = ITLIST update (MAP (λ(x,e). (x,e s)) asl) s
```

This relies on two standard list combinators:

```
|- (MAP f [] = []) ∧ (MAP f (CONS h t) = CONS (f h) (MAP f t))

|- (ITLIST f [] b = b) ∧ (ITLIST f (CONS h t) b = f h (ITLIST f t b))
```

We still need to deal with the declarations of variables and scoping rules. Dijkstra's system, given in Chap. 10, is somewhat unusual. He uses blocks as in Algol-like languages generally, but the usual distinctions between local and global variables are ramified. First, blocks are not allowed to inherit global values implicitly: all global variables used in the body must be explicitly declared. This means that as well as the usual declarations of local variables (which Dijkstra calls **pri** for private), blocks contain declarations of the imported ones. These imported variables are further divided into initialized (**glo** for global) and uninitialized (**vir** for virgin). Finally, all 'variables' are split into true variables (**var**) and constants **con**. The latter might simply be considered non-program variables, but we follow Dijkstra in considering them as program variables that aren't allowed to be assigned to. Hence it's permissible for a variable to be treated as a constant in an inner block and a variable outside. This means there are no fewer than six different classes of variable declaration: **privar**, **pricon**, **glovar**, **glocon**, **virvar** and **vircon**. Finally, all variables must be initialized before being referenced, using a statement similar to an assignment but with a type declaration. There are quite strong syntactic restrictions on variable initialization so that this condition can easily be checked statically.

Formalizing the above looks quite intimidating. But we elect to treat most of it extra-logically, as a series of static checks much like type checking that are performed before the parser even accepts a program and parses it into its HOL form. (And in fact at present we haven't bothered to implement any static checks at all, partly because it's not very interesting, partly because we assume Dijkstra will manage to stick to his own rules.) This means that distinctions between constants and variables, and between global and virgin variables, need have no semantic interpretation. These variable declarations are all represented as separate constants, to keep the representation of program texts invertible, but most of them mean the same semantically. All of them are represented not as individual commands, but rather as functions building a command from a declaration and a command, which is the remainder of the block, possibly including other declarations. This makes it easy to restore the value of a temporarily-overridden global variable. The main distinction is between imports of global names, which are semantically null, e.g.

```
|- Glovar x c = c

|- Vircon x c = c
```

and introduction of local names, which temporarily hide outer parts of the state, e.g.

```
|- (Privar x c s Loops = (∃a. c (update (x,a) s) Loops)) ∧
   (Privar x c s (Terminates s') =
    (∃a s''.
         c (update (x,a) s) (Terminates s'') ∧
         (s' = update (x,lookup x s) s'')))
```

Here we express the undefinedness of the initial value `a` for the new variable by making the semantics nondeterministic. In practice, this never matters since one always has an initialization before the variable is referenced. Initializing assignments are represented by a constants semantically equivalent to the usual assignment, but with an extra argument for the type; this is only retained for invertibility. Finally, we have another semantically null tagging constant to indicate the start of a block; without this we would have no means of deciding whether a series of declarations belong to the same block or several nested ones.

6 Conclusions and Future Work

Our formalization has almost covered all the 'foundational' parts of Dijkstra's monograph. The gaps that remain are (i) the parser does not perform any static checks like typechecking, making sure variables are initialized and that parallel assignments do not include repeated variables, etc., and (ii) the theory of arrays has not been fully developed. Once the latter at least is finished, we will be ready to try verifying some of the examples given later in the book. Doing so will reveal how successful the formalization we have chosen is in practice.

References

1. R. Back. *Correctness Preserving Program Transformations: Proof Theory and Applications*, Volume 131 of *Mathematical Centre Tracts*. Mathematical Centre, Amsterdam, 1980.
2. R. DeMillo, R. Lipton, and A. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, **22**, 271–280, 1979.
3. E. W. Dijkstra. Trip report visit ETH Zurich, EWD474, 3-4 February 1975. See [8], pp. 95–98.
4. E. E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
5. E. W. Dijkstra. Formal techniques and sizeable programs, EWD563. See [8], pp. 205–214, 1976. Paper prepared for Symposium on the Mathematical Foundations of Computing Science, Gdansk 1976.

6. E. W. Dijkstra. A somewhat open letter to EAA or: Why I proved the boundedness of the nondeterminacy in the way I did, EWD614, 1977. See [8], pp. 284–287.
7. E. W. Dijkstra. On a political pamphlet from the middle ages. *ACM SIGSOFT, Software Engineering Notes*, **3**, 14, 1978.
8. E. W. Dijkstra (ed.). *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag, 1982.
9. M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam (eds.), *Current Trends in Hardware Verification and Automated Theorem Proving*, pp. 387–439. Springer-Verlag, 1989.
10. J. Grundy. Predicative programming – a survey. In D. Bjørner, M. Broy, and I. V. Pottosin (eds.), *Formal Methods in Programming and Their Applications: Proceedings of the International Conference*, Volume 735 of *Lecture Notes in Computer Science*, Academgorodok, Novosibirsk, Russia, pp. 8–25. Springer-Verlag, 1993.
11. W. H. Hesselink. *Programs, Recursion and Unbounded Choice*, Volume 27 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.
12. J. J. Joyce and C. Seger (eds.). *Proceedings of the 1993 International Workshop on the HOL theorem proving system and its applications*, Volume 780 of *Lecture Notes in Computer Science*, UBC, Vancouver, Canada. Springer-Verlag, 1993.
13. D. Syme. Reasoning with the formal definition of Standard ML in HOL. See [12], pp. 43–60.
14. M. VanInwegen and E. Gunter. HOL-ML. See [12], pp. 61–74.
15. J. von Wright, J. Hekanaho, P. Luostarinen, and T. Långbacka. Mechanizing some advanced refinement concepts. *Formal Methods in System Design*, **3**, 49–82, 1993.

This article was typeset using the LATEX macro package with the LLNCS2E class.