Experimenting with a Refinement Calculus

A. J. M. van Gasteren

Meeting Carroll Morgan, some time ago, started me doing what I had wanted to do for a long time, viz. investigating to what extent refinement calculus is a helpful vehicle in presenting the development of an imperative algorithm in such a way that

- reasoning is as calculational as possible;
- the design decisions stand out clearly and
- can be described concisely; and
- the structure of the development is highly visible in the presentation.

Here I show refinement calculus in action in an example programming problem. The example has been chosen because it illustrates a variety of properties of the calculus, and because of the way the solution developed avoids indexitis. My main goal is to give an introduction to the use of the calculus. The reader is assumed to be familiar with program development.

Preliminaries: refinement

A considerable amount of work has gone into laying the mathematical foundations of a refinement calculus for sequential, imperative programs. R. J. R. Back [1] did pioneering work; more recently, Carroll Morgan [2] at Oxford and Joseph M. Morris [3] at Glasgow joined in, partly redoing, partly extending Back's work, and continuing a development that was also inspired by others — such as Dijkstra, Hoare, and Jones. Here I shall only mention what is actually needed in the development of the example problem. The program notation used is Dijkstra's Guarded Command notation, except that we use a fat dot to delimit variable declarations; other notations and the calculational rules, to be presented next, have largely been adopted from [2].
Firstly, in order that a program development be a calculation, specifications and programs need to be expressions of the same kind. Here we choose a specification to be a program. The introduction of a special statement, the *specification statement*, enables us to do so:

\[ v : [P, Q] \text{ is a program establishing postcondition } Q \text{ from precondition } P, \text{ by changing variables of list } v \text{ of program variables.} \]

Formally, for all points in the state space and for all \( R \),

\[ wp.(v : [P, Q] ).R \equiv P \land (Av :: Q \Rightarrow R). \]

Sometimes we omit the precondition \( P \), writing \( v : [Q] \); we only do so if the precondition is valid and takes the form “viewed as an equation in \( v \), postcondition \( Q \) has a solution”.

Secondly, we need a notion of calculation. It is provided by “\( \sqsubseteq \)”, pronounced “refines to”:

\[ \text{for programs } S \text{ and } T, S \sqsubseteq T \text{ means } (AR :: \text{for all points in state space } wp.S.R \Rightarrow wp.T.R). \]

Program development amounts to refining a specification (statement) into a program that is free of specification statements. The refinement relation \( \sqsubseteq \) is reflexive and transitive, and the program constructors are monotonic with respect to it; therefore, expressions can be refined by refining their subexpressions.

Specification statements can be refined in all the familiar ways, such as

\[
\begin{align*}
x : [P, Q] & \sqsubseteq \text{skip} \quad \text{if for all states } P \Rightarrow Q \\
x : [P, Q] & \sqsubseteq x := E \quad \text{if for all states } P \Rightarrow Q(x := E)
\end{align*}
\]

Finally, it will prove to be convenient to use the construct \( v : [[\text{inv } Q \bullet S ]] \), for list \( v \) of program variables, predicate \( Q \), and statement \( S \), to stand for an “extension” of program \( S \) that —by assigning to variables of \( v \)— maintains \( Q \) over \( S \), i.e. if \( Q \) holds as a precondition of \( S \) it will hold as a postcondition of the construct as well. (That is why such a \( Q \) is called an invariant.) It typically appears where we generalize a pre- and a post-condition.

The most important property is that \( v : [[\text{inv } Q \bullet \ldots]] \) distributes over the program constructors; for instance,

\[ v : [[\text{inv } Q \bullet S ; T ]] \sqsubseteq v : [[\text{inv } Q \bullet S ]] ; v : [[\text{inv } Q \bullet T]] . \]

In this paper, only two elimination rules for such constructs are needed, viz. elimination rules (0) and (1):