Proof by Programming

Theodore S. Norvell [*]

September 16, 1999

But programming, when stripped of all its circumstantial irrelevances, boils down to no more and no less than very effective thinking so as to avoid unmastered complexity, to very vigorous separation of your many different concerns. – E.W.Dijkstra [1975]

$$\vdash \Psi(A, B)$$

can be translated by "B stands in relation Ψ to A" or "B is a result of an application of the procedure Ψ to the object A". – Gottlob Frege [1879]

There are plenty of equations, though not much math, for intermediate steps are fully presented. –Philip Morrison [1994]

One might say facetiously that the subject matter of formal logic is the study of the verifiable interpretations of the program consisting of the null statement. – Robert W. Floyd [1967]

Abstract

If you have any conjecture to prove, there is a programming problem such that, if a program can be written that demonstrably solves the programming problem, then the conjecture is true.

This observation leads to a new way of writing proofs and allows tools and techniques developed for the writing of demonstrably correct programs to be applied to proving theorems that ostensibly have nothing to do with programming. In particular techniques of problem decomposition used in structured programming can be applied to structuring proofs.

This approach neither enlarges nor shrinks the set of theorems provable in a given logic. What it does is provide a nice way to communicate the structure and content of a proof.

0 Overview

There are obviously great similarities between the activities of writing programs and writing proofs. Both involve decomposing problems into simpler problems until the problems are simple enough to be dealt with in an obvious way. In the area of programming this is called step-wise refinement and its importance has been articulated, for example, by Wirth [1971]. In metamathematics we see examples like Gentzen's natural deduction systems, which were a reaction to the unstructured Hilbert style of proof.

In this paper I will formalize the connection between proving and programming by showing that if a proposed theorem is treated as a specification for a program, it is true if and only if it is a feasible specification (in the sense of obeying Dijkstra's "law of excluded miracles").

The best known formalizations of the connection between proofs and programs are constructive type theories (see for example [Backhouse 1988]) in which there is no difference between proofs and programs. In constructive type theories, it is even possible to execute a proof in order to compute the quantity that is asserted to exist by the theorem.

[*]Faculty of Engineering, Memorial University of Newfoundland St. John's, NF, A1B 3X5 Canada, theo@engr.mun.ca

Basis

$$\begin{array}{rcl}
\mathrm{wp}.(w := E).C & \Leftrightarrow & C^{w}_{E} \\
\mathrm{wp}.\mathbf{skip}.C & \Leftrightarrow & C \\
\mathrm{wp}.\mathbf{abort}.C & \Leftrightarrow & \mathbf{false} \\
\mathrm{wp}.(w : [A, Q]).C & \Leftrightarrow & A \wedge \langle \forall w' \cdot Q^{v'}_{v} \Rightarrow C^{w}_{w'} \rangle
\end{array}$$

Constructors

$$\begin{array}{rcl}
\mathrm{wp}.(p; q).C & \Leftrightarrow & \mathrm{wp}.p.(\mathrm{wp}.q.C) \\
\mathrm{wp}.(\mathbf{if}\ E\ \mathbf{then}\ p\ \mathbf{else}\ q).C & \Leftrightarrow & (E \Rightarrow \mathrm{wp}.p.C) \wedge (\neg E \Rightarrow \mathrm{wp}.q.C) \\
\mathrm{wp}.(p \mathbin{[\!]} q).C & \Leftrightarrow & \mathrm{wp}.p.C \wedge \mathrm{wp}.q.C \\
\mathrm{wp}.(\mathbf{while}\ E\ \mathbf{do}\ p).C & \Leftrightarrow & \langle \mu D \cdot (E \wedge \mathrm{wp}.p.D) \vee (\neg E \wedge C) \rangle \\
\mathrm{wp}.\langle \mathbf{var}\ i \cdot p \rangle.C & \Leftrightarrow & \langle \forall i \cdot \mathrm{wp}.p.C \rangle
\end{array}$$

Table 0: Semantics for specifications

The idea to be presented in this paper is somewhat different. First, it is not restricted to constructive proofs. Constructive proofs may be written in the approach to be presented below, very much as they would in a constructive type theory —i.e., by writing a program to construct the object asserted to exist— but nonconstructive proofs are not excluded. Second, the programming language I use is not a functional language, but an imperative one. Third, and most important, I do not identify proofs with programs, but rather with derivations of programs, that is, with a history of a program's development. A program derivation, in turn relies on the proof of each of its steps. The proofs of these steps will be familiar as they are all things that would have to be proved if the proof were written more conventionally.

Since we do not ultimately escape the necessity to prove, what, then, is the advantage of recasting a theorem proving problem as a program writing problem? The primary advantage is that the programming constructs provide a mechanism for communicating the *structure* of the proof.

1 Refinement Calculi

A *refinement calculus* $(\mathcal{S}, \mathcal{P}, \sqsubseteq)$ consists of a set $\mathcal{S}$ whose members are called *specifications*, a subset $\mathcal{P}$ of $\mathcal{S}$ whose members are called *programs*, And a preorder $\sqsubseteq$ on $\mathcal{S}$. If $p \sqsubseteq q$ we say that q *refines* p or that p is a *specification of* q.

Each specification p is associated with a *programming problem*. This is to find a program q such that $p \sqsubseteq q$.

In this paper I will use the refinement calculus developed by Morgan [1990].[0] For this calculus, the set of specifications $\mathcal{S}$ is generated from assignment statements, **skip** statements, **abort** statements, and specification statements (to be explained shortly), by a set of constructors that includes semicolon, **if-then-else**, nondeterministic choice, **while-do**, and variable declaration. I will consider other specification constructors as the need for them arises. The set of programs $\mathcal{P}$ is generated from a basis that includes assignment statements, **skip** statements, and **abort** statements, but not specification statements. The set of constructors for programs again includes semicolon, **if-then-else**, nondeterministic choice, **while-do**, and variable declaration. Each specification p is mapped to a predicate transformer wp.p. This mapping is described in Table 0, in which $\langle \mu D \cdot f.D \rangle$ means the strongest condition that is a fixpoint of f. The refinement relation $\sqsubseteq$ is defined by $p \sqsubseteq q$ iff

$$\mathrm{wp}.p.C \Rrightarrow \mathrm{wp}.q.C$$

for all conditions C. (I write $A \Rrightarrow B$ to mean A is as strong a condition as B, that is A implies B for all states.)

[0] However we could use any similar refinement calculus which shares some of the same properties.

Here is an example of a specification statement

$$x : [y \geq 0, y - 0.001 \leq x'^2 \leq y + 0.001]$$

Informally it is a command to change variable x, such that the final value of x squared is within 0.001 of the value of y; all this is under the assumption that y is initially nonnegative. In the case where y is initially negative, 'execution' of the specification statement can lead to any behaviour at all, including nontermination or alteration of variables other than x. The general form of a specification statement is

$$w : [A, Q]$$

where w is a list of variables, A is a condition (that is a set of states) and Q is a relation (that is a set of pairs of states). The informal meaning of $w : [A, Q]$ as an operation is as follows. If started in a state σ in A, then $w : [A, Q]$ terminates in a state σ' such that (σ, σ') is in Q, and such that σ and σ' differ only at variables in the list w. If started in a state σ not in A, then $w : [A, Q]$ need not terminate and if it does terminate, may terminate in any state at all.

In this paper (as can be seen from the above example), conditions are represented by predicates mentioning the program variable names. Relations are represented by predicates in which the program variable names represent the values of program variables in the first (initial) state and primed variables (x', y', etc.) represent the values of program variables in the second (final) state.[1] With this convention established, we can define the predicate transformer associated with each specification statement:

$$\text{wp}.(w : [A, Q]).C \Longleftrightarrow A \wedge \langle \forall w' \cdot Q_v^{v'} \Rightarrow C_{w'}^{w} \rangle$$

where v is all variables not in w, R_E^x means predicate R with variables x replaced by expressions E, and $\Longleftrightarrow$ is equality of predicates (equivalence over all states).

As a abbreviation, I write simply $[A, Q]$ for $w : [A, Q]$ when w includes all variables.[2] I also will write $w : [Q]$ for $w : [\mathbf{true}, Q]$.

Specification statements allow one to express things that can not express with a program. One is unbounded nondeterminism; the square root example above is an example, if the variables are understood to range over the rational numbers. The other is self-contradictory specifications such as

$$x : [\mathbf{true}, y - 0.001 \leq x'^2 \leq y + 0.001]$$

Consider this specification being 'executed' with an initial state in which $y < -0.001$. Assuming that the type of x does not include any nonreal numbers, the specification requires that the impossible be accomplished, that is, termination in a state where x^2 is negative. The problem with this statement is that the domain of its relation does not include all of its condition. Such a specification statement is called *infeasible*. For specifications in general, we can define p to be *feasible* exactly if

$$\text{wp}.p.\mathbf{false} \Longleftrightarrow \mathbf{false}$$

and infeasible otherwise. As mentioned above infeasibility does not arise in programs:

theorem "All programs are feasible"

$$\langle \forall p \cdot p \in \mathcal{P} \Rightarrow (\text{wp}.p.\mathbf{false} \Longleftrightarrow \mathbf{false}) \rangle$$

The proof is by induction on the structure of programs.

[1]Readers familiar with [Morgan 1990] will notice that there undecorated variables represent the final state, and variables subscripted with 0 (x_0, y_0, etc) represent the initial state.

[2]This is another minor notational deviation from [Morgan 1990] where $[A, Q]$ is consider to have an empty variable list.

2 Derivations

For a refinement calculus $(\mathcal{S}, \mathcal{P}, \sqsubseteq)$ a *derivation* of p_n from p_0 is a sequence

$$\begin{array}{ll} & p_0 \\ \sqsubseteq & \quad \{\text{hint}_1\} \\ & p_1 \\ \sqsubseteq & \quad \{\text{hint}_2\} \\ & \vdots \\ \sqsubseteq & \quad \{\text{hint}_n\} \\ & p_n \end{array}$$

where each $p_i \sqsubseteq p_{i+1}$. I will call a derivation *clear* if the correctness of each step is easy to see in light of the hint. In a sense, a clear derivation of q from p is a proof that $p \sqsubseteq q$, and if q is a program, then a clear derivation of q from p demonstrates a solution to a programming problem.

In situations where we want a completely formal system of proof, for example in proof checkers or theorem provers, it would be necessary to formalize the notion of a step 'being easy to see'. That is we may want to describe a set of triples (p, q, hint) which constitute an allowed step in clear derivations. In other situations, for example communication between humans, it may be better to rely on the taste of the writer of the derivation to choose stepping stones the right distance apart for the intended reader and the best hints to guide the reader. In this paper I will leave unexplored the possibility of formally restricting the steps allowed in a derivation —and hence having an objective definition of 'clear' derivations—, because the ideas presented are independent of whether or not the distinction between clear and unclear derivations is subjective or objective.

All the specification constructors in Table 0 are *monotone* in the sense that for any unary constructor f

$$(p \sqsubseteq q) \Rightarrow (f.p \sqsubseteq f.q)$$

and for any binary specification constructor g

$$(p \sqsubseteq q) \wedge (r \sqsubseteq s) \Rightarrow (g.p.r \sqsubseteq g.q.s)$$

One can shorten and improve the clarity of derivations containing monotone constructors by writing them in a tree-like fashion. For example:

$$\begin{array}{ll} & p \\ \sqsubseteq & \quad \{\text{hint why } p \sqsubseteq q; r\} \\ & \begin{array}{lll} q & ; & r \\ \boxed{\begin{array}{ll} \sqsubseteq & \quad \{\text{hint why } q \sqsubseteq s\} \\ & s \\ & \vdots \end{array}} & & \boxed{\begin{array}{ll} \sqsubseteq & \quad \{\text{hint why } r \sqsubseteq \textbf{if } b \textbf{ then } t \textbf{ else } u\} \\ & \textbf{if } b \\ & \textbf{then } \ t \\ & \qquad \boxed{\vdots} \\ & \textbf{else } \ u \\ & \qquad \boxed{\vdots} \end{array}} \end{array} \end{array}$$

Each box shows the part of the derivation that further refines the specification above it. Within such a sub-derivation, certain contextual information is generally considered to be available. For example in the refinement of t, the fact that b is true initially may be used.

Such hierarchical derivations have the advantages, over the linear format, of focusing the reader's attention on what is relevant, of reducing the amount of copying, and of better expressing the order of dependence of steps. In short the *structure* of the programming process is clearly shown.

This hierarchical format could obviously be applied directly to proofs, using the rule of Leibniz or the monotonicity of implication in place of the monotonicity of refinement. But there are advantages to considering the circuitous route of translating conjectures to programming problems and presenting solutions to the programming problems in place of proofs of the proposed theorems.

3 From conjectures to programming problems

Now we come to the central ideas of this paper.

Suppose we have a closed boolean formula $\langle \forall w \cdot Q \rangle$ with no primed variables occurring in the list of variables w. Although predicate Q mentions only unprimed variables, we can still regard it as a relation (it relates each point in its domain to every state) and consider the specification $[Q]$.

Under what condition on the specification $[Q]$ is $\langle \forall w \cdot Q \rangle$ true? One answer is that $[Q]$ is feasible.

theorem "Feasibility implies theoremhood" *For* $\langle \forall w \cdot Q \rangle$ *closed and* w *having no primed variables*

$$\langle \forall w \cdot Q \rangle \Leftarrow (\text{wp}.[Q].\mathbf{false} \Longleftrightarrow \mathbf{false})$$

Since all programs are feasible and any specification refined by a feasible specification is feasible we have as a corollary:

theorem "Proof by programming" *For* $\langle \forall w \cdot Q \rangle$ *closed and* w *having no primed variables*

$$\langle \forall w \cdot Q \rangle \Leftarrow \langle \exists p \in \mathcal{P} \cdot [Q] \sqsubseteq p \rangle$$

Although the first theorem is more general, the second has the advantage that membership in $\mathcal{P}$ is a syntactic matter and can be seen at a glance. (There will be an exception to this when we come to parallelism.) In fact both of these theorems can be strengthened to equivalences as can be seen from the following:

theorem "Refinability by **skip** equals theoremhood" *For* $\langle \forall w \cdot Q \rangle$ *closed and* w *having no primed variables*

$$\langle \forall w \cdot Q \rangle \equiv ([Q] \sqsubseteq \mathbf{skip})$$

But we are mostly interested in the $\Leftarrow$ direction.

Now suppose that $\langle \forall w \cdot Q \rangle$ is a conjecture. If we can find a program p that refines $[Q]$, then $\langle \forall w \cdot Q \rangle$ is a theorem. A clear derivation of p from $[Q]$ constitutes a proof of $\langle \forall w \cdot Q \rangle$.

4 Case based reasoning

As a first example we will look at proof by cases. The standard law of **if** introduction

$$w : [A, Q] \sqsubseteq \mathbf{if}\ B\ \mathbf{then}\ w : [A \wedge B, Q]\ \mathbf{else}\ w : [A \wedge \neg B, Q]$$

allows for case wise reasoning. The definition of $\mathcal{P}$ can allow any B here.

Figure 0 shows Euclid's proof of the infinity of primes. The notation $\{0,..n\}$ means the set of the first n naturals. Most of the interesting parts of the proof are left informal in the hints; the **if** statement serves to show the structure of the proof.

Other statement forms such as Dijkstra's **if-fi** statement, or various varieties of **case** statements can also be used for various sorts of case wise reasoning.

5 Constructive existence

Existence is often demonstrated by describing a method that computes an example of what is asserted to exist. This technique is formally justified by the refinement law:

Let Q be $\langle \exists i : \mathsf{primes} \cdot prime.(n-1) < i \leq 1 + \langle \times j : \{0,..n\} \cdot prime.j\rangle\rangle$

$$
\begin{array}{ll}
 & [Q] \\
\sqsubseteq & \quad \{\textbf{if } \text{introduction}\} \\
 & \textbf{if } prime.(1 + \langle \times j : \{0,..n\} \cdot prime.j\rangle) \\
 & \textbf{then } \ [prime.(1 + \langle \times j : \{0,..n\} \cdot prime.j\rangle), Q] \\
 & \qquad \boxed{\begin{array}{ll} \sqsubseteq & \left\{\begin{array}{l}\text{Since } 1+\langle \times j : \{0,..n\} \cdot prime.j\rangle) \text{ is prime it shows } Q \text{ is true provided} \\ prime.n < prime.(1 + \langle \times j : \{0,..n\} \cdot prime.j\rangle).\end{array}\right\} \\ & [true] \\ \sqsubseteq & \textbf{skip} \end{array}} \\
 & \textbf{else } \ [\neg prime.(1 + \langle \times j : \{0,..n\} \cdot prime.j\rangle), Q] \\
 & \qquad \boxed{\begin{array}{ll} \sqsubseteq & \left\{\begin{array}{l}\text{Since } 1 + \langle \times j : \{0,..n\} \cdot prime.j\rangle) \text{ is composite but not divisible by} \\ \text{one of the first } n \text{ primes, there is prime bigger than the first } n\text{, but} \\ \text{smaller than } 1 + \langle \times j : \{0,..n\} \cdot prime.j\rangle. \text{ Thus } Q \text{ is true.}\end{array}\right\} \\ & [true] \\ \sqsubseteq & \textbf{skip} \end{array}}
\end{array}
$$

Figure 0: An example of proof by cases

theorem "Exists to **var**"

$$w : [A, \langle \exists i' \cdot Q\rangle] \sqsubseteq \langle \mathbf{var}\ i \cdot w, i : [A, Q]\rangle$$

provided i is not free in A or Q, and i is not in w.

Figure 1 shows a proof of half the fundamental theorem of arithmetic. The variable types are $n \in \{1,..\}$, $m \in \{0,..\}$, $p \in \mathsf{seq}_m\mathsf{primes}$ (a sequence of m prime numbers), $a \in \mathsf{seq}_m\{1,..\}$. Note that $\{0,..m\}$ refers to the set of integers up to but *not* including m. The proof is somewhat shortened, for the purposes of the example, as the reader likely knows how to fill in the development of the **while-do** loop.

6 Induction

The induce statement introduced in [Norvell 1993a] can be used to express proofs by (simple) induction. (A similar construct is introduced in [Burstall 1987].) The induce statement is used in conjunction with initial data types, such as natural number, lists, and trees. Figure 2(a) shows a skeletal induce statement for the natural numbers. The statement has a name (h in this case), which is local to the statement, an argument (a natural number expression represented by E), and a clause (beginning with the key word **when**) for each of the generators of the data type; each clause consists of a header and a specification (p and q in this case). Execution begins with the evaluation of the expression and selection of a clause by pattern matching. Any variables mentioned in a clause header (in this case j) are local to the specification following and are constants. Within a clause, the statement as a whole may be recursively called by its name, but only if the argument of the call is one of the variables introduced in the clause header. Figure 2(b) shows a program for a familiar algorithm written with an induce statement.

The semantics, elaborations, and expressive power of this statement are discussed in [Norvell 1993a]. Here we will be content with one refinement law

```
    $[\langle \exists m', p', a' \cdot n = \langle \times i \in \{0,..m'\} \cdot p'.i' ** a'.i\rangle \wedge \langle \forall i, j \cdot i \neq j \Rightarrow p'.i \neq p'.j\rangle\rangle]$
$\sqsubseteq$   $\{$exists to $\mathbf{var}\}$
    $\langle \mathbf{var}\ m, p, a \cdot$
      $[n = \langle \times i \in \{0,..m'\} \cdot p'.i' ** a'.i\rangle \wedge \langle \forall i, j \cdot i \neq j \Rightarrow p'.i \neq p'.j\rangle]$
      $\sqsubseteq$   $\{$introduce $\mathbf{var}\}$
          $\langle \mathbf{var}\ j \in \{1,..n\} \cdot$
           $j, m, p, a := n, 0, \mathsf{nil}, \mathsf{nil}$
             $\left[\begin{array}{l} n = j \times \langle \times i \in \{0,..m\} \cdot p.i ** a.i\rangle \\ \wedge \langle \forall i, j \cdot i \neq j \Rightarrow p.i \neq p.j\rangle \\ \wedge \langle \forall i \cdot p.i \nmid j\rangle, \\ n = \langle \times i \in \{0,..m'\} \cdot p'.i' ** a'.i\rangle \\ \wedge \langle \forall i, j \cdot i \neq j \Rightarrow p'.i \neq p'.j\rangle \end{array}\right]$
             $\sqsubseteq$   $\{$standard techniques$\}$
                 $\mathbf{while}\ j \neq 1\ \mathbf{do} \langle \mathbf{var}\ q, b \cdot$
                         $q := \langle \mathsf{max}\ q \in \mathsf{primes} \cdot q \mid j\rangle$
                         $b := \langle \mathsf{max}\ b \cdot q ** b \mid j\rangle$
                         $j, m, p, a := j/(q ** b), m + 1, \mathsf{cons}.q.p, \mathsf{cons}.b.a$
                         $\rangle$
                 $\mathbf{od}$
          $\rangle$
    $\rangle$
```

Figure 1: A proof of half the fundamental theorem of arithmetic

```
                          $\langle \mathbf{var}\ a, b, c := 0, 1, 2 \cdot$
                           $\mathbf{ind}\ h.7$
$\mathbf{ind}\ h.E$                  $\mathbf{when}\ 0 \cdot \mathbf{skip}$
$\mathbf{when}\ 0 \cdot p$              $\mathbf{when}\ \mathsf{succ}.j \cdot b, c := c, b; h.j; b, c := c, b$
$\mathbf{when}\ \mathsf{succ}.j \cdot q$          $\mathbf{print}\ a$ “ to ” $b$
$\mathbf{dni}$                                $a, c := c, a; h.j; a, c := c, a$
                           $\mathbf{dni}$
                          $\rangle$
```

Figure 2: (a) (b)

```
        $[f^n.q \le p]$
$\sqsubseteq$       $\{$**ind** introduction$\}$
        **ind** $h.n$
        **when** $0 \cdot$  $[f^0.q \le p]$
                        $\sqsubseteq$  $[q \le p]$
                        $\sqsubseteq$      $\{q \le p\}$
                            $[$**true**$]$
                        $\sqsubseteq$  **skip**
        **when** $\mathsf{succ}.j \cdot$  $[f^j.q \le p \sqsubseteq h.j,\ \ f^{j+1}.q \le p$          $]$
                                            $\Leftrightarrow$  $f.(f^j.q) \le p$
                                            $\Leftarrow$      $\{f.p \le p\}$
                                                 $f.(f^j.q) \le f.p$
                                            $\Leftarrow$      $\{f$ monotone$\}$
                                                 $f^j.q \le p$
                        $\sqsubseteq$      $\{$consolidating$\}$
                            $[f^j.q \le p \sqsubseteq h.j, f^j.q \le p]$
                        $\sqsubseteq$  $h.j$
        **dni**
```

Figure 3: A proof using an induce statement.

theorem "Induce introduction"

$$
\begin{array}{l}
w : [A, Q.E] \sqsubseteq \textbf{ind}\, h.E \\
\quad \textbf{when}\, 0 \cdot w : [A, Q.0] \\
\quad \textbf{when}\, \mathsf{succ}.j \cdot w : [A \wedge (w : [A, Q.j] \sqsubseteq h.j), Q.(\mathsf{succ}.j)] \\
\quad \textbf{dni}
\end{array}
$$

Notice that no mention of a variant or invariant is needed.

Figure 3 shows a proof of $\langle \forall n : \mathsf{nat} \cdot f^n.q \le p \rangle$ from the assumptions that (a) f is monotone, (b) $q \le p$, and (c) $f.p \le p$.

Proofs by Noetherian, rather than simple, induction can be expressed (more awkwardly) with **while-do** loops or with recursive procedures.

7 Parallelism and Conjunction

One way of dealing with conjunction is to use the following law:

theorem "Parallel introduction"

$$w : [A, Q \wedge R] \sqsubseteq w : [A, Q] \mid\mid w : [A, R]$$

The parallel composition is defined by

$$\mathrm{wp}.(p \mid\mid q) = \lceil \lfloor \mathrm{wp}.p \rfloor \wedge \lfloor \mathrm{wp}.q \rfloor \rceil$$

where $\lfloor f \rfloor$ (for f a predicate transformer) is defined as the relation

$$\langle \forall C \cdot f.C \Rightarrow C^v_{v'} \rangle$$

and $\lceil Q \rceil . C$ is defined as the condition

$$\langle \forall v' \cdot Q \Rightarrow C^{v}_{v'} \rangle$$

(with v being all variables) [Morgan 1994, Norvell 1993b]. The definition of the parallel operator roughly says that the final state reached by $p \mid\mid q$ must be acceptable to (i.e. within the range of nondeterminism of) both p and q. As a model of parallel processing, the $\mid\mid$ operator appears to leave something to be desired as it does not provide any means of communication between the parallel processes. In particular intermediate values of variables set by one process will not affect the other process. Hehner [1993] shows how interprocess communication can be dealt with in a similar model of computation.

There is a hitch with parallelism. Unlike other programming constructs, even if p and q are feasible, $p \mid\mid q$ may not be. For programs p and q, we say that $p \mid\mid q$ is a program only if it can be shown to be feasible. How can we tell if $p \mid\mid q$ is feasible? One way is to define a new program constructor

$$\text{wp}.(\textbf{forget}\ v\ \textbf{after}\ p).C = \text{wp}.p.\langle \forall v \cdot C \rangle$$

Then

$$(\textbf{forget}\ v\ \textbf{after}\ p) \mid\mid (\textbf{forget}\ w\ \textbf{after}\ q)$$

is feasible provided p and q are both feasible and universally conjunctive, and v, w comprises all variables.

We have seen that existential quantification can be proved with the **var** construct. What of universal quantification? We can define a programming construct

$$\text{wp}.\langle \mid\mid i \cdot q.i \rangle = \lceil \langle \forall i \cdot \lfloor \text{wp}.(q.i) \rfloor \rangle \rceil$$

and obtain the law

theorem "Process array introduction"

$$w : [A, \langle \forall i \cdot Q \rangle] \sqsubseteq \langle \mid\mid i \cdot w : [A, Q] \rangle$$

provided i not free in A.

The same difficulties with feasibility occur as with the binary operator.

The use of these constructs in proving (and, within this refinement calculus, programming) remains to be explored.

8 Connections

The idea of proof by programming is closely associated with "predicative programming" [Hehner 1984, Hehner 1993], which can be thought of as programming by proof. By interpreting programming constructs as operators on relations —as Hehner does—, the activities of proving and programming become blurred.

The main reason for using the refinement calculus of [Morgan 1990] in the presentation of this paper rather than predicative programming is so that the activities of proving and programming are kept distinct and thus the contribution of each to the other is more easily seen.

As mentioned above, there are also strong ties with constructive proof and particularly its expression in constructive type theory. These ties are an area for future investigation.

9 Conclusion

The idea presented here does not solve any fundamental problems in proving; nor does the use of the same constructs in programming. However in both cases they provide a mechanism for controlling complexity. In constructing a proof, they help the prover keep track of which assumptions are currently available and what remains to be proved; that is they provide a (familiar) notation for expressing the structure of the proof. In the reading of a proof, understanding the structure of the proof is often the main difficulty; it is helpful to have an unambiguous notation expressing the structure.

An unambiguous way of expressing proof structure is particularly important in communication between humans and machines. With the development of proof editors, proof checkers, and theorem provers an unambiguous and cognitively efficient way of communicating proofs is required.

Acknowledgments

Thanks are due to Ric Hehner for many discussions. I am also pleased to thank the Natural Sciences and Engineering Research Council of Canada for financial support.

Bibliography

[Backhouse 1988] Roland C. Backhouse. Constructive type theory: A perspective from computing science. In Edsger W. Dijkstra, editor, *Formal Development of Programs and Proofs*, pages 1–32. Addison-Wesley, 1988.

[Burstall 1987] R.M. Burstall. Inductively defined functions in functional programming languages. *Journal of Computer and System Sciences*, 34:409–421, 1987.

[Dijkstra 1975] Edsger W. Dijkstra. EWD 512 Comments at a symposium, 1975. Republished in *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982.

[Floyd 1967] Robert Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics, Volume XIX*, 1967.

[Frege 1879] Gottlob Frege. Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought, 1879. Translation published in *From Frege to Gödel*, Jean van Hiejenoort, editor, Harvard, 1967.

[Hehner 1984] Eric C.R. Hehner. Predicative programming. *Communications of the ACM*, 27(2):134–151, 1984.

[Hehner 1993] Eric C.R. Hehner. *A Practical Theory of Programming.* Springer-Verlag, 1993.

[Morgan 1990] Carroll Morgan. *Programming from Specifications.* Prentice Hall International, 1990.

[Morgan 1994] Carroll Morgan. The cuppest capjunctive capping, and Galois. In Bill Roscoe, editor, *A Classical Mind.* Prentice Hall International, 1994.

[Morrison 1994] Philip Morrison. The whale as hot-oil balloon. *Scientific American*, June 1994. Review of *Air and Water: The Biology and Physics of Life's Media,* by Mark Denny.

[Norvell 1993a] Theodore S. Norvell. Induce-statements and induce-expressions: Constructs for inductive programming. In *Sixth Conference on Foundations of Software Technology and Theoretical Computer Science, 1993*, number 761 in LNCS, 1993.

[Norvell 1993b] Theodore S. Norvell. Reconnecting predicates and transformers, April 1993. Unpublished memorandum.

[Wirth 1971] Niklaus Wirth. Program development by stepwise refinement. *Communication of the ACM*, 14(4):221–227, 1971.