Generating Algebraic Laws from Imperative Programs

H. Peter Gumm

Dept. of Mathematics, Philipps Universität Marburg
gumm@mathematik.uni-marburg.de

Abstract. The use of verifiers for proving the correctness of concrete programs is well known and has been amply described in the literature. Here we focus on further, perhaps more general tasks such verifiers can perform. Given a program that is assumed to be correct, we derive a set of axioms for the data structures involved. In the simplest case, we study an abstract program interchanging the contents of two variables. The verification conditions generated by our verifier, NPPV, are a set of equations specifying quasigroups. Other examples reveal the notion of "strategy" from the verification of an abstract game playing program, or show the correspondence between inductive proofs of numeric properties and verification of a program searching for a counterexample. Finally we apply NPPV on Wand's example showing the incompleteness of Hoare's logic. We also give a simplified proof of Wand's result.

1 Algorithm = Data Structure + Control

According to standard definitions [4], an algorithm is a *detailed and explicit instruction for the stepwise solution of a given problem.* This means that there must be given a repertoire of elementary (or atomic) steps which are to be combined according to the instructions of the algorithm. In a general sense it is of course allowed to think of atomic steps such as "*add a cup of flour*", "*stir*", and of combinations of instructions such as "*add a cup of flour* `then` *stir* `until` *smooth*", but we shall not deal with recipes, rather with algorithms computing functions over sets of data.

Here an atomic step consists of calculating a data value according to a given set of operations and storing the result in a memory cell. In describing how to combine such elementary steps a small set of instructions including composition (;), conditional (`if-then-else`) and loops (`while` or `repeat`) is commonly used.

This way a separation of concerns is achieved. A *data structure* defines the admissible atomic steps and a *control structure* determines how these steps are to be combined to yield the desired algorithm. This view is stated very succinctly in the well known slogan *"algorithm = data structure + control"*.

The border separating data structure and control may slide towards either side depending on the application. As an example we may assume to have multiplication "$*$" of natural numbers available as elementary arithmetical instruction, yet we may also get by with the operators of Presburger arithmetic $(0, succ, +, <)$ and construct an algorithm for multiplication. All programming languages provide mechanisms to augment the data structure by such defined functions.

The main purpose of this article is a demonstration together with a set of some succinct examples that show how Wirth's "equation" may be solved for an unknown data structure too. That is, given the specification of an algorithm and given a control structure, automatically determine axioms for a data structure required to fulfil the specification. A vehicle for finding these examples is a program verifier (NPPV) that we have constructed for educational purposes and used in many courses on program verification. With its help we can not only semi-automatically verify concrete programs, but also investigate "abstract programs" and reveal relationships between programs, specifications, invariants and data structure requirements. As a simple example, for instance, we shall show that a program to interchange the value of two variables works correctly precisely if the data structure contains a quasigroup operation, or that the failure of a program to find a counterexample to a conjecture leads to an induction axiom for the data type.

NPPV (New Paltz Program Verifier), has been implemented on an IBM compatible PC and has been developed for, and successfully used in courses devoted to the mathematics of program verification and abstract data types. The software is embedded in an "integrated development environment" with built-in editor, pull-down menus and pop-up windows. It is freely available for demonstration and course use.

2 Data Structures

Definition 1 (Signature). A *many-sorted signature* is a triple $(S, \mathcal{F}, \tau)$ where

- S is a set (whose elements are called *sorts*)
- $\mathcal{F}$ is a set (of *operation symbols*), and
- $\tau : \mathcal{F} \longrightarrow S^* \times S$ is a map associating with every $f \in \mathcal{F}$ a tuple of argument sorts and a result sort.

Definition 2 (Data Structure). A *data structure* of signature $(S, \mathcal{F}, \tau)$ is a tuple $\mathcal{A} = (A, F)$, consisting of a family $A = (A_s)_{s \in S}$ of nonempty sets together with a family of *fundamental operations* $F = (f_{\mathcal{A}})_{f \in \mathcal{F}}$ so that the type assignment τ is respected, i.e. if $\tau(f) = ((s_1, \dots, s_n), s)$, then $f_{\mathcal{A}} : A_{s_1} \times \dots \times A_{s_n} \longrightarrow A_s$.

Examples of data structures are groups of sort $(\{G\}, \{\circ, ^{-1}, e\}, \tau)$ where τ specifies that:

$$\begin{aligned} \circ\ &: G \times G \longrightarrow G \\ ^{-1}\ &: G \longrightarrow G \\ e\ &: \longrightarrow G. \end{aligned}$$

The two-element Boolean algebra is $(\{\mathbb{B}\}, \{\wedge, \vee, \neg, \mathtt{true}, \mathtt{false}\})$, Presburger arithmetic is the data structure $(\{\mathbb{N}, \mathbb{B}\}, \{+, succ, 0, <, =\})$, and standard arithmetic is $(\{\mathbb{N}, \mathbb{B}\}, \{+, *, 0, 1, <, =\})$ where in the latter three data structures the signature is evident.

2.1 Terms

Terms are expressions built from variables and fundamental operations. Assuming that for each sort $s \in S$ we are given a set of variables Var_s, we can define recursively the notion of a *term of type* s:

Definition 3. (i) Every variable $v \in Var_s$ is a term of type s.
(ii) If $\tau(f) = ((s_1, \ldots, s_n), s)$ and t_i is a term of type s_i for every $i \leq n$, then $f(t_1, \ldots, t_n)$ is a term of type s.
(iii) Given a term u of type s and a variable $v \in Var_s$, then for every term t, $t[v/u]$ denotes the term obtained by replacing every occurrence of variable v in t by the term u.

2.2 Logical expressions

Data types used for programming are always assumed to extend the Boolean data type. [1] *Boolean expressions* are simply terms of type $\mathbb{B}$. Boolean expressions are used in programs to determine the flow of control. *Predicate logic expressions* extend Boolean expressions by allowing quantifiers $\forall$ and $\exists$, that is:

Definition 4. (i) Every Boolean expression is a predicate logic expression,
(ii) If p, q are predicate logic expressions, then so are $p \wedge q$, $p \vee q$, and $\neg p$.
(iii) If p is a predicate logic expression, and x is a variable, then $\forall x.p$ and $\exists x.p$ are predicate logic expressions.

2.3 States

A *state* is an assignment of values to variables. More precisely, a state σ is a family of mappings $\sigma_s : Var_s \longrightarrow A_s$. Given a variable x of type s, we write $\sigma(x)$ instead of $\sigma_s(x)$. The canonical extension of σ to a map from terms to values is also denoted by σ. (In functional programming terminology this extension is often called $\texttt{eval}_\sigma$.)

Given state σ, variable v and value M we let $\sigma + [v = M]$ denote the "new" state σ' with $\sigma'(v) = M$, and $\sigma'(w) = \sigma(w)$ for every $w \neq v$. Observe that for any state σ, terms t, r and variable v :

$$\sigma(t[v/r]) = (\sigma + [v = \sigma(r)])(t).$$

Unfortunately, it turns out that any sufficiently rich model of computation will allow calculations that never terminate. We therefore include a pseudo-state $\bot$, pronounced "bottom" or "undefined". A nonterminating computation is then said to return $\bot$.

[1] Even if the Boolean operations are only derived operations, such as in the C language.

3 Control

The purpose of a (sequential) calculation is to proceed from an initial state σ to a final state σ' in which certain variables have some desired value. A program calculating the gcd of two numbers, e.g., is started in any state $[x = M\ ,\ y = N]$ where variables x and y are assigned positive integer values M, resp. N, and is supposed to reach final state in which a variable **z** is assigned $\gcd(M, N)$. Thus a program is (the description of) a state transformation.

3.1 Commands

Control structures describe state transformations and their combinations. Given a state, a *command* specifies how the next state is to be achieved. With $[\![C]\!]$ we describe the state transformation specified by command C, so $[\![C]\!](\sigma)$ is the state achieved after starting the execution of C in state σ. We set $[\![C]\!](\perp) = \perp$ for all commands C.

3.2 Assignment

The most basic command is given by a variable v and a term t. The phrase

$$v := t$$

is called an *assignment* and it is meant to denote the map transforming state σ to $\sigma + [v = a]$, where a is the value of t in state σ, i.e.

$$[\![v := t]\!](\sigma) = \sigma + [v = \sigma(t)].$$

Non-conflicting assignments my be executed in parallel

$$v_1, \ldots, v_n := t_1, \ldots, t_n,$$

where the values of $v_1, \ldots, v_n$ are updated with the (simultaneously computed) values of $t_1, \ldots, t_n$ respectively, i.e.

$$[\![v_1, \ldots, v_n := t_1, \ldots, t_n]\!](\sigma) = \sigma + [v_1 = \sigma(t_1)] + \cdots + [v_n = \sigma(t_n)].$$

3.3 Sequencing

Given commands C_1 and C_2 the sequential execution of "*first* C_1, *then* C_2", is described by "$C_1\ ;\ C_2$", that is

$$[\![C_1; C_2]\!] = [\![C_2]\!] \circ [\![C_1]\!].$$

At this point we note that assignment and sequencing alone do not add "computational power" going beyond the evaluation of terms in the data type. That is, a sequence of assignments can always be replaced by one single parallel assignment.

3.4 Skip

Occasionally it is convenient to have a command `skip` available. `skip` denotes the identity state transformation and could be simulated by a trivial assignment v := v. Clearly, ";" is associative with two-sided unit `skip` .

3.5 Conditionals

Given a Boolean expression B and two commands C_1 and C_2, the command

$$\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2$$

will be the same as C_1 when started in a state where B is true and C_2 otherwise, that is :

$$[\![\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2]\!](\sigma) = \begin{cases} [\![C_1]\!](\sigma), & \text{if } [\![B]\!](\sigma) = \texttt{true} \\ [\![C_2]\!](\sigma), & \text{if } [\![B]\!](\sigma) = \texttt{false}. \end{cases}$$

A (complex) command built from assignments using only sequencing and conditionals is called a *"straight line program"*. Note, that in the case of a finite set A every map $f : A \longrightarrow A$, (more generally, every operation on A) can be realized with a straight-line program P : If $A = \{a_1, \ldots, a_r\}$, and the desired map is given as $a_1 \mapsto b_1, \ldots, a_r \mapsto b_r$, let P be the straight line program

```
     if  x = $a_1$ then z := $b_1$
else if  x = $a_2$ then z := $b_2$
     ...
else                    z := $b_r$
```

With straight line programs we therefore go beyond evaluation of terms, i.e. the computational mechanism afforded by Universal Algebra, unless there is an "if-then-else" at the term level. Such algebras are well studied, they are called functionally complete. A term simulating the if-then-else is usually introduced as

$$d(a, b, c) = \begin{cases} c, & \text{if } a = b \\ a, & \text{if } a \neq b. \end{cases}$$

3.6 While

Given a Boolean expression B and a command C, the command

$$\texttt{while } B \texttt{ do } C$$

specifies a computation that repeatedly executes C, as long as B is satisfied. Such a computation need not terminate. It might in fact be defined as a countable sequence of if-commands :

```
if $B$ then $C$ else skip ;
if $B$ then $C$ else skip ;
...
```

If after finitely many steps a state is reached satisfying $\neg B$, then that state is the result of the computation, otherwise the result is the state $\bot$.

The given constructs suffice to specify all functions that are computable over a given data structure. Moreover, by structural induction it is not hard to see that every program C may be transformed into an equivalent program containing only a single while-loop, i.e. into a program of the form

$$I \;;\, \texttt{while}\ B\ \texttt{do}\ D$$

where I is a sequence of assignments and D is a straight-line program.

4 The Hoare Calculus

4.1 Specifications

The purpose of a program is to achieve a desired state transformation. A *specification* is a "declarative" description of such a transformation, that is it specifies the desired net effect of a transformation without concerning itself about how this effect is achieved using the available commands.

The classical method of C.A.R.Hoare ([5],[1])presents a specification as a pair (P, Q) of expressions in the predicate logic over the underlying data structure. The idea is that a command C satisfies the specification (P, Q), if for any state σ satisfying P the state achieved after executing C satisfies Q. However, the possibility that $[\![C]\!](\sigma) = \bot$ must be taken into account, so we distinguish between *partial correctness*

$$\{P\}\,C\,\{Q\} :\Longleftrightarrow \forall\sigma.(\sigma \models P \land [\![C]\!](\sigma) \neq \bot) \Rightarrow ([\![C]\!](\sigma) \models Q).$$

and *total correctness*:

$$[\,P\,]\,C\,[\,Q\,] :\Longleftrightarrow \forall\sigma.(\sigma \models P) \Rightarrow ([\![C]\!](\sigma) \neq \bot \land [\![C]\!](\sigma) \models Q).$$

Thus given a specification (P, Q), it may be considered the programmers job to solve it by finding a program X such that $\{P\}\,X\,\{Q\}$, or even $[\,P\,]\,X\,[\,Q\,]$ is true.

4.2 Hoare rules

C.A.R. Hoare has presented a calculus to derive theorems of the form $\{P\}\,C\,\{Q\}$, where (P, Q) is a specification and C a program. There are two general logical rules, an assignment axiom and one rule for every control construct .

Logical rules:

$$\frac{P' \Rightarrow P, \{P\}\, C\, \{Q\}}{\{P'\}\, C\, \{Q\}} \qquad \text{(pre-strengthening)}$$

$$\frac{\{P\}\, C\, \{Q\}, Q \Rightarrow Q'}{\{P\}\, C\, \{Q'\}} \qquad \text{(post-weakening)}$$

Axiom:

$$\frac{P \Rightarrow Q[v/t]}{\{P\}\, v := t\, \{Q\}} \qquad \text{(assignment axiom)}$$

Structural rules:

$$\frac{\{P\}\, C_1\, \{R\},\ \{R\}\, C_2\, \{Q\}}{\{P\}\, C_1\ ;\ C_2\, \{Q'\}} \qquad \text{(sequence rule)}$$

$$\frac{\{P \wedge B\}\, C_1\, \{Q\},\ \{P \wedge \neg B\}\, C_2\, \{Q\}}{\{P\}\, \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\, \{Q\}} \qquad \text{(conditional rule)}$$

$$\frac{P \Rightarrow I,\ \{P \wedge B\}\, C\, \{I\},\ I \wedge \neg B \Rightarrow Q}{\{P\}\, \texttt{while}\ B\ \texttt{do}\ C\, \{Q\}} \qquad \text{(while rule)}$$

The rules can be formulated in several equivalent ways. Here they are presented in a form that makes them appropriate for backward proof, that is, given a program X, specification (U, V), to check that $\{U\}\, X\, \{V\}$ is true, proceed according to the form of X and use the rules backwards: If X is a while loop, use the while-rule, if X is an assignment, use the assignment rule, etc. There are, unfortunately, several rules where a logical formula appears in the premise, but not in the conclusion. In a backwards proof, this formula will have to be guessed. This concerns the logical rules, the sequence-rule and the while-rule. Fortunately it turns out that except for the while-rule, the unknown expressions in the premises can be chosen in a standard way, as so called *weakest preconditions.*

The logical expression I in the while-rule is called an *invariant.* There is no standard way to guess a proper invariant in a backwards proof, although a number of heuristics are available. We shall see later that finding a proper invariant is at least as hard as finding a proper induction hypothesis in an inductive proof.

The rules are easily seen to be correct. Since the premises contain predicate logic expressions that must be shown valid in the data structure, it is clear that logical completeness of the above set is out of the question. However, we can ask for *relative completeness*, that is completeness under the assumption of an oracle for the valid formulas of the data structure. It turns out that the rules are indeed relative complete in that sense, provided the data structure is *expressive*, a notion introduced below.

4.3 Weakest liberal precondition

Given a set $W \subseteq S$ of states and a program C, the *weakest liberal precondition* of C and W is defined as

$$wlp(C, W) := \{\sigma \in S \mid [\![C]\!](\sigma) \in W\}.$$

Usually S will be denoted by a logical expression Q, so we set correspondingly

$$wlp(C, Q) := \{\sigma \in S \mid [\![C]\!](\sigma) \models Q\}.$$

For a straight line program C and a logical expression Q, $wlp(C, Q)$ is again definable by a logical expression :

$$\begin{aligned}
&wlp(\texttt{x := }t, Q) &&= Q[\texttt{x}/t] \\
&wlp(C_1 \texttt{ ; } C_2, Q) &&= wlp(C_1, wlp(C_2, Q)) \\
&wlp(\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, Q) &&= (B \wedge wlp(C_1, Q)) \vee (\neg B \wedge wlp(C_2, Q))
\end{aligned}$$

If C is a while-loop, $wlp(C, Q)$ need not be first order definable. Notice that according to our characterization of the while-loop as a countable sequence of conditionals, we can always write it as a countable disjunction

$$wlp(\texttt{while } B \texttt{ do } C, Q) = \bigvee_{n=0}^{\infty} wlp(D_n, Q),$$

where D_n is the straight line program consisting of the n-fold iteration of the command "`if` B `then` C `else skip` ".

4.4 Expressiveness and completeness

A data structure is called *expressive*, if the previous infinite disjunction is always first order definable. It turns out that standard arithmetic is expressive, whereas Presburger arithmetic is not. For expressive data structures, the Hoare calculus is relatively complete[3]. For a program C over an expressive data structure we therefore have :

$$\{P\}\, C\, \{Q\} \iff P \Rightarrow wlp(C, Q).$$

5 Mechanizing the Hoare calculus

The Hoare calculus is meant to be used on practical programs such as programs that search or sort arrays, calculate number theoretic functions, play games or that use clever tricks to implement an algorithm efficiently. Given a specification and a nontrivial algorithm, a paper and pencil verification of the corresponding program using the Hoare rules may present a formidable task. Typically, early versions of the program contain bugs, first attempts at formulating an invariant for a while-loop are incorrect, leading to a new proof attempt for every small correction. Each backwards proof attempt in turn produces a plethora of logical expressions, so called "verification conditions" that have to be shown valid in the data structure. For this reason, it is absolutely necessary, to have some machine support, if the calculus is to be useful.

In a somewhat weaker sense the same holds true in the teaching of program-verification. It is very hard to go beyond some very trivial examples because of the sheer number of verification conditions that are freshly generated with each proof attempt.

5.1 NPPV

For the above reasons we have implemented the program verifier NPPV. The acronym stands for "New Paltz Program Verifier". This MS-DOS Program presents a user interface familiar from virtually all programming language implementations, collectively termed as "interactive development environment" (IDE).

To be specific, the main screen shows an editor window in which the program together with its specification can be edited. A menu bar above the main window provides the most important commands, such as "edit", "prove" or "help". Others lead to further pull-down sub-menus, all in all providing a comfortable proof development environment.

5.2 Annotated programs

In order to prove partial correctness of a program, NPPV expects as input a specification consisting of

- a precondition,
- a program, in which every while loop is annotated with an invariant
- a postcondition.

If desired, the user may additionally include after any semicolon "`;`"an *intermediate assertions*, i.e. a logical expression that he expects to be true at that point in the program. Annotations appear within comment braces "{" and "}".

Such an *annotated program* is entered and edited in NPPV's main window. Syntactical and similar errors are immediately detected with the cursor placed at the offending position and a meaningful error explanation at the bottom of the screen.

Each annotation must be an open formula in an extension of the language over the data type used in the program. Essentially, it must be a Boolean formula, but in addition to the program variables (also called *mutable variables*) and to the fundamental operations, formulas in annotations may contain extra variables and functions not declared for the data type. Those so called *logic variables* and *Skolem functions* may not be read or written by the program. As a convention, NPPV expects logic variables to start with an uppercase letter.

To see the need for this distinction, consider a specification that asks for a program to exchange the contents of the variables `x` and `y`. Thus we are looking for a program C solving the following specification where the logic variables M and N stand for some arbitrary but fixed values :

$$\{\mathtt{x} = M \land \mathtt{y} = N\}\, C\, \{\mathtt{x} = N \land \mathtt{y} = M\}.$$

If M and N were program variables, then the program

$$M := N$$

would be a solution. If C was allowed merely to read M and N, then we still would have the unintended solution

$$\mathtt{x} := N \;;\; \mathtt{y} := M.$$

Rather we intended to specify a program C which does not contain M or N and which satisfies

$$\forall M, N.\{\mathtt{x} = M \wedge \mathtt{y} = N\}\, C\, \{\mathtt{x} = N \wedge \mathtt{y} = M\}.$$

We shall never use quantifiers explicitly in our specifications. Existential quantifiers can be eliminated through Skolemizations, universal quantifiers are assumed to bind every free logical variable.

5.3 Verification conditions

Given an annotated program, i.e. a construct $\{P\}\, C\, \{Q\}$, where every while-loop in C is annotated with an invariant, we could attempt to calculate $wlp(C, Q)$ and check whether this is implied by P. However, we have seen that $wlp(C, Q)$ need not exist when C contains a while loop. Even if it did, the resulting logical expression, if not valid, would hardly give us a clue as to the source of the error. Therefore we use a "localized" approach: First, we replace the *wlp*-function by the simpler function *pre*, defined on annotated programs as

$$\begin{array}{ll}
pre(\mathtt{x} := t, Q) & = Q[\mathtt{x}/t] \\
pre(C_1 \;;\; C_2, Q) & = pre(C_1, pre(C_2, Q)) \\
pre(\mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2) & = (B \wedge pre(C_1, Q)) \vee (\neg B \wedge pre(C_2, Q)) \\
pre(\mathtt{while}\, B\, \mathtt{do}\, \{I\}\, C, Q) & = I
\end{array}$$

Then we generate a set of simple logical expressions, so called *verification conditions* :

$$\begin{array}{ll}
vc(P, \mathtt{x} := t, Q) & = \{P \Rightarrow Q[\mathtt{x}/t]\} \\
vc(P, C_1 \;;\; C_2, Q) & = vc(P, C_1, R) \cup vc(R, C_2, Q) \\
 & \quad \text{where } R = pre(C_2, Q) \\
vc(P, \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, Q) & = vc(P \wedge B, C_1, Q) \cup vc(P \wedge \neg B, C_2, Q) \\
vc(P, \mathtt{while}\, B\, \mathtt{do}\, \{I\}\, C, Q) & = \{P \Rightarrow I\} \cup vc(B \wedge I, C, I) \cup \{I \wedge \neg B \Rightarrow Q\}.
\end{array}$$

Each verification condition is associated with a certain identifiable place in the program. Verification conditions are propagated through straight line subprograms. If this is not desired, an intermediate assertion $\{R\}$ can be placed after a semicolon ";". In this case we calculate

$$vc(P, C_1 ; \{\, R\,\}\, C_2, Q) = vc(P, C_1, R) \cup vc(R, C_2, Q)$$

It can be shown that the given annotated program satisfies its specification if and only if all these verification conditions are valid. More precisely, we have:

Theorem 5. *Let (P, Q) be a specification and C a program over data type $\mathcal{D}$. Let C' be an annotated version of C. If every verification condition in $vc(P, C', Q)$ is valid in $\mathcal{D}$, then $\{P\}\, C\, \{Q\}$ is true. If $\mathcal{D}$ is expressive, then the converse holds, i.e. if $\{P\}\, C\, \{Q\}$ is true then there is an annotation C' of C by loop invariants such that all verification conditions in $vc(P, C', Q)$ are valid in $\mathcal{D}$.*

Proof. One direction follows by a straightforward induction over the structure of annotated program C'. For the other direction we assume that D is expressive, so $wlp(C, Q)$ always exists. Given that $\{P\}\, C\, \{Q\}$, we have to find an annotated version C' of C such that every verification condition in $vc(P, C', Q)$ is valid. We set $C' = C^Q$, where for arbitrary C, Q we define

$(x := t)^Q = x := t$
$(C_1 \ ; \ C_2)^Q = C_1^R \ ; \ C_2^Q$, where $R = wlp(C_2, Q)$
$(\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2)^Q = \texttt{if } B \texttt{ then } C_1^Q \texttt{ else } C_2^Q$
$(\texttt{while } B \texttt{ do } C)^Q = \texttt{while } B \texttt{ do } \{I\}C^I$ where $I = wlp(\texttt{while } B \texttt{ do } C, Q)$

With this annotation we find for every C, Q that $pre(C^Q, Q) = wlp(C, Q)$. Now assume $\{P\}\, C\, \{Q\}$, i.e. $P \Rightarrow wlp(C, Q)$. By induction over the structure of C we need to show that every verification condition in $vc(P, C^Q, Q)$ is true.

- The cases $C = x := t$ and $C = \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2$ are straightforward.
- Let $C = C_1 \ ; \ C_2$, then $\{P\}\, C_1\, \{R\}$ and $\{R\}\, C_2\, \{Q\}$ with $R = wlp(C_2, Q) = pre(C_2^Q, Q)$. Every verification condition in $vc(P, C_1^R, R)$ and in $vc(R, C_2^Q, Q)$ is valid by induction hypothesis. Since $C^Q = (C_1 \ ; \ C_2)^Q = C_1^R \ ; \ C_2^Q$, the claim is true for $C_1 \ ; \ C_2$.
- Suppose $C = \texttt{while } B \texttt{ do } C_1$ and $I = wlp(C, Q)$, then clearly $P \Rightarrow I$. From $\{I\}\, C\, \{Q\}$ and the identity

  $$[\![\texttt{while } B \texttt{ do } C_1]\!] = [\![\texttt{if } B \texttt{ then } (C_1 \ ; \ \texttt{while } B \texttt{ do } C_1) \texttt{ else skip}]\!]$$

  we conclude $I \wedge \neg B \Rightarrow Q$ and $\{I \wedge B\}\, C_1 \ ; \ \texttt{while } B \texttt{ do } C_1\, \{Q\}$, and therefore $\{I \wedge B\}\, C_1\, \{I\}$. By the inductive hypothesis, all verification conditions in $vc(I \wedge B, C_1^I, I)$ are valid, so the same is true for $vc(P, C, Q)$.

NPPV will prove many of these verification conditions by itself and list the remaining ones with the remark: "`Remains to prove :` ". The user will have to decide whether she accepts them as true, or whether she wants to store them in a log-file for later inspection.

5.4 Example: Swapping variables

As a first example we consider two versions of a program exchanging the values of two variables. The first (and standard) solution uses a temporary variable:

```
{ x = A and y = B}
  temp := x ;
  x := y ;
  y := temp
{ x = B and y = A }
```

The second version shows that two integer values may be interchanged without an auxiliary variable. Both versions can be entered into NPPV as shown and will be automatically proved correct.

```
{ x = A and y = B}
  x := x+y ;
  y := x-y ;
  x := x-y
{ x = B and y = A }
```

5.5 Example: Gauss

As a further example, we consider a program adding all natural numbers below a fixed number N. A correctly annotated program (annotations are enclosed in braces) is:

```
{ N > 0 }
begin
  i   := 0 ;
  sum := 0 ;
  while i < N do { sum = i*(i+1)/2 and i <= N }
    begin
      i   := i+1;
      sum := sum + i
    end
end
{ sum = N*(N+1)/2 }
```

Aside from the pre- and postcondition the program contains as annotation a loop invariant. Whilst the principal conjunct of this invariant seems clear, the second conjunct, `i <= N` would typically be forgotten in a first proof attempt. The resulting verification condition

```
sum=i*(i+1)/2 and i >= N   ==>   sum=N*(N+1)/2
```

is not a tautology. Strengthening the invariant by `and i <= N` yields `i = N` in the premise, and the tautology is automatically proved by the system. In fact, NPPV proves all verification conditions except for one :

```
i < N  ==>  (i+1) <= N.
```

This means that NPPV cannot decide whether the formula

$$\forall i.\forall N.i < N \Rightarrow (i+1) \leq N$$

is a tautology in the data structure. Since we have not specified whether `i` and `N` are supposed to be integers (so far they might be assumed real), we see that it is perfectly correct, for NPPV to leave us with the above verification condition. All that is by default assumed for the algebraic operations $+, -, *, 0$, and 1 is that they satisfy the axioms of a commutative ring with unit.

5.6 Verifying abstract program transformations

NPPV does not restrict the user to a fixed set of data structures. New operations and relations may be freely introduced. This feature opens the door to verifying not just fixed programs, but rather general program transformations.

As an example consider the transformation from recursive into sequential programs. Recursive programs are usually easier to specify than sequential ones, but recursive executions often require extra resources in time and space. Therefore, many methods have been invented to transform recursive programs into sequential ones. As a first example we will here only consider the transformation of tail-recursive programs into sequential ones.

Consider the recursive definition of a function f in terms of already available functions g, r and a relation P. The recursive definition is *tail-recursive*, if it is of the form

$$f(x) = \begin{cases} g(x), & \text{if } P(x) \\ f(r(x)), & \text{else.} \end{cases}$$

An imperative program to compute the same function f is given below. It has already been annotated with the proper pre- and postconditions and a loop-invariant.

```
{ x = M }
  WHILE not P(x) DO     { f(x) = f(M) }
      x := r(x);
  x := g(x)
{ x = f(M) }
```

The verification conditions generated by NPPV are :

```
P(x)     => f(x)=g(x)
not P(x) => f(x)=f(r(x)),
```

which is precisely the requirement of tail-recursivity.

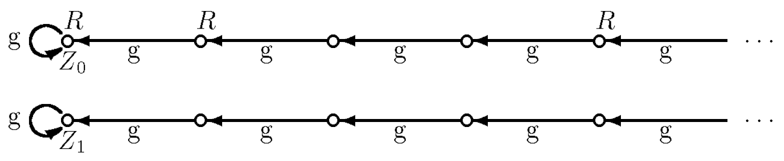
5.7 Verifying incompleteness

M. Wand [6] has presented a data structure $\mathcal{W}$ over which the Hoare calculus is incomplete. From our earlier remarks it follows that $\mathcal{W}$ is not expressive. The signature of $\mathcal{W}$ extends the Boolean signature by

$$\begin{array}{ll} g & : W \longrightarrow W \\ Z_0, Z_1, R & : W \longrightarrow \mathbb{B} \end{array}$$

and the operations are defined on the set $W = \mathbb{N} \times \{0,1\}$ via

$$\begin{array}{ll} g(n,i) & := (n \dot{-} 1, i), \\ Z_0(x) & :\Leftrightarrow x = (0,0), \\ Z_1(x) & :\Leftrightarrow x = (0,1), \quad \text{and} \\ R(x) & :\Leftrightarrow \exists k.\, x = (2^k, 0). \end{array}$$



Wand considers the following program C_W over $\mathcal{W}$:

$$\texttt{while}\ \neg(Z_0(x) \vee Z_1(x))\ \texttt{do}\ \mathrm{x} := \mathrm{g(x)}$$

It is obvious that $wp(C_W, Z_0(x)) = \{(n,0) | n \in \mathbb{N}\}$, which is the upper copy of $\mathbb{N}$ in the figure, so in particular the Hoare triple $\{R(x)\}\, C_W\, \{Z_0(x)\}$ is valid.

Assuming that this can be proven in the Hoare calculus, we submit the annotated program to NPPV. We have to supply the while-loop with an invariant, which, if it exists, must be a logical expression with at most **x** as free variable. NPPV allows us to enter such an "unknown" expression, so we enter

```
{ R(x) }
WHILE not (Z0(x) or Z1(x)) DO  { I(x) }
        x := g(x)
{ Z0(x) }
```

NPPV generates the following verification conditions

```
R(x) => I(x)
I(x) and not (Z0(x) or Z1(x)) => I(g(x))
I(x) and Z1(x) => Z0(x)
```

from which we conclude that `I(x)` describes the same set as before, namely $\{(n,0) | n \in \mathbb{N}\}$.

Using the Beth-definability theorem[2], Wand shows that the above set is not definable, contradicting the existence of an expression `I(x)`. We shall now give an elementary proof of this result.

Definition 6. We call a subset $S \subseteq W$ *thin*, if $\sum_{(n,i)\in S}^{\infty} \frac{1}{n}$ converges. Complements of thin sets are called *thick*.

Lemma 7. *Every definable set in Wand's algebra is either thick or thin.*

Proof. Thin sets form an order ideal, in particular, they are closed under finite unions and subsets of thin sets are thin. The set of all thin or thick subsets of W therefore forms a Boolean algebra $\mathcal{B}$. The basic predicates Z_0, Z_1 and R define sets in $\mathcal{B}$, therefore all quantifier free expressions in $\mathcal{W}$ define sets in $\mathcal{B}$. The proof is finished, if we can show that $\mathcal{W}$ allows quantifier elimination, i.e. every logical expression is equivalent in $\mathcal{W}$ to a quantifier free expression. For this it suffices to show that for every variable x and Boolean expression $B(x, \ldots)$ we can find another Boolean expression B' not containing x, so that

$\mathcal{W} \models B' \Leftrightarrow \exists x.B(x, \ldots)$. Since $B(x, \ldots)$ may be transformed into disjunctive normal form and since $\exists$ distributes over $\vee$, we may actually assume that $B(x, \ldots)$ is of the form $L_1 \wedge \ldots \wedge L_n$, where each L_i is atomic or negated atomic. Now in $\mathcal{W}$ every atomic Boolean expression contains at most one free variable, hence

$$(\exists x.L_1 \wedge \ldots \wedge L_n) \Leftrightarrow L_1 \wedge \ldots \wedge L_k \wedge (\exists x.L_{k+1} \wedge \ldots \wedge L_n),$$

where $L_{k+1}, \ldots, L_n$ are those L_i whose free variable is x. This expression clearly is equivalent either to $L_1 \wedge \ldots \wedge L_k \wedge \texttt{true}$ or to $L_1 \wedge \ldots \wedge L_k \wedge \texttt{false}$.

Corollary 8. *$\mathcal{W}$ is not expressive and Hoare's calculus is incomplete over $\mathcal{W}$.*

Proof. $I = wp(C, Z_0)$ is not first order definable, since it is neither thin nor thick.

6 Data Structure = Algorithm - Control

NPPV's proof-component will either succeed in proving a given verification condition, or simplify it to a (hopefully) simpler but logically equivalent statement. It will not force the user to prove these remaining statements, rather collect them into an "axioms file".

This gives rise to a novel perspective on program verification. Given a program together with an appropriate annotation, a set of data structure axioms will be generated such that

the algorithm satisfies the specification	$\Leftrightarrow$	the data structure axioms are satisfied.

Thus, given a desired algorithm, a data structure may be tailored so that the algorithm computes the desired function. We shall give a number of examples.

6.1 Gauss

Recall that the proof of the summation program succeeded automatically except for one verification condition that NPPV could not prove. This was the condition

```
i < N => i+1 <= N.
```

All that NPPV assumes about the operations $+, *$ and the relations $<$, resp. $\leq$, is that they form an ordered commutative ring with unit. The unproved verification condition can be thus interpreted as an axiom for the data structure needed to make the program work. In other words, the unproved property tells us that Gauss's summation formula is true provided the ring carries a discrete order.

For good reasons one might argue that we have proven Gauss's theorem rather than simply proving that the program sums all numbers up to N. So what we actually should be specifying in the postcondition is that

$$sum = \sum_{0}^{N} i.$$

Since the $\sum$-operator is not defined in NPPV, we simply specify in the postcondition :

```
{ sum = sumTo(N) }
```

and in the invariant :

```
{ sum=sumTo(i) and  i <= N }.
```

In addition to the previous

```
i < N ==> i+1 <= N
```

NPPV now generates the two conditions :

```
sumTo(0)= 0
i < N => sumTo(i+1) = sumTo(i)+i
```

which we are ready to accept as the definition of the summation operator.

6.2 Swap, revisited

Let us now investigate the reasons what made the earlier tricky exchange program work. In order to do that we formulate the same program structure using abstract terms p, q, r :

```
{ x = A and y = B }
   x := p(x,y)
   y := q(x,y)
   x := r(x,y)
{ x = B and y = A }
```

NPPV generates the following verification condition :

```
q(p(A,B),B) = A
r(p(A,B),A) = B
```

On close inspection we find that these are precisely the defining equations of a quasigroup. To emphasize this, let us replace p, q and r with infix symbols $*, /,$ and $\backslash$. We see that the equations specify that $*$ should be a binary operation which is both left- and right-cancellative, i.e.

```
(A * B) / B = A
A \ (A * B) = B.
```

Thus we find that the content of two variables can be switched by a sequence of three assignments, iff the underlying data structure contains a quasiqroup operation.

6.3 An abstract two-person game

Suppose we have a two-person game given by

- a set S of (game-)states
- subsets $Init, Terminal \subseteq S$
- a relation $R \subseteq S \times S$ characterizing the legal moves, such that $\forall \sigma \notin Terminal.\, \exists \sigma'.\, \sigma\, R\, \sigma'$.

A game starts in an initial state with two opposing players taking turns to move. A player wins if his move reaches a terminal state. We are looking for conditions that guarantee a win for the first player.

In the following program we model the players with the two-element data type

```
Player = ({You, Me }, = ).
```

Let *myMove* and *yourMove* be the functions realizing the moves of the players, that is once *You* are in state s you move to $yourMove(s)$, similarly, the function *myMove* determines my moves. We assume that $yourMove, myMove \subseteq R$.

The abstract game-playing program, together with the stipulation that Me should win is :

```
{ Init(s) }
turn := Me;
WHILE not Terminal(s) DO
  IF turn = Me
    THEN
      BEGIN  s := myMove(s)  ;  turn := You END
    ELSE
      BEGIN  s := yourMove(s);  turn := Me  END ;
IF turn = Me
  THEN winner := You
  ELSE winner := Me
{ winner = Me }
```

When verifying this program, we have to supply an invariant for the loop. In the absence of any further information about the rules of the game, we invent an abstract predicate depending on the relevant variables, `P(player,s)`.

NPPV generates four verification conditions, which we simplify slightly using the trivial axioms of the `Player`-data type: $t = You \lor t = Me$ and $\neg(t = You \land t = Me)$:

```
Init(s)  ==> P(s,Me)
P(s,Me)  ==> not Terminal(s)
P(s,Me)  ==> P(myMove(s),You)
P(s,You) and not Terminal(s) ==> P(yourMove(s),Me)
```

Let *MyPos* resp. *YourPos* be the sets defined by the unary predicates $P(s, Me)$, and $P(s, You)$. Note that in order to guarantee a win for every possible legal move the opponent (You) might make, the function *yourMove* must be considered a nondeterministic function, whereas $myMove$ can be thought of as a Skolem function, choosing an appropriate new state if one exists. With this in mind, the above axioms can be reformulated in set language as:

$$
\begin{aligned}
&Init \subseteq MyPos \\
&MyPos \cap Terminal = \emptyset \\
&\forall_{s \in MyPos} . \exists_{s' \in YourPos} . s \, R \, s' \\
&\forall_{s \in YourPos} . \forall_{s' \in S} . \, s \, R \, s' \Rightarrow s' \in MyPos
\end{aligned}
$$

Thus the set $MyPos$, if it exists, can be called a *strategy.* In order not to lose, I must (and can) always make a move resulting in a state within $YourPos$.

6.4 Programming = Proving

From the examples that we have seen so far, it may appear that programming is as hard (and in fact the same type of activity) as proving a mathematical theorem. In a very abstract sense we can demonstrate this fact using NPPV.

Assume that $X(n)$ is a property of natural numbers. $X(n)$ is obviously true, if and only if a program P that starts at 0 and checks all numbers until it finds one that does not satisfy X, will never stop. In this abstract framework we can write the program P where the fact that P never stops can be specified by the postcondition $False$. Thus we obtain :

```
{ True }
  n := 0 ;
  WHILE X(n) DO
    n := n+1
{ False }
```

NPPV will require us to annotate the loop with an invariant. Since it is not clear what this invariant should be, we just add an abstract predicate $I(n)$, which may depend on n, the only variable in the program. The verification conditions that NPPV generates show very succinctly the connection between programming and theorem proving :

```
I(0)
I(n) => I(n+1)
I(n) => X(n).
```

6.5 Abstract invariants

The last example shows quite clearly, that it is futile to hope for a widely applicable method for finding proper invariants. Still there are ways to proceed and a verifier may be helpful in this. Firstly, given a specification and a program we may use as invariant an abstract predicate $I(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are all variables occurring in the program. The verifier will then generate a set of verification conditions and the problem becomes to show that they are not contradictory.

In case where a program C is to compute a function $f(x)$, the specification will typically be $\{x = A\}\, C\, \{z = f(A)\}$. A while-loop calculating $f(A)$ will modify x, z and perhaps some auxiliary variables, but maintain an invariant specifying how at each moment $f(A)$ can be recovered from x, z, and the auxiliary variables. With an abstract function r, then $r(x, \ldots, z) = f(A)$ should be attempted as an invariant, where the "..." stand for the auxiliary variables.

The resulting verification conditions can be seen as a set of axioms for a data structure required to make the algorithm work. If a data structure exists making these axioms true, then the program can be accepted as a correct implementation of the specification. The next step will be to implement the data structure conforming to the axioms.

In the following we apply this method in showing how an arbitrary linearly recursive function may be implemented by a sequential program with the aid of a *stack*. To be specific, a function f is *linear recursive*, if it is of the form

$$f(x) = \begin{cases} g(x), & \text{if } P(x) \\ h(f(r(x)), x) & \text{else.} \end{cases}$$

An example of a linear recursive function is the previously discussed function *sumTo*. Moreover, every primitive recursive function is linearly recursive. The following program purports to implement the function f in general, using a stack. We have annotated the loops with invariants stating that in the first loop, $f(A)$ can *somehow* be recovered from $f(x)$ and s (by some as yet unknown function *prod*) whereas during execution of the second loop, $f(A)$ is recoverable in the same way from z and s.

```
{ x = A }
BEGIN
  s := empty;
  WHILE not P(x) DO { prod(f(x),s) = f(A) }
    BEGIN
      s := push(x,s);
      x := r(x)
    END ;
  z := g(x);
  WHILE s <> Empty DO { prod(z,s) = f(A) }
    BEGIN
      z := h(z,top(s));
      s := pop(s)
    END
END
{ z = f(A) }
```

To simplify matters, let us assume that the axioms for the "stack"-data type, are known to NPPV (in practice, they can be supplied in a "theory file"), then we remain with the verification conditions :

```
prod(x,empty) = x
prod(x,push(y,s)) = prod(h(x,y),s)
```

These equations can be considered as the defining equations for the unknown function *prod*. From the freeness axioms for the stack-operations *empty* and *push* it follows that they unambiguously define a total function. Thus we have shown that a proper invariant for the program exists, which is all we need to know.

7 Conclusion

The scope of program verification techniques can be extended beyond their original goals which was verifying correctness of individual programs. Assuming correctness of an implementation, axioms for a required data structure can be inferred. If these axioms are not contradictory, the data structure can be implemented in a second step, applying the same method again.

Mechanical program verifiers play an essential role in that task. They can be designed to handle abstract program schemata and thereby aid theoretical understanding and discussion of the mathematical foundations and interrelations.

References

1. Apt, K.R.: Ten years of Hoare's logic: A Survey – Part I. ACM Trans. Progr. Lang. and Systems **3**(1981) 431–483
2. Beth, E.W.: Formal methods. D. Reidel, Dordrecht-Holland 1962.
3. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM Joun. on Comp. **7**(1978) 70–90
4. Gumm, H.P., Sommer, M.: Einführung in die Informatik. Addison Wesley, 2nd ed. 1995
5. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(1969) 576–580
6. Wand, M.: A new incompleteness result in Hoare's system. Journ. ACM **25**(1978) 168–175

This article was processed using the LaTeX macro package with LLNCS style