3

Programs for Mechanical Program Verification

D. C. Cooper
Department of Computer Science
University College of Swansea

Abstract

A set of programs (written in POP-2) have been developed to be used in investigations into mechanical and mechanically-aided proofs about programs. These include programs for algebraic manipulation and simplification, logic manipulation, input, conversion to flow chart, and conversion to 'block form' of programs, the Presburger decision algorithm, the proof of convergence of programs by Floyd's well-ordering technique, and the proof of correctness of programs by attaching relations to blocks. The present stage of this project will be described.

INTRODUCTION

This paper describes the present stage of research aimed at building up a set of computer routines in the form of POP-2 functions to be used in mechanical and mechanically-aided proofs about programs – for example, proof of correctness and convergence. Certain functions will clearly be required – functions for algebraic and logic manipulations and for the input and manipulation of programs, others will probably be useful – various theorem proving routines such as a resolution-based prover and a prover for decidable subcases such as Presburger arithmetic, and others will be special to the particular task. As a particular problem we take the task of producing a mechanically-aided proof of convergence and correctness of programs (by convergence we mean that the program terminates, by correctness we mean that if it terminates its results satisfy some desired condition). The basis for the technique is that we associate relations (between values of variables at the start and at the end) with parts of the program and then prove these relations do indeed hold for any inputs. We give an outline of the necessary theory.

ALGEBRA AND LOGIC FUNCTIONS

Clearly functions for the manipulation of algebraic and logic expressions will be required. An expression is represented by a member of the record class EXPR with components to indicate the kind of expression and its subcomponents. There are operators to combine expressions according to all the usual rules of algebra and of the propositional and predicate calculi; conditional expressions may also be constructed. There are functions for input and output of expressions. Rather than break every expression into binary components, addition, multiplication, logical 'and', and logical 'or' are represented as n-ary operations. General functions and predicates may be represented in the system. Quantification is allowed; however the problem of clashes of bound variable is avoided by using unique names of a special form for the bound variables. Functions are provided for iterating through expressions, searching expressions for subexpressions with desired properties, and substitution. Conversion to conjunctive or disjunctive normal form is available.

All 'immediately recognizable' simplification is performed as soon as it occurs. Thus properties of 0, 1, *true* and *false* will always be used; in any addition or multiplication at most one number will appear; pure numeric subexpressions will be evaluated; and an addition will not appear as a subexpression of an addition. There is no recognition of common subexpressions, although numbers will be collected together. Thus 1+A * B+A * B+2 will automatically simplify to 3+A * B+A * B. Further simplification functions for algebraic expressions are provided and may be called on if needed. In the application so far, this has not proved necessary for the algebra part although the logic expressions do tend to grow. All this is programmed in an obvious manner.

THE PRESBURGER ALGORITHM

A formula is said to be a formula of Presburger arithmetic if it is formed from algebraic expressions (only allowing variables, integer constants, addition and subtraction), the arithmetic relations $<$ and $=$, the propositional calculus logical connectives, and quantification (either universal or existential). We may trivially add the other usual arithmetic relations. The prime omission is multiplication, although we can allow multiplication by a constant ($3 * a$ is an abbreviation for $a+a+a$). Many of the simpler formulae arising from proofs about programs, for example, those expressing results about the counting variables around loops, fall into this class, and it is useful to have an algorithm to check their validity or to simplify them by eliminating quantifiers. The basic Presburger algorithm will decide the validity (or satisfiability) of such formulae and is given in Hilbert and Bernays (1968). Our implementation is a modification of this including equality (rather than eliminating it by replacing $a=b$ by $a<b+1 \wedge b<a+1$, its use makes the algorithm more efficient), subtraction, and both positive and negative integers.

Basically the algorithm is a rule for eliminating an existential or a universal quantifier, and indeed this is its most frequent use in the programs. Any quantifiers introduced are immediately eliminated, if the quantified part is in Presburger arithmetic. Validity (or satisfiability) of a formula can be decided by universally (or existentially) quantifying all free variables, eliminating all quantifiers from the innermost to the outermost, and evaluating the resulting formula, which can have no variables.

Consider then the elimination of x from $(\exists x)F$ where F is quantifier free. Universal quantifiers may be eliminated by a dual process, or by using $(\forall x)F \equiv \neg(\exists x)\neg F$. By transforming F to disjunctive normal form and distributing the quantifier over the disjuncts we obtain a number of separate cases in which the F part is a conjunction of relations or negations of relations. Negation may easily be removed: replace $\neg(E<F)$ by $F<E+1$ and $\neg(E=F)$ by $E<F \vee E>F$. In this latter case it will be necessary to retransform to disjunctive normal form. We now have a formula of the form:

$$(\exists x)\left[\bigwedge_{i=1}^{p} a_i' < \alpha_i x \wedge \bigwedge_{i=1}^{q} \beta_i x < b_i' \wedge \bigwedge_{i=1}^{r} \gamma_i x = c_i'\right] \wedge G \tag{1}$$

where each relation has been 'solved for x'; a_i', b_i', and c_i', and G do not include x, and α_i, β_i and γ_i are positive integers. Either p, or q, or r may be zero indicating the corresponding formula does not appear.

Let δ' be the least common multiple of the α_i, β_i and γ_i. Then eq. (1) is clearly equivalent to the special case of eq. (2) in which $k=1$, $\delta_1=\delta'$ and $d_1=0$. The notation $\delta \mid x$ means x is exactly divisible by δ.

$$(\exists x)\left[\bigwedge_{i=1}^{p} \alpha_i < x \wedge \bigwedge_{i=1}^{q} x < b_i \wedge \bigwedge_{i=1}^{r} x = c_i \wedge \bigwedge_{i=1}^{k} \delta_i \mid x + d_i\right] \wedge G \tag{2}$$

To prove this, consider x of eq. (2) to be δx of eq. (1). We generalize on k as the introduction of the '|' function will persist to the next quantifier elimination, hence we must allow for it occurring in the original F. If the original formula included this function, we must also allow for its negation. This can either be eliminated or the following formula complicated to allow for this situation. However, in our examples this negation cannot appear and so we do not include it.

Let δ be the least common multiple of all the δ_i.

If r is not zero we may choose one of the equalities and trivially eliminate x by substitution.

If r is zero and either p is zero or q is zero then eq. (2) is equivalent to

$$\bigvee_{j=1}^{\delta} \bigwedge_{i=1}^{k} \delta_i \mid j + d_i \wedge G \tag{3}$$

If r is zero and neither p nor q is zero then eq. (2) is equivalent to

$$\bigvee_{j=1}^{\delta} \bigvee_{s=1}^{p} \left[\bigwedge_{i=1}^{k} \delta_i \mid a_s + j + d_i \wedge \bigwedge_{i=1}^{p} a_i < a_s + 1 \wedge \bigwedge_{i=1}^{q} a_s + j < b_i\right] \wedge G \tag{4}$$

This completes the process, but we make certain remarks concerning its efficiency. Eq. (3) may be replaced by G if $k=1$. Eq. (4) is asymmetric with regard to a_i and b_i. Clearly we can write down an equivalent formula concentrating on b_i rather than a_i, and this should be done if $q<p$ as then there will be fewer terms. In constructing the relations in eq. (4) it pays to simplify them by collecting like terms; this is particularly easy in Presburger arithmetic and a special simplification function is used. The case $i=s$ in the second conjunction need not be included; however it is simpler to leave it and let the simplification remove the term.

There are two sources of combinatorial explosion, the construction of eqs. (3) and (4) and the original conversion to disjunctive normal form. The unsatisfactory feature of eqs. (3) and (4) is that their size depends on δ, that is, on the integers occurring in the original formula. This seems unavoidable. However, we do not need to construct all the terms, that is, j can be left as a parameter and the fact that it is to take all integer values from 1 to δ recorded. Eqs. (3) and (4) are already in disjunctive normal form and, on the assumption that we never need to convert to conjunctive normal form, this parameter may remain undisturbed – we must expand it, though, to carry out the conversion. If the original formula has either all existential or all universal quantifiers at the front then we do not need to carry out this conversion. Additionally, if all α_i and β_i of eq. (1) are 1 then δ is 1 and there is no explosion. We have not so far needed any other case, although a simple example in which we will get 2^{20} terms is:

$$(\forall a)(\forall b)(\exists x)[a<20x<b].$$

After all eliminations have been carried out the final variable-free formula will contain these 'limited parameters' and can be tested by exhaustive evaluation; however, this is again impractical on even simple examples. However, results from the theory of solution of linear equations in integers may be used to produce a practical evaluation as follows:

We wish to find all possible solutions of a set of divisibility relations $\delta_i | E_i$ where δ_i are integers and E_i are linear expressions in the variables. These may be solved by reduction to triangular form and back substitution, the back substitution being on sets of values, as we have several solutions. The following theorem may be used to eliminate a variable:

$$\delta | mx+p \wedge \gamma | nx+p \quad \text{if and only if}$$

$$\delta\gamma | dx+p\gamma\psi+q\delta\phi \wedge d | pn-qm$$

where d is the greatest common divisor of δn and γm; and $d=\delta n\phi+\gamma m\psi$ (these ϕ and ψ will be automatically found if Euclid's algorithm is used to compute d).

To solve for a variable use:

$\delta | mx+p$ has solutions for x if and only if $d | p$; these solutions are

$$x=-\frac{\phi p}{d}+\frac{\delta}{d}t$$

where t is any integer, d is the greatest common divisor of m and δ, and $m\phi+\delta\psi=d$.

INPUT OF PROGRAMS AND CONVERSION TO GRAPH FORM

Functions are provided for the input of programs written in a simple language allowing assignment statements and conditional jumps (*see* example program later). With this simple language we can make a start, leaving elaboration to more realistic languages to the future. These programs are read in and stored in 'graph form', i.e., as flow charts.

The graph form of a program is a list of 'program nodes', the first of which is regarded as the entry point, the order of the rest being irrelevant. A program node is a pair consisting of a block and a list. A block may be regarded as a basic piece of code which, for the moment, is not to be analyzed further. It may have several exits. In the present version of the program there are four kinds of blocks, only one of which will occur in the graph form of a program (*see* next section for more detail on blocks). The one which occurs in the graph form may be thought of as standing for:

if P_1 **then** $V_1:=E_1$; **goto** *exit*1
elseif P_2 **then** $V_2:=E_2$; **goto** *exit*2
.
.
.
.
elseif P_n **then** $V_n:=E_n$; **goto** exit n

The P, V, and E expressions will be stored as members of a list in an array, the array subscript varying from 1 to n and indicating the particular exit. The list component of a program node will be a list of next nodes in the same order as the components of the array. A program node consisting of a special block with no exits and a null list indicates termination of the program.

BLOCK FORM

One line of investigation being followed is that of regarding programs as being made up entirely from blocks. A block is a section of the program with one input, one or more outputs, and a given internal structure which may itself contain sub-blocks. A definition of block form was given in Cooper (1968). However, that definition insisted on blocks having a single exit. This leads to difficulties in treating general programs as being composed from blocks; these difficulties disappear if we allow several exits to a block. Alternatively, we may view a block form program as one in which all loops are properly nested.

In the POP-2 implementation a block is a record with four components BNX, BT, BC, and BNUMB. BNX is an integer (the number of exits), BT is an integer from 1 to 4 (indicating the type of block), BC is a general item

giving information about the block depending on its type and BNUMB is an integer or word (its external name). The four types of block are:

Type 1. Basic block

TC is an array, $BC(I)$ is a list of three expressions, P_I, V_I, E_I. The semantics of this block has been defined in the previous section; it corresponds to basic computation and this block has no sub-blocks.

Type 2. Sequence block

BC is an array, $BC(I)$ is a list of pairs, each pair having a block and exit number as its components. This corresponds to a sequence of blocks, for example, if block A has as the value of $BC(2)$ the list of pairs $[B.3, D.1]$ then to exit through the second exit of block A, control must first go through block B exiting at its third exit and then through block D to its first exit. (Note that A, B, and D will actually appear as internal names for the blocks.)

Type 3. Loop block

BC is a pair with components of type block and integer. The loop block is created from the sub-block by taking its *integer*th exit and looping this back to the start, thus creating a new block with one less exit.

Bype 4. Merge block

BC is a pair with components of type block and array. The merge block is created by joining together several of its sub-block's exits to form a single exit. The Ith component of the array is a list of integers, and these exits from the sub-block must all be merged to form the Ith exit from the main block.

Whilst the main motivation for blocks is that they are natural units of programs about which to prove, or assume, sub-results (compare subroutines and library programs), yet they are completely general in that corresponding to any 'flow chart' program an equivalent block form program may easily be constructed. It is useful to have a function to carry out this transformation and this is provided. A method of carrying out the transformation is simply to eliminate the program nodes one at a time until only one is left: the block of that node will, together with all its sub-blocks, be the equivalent block form program. A node can be eliminated by the process illustrated in the transformation from figure 1 to figure 2, suitably generalized.

The final block form obtained depends on the order in which the node eliminations are performed. The simple method of eliminating the first node on the node list (after the entry node which is never eliminated) often leads to a very complicated block form; what is wanted is a process which produces the most natural blocking.

Worse, a disastrous combinatorial process of copying can occur which, although it will eventually terminate, makes the process impractical. Use of the following two heuristics produced natural blockings and no explosion on the examples tried:

(a) As soon as two or more exits from a node lead to the same program node, create a merge block. Note that the original algorithm did not produce any

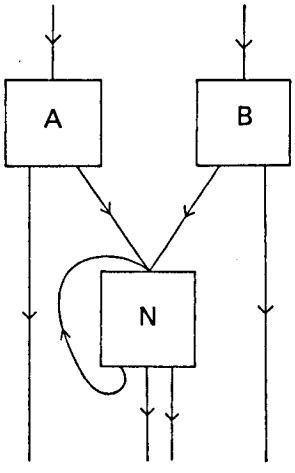

Figure 1. Before elimination of node N

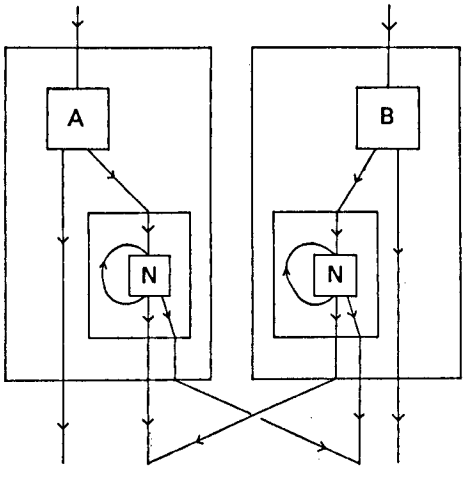

Figure 2. After elimination of node N

merge blocks. They are not theoretically needed except for one overall merge if we wish the complete program to have only one exit rather than several.
(b) Give preference to nodes with only one incoming arc.

ASSIGNING RELATIONS TO BLOCKS: THEORY

Floyd (1967) and Naur (1966) have independently developed the idea of attaching predicates to points in a program, these predicates having the property that if control reaches that point then the predicate will be true of the current values of the variables; in particular the predicate on the end point expresses the correctness of the program. Conditions may be derived relating these predicates, and if we can prove the validity of these conditions then we will have proved the correctness of the program. These predicates are not local properties of that point, they depend on the whole program. In some ways it seems natural to isolate a part of the program and specify its properties, then combine these properties to obtain the conditions to be verified. Thus one is led to consider block form, attaching relations (between values of variables on input and output) to blocks and deriving conditions between these relations depending on the particular block structure. This gives a theory analogous to that developed by Floyd; we now sketch that theory, and also extend it to cover proof of convergence. All proofs will be omitted.

Assume we have a universe, U, whose members are possible values for a program's state vector (alternatively, consider the case of a program with one variable: the universe is then the set of all possible values of that variable). Lower case italic letters, e.g., x, denote variables ranging over U, lower case

bold letters, e.g., **a**, denote sets of members of U, i.e., predicates, and capital bold letters, e.g., **A**, denote sets of ordered pairs of U, i.e., relations. The notation **a**x means that x is a member of **a** (or equivalently **a** is true of x) and x**A**y means that $\langle x, y\rangle$ is in the set **A** (or equivalently x is in relation **A** to y).

We define two operations. The relation **AB** is the usual composition of relations and is defined by

$$x\mathbf{AB}y \text{ is } (\exists z)[x\mathbf{A}z \wedge z\mathbf{B}y]$$

The set **A . b** is defined by

$$(\mathbf{A}\,.\,\mathbf{b})x \text{ is } (\forall z)[x\mathbf{A}z \rightarrow \mathbf{b}z]$$

where $\rightarrow$ is the logical implication operator. This may be interpreted programwise by noting that if **A** is a 'change' relation associated with the end of a block then **A . b** is the set of x such that if we start the block with value x and if control comes out of the given exit, then **b** must be true of the final value.

With each exit of a block is associated two sets ($\mathbf{a}^\dagger$ and **a** say) and two relations ($\mathbf{A}^\dagger$ and **A** say).

$\mathbf{A}^\dagger$ is the actual relation between input and output values, i.e., $x\mathbf{A}^\dagger y$ iff when the block is started with value x control leaves the block by the given exit with value y.

$\mathbf{a}^\dagger$ is the set of starting values which will cause control to leave the block by the given exit, i.e., the domain of $\mathbf{A}^\dagger$.

A is called a 'change relation' and is the relation between input and output values we are trying to prove. It would not normally be as strong as $\mathbf{A}^\dagger$, e.g., we might only wish to prove, in the single numerical variable case, that the value of the variable is increased – **A** would then be the set of $\langle x, y\rangle$ such that $x<y$.

a is called a 'convergence condition' and is the condition on the input values which we are trying to prove guarantees that the block is left by the given exit. It is normally stronger than $\mathbf{a}^\dagger$, thus we might only wish to prove that a program converges for $0<x<100$ when in fact it converges for $0<x$.

These quantities are related by:

$$\mathbf{a} \subseteq \mathbf{a}^\dagger = \text{domain } (\mathbf{A}^\dagger) \tag{5}$$

and

$$\mathbf{A}^\dagger \subseteq \mathbf{A} \tag{6}$$

Each block is either basic or contains sub-blocks. In the latter case we can mechanically obtain equations relating the above four quantities on the exits of the main block with those on exits of sub-blocks. Conditions can be obtained relating the actual relations (semantic equations), the change relations (change equations), and the convergence conditions (convergence equations). We shall not fully define these – they are obvious generalizations of the following special cases.

Basic block

Semantic equation: $\mathbf{A}^\dagger$ is explicitly defined for each exit.

Change equation: $\mathbf{A}^\dagger \subseteq \mathbf{A}$

Convergence equation: $\mathbf{a} \subseteq \text{domain } (\mathbf{A}^\dagger)$

Sequence block. Case of one-exit block ($\mathbf{b}, \mathbf{B}, \mathbf{B}^\dagger$ associated with exit) followed by one-exit block ($\mathbf{c}, \mathbf{C}, \mathbf{C}^\dagger$) to form new one-exit block ($\mathbf{a}, \mathbf{A}, \mathbf{A}^\dagger$).

Semantic equation: $\mathbf{A}^\dagger = \mathbf{B}^\dagger\mathbf{C}^\dagger$

Change equation: $\mathbf{BC} \subseteq \mathbf{A}$

Convergence equation: $\mathbf{a} \subseteq \mathbf{b} \cap \mathbf{B} \,.\, \mathbf{c}$ ($\cap$ is set intersection)

Merge block. Case of two-exit block ($\mathbf{b}_i, \mathbf{B}_i, \mathbf{B}_i^\dagger$; $i=1$ and 2) merged to form new one-exit block ($\mathbf{a}, \mathbf{A}, \mathbf{A}^\dagger$).

Semantic equation: $\mathbf{A}^\dagger = \mathbf{B}_1^\dagger \cup \mathbf{B}_2^\dagger$ ($\cup$ is set union)

Change equation: $\mathbf{B}_1 \cup \mathbf{B}_2 \subseteq \mathbf{A}$

Convergence equation: $\mathbf{a} \subseteq \mathbf{b}_1 \cup \mathbf{b}_2$

Loop block. Case of two-exit block ($\mathbf{b}_i, \mathbf{B}_i, \mathbf{B}_i$; $i=1$ and 2) in which exit 1 is looped back to start and exit 2 becomes the exit of the new one-exit block ($\mathbf{a}, \mathbf{A}, \mathbf{A}^\dagger$).

Semantic equation: $\mathbf{A}^\dagger$ is the minimal solution of $\mathbf{B}_1\mathbf{X} \cup \mathbf{B}_2 \subseteq \mathbf{X}$ for $\mathbf{X}$

Change equation: $\mathbf{X}_{\min} \subseteq \mathbf{A}$ where $\mathbf{X}_{\min}$ is the minimal solution, for $\mathbf{X}$, of $\mathbf{B}_1\mathbf{X} \cup \mathbf{B}_2 \subseteq \mathbf{X}$

Convergence equation: $\mathbf{a} \subseteq \mathbf{x}_{\min}$ where $\mathbf{x}_{\min}$ is the minimal solution, for $\mathbf{x}$, of $\mathbf{b}_2 \cup (\mathbf{b}_1 \cap \mathbf{B}_1 \,.\, \mathbf{x}) \subseteq \mathbf{x}$.

The semantic equations may be taken as a definition of the semantics of a program in block form. Alternatively if we have some other definition then they will have to be proved. Given that the semantic equations are accepted, the following two theorems may be proved by induction on the block structure:

Theorem 1. If with every exit of every block we have associated change relations, and if all change equations are valid, then eq. (6) is valid for every exit (and in particular for the final exit, thus proving correctness of the program).

Theorem 2. If with every exit of every block we have associated change relations and convergence conditions, and if all change and convergence equations are valid, then eq. (5) is valid for every exit (and in particular for the final exit, thus proving convergence of the program).

Equations for basic, sequence, and merge blocks may easily be used to obtain formulas of the first-order predicate calculus and standard theorem proving techniques used. However, in contrast to the Floyd approach, the equations for a loop block are inherently second order, because of the minimality conditions. The change equation may be proved valid by first-order means if we can produce an $\mathbf{X}$ such that $\mathbf{B}_1\mathbf{X} \cup \mathbf{B}_2 \subseteq \mathbf{X} \subseteq \mathbf{A}$ (often $\mathbf{A}$ can be used for $\mathbf{X}$). This corresponds to producing a formula on which to use mathematical induction or to finding a Floyd predicate of just the right 'power' to associate with an arc in a loop. Similar remarks may be made concerning the convergence equation: here we have to provide a proposed predicate which expresses the condition that the loop is traversed exactly n- times; and the proof can then be made by first-order means.

The change equation minimal solution is clearly

$$\mathbf{X}_{\min} = \bigcup_{n=0}^{\infty} \mathbf{B}_1^n \mathbf{B}_2$$

where $\mathbf{B}_1^n$ is the composition of $\mathbf{B}_1$ n-times and $\mathbf{B}_1^0$ is the identity relation. If we know $\mathbf{B}_1$ and $\mathbf{B}_2$ then we can start constructing successive terms and, in examples tried, it soon becomes obvious what $\mathbf{X}_{\min}$ is. Similarly for the convergence equation, if we assume first that the operation $\mathbf{B}_1.\mathbf{x}$ is continuous [in the sense defined, for example, in Park (1970)]; secondly that $\mathbf{b}_1 \cup \mathbf{b}_2$ is the universe, and thirdly that the domain of $\mathbf{B}_1$ is disjoint from $\mathbf{b}_2$, then it can be proved that the minimal solution is

$$\mathbf{x}_{\min} = \bigcup_{n=0}^{\infty} \mathbf{B}_1^n \,.\, \emptyset$$

where $\emptyset$ is the empty set. Programwise $\mathbf{B}_1^n \,.\, \emptyset$ is the condition that control leaves the loop block in less than n iterations.

The second condition (on $\mathbf{b}_1 \cup \mathbf{b}_2$) can easily be removed, the third condition can easily be satisfied by redefining $\mathbf{B}_1$ to exclude members of $\mathbf{b}_2$ from its domain, but the first condition is a real restriction. The dot operator is not necessarily continuous, although it is if $\mathbf{B}_1$ is a partial function. It is easy to give an example in which we do not have continuity. Consider the following loop:

L: **if** $x=1$ **then goto** *exit*
elseif $x=0$ **then** [set x to be any integer greater than 0].
else $x-1 \rightarrow x$ **close; goto** *L*;

Consider the part in square brackets to be any subroutine about which all we know is the given fact.

The loop clearly converges for all non-negative initial values of x. However, the $\mathbf{B}_1$. $\mathbf{x}$ operator is not continuous. This corresponds to the fact that if the initial value of x is zero we cannot predict how many times it will go round the loop even though we know it must terminate.

These infinite union methods are practicable in the sense that, on examples tried, evaluation of the first few terms soon shows one what the infinite union must be. However, we have no program that can make this inductive step.

ASSIGNING RELATIONS TO BLOCKS: PRACTICE

There are various ways in which the technique of assigning-relations-to-blocks technique could be used. The user could supply relations and we could attempt mechanically to verify the change equations, or he could supply convergence conditions and we could also verify the convergence equations. In practice we should probably not wish to adopt a pure relation-to-block approach, or a pure predicate-to-points approach, but mix the two. Direct transformation between Floyd predicates and our relations may be made – roughly if $\mathbf{p}(x)$ is the predicate on the input, $\mathbf{q}(x)$ that on an exit, and $\mathbf{A}$ the change condition then $\mathbf{p}(x) \rightarrow \mathbf{q}(y)$ is $x\mathbf{A}y$. Indeed this may be elaborated

into a simple direct proof of theorem 1 by deducing it from the corresponding Floyd result, or vice versa deducing Floyd's result from ours.

The particular task we have chosen is to prove the convergence of a program as mechanically as possible. This, on the programs usually arising in practice, should be a lot easier than trying to prove their correctness. We adopt the convergence equation approach attaching convergence conditions (and change relations where necessary) to exits from blocks. At first we ask no help from the user, the program trying to deduce conditions under which a program converges. The program is controlled in a recursive manner by the block structure. Thus there are two basic functions: CHANGE(block, exit number) and CONV(block, exit number), giving the change relation and convergence condition of a block exit. These look at the block structure, call on themselves recursively for sub-blocks, and then combine these results according to the appropriate equation, e.g., for the sequence block convergence condition example of the last section: compute **b**, **B**, **c**, and then take as the convergence condition $\mathbf{b} \cap \mathbf{B} \,.\, \mathbf{c}$. Results of all calls are recorded so that if needed again they do not have to be recomputed.

For the loop case we have no general method, but simple counting loops are recognized – specifically if only one variable occurring in the sub-block convergence conditions is altered in the sub-block. If that variable is altered in a linear manner and occurs in a linear manner in the convergence condition, then CONV can produce an answer. One can imagine other cases, or some kind of inductive technique. If CONV cannot deduce the condition then it invents a new predicate name, prints this name and relevant details of the loop, and exits with this new predicate as the answer. Alternatively provision is made for CONV to be given enough information to prove its result by the well-ordering method of Floyd (1967). The user must supply the convergence condition **a** and a set of functions $\mathbf{f}_1 \ldots \mathbf{f}_n$ mapping the values of the variables onto the non-negative integers. Let, as before, $\mathbf{b}_1$ be the convergence condition on the sub-block exit which loops back, $\mathbf{B}_1$ its change condition, and $\mathbf{b}_2$ the convergence condition on the other exit. Then **a** will be a convergence condition if we can prove:

$$\mathbf{a}x \rightarrow \mathbf{b}_1 x \vee \mathbf{b}_2 x$$

$$\mathbf{b}_1 x \wedge \mathbf{a}x \wedge x\mathbf{B}_1 y \rightarrow \mathbf{a}y$$

$$\mathbf{b}_1 x \wedge \mathbf{a}x \rightarrow \mathbf{f}_i x > 0 \qquad \text{for } i = 1 \ldots n$$

$$\mathbf{b}_1 x \wedge \mathbf{a}x \wedge x\mathbf{B}_1 y \rightarrow [\mathbf{f}_1 y, \mathbf{f}_2 y, \ldots, \mathbf{f}_n y] < [\mathbf{f}_1 x, \mathbf{f}_2 x, \ldots, \mathbf{f}_n x]$$

We have changed from a set notation to a predicate notation, **a** is now considered a predicate of one argument and $\mathbf{B}_1$ a predicate of two arguments; $\mathbf{a}x$ is more usually written $\mathbf{a}(x)$ and $x\mathbf{B}_1 y$ written $\mathbf{B}_1(x, y)$. The functions $\mathbf{f}_1, \ldots, \mathbf{f}_n$ map the values of the variables onto the set of ordered n-tuples of non-negative integers. These are well ordered by the relation $<$. This can be any suitable relation, but in the program ordering on the first member of the n-tuple is used, if these are equal on the second, and so on. The fact that

the above equations do indeed show convergence can be proved from the theory of the previous section, or directly from the semantics, or taken as obvious. These equations are formed and passed to a theorem-prover; at present this is the Presburger algorithm (many cases fall into Presburger arithmetic) but we have begun experiments with a resolution type theorem prover.

One other important practical point, and restriction, should be mentioned. So far we have taken a state vector approach, i.e., our predicates and change relations depend on the values of all the variables occurring in the program. For many proofs of convergence, or particular aspects of correctness, only some of the variables will be involved. Clearly we do not wish to deal with the program variables of no consequence in the proof. Change relations computed need only consider that part of the relation which indicates how the variables of interest are affected. Our philosophy takes this to the extreme in that all our change relations give information only on how the final value of one particular variable depends on the initial values of variables, and also which variables' initial values can possibly affect the final value. Thus we have a record class 'CH1' whose members contain this information; we do not have the full CHANGE function mentioned earlier but only CHANGE1(block, exit number, variable). Effectively if we have n variables then a change relation is regarded as the intersection of n change relations, each of which specifies how one variable is changed and is a 'don't care' on the rest. Thus, even if our program cannot prove the convergence of some loop, it will isolate those variables which can affect the looping and point out how these are changed in the loop. This use of CHANGE1 is a restriction: thus we could not prove results about the final value of the sum of two variables without first proving individual results about each variable, which could be much more difficult. There is of course no difficulty in principle in removing this restriction.

COMMENTS ON SOME PRELIMINARY RESULTS

So far we do not have very much experience with the programs. However there is one particular example program which has been used all along as a test vehicle. This is a program to calculate how a sum of money may be paid using a minimum number of coins (in the British monetary system before the demise of the half crown, which creates an extra problem because it is the only coin not a multiple of the next lower-valued coin). The program involves an array. As yet arrays are not properly dealt with and so all references to the array **B** [i.e., all occurrences of **B**(I)] are replaced by the single variable BI. The following is the complete test program, as input to our program. The only omissions are assignments at the start to the **B** array (**B**(I) is the value in pence of the Ith coin) and the calculation of C (the sum of money). This latter does not affect convergence, and there is no cheating involved here as our program would easily detect this. We would

hope that the program could come up with the overall convergence condition $BI>0$.

```
      MIN:=1000000; MINX:=0; I=1;
L1:   NC:=0;
L2:   IF C>BI GOTO L3;
      IF I=5 GOTO L7;
      IF NOT(I=10) GOTO L4;
      IF NOT(MINX<MIN) GOTO L5;
      MIN:=MINX;
L5:   IF SNC=0 GOTO L6;
      MINX:=SMINX; C:=SAVEC+30; SNC:=SNC-1; I:=5;
L4:   I:=I+1; GOTO L1;
L3:   MINX:=MINX+1; C:=C-BI; NC:=NC+1; GOTO L2;
L6:   STOP;
L7:   SNC:=NC; SMINX:=MINX; SAVEC:=C; GOTO L4;
```

Of course this program can be written more legibly in a language with better syntax, but that is not the problem we are concerned with here. Basically there is an inner loop (lines *L2* and *L3*) within an outer loop counting on *I*, but on termination of this outer loop it may be restarted with $I=6$ depending on a count in SNC. Proof of its termination is not therefore completely trivial.

The program succeeds in obtaining the block form of this program, this block form corresponding naturally to the blocks a user would put on the flowchart (except in one respect we comment on later). It was then given the task of obtaining the convergence condition for the overall coin program. It printed out (in a less readable fashion) that the convergence condition was $\mathbf{f}(\text{SNC}, 1, \text{BI})$ where $\mathbf{f}(\text{SNC}, \text{I}, \text{BI})$ is the convergence condition for the loop with the following properties:

Exit condition: $\text{BI}>0 \wedge \text{I} \neq 5 \wedge \text{I}=10 \wedge \text{SNC}=0$

Loop condition: $\text{BI}>0 \wedge \text{I} \neq 5 \wedge \text{I}=10 \wedge \text{SNC} \neq 0 \quad \vee$
$\text{BI}>0 \wedge \text{I} \neq 5 \wedge \text{I} \neq 10 \quad \vee$
$\text{BI}>0 \wedge \text{I}=5.$

Change1 relation for SNC: $\text{P} \vee \{[\text{Q} \vee \text{R}] \wedge [\text{S} \vee \text{T}]\}$

where P is $\text{SNC}'=\text{SNC}-1 \wedge \text{BI}>0 \wedge \text{I} \neq 5 \wedge \text{I}=10 \wedge \text{SNC} \neq 0$

Q is $\text{SNC}'=\text{SNC} \wedge \text{BI}>0 \wedge \text{I} \neq 5 \wedge \text{I} \neq 10$

R is $\text{SNC}' \geqslant 0 \wedge \text{BI}>0 \wedge \text{I}=5$

S is $\text{BI}>0 \wedge \text{I} \neq 5 \wedge \text{I} \neq 10$

T is $\text{BI}>0 \wedge \text{I}=5$

(note SNC′ denotes the final value of SNC).

Change1 relation for *I*: a similar, but longer ($2\frac{1}{2}$ line printer lines) expression which we will not give here.

Change1 relation for *BI*: $BI'=BI$, i.e., no change.

The CONV function is not sufficiently powerful to evaluate this function.

However, as mentioned earlier, the program can attempt to prove convergence by the well-ordering technique. The above loop was given to the well-ordering function with the following additional information:

convergence condition is $BI>0\wedge[1\leqslant I<5\vee\{6\leqslant I\leqslant 10\wedge \text{SNC}\geqslant 0\}]$; well order by mapping onto a 3-tuple with

$\mathbf{f}_1$ is **if** $I\leqslant 5$ **then** 1 **else** 0

$\mathbf{f}_2$ is **if** $I\leqslant 5$ **then** 0 **else** SNC

$\mathbf{f}_3$ is $10-I$.

There are then four theorems to be proved, all within Presburger arithmetic. These theorems have all been printed out; we omit them here but the fact that they take 1, 4, 2, and 7 lines of full 120 character lines gives an indication of their size. The first has been proved by the Presburger algorithm, the third could be, but the other two would take too much computing time to be worth trying. The inefficiency is two-fold. Not too much trouble has been taken with writing efficient programs, much use being made of small functions. Recoding the basic functions in machine code would greatly increase the efficiency. Much more serious are inefficiencies in the algorithm: either more automatic simplification must be carried out so that the Presburger algorithm has a simpler formula to work on or we must find a better decision procedure. The main inefficiency arises in the initial reduction to disjunctive normal form. The formula proved by the system had 55 disjuncts: each of these has to be proved inconsistent. In fact the variations on the clauses came for the most part in parts of the clause not contributing to the inconsistency and so much work was repeated. The formulae themselves, whilst not too easily comprehended by a human, yet are not prohibitively large. The transformation to disjunctive normal form, however, can be disastrous. It is not difficult to think of heuristics which would soon enable the computer to prove these theorems. For example, the variable I occurs frequently, and most occurrences are in subpredicates involving no other variable. This suggests splitting the formula into a number of cases depending on the value of I. This leads to a set of much simpler formulae with which the Presburger algorithm should have no difficulty.

Several points arose in developing the program to its present state: conditional expressions were not included at first in the algebraic expression as we also had logic and arithmetic relations. Apart from the general consideration that they are basic to computation, they were found to be needed for the user communication of functions to be used in the well ordering. The necessity for the two heuristics mentioned in the previous section on conversion to block form was shown up by the program, and also their adequacy on the examples tried. Some choice is available on what to use as a change relation – in particular, as we also have the convergence conditions, do we need to restrict the domain of the change relation? For example, should we use $x\mathrm{B}y$ is $x>0\wedge y=x+1$ or just $y=x+1$? If the first is a valid

change relation, then so is the second. If we wish to know the condition for the path to be traversed we have to ask for the convergence condition anyway, and so it seems reasonable not to include $x>0$ in the change relation. This is indeed the policy adopted, we only use change relations to indicate how variables are altered. This led to trouble in connection with the merge block change equation, $\mathbf{B}_1 \cup \mathbf{B}_2 \subseteq \mathbf{A}$. Suppose $\mathbf{B}_1$ said the final value of x was 0, $\mathbf{B}_2$ said it was 1; then **A** (taken to be $\mathbf{B}_1 \cup \mathbf{B}_2$) will just say that x is either 0 or 1. This, as indeed the example showed, is probably insufficient to prove the required result. In this case then (when we are merging two different change1 conditions) we do restrict the change relations to the domain of the convergence conditions (by calling on CONV) and form a conditional expression as the change condition for the merge block.

Whether the calls on CONV just mentioned need to be done or not depends on why the change relation is needed, and this shows up a basic weakness in the system caused by the recursive control. This means that each level produces its best possible result, whereas it may be that a more quickly obtainable, but weaker, result will suffice. Various control strategies are possible, for example, a calling routine indicating how much resources (presumably computer time) a subroutine is allowed, or a subroutine producing a quick answer, but willing to be re-entered to find a stronger result. Whilst we certainly need more work to improve the power of our techniques, yet it remains probable that we shall always want some heuristic-type organization to make the best use of them.

A more serious point arose in connection with the merge block convergence equation: indeed the example showed up an incompleteness in the convergence part of the theory. The situation is given in figure 3 in a simplified form. Block C is a block about which we have proved that control cannot get stuck here and that the value of x is not decreased. It is trivial to see that the A block always converges, but our theory cannot prove it! The convergence condition $\mathbf{b}_1$ is empty, no initial value of x can guarantee that control comes out of the 1 exit. Convergence condition $\mathbf{b}_2$ is $x \geq 10$, hence the strongest condition we can give for $\mathbf{a}$ ($\mathbf{b}_1 \cup \mathbf{b}_2$) is $x \geqslant 10$. This is certainly a convergence condition, but not the best. In this case a different form of blocking would solve the problem, the more natural blocking with the test and its two exits forming a block would cause no trouble. This seems more natural and the block conversion algorithm could be altered to recognize this situation (and if we had done that in the first place we would probably still be unaware of the incompleteness!). However that does not make the theory right: as given in figure 3, we have an always converging blocked program and we should be able to prove it. The extension to the convergence theory seems to be that we need quantities like $\mathbf{b}_{12}$ – a condition on the input which guarantees that control comes out of either the 1 exit or the 2 exit. It may be, as in this example, that we can do better than take $\mathbf{b}_1 \cup \mathbf{b}_2$ for this condition. The theory can readily be extended in this way – but then we could construct an example

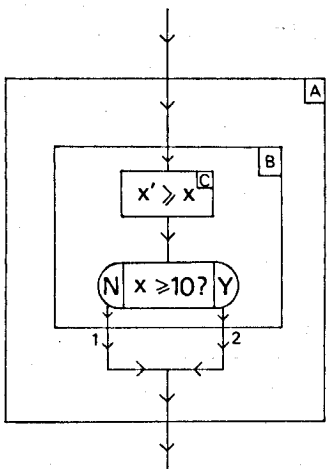

Figure 3. Incompleteness example

which needs $\mathbf{b}_{123}$ say. So at this point we give up and say more work is needed.

King (1969) has implemented a verifying compiler, i.e., assertions are placed by the user (at least one to a loop), formulae are produced, and the program attempts to prove their validity. This corresponds to the change part of our program in which the user would supply a proposed change relation for '*n* times around a loop'. He has a method for dealing with arrays. His theorem-prover also requires first transforming to disjunctive normal form; however as he has carried out logic simplifications whenever possible in the building up of expressions, this could be a simpler task than ours. The theorem-prover consists of a number of steps, each of which may prove the theorem, but if it does not then the prover moves to the next step; thus the prover can be very efficient on simple theorems. The whole is not a complete decision procedure for Presburger arithmetic; however King suggests the missing cases may well not arise in practice. It also extends Presburger in the sense that non-linear terms can sometimes be dealt with, for example, if the prover wishes to make substitution of an integer for x, say, then if the original formula had a term $x*y$ that term would now come into Presburger arithmetic; the alternative device of substituting z for $x*y$ throughout may work. King has taken trouble over the efficiency of his program, in particular it is written in an assembly language with heavy use of macros. Thus, on the problems it can deal with, it can act as a benchmark for future efforts.

This paper has presented a package of POP-2 programs for use in mechanical program verification, and an example of their use in a convergence-proving program based on the 'relations to blocks' theory. This theory is not put forward as the way to deal with programs – it was the development of an

idea for which practical test seemed to require mechanical aids (relations are not easy things to deal with by hand). The kind of problems which have arisen, and the kind of arithmetical theorems we need to prove, will probably arise in any theory – indeed the theorems can give practical motivation to research in the area of mechanical theorem-proving. We have described the present stage of the project; this paper should in no sense be interpreted as presenting definite conclusions and results, since limitations of our programs are still being worked on.

Acknowledgements

This work was done with the aid of a grant from the Science Research Council and I am grateful to my colleagues for their comments. In particular Martin Weiner contributed to the particular Presburger algorithm used and Malcolm Bird and Robin Milner to the algorithm for transformation to block form.

REFERENCES

Cooper, D.C. (1968) Some transformations and standard forms of graphs with applications to computer programs. *Machine Intelligence 2*, pp. 21–32 (eds Dale, E. & Michie, D.). Edinburgh: Edinburgh University Press.

Floyd, R.W. (1967) Assigning meanings to programs. *Mathematical Aspects of Computer Science*, pp. 19–32. Providence, Rhode Island: Amer. Math. Soc.

Hilbert, D. & Bernays, P. (1968) *Grundlagen der Mathematik I*, pp. 366–75. Berlin: Springer-Verlag.

King, J.C. (1969) *A program verifier*, Ph.D. thesis. Carnegie-Mellon University, Pittsburgh.

Naur, P. (1966) Proofs of algorithms by general snapshots. *B.I.T.*, **6**, 310–16.

Park, D. (1970) Fixpoint induction and proofs of programs properties. *Machine Intelligence 5*, pp. 59–78 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.