Distribution Category:
Mathematics and Computers (UC-32)

ANL-86-44

ANL--86-44

DE87 003295

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

A FORMAL NOTATION FOR HARDWARE AND SOFTWARE VERIFICATION

Richard O. Chapman and John R. Gabriel

Mathematics and Computer Science Division

October 1986

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38.

CONTENTS

A Formal Notation for Software and Hardware Verification*

R. O. Chapman† and J. R. Gabriel
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439-4844

1. Program Representation

Computation might be viewed as the replacement of one symbol by another symbol according to some finite and completely specified set of rules. That view, in itself, is useless for accomplishing any reasonable purpose, unless there is an interpretation of those symbols in which some property of the symbols is preserved by the replacements, while some other desired property is added. Another way of phrasing the same thing is to say that computation is the assignment of values to variables. Said this way, a program might be described as a relation between a set of starting symbols (the possible combinations of values of the input variables) and a set of stopping symbols (the resulting combinations of values of output variables). Of course, given a specification for a computation in such a form, the set of non-trivially different programs that meet the specification is large.

This paper develops a formalism appropriate to demonstrating the functional equivalence of two programs. That is to say, we develop a formalism in which one might use an automated or human reasoner to rewrite one program's representation in the formalism to the representation of any functionally equivalent program, using a set of rewriting rules that specify how a representation may be rewritten such that the function of the program it represents is not altered. This project is motivated by the success of similar notations for representing digital logic, most notably those of H. Barrow's program VERIFY [Barrow 1984] and M. Gordon's papers on modelling and verifying hardware [Gordon 1981].

2. Programs and Digital Circuits

Digital circuitry also performs computations, though a circuit does not sequentially follow an algorithm. The computation consists of the generation of certain voltage levels on the output wires, that being a function of the state of the machine and the voltage levels present on the input wires. The algorithms of Gordon [1981: p. 4.6] provide a means of creating such a representation of a digital circuit by tracing the flow of data between inputs and outputs, building expressions for each output to indicate its dependence on the inputs. We propose a similar method for representation of programs.

In some sense, a data flow model such as this is the "natural" model for hardware representation, since the idea of flow of control is little used in a typical digital logic system. Usually, all parts of the circuit are operating simultaneously. By modelling software using the same sort of data-flow algorithms, we hope to achieve two goals: to shed some light on

* This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38.

†Current address: Wadham College, Oxford University, Oxford OX1 3PN, United Kingdom

methods for verifying combined systems with both hardware and software components, and to increase the domain of verifiable programs and circuits.

A number of parallels between the structures of digital logic and programs led to this approach. One might think of the input wires to the circuit as corresponding to the input variables of a program, and similarly the output wires to the output variables. The modules of the circuit, which produce "local" output signals, might be viewed as similar to those subprograms that compute new values using existing values (a program statement may itself be viewed as a subprogram in the assembly language representation of the program). The wires of the circuit connect outputs of some modules to inputs of others, much as variables connect values returned by one subprogram to the argument list of another. This is the motivating idea behind our claim that a similar notation may be appropriate for hardware and software, and it sheds some light on the sorts of programs for which our representation is appropriate. In particular, we abstract away from the entire idea of a "user interface" between computational results and some human being. We perceive the problems of a program of that sort as being of an entirely different nature than those of program verification.

A major advantage to describing hardware in these terms is that one may exploit of the hierarchical nature of digital circuit organization, as pointed out by Barrow [1984: p. 446]. To a somewhat lesser extent, the same sort of hierarchical organization is present in software. A program typically consists of a "main" part which calls numerous subroutines, each of which in turn may call subroutines of its own. However, unlike the components of digital circuitry, chosen from a relatively small set of integrated circuits in some logic family, each program uses its own unique subroutines, tailor-made for the particular application. Barrow's success in verifying hardware may provide yet further evidence for several pieces of by-now conventional programming wisdom: one should program in a structured fashion, prohibit a called routine from modifying the values of global variables, and make extensive use of preverified library routines whenever possible, rather than writing new code. These practices may be essential if a program is to be verifiable.

3. Description of Digital Hardware

The Gordon notation [Gordon 1981] represents a circuit as a list of parts or modules and a list of the interconnections between the parts (corresponding to wires or buses in the physical circuit). The parts are then defined recursively. Each module may itself be represented in a similar fashion, but at some level each circuit or subcircuit must be made of simple parts. That is, at some level each module must be made of submodules that are not defined in terms of their own constituent parts (the recursion ends). Simple parts are assumed to function as intended, and accordingly they are described by sets of equations for computing the values of the output signals and new states as a function of the current state and input signals.

Using the list of interconnections between modules and the equations for the values of the outputs of simple modules, for each level in the hierarchy of description one may arrive at a "system of equations" relating the outputs of a circuit to its inputs, in terms of the relations and functions of the circuits parts. The interconnection list specifies what equation to use for the value present at each of the input ports of the modules that make up the circuit. So, using substitution, one may derive equations for the "final" outputs of the circuit in terms of its "initial" inputs. Hence, the intermediate modules and connections do not appear in the derived equations, and the resulting equations are a "black box" description of the circuit. In Gordon's notation the derived equation takes the form of an expression of lambda calculus, while Barrow [1984: p. 451] actually substitutes the value present at an input for the name of the input in a module's output equation.

Such a scheme would be completely adequate for describing a purely combinational circuit, that is, one for which the output did not depend on the history of the circuit. One must consider as well the state of the circuit at the time of the input if the circuit is sequential. Sequential behavior in digital logic arises from the presence of feedback loops in the circuit, such as the case of the cross-coupled NAND gates in the R-S latch [Mead and Conway 1980: pp. 82-83]. Such circuits cannot be adequately explained in discrete terms; the latch's working as a memory device is described not by its construction from digital devices, but by the solutions to differential equations describing its behavior as an analog device. (For a more complete treatment, see Gabriel and Roberts [1984].) The equations may be solved to show that an R-S latch has two stable states, and that certain perturbations of the voltage on the input wires can lead to a transition from one state to the other, while other perturbations lead to no change. In physical terms, the latch works because of the temporary charge storage by the NAND gates.*

The Gordon model, which we adopt, abstracts away from this level of description, adopting implicitly the Mead-Conway fiction of a two-phase system clock to describe the behavior of a sequential system. All signal transfers take place either in phase 1 or phase 2 of the clock cycle. Signals arising from purely combinational logic are computed in phase 1, and these values are latched into a series of hypothetical state variables in phase 2. State variables are introduced in each loop of the circuit's signal flow, and the set of values of those state variables defines the circuit's state. Hence the behavioral description for a sequential module consists of a set of equations for computing the output signals from input signals and the current values of the state variables, and a set of equations for computing the new values of the state variables from the previous inputs and state variable values. Otherwise, any attempt to evaluate sequential circuits according to the described program of substitution would never terminate; one would repeatedly replace the name of an input signal in a loop with the expression for its value, which contained the name of the signal to be substituted for, and hence required another substitution, *ad infinitum*.

The Gordon procedure for producing these expressions describing circuits resembles a Markov algorithm [Galler and Perlis 1970: pp. 1-4]. The process consists of applying a number of rules for rewriting one string to produce another by performing substitutions among a number of perhaps mutually recursively defined equations. One may make the substitution ordered by any rule that applies, until one reaches a termination condition: stop evaluating recursively when the expression to be evaluated denotes either a state variable or an input port to the circuit [Gordon 1981: pp. 4-6]. Reaching a state variable indicates that one has evaluated the part of the circuit computed in phase 1 of the clock cycle. We observe that in the case of sequential logic constructed out of combinational components, a termination condition is never reached according to Gordon's rules; each application of a substitution rule makes it possible to apply another substitution rule, since the outputs of the gates are recursively defined (that is equivalent to saying that there is a loop in the signal flow).

In fact, one may algorithmically determine the location of state variables necessary in a given system.† When recursively evaluating a circuit by substituting a computed value for the name of an input signal, one need keep track only of the signal path described by the sequence of substitutions (in a list of interconnections traversed, for example), checking each time a substitution is used that it has not already been used. If it has not, it is added to the list. If it has, then we know that the path examined forms a loop, and a state variable must be introduced

*In fact, it is possible to build an R-S latch from NAND gates from early TTL series chips which behave not as a latch, but as an oscillator; the NAND gates lack sufficient hysteresis.

†This point was suggested, but not discussed, by Barrow [1984: p. 451].

somewhere along that loop to prevent its being traversed repeatedly.*

There is, interestingly, a similar problem to be dealt with when attempting to represent loops in software. In that case also, new variables must be introduced to describe the program. In the hardware case the loop in the signal flow is described by the equation for the value of the state variable, a recurrence relation typically relating a part of the state of the machine at time t to its state at time $t-1$. A loop in a program transforms some variables (part of the program's state) according to a similar recurrence relation, defined by the transformation coded in the body of the loop. In the circuit, however, that transformation is continuously applied to whatever signals are present at the loop's inputs; all parts of the circuit operate simultaneously, and there is little or no "flow of control." In a program the loop transformation is applied a finite number of times to one set of input values. The program loop has an exit condition as well as a recurrence relation. If we knew the number of iterations of the loop, we could specify an equation for its effect on the program state using that number as one of the parameters in the equations. So, we will introduce a new variable to keep track of the number of loop iterations to be performed, rather than a set of state variables. This process will be described below.

4. Languages of Description

The notion of equivalence of function depends very much on the language, natural or otherwise, in which the description of function is expressed. For example, consider the class of programs that update a general ledger. At that level of description, all programs in the class are identical. If we add more descriptive power to our language, the class will be broken into subclasses naturally, according to the coarseness or fineness of the language in which they are described. ALGOL-like computer languages are probably all approximately equivalent in descriptive power. Thus, using the notation to be described as a tool for showing equivalence between two different programs should not require consideration by the user of the question of appropriate scope for the representation language.

On the other hand, using the notation to verify that a program behaves as it was intended seems to be a very different case. If the verification is to serve any purpose at all, it must be somewhat easier (in the sense of less prone to error) to formulate descriptions of programs in the language of behavior specifications than in the computer language in which the program itself is written. If not, then it seems that program verification is little more than a sophisticated version of the old method of ensuring error-free keypunch entry by having the data typed twice and comparing the two versions for discrepancies. That being the case, it becomes an important question—and one not addressed by this note—what expressive power is appropriate to a general-purpose language for describing behaviors of computer programs, for as a language gains the power to describe in more detail, the user of that language is faced with a more difficult task in translation from his native language into the language of behaviors.

5. Von Neumann Machines

For the following discussion we limit ourselves to Von Neumann-model computers executing sequential programming languages such as Fortran or C (we expect to consider a generalization of the representation for MIMD machines in another paper). Input, output, flow of control, and the assignment of values to variables (which one may think of as reading and

*We are implementing such a system in Prolog, using the notation of Barrow [1984]. The circuit parts and connection lists are Prolog clauses, and the program asserts new clauses for parts called state variables, as needed, inserting them along loops in the signal paths. We hope to publish details of this work in another report.

writing to memory locations) are the fundamental actions of any Von Neumann machine. However, all of these kinds of action are discernible externally only in the ways they alter the output values. Thus, to evaluate a program functionally, we need only derive expressions in terms of input variables for the values returned by each of the output statements. Church's lambda calculus provides a formal system for representation of programs in these terms, and consequently we attempt to produce expressions whose evaluation according to the rules of lambda calculus correspond to execution of the programs from which the expressions were generated.

6. Evaluation Procedure for Programs

The evaluation procedure for a program is a set of formal rules for producing a representation of a computation in a language like lambda calculus, given a request for a computation in a programming language. The representation consists of a set of lambda expressions, one for each output statement of the program. To evaluate an output statement, one builds an appropriate lambda expression for that statement's value. One starts with the output statement (in which some variable names usually appear) as the "string" under consideration. Working backwards through the program, one determines what transformations the variables in the string underwent, at each stage relating (or binding) the name of a variable to the previous expression computing its value, and then recursively evaluating any other variables whose names appear in the resulting expression. Each programming statement that modifies a variable whose value is used in computing the value of the output variable specifies a substitution that can be made for the value of that output variable. As a motivational example, consider the following program fragment:

```
begin
    input(x);
    input(y);
    z:=x+y;
    write(z);
end;
```

If one were to analyze the *z* that appears in "write(z)", one would find the statement "z:=x+y" as the last statement which changed the value of *z*. Thus an expression such as "write(x+y)" ought to be derivable from "write(z)" by substituting the expression for the value of *z* in place of *z*'s name. We can see that the easiest form of programming statement to view in this manner is the direct assignment of a value to a variable, either by an assignment statement or by a subroutine returning a value. The rule corresponding to this kind of statement will say that one may substitute (obeying the rules of lambda calculus) the expression assigned to the variable in place of the variable's name in any expression after the assignment in which the variable appears. The evaluations produced by this procedure give, in effect, a dataflow description of the program.

7. Other Kinds of Program Statements

The statements in a program other than the assignment statements either perform I/O functions or alter the flow of control in the program. In describing flow of control we restrict ourselves to consideration of conditional branching and loops. We describe each in terms of what effects it may have on assignment of values to output variables, since that will dictate the form of the representation of that statement in terms of the substitutions in strings it specifies.

Think of an input statement as a zero-argument function producing a value, which can be substituted for the name of the input variable in any later expression in which it appears. In the recursive analysis of relations between variables and value, input statements serve as recursion terminators; we cannot express the value of variable read by an input statement in any simpler terms.

Think of output statements as the "outermost" functions whose arguments are to be substituted with values. As said earlier, these are the only variables whose values matter from a functional standpoint.

Think of a branch instruction as a triadic operator, whose value is either of two functions, each consisting of the transformations specified by one of the two paths of the branch, based on the value of a third function (which usually evaluates a Boolean expression). Any variable whose value is changed by statements that form one of the two paths of a branch instruction may have the name of the variable replaced by an expression that gives the Boolean condition and the two different transformations of the variable's value based on the value of the Boolean expression.

Loops are more difficult to handle, and are the subject of the next section.

8. Exiting from a Loop

It is difficult to specify a value to substitute for a variable transformed by statements in the body of a loop, since the number of iterations of the loop is not known before the program is run; it typically depends on the values of the input variables. Consequently, the number of transformations of the state of the program is not known prior to running the program with particular inputs. This situation poses a difficulty in explicitly describing the function of the program, since, in general, the question of whether a program executing a loop will exit that loop is equivalent to the halting problem, and hence insoluble. That is to say, we cannot write a program that will answer the halting question for every loop in every program.

But, in fact, we can assume it is answered positively for all correctly written programs,* for the following reason. If we are creating a functional description of a program by making relations between variables that are output and the values they take, we know that if in the course of tracing backwards it becomes necessary to evaluate a variable whose value was transformed by a loop, the loop was indeed exited; otherwise the output statement that is influenced by the variable we are analyzing would never be executed. Since we may rightly claim that the programmer should not have coded a statement he never intended to be executed, we assert that in any well-formed program all loops will be exited after a finite number of iterations.† In addition to these arguments, it may be remarked that if programs execute only on finite state (actual) machines, then the halting problem is decided perhaps by an error exit of some kind, enforced by the operating system or by the hardware.

This is an assumption made by programmers, even though not all programs are written so that the assumption is made clear. That is to say, the programmer has some idea, based on his

*That is, if for a given program, the halting problem cannot be decided, then the program is in some sense proved incorrect.

†Some loops in operating systems behave a little differently. They are intended to run indefinitely, performing the identity transformation on the program state (the only transformation that can be performed an indefinite number of times on a finite state machine), until a request for service is received. When a request is received, the loop is exited, the service performed, and the loop resumed. However, our assumption about the loop having been "exited" when the output was generated remains true. When a line gets written to a device, only a finite number of iterations of the loop preceded it, and since the transformation performed was the identity transformation, it may be ignored.

knowledge of things that are not necessarily represented to the program, that the loop he codes will terminate. That knowledge usually is based on what he knows can be assumed about the system of program, computer, and input. This can be seen in the fact that a programmer almost always has some idea how long would be appropriate for the program to spend executing the loop. For example, a program to read a file and send it to an output device probably contains a loop to read a character from the file, process it, and repeat until an end-of-file marker is reached. Should the programmer run this program and notice that after spending five minutes waiting for it to print a ten-line file he still had no output, he would probably conclude that the program was stuck in an infinite loop, and he would terminate its execution to do some debugging. In this case, the programmer knew something about what length of input file to expect, something that the program did not.

Our method will be to assume that the loop is exited; if we arrive at a contradiction in evaluation of the loop, we may conclude that the program containing the loop is not well formed. This conclusion rests on the proof by contradiction that if a program's termination leads to a contradiction, we can conclude that it does not terminate. That, in turn, requires the Law of the Excluded Middle, which may seem untenable in this case, given the undecidable character of the halting problem. However, in our world programs run on real computers, not Turing machines. Input streams and memory capacities are finite, and we may think of the system of computer/program/input as a finite state machine, whose halting problem *is* decidable. Hence our premise is a valid one.

We do not believe that requiring a finite number of loop iterations is overly restrictive in light of what has been said about human techniques for verifying that a program functions correctly. A mechanical verifier should have equally as much information to use in evaluating a program as the designer had in deciding that he had written a correct program. We use the finiteness requirement to ensure that the expressions we write to describe program behavior can be manipulated according to the rules of lambda calculus. Having required some finite number of recursive calls, we may write down a formula in which all possible transformations of program state by the loop are represented, implicitly, by allowing the loop transformation to be recursively applied to itself any finite number of times.

An estimation of how many iterations a loop might execute depends on many things, including the hardware running the program, the programming language being used, and the purpose of the program. Our purpose is to present a theoretical framework in which programs can be shown equivalent or meeting some specification. Hence we do not concern ourselves in great depth with the question of deriving appropriate upper bounds for loops. As we have mentioned, one appropriate scheme would require the programmer himself to estimate the number of iterations needed. Techniques of automated reasoning might be employed to determine an upper bound based on the function of the loop. In practice it seems appropriate to delay calculation of the bound and expansion of the recursive loop definition until the last possible moment (that is, until something is known about the input variables that determine how iterations of the loop are to be performed).

9. Representation of Loops

The number of iterations of a loop is typically determined by the truth or falsehood of some Boolean expression whose arguments are themselves transformed by execution of the body of the loop. We represent this in the lambda expression for a loop's transformation of program state by annotating the expression with a new unbound variable, say n, representing the number of iterations of the loop to be performed. The variable n must remain uninstantiated in a sense; we do not have a "good" formula to substitute for n from which its value can be calculated in terms of the input variables, unless the loop is one (e.g., a "For" loop in

Pascal) for which the number of iterations is explicitly stated or calculated in advance. In that case, n may be bound to its value in the normal manner.

We describe the action of a loop on the state of the program as a sequence of recursive calls to a transformation having the form of a branch instruction, with one path of the branch representing the effect on the program state of one pass through the loop, and the other the identity transformation. The loop transformation branch is chosen if the exit condition is not met, and the identity transformation otherwise. As described above, we introduce a new unbound variable, and that variable denotes the number of recursive applications of the loop transformation to be done. If we expand the recursive expression one time, the resulting expression should say that we will recursively apply the transformation $n-1$ times to an expression representing one transformation. We have gone to such lengths to provide some guarantee that the number of recursive calls will be finite, so that we may be sure that the lambda expressions we write will be expandable only to a finite length. Since we cannot explicitly write out all transformations of the program state prior to running the program, we represent the loop in such a fashion so that it may be expanded into any finite number of transformations, depending on what value the new variable for the number of iterations is assigned based on input values.

10. Derivation of a Representation of a Program

Two kinds of rewritings are involved in building the representation of a program. The first set of rules specifies how to write an expression in our extended lambda calculus for the value of an output of a program. When this step has been done, the only unbound variables remaining in the expression will be the input variables to the program and perhaps some variables representing the number of iterations for loops. The second step is to evaluate the result of the first step using the substitution and renaming rules of the lambda calculus to reduce the expression for the program. The aim is that one should be able to use this form to do a computation by binding the input variables to appropriate values and reducing the expression further, specifying some upper bounds for the number of loop iterations so that they can be expanded appropriately. More important, one could use a mapping between a program and its specification (provided by the designer or an automated theorem prover) to relate two representations for a computation in such a way that one could conclude that they represented functionally equivalent computations.

The main idea in our notation, as in the lambda calculus, is that of replacement of the name of a variable with an expression for its computed value. The fundamental rewriting describes binding a variable in an existing lambda expression, say $f(x)$, with a previously computed value, say G, for the free variable x. So, given the lambda expression $f(x)$ in which x is a free variable, we may replace $f(x)$ by

$$(\lambda x(f(x))G), \tag{10.1}$$

where $f(x)$ denotes a lambda expression in which x is a free variable and G is the expression to be substituted (bound) for x in $f(x)$. When discussing a program, because of the large number of variables, we will represent the states of all the variables of a program collectively with the symbol Σ to represent the state of the program. We will use Σ' to denote a different state of the program (e.g., Σ' might represent the state before some variable was assigned a value by a program statement, and Σ might represent the state after the assignment). This is only a notational convenience. One could imagine binding Σ (the entire state of the program) as the successive bindings, one by one, of each variable in Σ that is transformed by the transformation of Σ' under consideration. In particular, if the variables of Σ are bound in the order in which the program statements that make the bindings are executed, then left-to-right substitution is guaranteed to be non-ambiguous. Returning to the lambda expression given above, we would represent it as

$$(\lambda\Sigma(T(\Sigma)G(\Sigma'))). \tag{6.2}$$

11. Rules for Building a Lambda Expression from a Program

Given an output statement from the program, one relates the variable names with their values as described in the previous section, and then one recursively does the same thing for all the variable names that appear in those substituted values, until the output statement's only unbound variables are those which are given values by means of input statements. When beginning to evaluate a program statement, one might think of placing a marker for each unbound variable in the expression at the line in the program being evaluated. When binding one of those unbound variables, the marker for that variable is moved up in the program to the statement that was used to bind it. The value given by that statement is bound to the variable according to the rules given in the next section. Then, that marker is removed and markers are placed there for all the unbound variables appearing in the newly substituted value. Analysis of a statement is complete when none of the markers can be moved up any further and no new markers can be placed. In the rules below, variables and state symbols that are unbound, or free, are printed in boldface. Keep in mind that since lambda expressions are created "backwards," starting at an output statement and working toward the input statements of a program, when one builds a new lambda expression from a previously constructed expression, one is incorporating into the existing lambda expression part of the program that was executed *before* the part from which the existing lambda expression arose.

1. If $S(x1,x2,...xn)$ is an output statement in a program, then one may form the lambda expression

$$(S(\mathbf{x1},\mathbf{x2},\cdots,\mathbf{xn})). \tag{11.1}$$

As described above, such a lambda expression could be written

$$(S(\mathbf{\Sigma})), \tag{11.1a}$$

where Σ denotes the states of all variables in the program, including $x1,...,xn$.

2. If $(S(\Sigma))$ is a previously constructed lambda expression in which at least one free variable appears, and $G(\Sigma')$ is a lambda expression for the transformation applied to Σ' to yield Σ, then the lambda expression

$$(\lambda\Sigma(S(\Sigma))(G(\mathbf{\Sigma'}))) \tag{11.2}$$

can be made from $S(\Sigma)$.

3. If $S(\Sigma)$ is a lambda expression, and the state transformation in the program producing Σ was an input statement, in which some variable xi of Σ had its value set by an input device, then the following lambda expression is equivalent:

$$(\lambda xi(S(\mathbf{\Sigma{-}xi},xi))(\mathbf{input\_value})), \tag{11.3}$$

where "input_value" is a unique identifier denoting the value read in for the variable whose value was input, and "Σ-xi" denotes the state of all variables but xi. Since this "input_value" variable name appears nowhere else in the program, there are no other rules to bind it, and it will remain a free variable. In this case the Σ-notation is less appropriate.

4. If the last modification of Σ', producing Σ, where the expression $S(\Sigma)$ has been previously produced, occurs in one of the two paths of a conditional branch instruction (each of which is a basic block), then the following is equivalent:

$$(\lambda\Sigma(S(\Sigma))(?boolean(\Sigma'):true\_branch(\Sigma'),false\_branch(\Sigma'))). \quad (11.4)$$

The lambda expressions "true_branch" and "false_branch" are the transformations of Σ' which would be performed by each branch of the conditional. Thus, they will generally contain free variables of their own. The expression "boolean(Σ')" denotes the condition whose truth or falsehood determines the path taken. The operator "?A:B,C" evaluates to B if A is true, and C if A is false.

The remaining rules deal with representation of loops.

5. If, given a lambda expression S(Σ) , the transformation producing Σ is done in the body of a loop, the following is equivalent, letting G(Σ') represent the transformation by the loop body producing Σ from Σ', and letting "boolean(Σ')" represent the condition on which the loop is entered again or exited:

$$(\lambda^n\Sigma(S(\Sigma))(?boolean(\mathbf{\Sigma'}):G(\mathbf{\Sigma'}),\mathbf{\Sigma'})^n). \quad (11.5)$$

Here, the superscript n is a new free variable denoting the number of times the loop is to be executed. We address how it may be bound in rule 8. Note that, in general, the variables that are parameters to the transformation G and the Boolean expression may now be free, as denoted by the bold $\mathbf{\Sigma'}$. However, because of the way rule 6 will be applied to expressions of the form 11.5, we must prohibit the use of the reduction rule on free variables in superscripted expressions. The guarantee that n is finite means that rule 7 can eventually be applied after repeated applications of rule 6 to remove superscripted expressions.

6. Given a lambda expression of the form

$$(\lambda^n\Sigma(S(\Sigma))(?boolean(\mathbf{\Sigma'}):G(\mathbf{\Sigma'}),\mathbf{\Sigma'})^n), \quad (11.6)$$

one may replace it with the equivalent lambda expression:

$$(\lambda\Sigma''(\lambda^{n-1}\Sigma(S(\Sigma))(?boolean(\Sigma''):G(\Sigma''),\Sigma'')^{n-1})(?boolean(\Sigma'):G(\Sigma'),\Sigma')). \quad (11.6a)$$

7. An expression of the form

$$(\lambda^n\Sigma(S(\Sigma))G^n), \quad (11.7)$$

where n is either equal to zero or negative, may be rewritten

$$S(\Sigma). \quad (11.7a)$$

8. In a lambda expression of the form

$$(\lambda^n\Sigma(S(\Sigma))(?boolean(\mathbf{\Sigma'}):G(\mathbf{\Sigma'}),\mathbf{\Sigma'})^n), \quad (11.8)$$

where S(Σ) is some previously constructed lambda expression and G(Σ) is a transformation of the state of the program resulting from one iteration of a loop, n may be bound as follows:

$$(\lambda n(\lambda^n\Sigma(S(\Sigma))(?boolean(\Sigma'):G(\Sigma'),\Sigma')^n)(upper\_bound)), \quad (11.8a)$$

where "upper_bound" is some constant determined either by the program explicitly stating a bound on the number of iterations, as in a "For" loop, or by the programmer, based on knowledge of what the program will do, as discussed above. The form of rules 5 and 6 ensure that if n is too large, the identity transformation will be performed for any iterations of the loop specified after the exit condition is satisfied. This notation for a loop allows loops specified by rule 5 to be expanded by repeated application of rule 6, so that the superscript keeps track of how many recursive expansions of the loop transformation have been carried out. Rules 7 and

8 guarantee that the recursion will eventually end, as we argued earlier is necessary.

12. Reducing a Lambda Expression Algorithmically

Once the lambda expressions for the outputs of a program have been created according to the above rules, one may perform reductions on those lambda expressions, rewriting each expression with the bound variables replaced by their values, according to an algorithmic procedure that obeys certain other rules, given below [Wegner 1968: p. 183].

1. A lambda expression of the form

$(\lambda x(M(x))(G))$,

where M is an expression in which the symbol x may or may not appear, may be reduced to $M(G)$, replacing each instance of x in the string M with string G, provided that no variable that appears free in G is bound in M and provided that the symbol x does not appear bound in M.

2. The conditions described above which prohibit reduction can often be remedied by renaming the variables that appear bound in M, as follows. Given any part of a lambda expression, say M, except the name of a variable occurring immediately to the right of Σ, in which a variable, say x, appears bound, then all bound occurrences of x may be replaced by some other symbol not appearing in M, say y, provided that x does not appear free in M.

These rules apply for expressions of pure lambda calculus. Our representation language of lambda expressions differs in that application of these two rules is prohibited in the instance in which the variables to be substituted for or renamed appear in one of the superscripted parts of an expression of the form 11.5, as discussed in the rules (that is, if the variables represent variables that may be transformed by a loop whose number of iterations is not yet "known" in the sense of having an upper bound for it). Because of the implicit renaming that happens when rule 7 is applied, reduction may then be used on the fully expanded lambda expression for the transformation by a loop.

13. The Order of Reductions

In specifying the rules for reduction of a lambda expression, the effect of the order of the substitutions must be addressed. Our goal is that the results of evaluating a program's lambda expression give the same result as execution of the program. In fact, the proper order to perform the reductions depends on the way the programming language being modelled passes parameters to procedures. We speak of two different orders for performing two reductions: left-to-right (or inside-out) and right-to-left (or outside-in).

If a lambda expression is evaluated left-to-right, we search through the expression from left to right looking for occurrences of bound variables and then using the reduction rule on each. This means that we first reduce the variables that were bound in the last transformation of program state performed before the program terminated, then those in the next to last, and so on, until the only unbound variables to be instantiated are the input variables. This corresponds to the method of parameter evaluation referred to as "call by name" in real programming languages. A procedure call is evaluated as if the code of the procedure were copied into the calling routine, with the names of formal parameters substituted for the names of the actual parameters, then the copied code executed. This can lead to unexpected results when there is some relationship between the actual parameters not assumed between the formal parameters. As an example, consider the result of passing an integer variable I and the array element $A[I]$ to the following routine, which evaluates parameters by name:

```
set2(I, A[I]);
/* where set2 is defined */
procedure set2(var x, y:integer);
    begin
      x:=2;
      y:=2;
    end;
```

No matter what the initial value of *I*, this routine always sets *A*[2] equal to 2, when in fact it was perhaps intended that the value of *I* before it was set to 2 should have been used in calculating which element of *A* to set to 2.

Right-to-left evaluation first substitutes the names of the input variables into the expression for the first transformation of the program, then the values derived from the first transformation into the expression for the second, until the final transformation can be performed. This order of evaluation corresponds to call-by-value evaluation of parameters. It is the order commonly used in modern programming languages, and thus we will adopt it here. Left-to-right evaluation might be more appropriate, for example, if we were modelling ALGOL.

Some note should be made of the shortcomings of this order of evaluation for lambda expressions. In arbitrary expressions of pure lambda calculus, the Church-Rosser theorem guarantees that if a lambda expression can be reduced by two different finite sequences of reductions, each resulting in an irreducible lambda-expression, then the two resulting lambda expressions must be the same, up to notation [Wegner 1968: pp. 185-86]. Further, if two sequences of reductions lead to different lambda expressions, one of the sequences of reductions must not be finite. Finally, we can say that if a lambda expression has two different sequences of reductions leading to two different irreducible expressions, then it must contain a well-formed subexpression of a form that can be reduced to $(\lambda x(M)(A))$, in which M has no occurrences of x, and where A cannot be reduced by a finite sequence of reductions [Wegner 1968]. In a real program this corresponds to a situation in which one of the parameters to a procedure is a value produced by a second procedure that cannot be computed in finite number of steps (because of the way the second procedure is coded) and in which the computed value of the parameter is never used in computing an output of the first procedure, for example,

```
program example(input,output);

   var result:integer;

/* argument a has no effect on the returned value c */
    procedure calculate(var c:integer; a,b:integer;);
      begin
        b:=2*c;
      end; /*calculate*/

/* a call to badlydefined results in the program being caught in a loop */
      function badlydefined(x: integer);
      begin
         while true do
            begin
            end;
          x:=2;
      end; /*badlydefined*/
```

```
begin /* program example */
      calculate(badlydefined(1),result,3);
end.
```

This sort of pathological situation does not arise in real programs unless they are in error. Hence, because the lambda expressions we build are not arbitrary but based on programs, we may conclude that the possibility of this sort of situation not leading to a finite reduction is a reasonable one since, in the languages we will be modelling, execution of a program written in that manner would suffer the same problem as the evaluation of its lambda expression.

14. Conclusion

The notation we have described provides a means of representing the function of a program, while discarding information about part of its internal structure, as we move more of the specification of a computation into the background, so to speak. Any person or machine using the representation must possess the ability to infer the steps described by the information that was discarded. That ability is provided by the set of rules described in this paper, or by a program following those rules.

When we talk about the "function" of a program, we are assuming a certain level of abstraction appropriate to description of function. Function may mean producing characters on a screen, or a floating-point number, or a voltage on a wire (most generally an "output") given some starting state. For any reasonable interpretation of an abstract specification of function there must exist a catalog of primitive functions, the results of which are silently agreed upon, and a translation table between the catalog and the elements of the representation language. Clearly, the translation table must not be ambiguous, as programming languages are not; also, it should not be overly restrictive with respect to the knowledge of the device intended to carry out the function. We think that ALGOL-like languages are overly restrictive as representations of programs to theorem-provers, and consequently our notation is an attempt at discerning the character of a more abstract way of specifying computation.

References

Barrow, H. G. 1984. "VERIFY: A Program for Proving Correctness of Digital Hardware Designs," *Artificial Intelligence*, 24 (1984) 437-491.

Gabriel, J. R., and P. R. Roberts 1984. "A Signal Flow Model for Sequential Logic Built from Combinational Logic Elements and Its Implementation in Prolog," Argonne National Laboratory Report ANL-84-89.

Galler, B. A., and A. J. Perlis 1970. *A View of Programming Languages*, Addison-Wesley, Reading, Massachusetts.

Gordon, M. 1981. "Two Papers on Modelling and Verifying Hardware." Computer Laboratory, Cambridge University, England.

Mead, C., and L. Conway 1980. *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts.

Wegner, P. 1968. *Programming Languages, Information Structures, and Machine Organization*, McGraw-Hill, New York.

Appendix: Examples

Suppose we want to rewrite the following program in lambda notation, using boldface to identify the variables of the program. Here we explicitly list all bindings, for clarity, rather than use the Σ-notation for program states.

```
begin
    string: ="";
    char:="A";
    while (char <> EOF) do begin
      string:=string+char;
      read(char);
    end;
    write(string);
end;
```

The first (and only) output statement is "write(string);", so all that needs to be done to derive the functional specification for the program is to determine the value of "string" at the time it is written. We start out with the expression

$$\mathit{write}(\mathbf{string}). \tag{A-1}$$

The most recent statement before the "write" that gives "string" a value is the assignment statement "string:= string+char" which is part of the "while" loop. Applying the rule to denote a transformation that is part of a loop, we arrive at

$$(\lambda^{n}\mathit{string}(\mathit{write}(\mathit{string}))(?(\mathbf{char}\diamond \mathit{EOF}):\mathbf{string{+}char},\mathbf{string})^{n}) \tag{A-2}$$

recursively binding each occurrence of "string" to the value assigned it during the previous iteration of the loop, with the value of the first iteration remaining unbound. The value of "string" specified by that assignment statement depends not only on the value of "string" during the previous iteration but also on the value of "char." Also, the loop iteration counter n remains unbound. Before it can be bound, all variables that the loop transforms must be bound. To bind the variable "char," we are faced with two possibilities: either the value of "char" was set to a new variable representing the character read in at runtime (call it "char_value") by the statement "read(char);" during the previous iteration of the loop, or we are looking at the first iteration of the loop and the value was set by "char:="A";". We may denote the action of the loop on the value of "char" by the following expression:

$$(\lambda^{n-1}\mathit{char}(\lambda^{n}\mathit{string}(\mathit{write}(\mathit{string})) \tag{A-3}$$
$$(?(\mathbf{char}\diamond \mathit{EOF}):\mathbf{string{+}char},\mathbf{string})^{n})(?(\mathbf{char}\diamond \mathit{EOF}):\mathbf{char\_value}_{i},\mathbf{char})^{n-1})$$

That leaves exactly one instance of variable "char" unbound. The statement "char:="A";" is bound to that instance, so our expression becomes

$$(\lambda \mathit{char}(\lambda^{n-1}\mathit{char}\ (\lambda^{n}\mathit{string}\ (\mathit{write}(\mathit{string})) \tag{A-4}$$
$$(?(\mathit{char}\diamond \mathit{EOF}):\mathbf{string}{+}\mathit{char},\mathbf{string})^{n})(?(\mathit{char}\diamond \mathit{EOF}):\mathbf{char\_value}_{i},\mathit{char})^{n-1})(\text{"}\mathit{A}\text{"}))$$

The remaining variable "string" is instantiated by the assignment statement "string:=""" which occurs before the loop. Thus, the final expression in lambda notation for the program given is

$$(\lambda \mathit{string}\ (\lambda \mathit{char}\ (\lambda^{n-1}\mathit{char}\ (\lambda^{n}\mathit{string} \tag{A-5}$$

$(\mathit{write}(\mathit{string}))$

$(?(\mathit{char}{<>}\mathit{EOF}){:}\mathit{string}+\mathit{char},\mathit{string})^{n})(?(\mathit{char}{<>}\mathit{EOF}){:}\mathbf{char\_value}_{i},\mathit{char})^{n-1})(\text{"}A\text{"}))(\text{""}))$

We may instantiate *n* as follows. Suppose the computer on which this program is to be verified keeps track of the the length of a string in a two-bit word (remember, this is an example written for pedagogical reasons). Thus, if the loop goes for more than four iterations, its behavior is unimportant since it goes beyond the limitations of the hardware. So, an appropriate value for *n* is 4. The final lambda expression for our example program becomes

$$(\lambda n\ (\lambda \mathit{string}\ (\lambda \mathit{char}\ (\lambda^{n-1} \mathit{char}\ (\lambda^{n} \mathit{string} \tag{A-6}$$

$$(\mathit{write}(\mathit{string}))$$

$$(?(\mathit{char}{<>}\mathit{EOF}){:}\mathbf{char},\mathit{string})^{n})$$

$$(?(\mathit{char}{<>}\mathit{EOF}){:}\mathbf{char\_value}_{i},\mathit{char})^{n-1})$$

$$(\text{"}A\text{"}))\ (\text{""}))(4))$$

Equation A-6 is the complete representation of our program.

Distribution for ANL-86-44

Internal:

J. M. Beumer (2)
J. R. Gabriel (10)
A. B. Krisciunas
P. C. Messina
G. W. Pieper (59)
R. W. Springer (2)
E. P. Steinberg

ANL Patent Dept.
ANL Contract File
ANL Libraries
TIS Files (5)

External:

DOE-TIC, for distribution per UC-32 (168)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
J. L. Bona, Pennsylvania State University
T. L. Brown, U. of Illinois, Urbana
P. Concus, LBL
S. Gerhart, MCC, Austin, Texas
G. H. Golub, Stanford U.
W. C. Lynch, Xerox Corp., Palo Alto
J. A. Nohel, U. of Wisconsin, Madison
D. Austin, ER-DOE
L. Beltracchi, NRC (2)
R. Chapman, Wadham College, England (10)
W. Livingston, EBASCO Services (2)
G. Michael, LLL