Garbage In/Garbage Out

COMFY—A Comfortable Set of Control Primitives for Machine Language Programming

Author: *Henry G. Baker, http://home.pipeline.com/~hbaker1/home.html; hbaker1@pipeline.com*

Henry G. Baker[1]
Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139

Structured programming advocates the elimination of the GOTO from the set of primitives for a computer language. Yet at the machine language level, almost all machines offer only the conditional or unconditional branch as the basic control primitive. Vaughan Pratt has suggested using instead pseudo-non-deterministic programming (as well as the COMFY name). We present a new set of machine language control primitives based on Pratt's suggestion which are simpler to implement than if-then-else and do-while and are more flexible at the same time. The machine language implementation of these primitives requires nothing more complicated than a stack; the size of the stack needed (exclusive of programmer-defined subroutine control points) is $O(d)$, where d is the operator depth of the program. This is much less space and machine complexity than would be required for a true backtracking control structure. The user can emulate if-then-else, do-while, repeat-until, etc., with macros using these primitives if he so desires.

A machine language program using our primitives is built up from individual *actions* and *tests* which are combined into *sequences*, *alternatives*, and *loops*. Actions are primitive instructions executed for their side-effects such as "`add x to accumulator 3`", "`move a to b`", or "`increment z`". Tests are primitive instructions executed to produce a boolean value; e.g., "`x>y`", "`a=0`", or "`overflow?`". Tests are distinguished from actions in that they can either *succeed* or *fail*, while actions can only *succeed*. The convention is that if the predicate value is true then the predicate succeeds, and otherwise it fails.

The semantics of success and failure depend entirely upon the semantics of sequences, alternatives, and loops. A *sequence* formed from the two program fragments A and B, written "`A;B`", is executed by first executing A and if it succeeds then executing B. If A fails, then B is not executed and the whole sequence fails. Likewise, if A succeeds and B fails, the whole sequence fails. COMFY is a poor man's non-deterministic programming language, because *any side effects produced by a program fragment before failing are not undone in the course of failing*. Since "`;`" is obviously an associative operator, *long sequences* can be written without parentheses and still be unambiguously interpreted; e.g., "`A;B;C;D`". A long sequence will succeed only if every fragment succeeds; it will fail on the leftmost failing fragment.

We will abbreviate long sequences of the form `A;A;...;A` (m A's) as A^m, where m is a positive integer. We also define $\mathsf{A}^0 \equiv$ *succeed* for completeness. This notation is well-defined due to the associativity of sequences.

Alternatives

An *alternative* consists of two program fragments A and B linked by the alternative operator "`|`" as in "`A|B`". The semantics of the alternative are dual to those of the sequence. First A is executed. If it succeeds, then the whole alternative succeeds immediately without executing B. If A fails, then B is executed and its success or failure becomes the success or failure of the whole alternative expression. Like the sequence operator, the alternative operator is associative, allowing long chains of alternatives to be written unambiguously without parentheses. We notice that for a chain of alternatives to succeed, at least one must succeed and only the leftmost successful alternative

[1] This previously unpublished note was written June 29, 1976, while the author was a graduate student at M.I.T. It has not been modified except for light editing. This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661.

will be executed. Furthermore, the chain will fail if and only if all of the alternatives fail.

Loops

With only sequences and alternatives, only finite (tree-like) programs can be written. The loop expression allows us to write programs whose execution length is not bounded. A *loop* consists of a program fragment A followed by the operator ∞, as in A^{∞}. The semantics of this loop expression are equivalent to "$(A; A; ...)$", where "..." means "and so on, to infinity." In other words, A is executed over and over again until it fails; then the whole loop fails. If A never fails, then the loop expression will continue executing forever; it will never *succeed*.

Inversion

Inversion is a monadic operator which takes an expression, executes it, and inverts the meaning of its success and/or failure. Inversion of an expression A is written as "¬A". Inversion can not be simulated by sequences, alternatives, *succeed* and *fail*, in an uninterpreted (pure) COMFY because both sequence and alternative are monotonic operations, composition preserves monotonicity, and inversion is not monotone.

Subroutine Calling

COMFY allows the definition of subroutines with mnemonic names. The existence of a stack for saving return points is assumed; hence subroutines can be recursive. Like most machine languages, COMFY avoids arguments by avoiding parameters; arguments are the programmer's responsibility. Likewise, returned values are the programmer's responsibility.[2]

In a machine implementation using only one stack, stack (LIFO) discipline must be maintained by both subroutine call and return as well as by the sequence, alternative, loop, and inversion primitives. Therefore, a return from a subroutine cannot be made from other than parenthesis level zero; this is tantamount to running off the end of the code of a sequence which implicitly succeeds or running off the end of an alternative which implicitly fails.

(In this context, we point out that COMFY would make a good control structure for APL, which currently has none. The issues COMFY addresses are all orthogonal to those that APL addresses, so the two might make a good marriage. The infix notation and shallow dynamic variable binding of APL make a good interactive language. COMFY would enhance that ease-of-use-interaction feature.)

Syntax

We will assume from now on that the four operators "|", ";", "¬", and "∞" have binding power (precedence) which increases from left to right in the preceding list.

With these four constructs and the two trivial tests *succeed* and *fail*, we can simulate many other useful constructs. Some of the simulations follow.

If-then-else

Before trying to simulate the construct "`if B then A else C`" in COMFY, we must define what it means. The intuitive meaning would be: execute B; if it succeeds, execute A and its success or failure becomes the success or failure of the whole expression; if B fails, then execute C and its success or failure becomes that of the whole expression. These semantics are analogous to the meaning of the Algol "`if B then A else C`" in the case that A and C are boolean-valued expressions.

But if-then-else in general cannot be simulated exactly with ";", "|", and "¬"! We mean by exact simulation one which would place no restrictions on the expressions B, A, and C. However, in the important special case that B *has no side-effects and is not affected by* A's *side-effects*, "`if B then A else C`" can be simulated by "$B; A|\neg B; C$", because "¬B" protects C from executing if A fails. This can be made a bit more readable by defining "$\rightarrow$" to be equivalent to ";": "$B \rightarrow A|\neg B \rightarrow C$". Finally, *if* A *cannot fail*, then the simulation optimizes to "$B \rightarrow A|C$", which is exactly BCPL syntax for conditionals.[3]

Do-while

The semantics for the construct "`while B do E`" are not usually defined for the case where E can fail; therefore we are free to define them as we please. We emulate this expression in COMFY by $\neg(B; E)^{\infty}$ which either loops forever or fails. It can fail either by B failing

[2] A "condition code" is a convenient place to return the boolean value from a subroutine which implements a predicate/test.

[3] In our implementation of COMFY, if-then-else is primitive, so it does not have to repeat the evaluation of B, and thereby avoids the limitations that that redundant evaluation entails.

or E failing. If the programmer does not wish to terminate the loop upon failure by E, he should instead use $\neg(\mathsf{B};(\mathsf{E}|\mathit{succeed}))^{\infty}$. Using "$\rightarrow$" for "`;`", the while construct becomes $\neg(\mathsf{B}\rightarrow\mathsf{E})^{\infty}$.

Repeat-until

The construct "`repeat E until B`" can be written in our language $\neg(\mathsf{E};\neg\mathsf{B})^{\infty}$. This works because of our convention that E is executed before B, and any side-effects are not nullified if B fails.

Extended if-then-else; LISP's `COND`

The extended if-then-else expression, equivalent to LISP's `(COND (`B_1 E_1`)(`B_2 E_2`)...(`B_n E_n`))` can often be emulated as $\mathsf{B}_1;\mathsf{E}_1|\mathsf{B}_2;\mathsf{E}_2|...|\mathsf{B}_n;\mathsf{E}_n$ or more readably as $\mathsf{B}_1\rightarrow\mathsf{E}_1|\mathsf{B}_2\rightarrow\mathsf{E}_2|...|\mathsf{B}_n\rightarrow\mathsf{E}_n$.[4]

Case

Unfortunately, this structured programming primitive resists attempts to simulate it straight-forwardly in terms of the other primitives. Therefore, it must remain primitive in order to gain efficiency during execution.

Backtracking

Although COMFY does not support true nondeterministic programming, backtracking can be done explicitly within it. If, for example, in the progam A|B we would like to undo any side-effects caused by A before failing, we could insert another alternative A^{-1}, which is charged with undoing A, i.e. $(\mathsf{A}|\mathsf{A}^{-1};\mathit{fail}|\mathsf{B})$. Thus, when A fails, A^{-1} is executed to undo A's side-effects and then fails again so B can be executed. (A^{-1} is only a mnemonic device; this -1 superscript is not a COMFY operator.) If A can fail at more than one point, it may be more convenient to undo many side-effects at once, through saving and then restoring some state information. If "S" is a save sequence, "E" is a program which can fail, "`I`" is an ignore-saved-information sequence, and "R" is a true restore sequence, then "`S;E;I|R;`*fail*" will have no side-effects if E fails and will have E's side-effects if E succeeds.

[4] Of course, for this emulation the E_i must not fail; this limitation is removed in the implementation by expressing extended if-then-else in terms of the primitive if-then-else.

Programming Examples

We give a few examples to show how the notation is used in real programming. The first program is trivial; it prints the integers from 1 to 10.

In ALGOL-like notation:

```
x:=1;
while x<11 do
 begin
  print x;
  x:=x+1
 end;
```

In COMFY notation:

`x:=1;` $\neg$`(x<11` $\rightarrow$ `print x; x:=x+1)`$^{\infty}$

The second program is a subroutine that computes the greatest common divisor of positive X and positive Y and leaves the result in Y.

```
GCD: while X>0 do
       if X>Y then X:=:Y else Y:=Y-X;
```

In COMFY notation:

`GCD:` $\neg$`(X>0` $\rightarrow$ `(X>Y` $\rightarrow$ `X:=:Y | Y:=Y-X))`$^{\infty}$

The third program scans a floating point number in FORTRAN format from an input stream. "`S'string'`" tests if `'string'` is next in the input stream. If not, it fails. If so, it succeeds and moves the buffer pointer past `'string'`. The program is simply an edited form of the BNF syntax for these numbers where ϵ is *succeed* and $\emptyset$ is *fail*! We will not attempt to give an equivalent Algol-like program for this example.

```
NUMBER: INTEGER;(S'.';DIGSEQ);(S'E';INTEGER)

INTEGER: (S'+'|S'-'|succeed);DIGSEQ

DIGSEQ: ¬(S'0'|S'1'|S'2'|S'3'|S'4'|S'5'|S'6'|S'7'|S'8'|S'9')^∞
```

Algebraic properties of sequences, alternatives, loops, and inversions

Sequences, alternatives, and inversions have a remarkable similarity to the boolean operators *and*, *or*, and *not*. In fact, for any boolean expression used for program control purposes, such as loop or conditional control, the analogy is exact. Thus, "`if A`$\wedge$`B then C else D`" can often be simulated by "`(A;B);C|D`" (the parentheses

are used only for emphasis; they are not required here), "`while A∨B do C`" is simulated by "$\neg((\mathsf{A}|\mathsf{B});\mathsf{C})^{\infty}$" (here the parentheses are required). Again, we can make these simulations much more readable by writing "∧" for "`;`" and "∨" for "`|`"; the above program fragments become "$\mathsf{A} \wedge \mathsf{B} \rightarrow \mathsf{C}|\mathsf{D}$" and "$\neg(\mathsf{A} \vee \mathsf{B} \rightarrow \mathsf{C})^{\infty}$" (assuming "∨" is the same as "`|`" except for syntactic binding power).

The looping operator "∞" reminds one of the "$\forall x$" operator of the first order predicate calculus; it is a "while" looping operator because success keeps it looping. We can define a dual operator "*" such that $\mathsf{A}^* \equiv (\mathsf{A}|\mathsf{A}|...)$. This is an "until" looping operator because success stops it from looping.

We first list properties which are true, followed by some properties which are not true in general, but are true for interesting special cases.

1. $(\mathsf{A};\mathsf{B});\mathsf{C} \equiv \mathsf{A};(\mathsf{B};\mathsf{C})$. Also, $(\mathsf{A}|\mathsf{B})|\mathsf{C} \equiv \mathsf{A}|(\mathsf{B}|\mathsf{C})$. (Associativity).

2. A; *succeed* $\equiv$ *succeed*; $\mathsf{A} \equiv \mathsf{A} \equiv \mathsf{A}|$*fail* $\equiv$ *fail*$|\mathsf{A}$. (Identities).

3. *succeed*$|\mathsf{A} \equiv$ *succeed*. *fail*; $\mathsf{A} \equiv$ *fail*. (Nil-potence).

4. $\mathsf{A}^m;\mathsf{A}^n \equiv \mathsf{A}^{m+n}$. $(\mathsf{A}^m)^n \equiv \mathsf{A}^{mn}$. (Laws of exponents).

5. $\neg\neg\mathsf{A} \equiv \mathsf{A}$. (Two-valued logic).

6. $\neg(\mathsf{A}|\mathsf{B}) \equiv \neg\mathsf{A};\neg\mathsf{B}$. Also, $\neg(\mathsf{A};\mathsf{B}) \equiv \neg\mathsf{A}|\neg\mathsf{B}$. (DeMorgan's Laws).

7. $(\mathsf{A}^{\infty})^{\infty} \equiv \mathsf{A}^{\infty}$. $\mathsf{A};\mathsf{A}^{\infty} \equiv \mathsf{A}^{\infty}$. $\mathsf{A}^{\infty};\mathsf{A}^{\infty} \equiv \mathsf{A}^{\infty}$. In fact, $\mathsf{A}^{\infty};\mathsf{B} \equiv \mathsf{A}^{\infty}$, for any `B`! (Closure).

8. $\neg\mathsf{A}^{\infty} \equiv (\neg\mathsf{A})^*$. Also $\neg\mathsf{A}^* \equiv (\neg\mathsf{A})^{\infty}$. (Generalized DeMorgan laws).

-1. "`;`" is neither idempotent nor commutative, in general.

-2. "`|`" is neither idempotent nor commutative, in general. However, a true non-determinism would make "`|`" both.

-3. "`;`" distributes over "`|`" only sometimes. "`|`" almost never distributes over "`;`".

-4. $\mathsf{A}|$*succeed* is not the same as *succeed*. Also, A; *fail* is not the same as *fail*.

Redundancy and Completeness

DeMorgan's laws tell us that alternatives can be simulated by sequence and inversion; similarly, sequence can be simulated by alternative and inversion. Therefore, at the machine language level, we need only implement either sequences or alternatives, and can simulate the other. However the sets {sequence, inversion, loop} and {alternative, inversion, loop} are both independent, meaning that neither set can be reduced without reducing the power of the language.

Sequences, inversion, and loops are not complete because there are at least two constructs which have non-optimum COMFY simulations. One is the "case" statement or "computed GOTO" which allows a quick selection from among a host of alternatives. The other is the multiple-exit loop—a loop with several exit conditions, each going to a different place. The semantics can be simulated, but only at the cost of repeating the exit tests after loop termination to determine what to do next. For example, a loop with body `B` and termination conditions `C` and `D` might want to goto `E` and `F`, respectively, upon termination. This could be simulated by $(\mathsf{C} \vee \mathsf{D}|\neg\mathsf{B})^*;(\mathsf{C} \rightarrow \mathsf{E}|\mathsf{D} \rightarrow \mathsf{F})$. Actually, in this case, repeating the condition for how control got to that place in the program might be a good idea for readability and ease in understanding a program.

Sources

Although the immediate source for COMFY was Vaughan Pratt [Pratt76], there are some striking similarities between it and a series of syntax-directed compilers, all called META, started by Schorre [Schorre64] and continued by Schneider and Johnson [Schneider64]. These META compilers were recursive parsers for deterministic context-free languages constructed by converting each BNF syntax equation of a suitable grammar into a recursive subroutine. Non-determinism was avoided by proper choice of the equations and clever factoring (property -3, above). Left recursive syntax equations were rewritten as loops to avoid stack overflow.

Schorre's META II had sequences specified by juxtaposition), alternatives specified by "`/`", loops specified by `$E` (meaning $\neg\mathsf{E}^{\infty}$ in COMFY), and recursion without parameters. Tests and actions were extremely rudimentary; the only test besides *succeed* (called `.EMPTY`) was the comparison of a character string with an input buffer for a match and update the buffer pointer if successful. The only action was `.OUTPUT(...)`, which placed information into an output buffer.

Schneider and Johnson's META-3 added tests which did not reference or affect an input stream and expanded the class of actions to include any imperative operation

which did not affect the succeed/fail flag. Thus, it was the first META which could be called a programming language.

The only significant difference between META's and COMFY's semantics seems to be in the treatment of sequences. META's sequences were required to start with a test; a failure at any other than the first position caused an error stop (this generalization is due to Pratt). Presumably, this restriction was to keep the semantics of the language consistent with that of a true non-deterministic (backtracking) implementation. However, the addition of unrestricted tests and actions in META-3 made that consistency impossible.

The duality of sequence and alternative noticed by Pratt does not appear to have been known by the META designers. Furthermore, the inversion operator which makes this duality explicit was not used by either META or Pratt; it is new. Finally, the current definition of loop in COMFY, which for completeness requires an inversion operator in the language, is also new.

In a future column, we will show how to program the COMFY compiler.

References

[Pratt76] Pratt, Vaughan. Personal communication, 1976.

[Schorre64] Schorre, D.V. "META II—A Syntax-oriented Compiler Writing Language." *Proc. 19th ACM Nat'l. Conf.*, Aug. 1964.

[Schneider64] Schneider, F.W., and Johnson, G.D. "META 3—A Syntax-Directed Compiler Writing Compiler to Generate Efficient Code." *Proc. 19th ACM Nat'l. Conf.*, Aug. 1964.

Henry Baker has been diddling bits for 35 years, with time off for good behavior at MIT and Symbolics. In his spare time, he collects garbage and tilts at windbags. *This column appeared in ACM Sigplan Notices 32,6 (Jun 1997), 23-27.*