

Prospective values and Forth

Bill Stoddart, Frank Zeyda

October 12, 2023

Abstract

We use $S \diamond E$ to represent the value expression E would have were it to be evaluated after the execution of program S . We call this the prospective value of E after S . This form is expressive enough to describe the semantics of an extended form of sequential programming language that incorporates backtracking and speculative computations. Here we try it out with Forth.

1 Introduction

We write $S \diamond E$ for the value expression E would have were it to be evaluated after the execution of program S . We call this the prospective value of E after S . In this paper, where we apply this idea to Forth, S is a Forth program, i.e. some self contained Forth code, and E is a mathematical expression.

For example $x + 1$ to $x \diamond 10 * x = 10 * (x + 1)$

Note the large equals $=$ is a very low priority equals symbol. The symbol \diamond is next lowest in priority.

We have developed the theory of prospective value semantics (PV semantics) in a series of papers, most recently in [DFM⁺23], and shown that it provides a sufficient formalism for describing backtracking and reversible computations. Our theory is developed as an extension of the B-Method. [Abr96]. We have developed a reversible Forth [SZL10] to act as an implementation platform, and developed some compilation techniques that make special use of Forth's essential features, for example executing type tagged parse trees as Forth programs [RS10]. One motivation for providing a PV semantics for Forth itself would be to provide a means of checking the validity of the code produced by such a compiler.

When constructing a PV semantics for Forth we have to take into account the following:

- The way Forth expressions are written, in an extended postscript notation with explicit stack manipulations, is so different from how expressions are written in mathematics that we will have to abandon the convenient and

unspoken fiction that program expressions and mathematical expressions are one and the same.

- Forth has an explicit stack so we need a way to represent the stack as a mathematical expression,

2 The stack, part 1

Our semantics uses the typed set theory of B. A stack may hold items of different type, and this prevents us from representing it as a sequence. However, we can represent a stack containing different types of value as a tuple ¹.

We use the symbol ε to represent the empty parameter stack, and in our mathematical universe we give the parameter stack the name s .

Here are some examples showing the value taken by the stack following some simple Forth code.

```
SP!  $\diamond s = \varepsilon$ 
SP! 1  $\diamond s = \varepsilon \mapsto 1$ 
SP! 1 2  $\diamond s = \varepsilon \mapsto 1 \mapsto 2$ 
SP! 1 2 10  $\diamond s = \varepsilon \mapsto 1 \mapsto 2 \mapsto 10$ 
SP! 1 2 10 +  $\diamond s = \varepsilon \mapsto 1 \mapsto 12$ 
```

Since the stack always consists of a tuple that commences with ε we can take the liberty of omitting the ε when the stack is non-empty and replacing the maplet symbol \mapsto by a space. This allows the above results to be expressed as follows.

```
SP!  $\diamond s = \varepsilon$ 
SP! 1  $\diamond s = 1$ 
SP! 1 2  $\diamond s = 1 2$ 
SP! 1 2 10  $\diamond s = 1 2 10$ 
SP! 1 2 10 +  $\diamond s = 1 12$ 
```

3 Expressions and the semantics of assignment

Let E be Forth code that's only effect is to leave one item on the stack, i.e. it causes no change of state of program variables or any memory; we will call such a fragment of code an “expression”. We will want to use the value left by E in some of our semantic equations.

In the form $S \diamond E$, The text to the left of the diamond is Forth code, and that to the right is a mathematical expression, i.e. the diamond separates Forth code from mathematical text, and we need a notation to translate the Forth expression E into the mathematical world. We enclose E in semantic brackets $\llbracket E \rrbracket$ to represent this translation. Some examples will make this clear.

¹This means the type of the stack will change every time we push or pop a value. Thus the stack has no identifiable type, but every state of the stack does have a type.

Suppose x is a Forth VALUE holding an integer. We translate x into the mathematical value x

$$\llbracket x \rrbracket = x$$

Next consider the translation of a simple expression:

$$\llbracket x \ 10 \ + \rrbracket = x + 10$$

In the next example we use a symbolic stack trace to perform the translation $\llbracket x \ \text{DUP} \ \text{DUP} \ * \ + \rrbracket = x^2 + x$

Forth commands	Stack
x	x
$\text{DUP} \ \text{DUP}$	$x \ x \ x$
$*$	$x \ x^2$
$+$	$x^2 + x$

Here is an symbolic trace for an example where three arguments $a \ b \ c$ are provided from the stack:

$$\llbracket a \ b \ c \ -- \ 2\text{DUP} \ * \ -\text{ROT} \ + \ \text{SWAP} \ -\text{ROT} \ * \ + \ 2* \rrbracket$$

Forth commands	Stack
$a \ b \ c \ -- \ 2\text{DUP}$	$a \ b \ c \ b \ c$
$*$	$a \ b \ c \ b*c$
$-\text{ROT}$	$a \ b*c \ b \ c$
$+$	$a \ b*c \ b+c$
SWAP	$a \ b+c \ b*c$
$-\text{ROT}$	$b*c \ a \ b+c$
$*$	$b*c \ a*(b+c)$
$+$	$b*c + a*(b+c) = a*b + a*c + b*c$
$2*$	$2*(a*b + a*c + b*c) = 2*ab + 2*ac + 2*bc$

4 Assignment

To avoid continual use of the semantic brackets $\llbracket \cdot \rrbracket$ we will use a change in typeface, by which the Forth expression E is translated as the mathematical expression E .

The semantics of changing variable states is expressed in lambda notation. $(\lambda x \bullet F)E$ represents the rewriting of F with with the term E substituted for each occurrence of x in F . For example $(\lambda x \bullet 2 * x)(y + 10) = 2 * (y + 10)$.

In Forth, and with E an expression as defined above (i.e. Forth code that leaves a value on the parameter stack and caused no other change of state) $E \ \text{to} \ x$ represent the assignment of the value left by E to the Forth VALUE x . We can give its semantics by describing its effect on a general expression F :

$$E \ \text{to} \ x \ \diamond \ F = (\lambda x \bullet F)E$$

For example:

$x \ 10 +$ to $x \diamond 2 * x + y =$ by rule for assignment
 $(\lambda x. 2 * x + y) \llbracket x \ 10 + \rrbracket =$ by semantics of expression
 $(\lambda x. 2 * x + y)(x + 10) =$ by lambda evaluation
 $2 * (x + 10) + y$

5 The stack, part 2

We represent the stack mathematically as a tuples, so let us review tuple notation. We write $x \mapsto y$ for the tuple consisting of the pair of values x and y , $x \mapsto y \mapsto z$ for the tuple consisting of the triple of values x , y , and z . The tuple operator is a left associative binary operator, so $x \mapsto y \mapsto z = (x \mapsto y) \mapsto z$.

We can decompose a tuple into its first and second components with the functions L (left) and R (right).

In line with Forth usage in referring to the top and next from top elements of the stack we define the following functions. $top(s) \hat{=} R(s)$
 $next(s) \hat{=} R(L(s))$

Unlike an assignment to a VALUE, e.g. 3 to X, which changes the whole of X, stack operations may affect only part of s . So our approach will be to use “helper functions” to describe the whole new stack, and assign this whole new state.

For following are examples of these helper functions:

$drop(s) \hat{=} L(s)$
 $twodrop(s) \hat{=} L^2(s)$
 $nip(s) \hat{=} L^2(s) \mapsto R(s)$
 $swap(s) \hat{=} L^2(s) \mapsto top(s) \mapsto next(s)$
 $plus(s) \hat{=} L^2(s) \mapsto (next(s) + top(s))$
 $minus(s) \hat{=} L^2(s) \mapsto (next(s) - top(s))$

and so on

Then to describe the value of expression E after a stack operation OP we have

$$OP \diamond E = (\lambda s. E)op(s)$$

Where E is a stack expression, such as $L(s)$, or just s . An example in the next section should help to make this clear.

6 Sequential Composition

Our semantic rule for sequential composition is:

$$\text{sequential composition} \quad S \ T \diamond E = S \diamond T \diamond E$$

Note that \diamond is right associative, so:

$$S \diamond T \diamond E = S \diamond (T \diamond E)$$

We can use this rule to show that the effect of NIP on the stack s is equivalent to that of SWAP DROP.

SWAP DROP $\diamond s =$ by rule for sequential composition
 SWAP \diamond DROP $\diamond s =$ by semantics of DROP
 SWAP $\diamond (\lambda s \bullet s)drop(s) =$ by lambda evaluation
 SWAP $\diamond drop(s) =$ by semantics of SWAP
 $(\lambda s \bullet drop(s))swap(s) =$ by lambda evaluation
 $drop(swap(s)) =$ applying *swap*
 $drop(L^2(s) \mapsto top(s) \mapsto next(s)) =$ applying *drop*
 $L^2(s) \mapsto top(s) =$ semantics of NIP
 NIP $\diamond s$

It seems strange that we compute the effect of SWAP DROP on the stack by first computing the effect of DROP and *then* computing the effect of SWAP, but intermediate result step $drop(swap(s))$ shows the helper function for SWAP is applied before that of DROP in obtaining the result.

7 Guard, choice and backtracking

Let g be a condition test that leaves either a true flag or a false flag on the stack and has no other side effect. The construct $g \rightarrow$ is a guarded no-op. If g leaves a true flag, the guarded no-op removes the flag and execution continues ahead. If g leaves a false flag, its effect is to reverse computation. In this case there is no state after g . Mathematically, we represent this as *null*, where *null* represents nothing. In our mathematical semantics we capture the idea of nothing by using Eric Hehner's Bunch Theory [Heh93]; this is a reformulation of set theory in which the collection and packaging of elements are orthogonal activities. This gives us access to unpackaged collections. We use $\sim S$ to represent the unpacking of set S . For example $\sim\{1, 2\} = 1, 2$ where $1, 2$ is an unpackaged collection. The comma in $1, 2$ is now a mathematical operator, known as bunch union. We obtain *null* by unpacking the empty set.

$$null = \sim\{ \}$$


Bunch union has the properties: $S, T = T, S$ and $S, null = S$, and an additional property of *null* is $\{ null \} = \{ \}$.

Corresponding to the programming guard \rightarrow , we have a bunch guard \rightarrow in our mathematical notation, defined by the following equations:

$$true \rightarrow E = E, \quad false \rightarrow E = null$$

so the expression $x = 1 \rightarrow x$ has the value 1 if $x=1$, and is equal to *null* for any other value of x .

Admittedly, this is pretty weird till you get used to it, and these concepts are not widely known. When we asked chatGPT about "nothing" we got the following response:

 Has the concept of "nothing" been formulated mathematically?



An error occurred. If this issue persists please contact us through our help center at help.openai.com.

Our semantic rule for guard is

$$g \rightarrow \diamond E = g \rightarrow E$$

here g is the mathematical translation of the Forth guard g , which for any specific g we can represent more fully using our semantic brackets, e.g.

$$\llbracket x = 1 \rrbracket = x = 1$$

Guards combined with choice can describe control structures, including backtracking.

We introduce a Forth choice operation. $S_1 \parallel S_2$ presents a choice between executing S_1 or S_2 . This choice has to be bracketed, rather like an IF construct, as

$\langle \text{CHOICE } S_1 \parallel S_2 \parallel \dots \text{ CHOICE} \rangle$

The semantic rule for choice is:

$$S \parallel T \diamond E = (S \diamond E), (T \diamond E)$$

Here the comma on the RHS is the bunch union operator that we defined above. the rule does not say which choice is tried first.

For example:

$$\langle \text{CHOICE } 1 \text{ to } x \parallel 2 \text{ to } x \text{ CHOICE} \rangle \diamond x = 1, 2$$

The combination of choice and guard allows us to express backtracking. Consider:

$$\langle \text{CHOICE } 1 \text{ to } x \parallel 2 \text{ to } x \text{ CHOICE} \rangle x = 2 \Rightarrow \diamond x$$

This has the following operational interpretation: a choice is made to assign either 1 or 2 to x , then a guard checks if $x=2$, and if not forces backtracking. Computation returns to the previous choice and continues ahead once more with the unused choice being selected. this time the guard lets computation continue ahead, with $x=2$. This simple example shows how we can use a guard to retrospectively select from two choices.

The semantic analysis goes as follows:

$$\langle \text{CHOICE } 1 \text{ to } x \parallel 2 \text{ to } x \text{ CHOICE} \rangle x = 2 \Rightarrow \diamond x = \text{ by semantics of sequential composition}$$

$$\langle \text{CHOICE } 1 \text{ to } x \parallel 2 \text{ to } x \text{ CHOICE} \rangle \diamond x = 2 \Rightarrow \diamond x = \text{ by semantics of program guard}$$

$\langle \text{CHOICE } 1 \text{ to } x \parallel 2 \text{ to } x \text{ CHOICE} \rangle \diamond x = 2 \rightarrow x =$ by semantics of choice
 $1 \text{ to } x \diamond x = 2 \rightarrow x, 2 \text{ to } x \diamond x = 2 \rightarrow x =$ by semantics of assignment
 $1 = 2 \rightarrow 1, 2 = 2 \rightarrow 2 =$ by property of bunch guard
 $\text{null}, 2 =$ by property of null
 2.

7.1 Conditionals

We can think of $\text{IF } S \text{ ELSE } T \text{ THEN } \diamond E$ as a bunch union of two terms, corresponding to the two branches of the conditional, and with the term corresponding to the branch not taken being equal to *null*. In our semantics this is expressed as follows:

$$\text{IF } S \text{ ELSE } T \text{ THEN } \diamond E = (\text{top}(s) \neq 0 \rightarrow S \diamond E), (\text{top}(s) = 0 \rightarrow T \diamond E)$$

8 Speculative computation

In our semantics $S \diamond E$ expresses the value E would take after executing S . We can use the same semantics to describe a speculative computation which executes S , evaluates and saves the result of E , then reverses, restoring any changes made in the forward execution of S . Thus we obtain, in our program, the value E would have after S but without incurring any of the side effects produced by executing S .

As with choice we needs brackets to express this:

$\langle \text{RUN } S \text{ E RUN} \rangle$

is a programming structure which adds to the stack the value E produces if executed after S , but without incurring the side effects that execution of S may produce. Its semantics is:

$$\langle \text{RUN } S \text{ E RUN} \rangle \diamond s = s \mapsto (S \diamond E)$$

If S contains choices there may be a plurality of values that E could take, and we can collect. If these are integer values the construct to do this is:

$\text{INT } \{ \langle \text{RUN } S \text{ E RUN} \rangle \}$

In this case $S \diamond E$ will be a bunch, and we have the following semantic rule:

$$\text{INT } \{ \langle \text{RUN } S \text{ E RUN} \rangle \} \diamond s = s \mapsto \{ S \diamond E \}$$

8.1 Example, Pythagorean triples, with a new concept of function application

We need to introduce some additional aspects of the RVM sets package.

The mathematical notation $m..n$ where $n \geq m$, represents the set of numbers $\{m, m + 1, \dots, n\}$. We provide this as a postfix operator in RVM Forth, used as, e.g.

```
1 4 .. .SET <cr> {1,2,3,4} ok
```

We have CHOICE from a set, used as in the following example. CHOICE makes a provisional choice from a set that may be revised by backtracking.

```
INT { <RUN 1 4 .. CHOICE 10 * RUN> } .SET <cr> {10,20,30,40} ok
```

We now consider a program to produce a set of Pythagorean triples $\{a, b, c\}$ where $a^2 + b^2 = c^2$. In the code we choose values for a and b , calculate $a^2 + b^2$ and then apply a perfect square root function PERF. This function illustrates the "new concept of function application" we mentioned above. The idea of n PERF is that it returns the perfect square root of n , if that exists, or otherwise triggers backtracking. In the following examples we see that if backtracking continues back to the user console, we get the prompt `ko` rather than `ok`.

```
0 PERF .<cr> 0 ok
1 PERF .<cr> 1 ok
2 PERF .<cr> ko
3 PERF .<cr> ko
4 PERF .<cr> 4 ok
```

This may seem a programming trick - we have just included the guard that triggers backtracking within the code for PERF. However, we have *mathematical* reason to claim that this is indeed a new idea of function application. Working with integers and using \sqrt{n} to represent the perfect integer square root of n , it is clear for example that no integer satisfies $\sqrt{2}$, and we capture this in our theory by saying $\sqrt{2} = \text{null}$. We also recall that from the semantics of guards, it is a *null* result that triggers backtracking. The new concept of function application is that a function application might represent "nothing", which we cannot express without the *null* of bunch theory. To express the stack effect of PERF we need to specify that if the stack input parameter n has an integer square root m then that will be the stack output parameter, otherwise *there will be no stack after state*. To do this we use *null*, as follows.

```
PERF ( n -- if  $\exists m \bullet m^2 = n$  then m else null end )
```

Now for the program to produce set of Pythagorean triples with perpendicular sides less than n . The set we are producing here is a set of sets of numbers, and its mathematical type is $\mathbb{P}(\mathbb{N})$. This is represented in our Forth sets package, in postfix, by the signature INT POW.

```
: TRIPLES ( n -- s, s is a set of Pythagorean triples with adjacent sides  $\leq n$  )
  (: n :)
  INT POW {
    <RUN
      1 n .. CHOICE to A
      A n .. CHOICE to B
      A B COPRIME =>
```



```

      A DUP * B DUP * + PERF to C
      INT { A , B , C , }
    RUN>
  } ;

```

In this code, A, B and C are global VALUES. The COPRIME guard prevents similar triangles being included, for example {3,4,5} and {6,8,10}.

Here is an example run

```

100 TRIPLES .SET <cr> {{3,4,5},{5,12,13},{7,24,25},{8,15,17},{9,40,41},
{11,60,61},{12,35,37},{13,84,85},{16,63,65},{20,21,29},{20,99,101},{28,45,53},
{33,56,65},{36,77,85},{39,80,89},{48,55,73},{60,91,109},{65,72,97}}ok

```

9 Preconditions

In general, in the field of formal semantics, operations are taken to have specific conditions which render them safe for use. These “preconditions” are there to protect us attempting to access the 20th element of a 10 element array, taking the square root of a negative number, dividing by zero etc. Unlike a guard, a pre-condition does not control whether an operation can take place, rather it is part of the instructions of using the operation. In Forth the situation with respect to pre-conditions is complex, because the programmer takes responsibility for an operation being meaningful in a particular context. For example, in 32 bit arithmetic, 7FFF 1 + violates the a precondition of + if we are using signed arithmetic, but not for unsigned arithmetic. However, one universal precondition of + is that it requires at least two elements to be on the stack.

We use the symbol \perp to express the effect of violating a precondition. The idea is that \perp represents absolute unpredictability - more unpredictable than just allowing any possible result - there might be no result because the computation does not terminate, or the machine might blow up!

We use $P \mid S$ to represent P as the pre-condition for S. Our rule for preconditions is:

$$P \mid S \diamond E = (P \rightarrow S \diamond E), (\neg P \rightarrow \perp)$$

We interpret \perp as a maximally non-deterministic bunch. We can think of the unpackaged collections of bunch theory as representing nondeterminism or uncertainty, e.g. the bunch 1,2 representing a value that might be 1 or might be 2. In this knowledge based order the value \perp represents a value about which nothing can be known, not even whether it exists, and *null* can be taken as the object about which too much is known, to the point of contradicting its possible existence. It is at the other end of the scale from \perp .

10 Loops

We consider the treatment of a WHILE loop

```
BEGIN g WHILE S REPEAT
```

Here g is some code which leaves a flag in the stack and otherwise leaves the program state unchanged.

Following the B-Method (and adapting it to Forth) the programmer is required to provide formal comments which identify an invariant expression I and a variant expression V for the loop.

The invariant expression must have the property

$$S \diamond E = E$$

When the loop terminates the invariant expression will still have the same value, but the condition reported by g is false. This allows us to draw a conclusion about the effect of the loop.

The variant expression serves the purpose of ensuring that the loop *does* terminate. It has to be an expression that is greater than 0 and decreased by S . Obviously this cannot continue for ever, so the existence of such an expression implies that the loop must terminate. Its formal property is:

$$V > 0 \wedge S \diamond V < V$$

We illustrate this method using Euclid's algorithm for the calculation of the greatest common divisor of two numbers.

```
: GCD ( a b - c, a>0 ^ b>0 | c = gcd(a,b) )
  BEGIN (
    INVARIANT gcd(top(s), next(s))
    VARIANT top(s) + next(s) )
    2DUP ≠
    WHILE
      2DUP > IF SWAP THEN
      OVER -
    REPEAT DROP ;
```

First note that we have a pre-condition that requires $a > 0$ and $b > 0$. Since a , b are names for the top two stack elements, this ensures that our variant property $V > 0$ holds. Depending on the branch taken by the IF, we have $S \diamond V = top(s)$ or $S \diamond V = next(s)$, and since $V = top(s) + next(s)$ in both cases we have $S \diamond V < V$. So the variant properties are satisfied and we can be sure the loop terminates.

That the loop invariant holds follows from the mathematical property $y > x \Rightarrow gcd(x, y) = gcd(x, y - x)$. When the loop terminates the loop condition tells us that $next(s) = top(s)$ and the loop invariant tells us $gcd(top(s), next(s)) = gcd(a, b)$. Thus we have two copies of the required result on the stack and just have to drop one of them to complete the computation.

11 Conclusions

When transporting prospective value semantics from our usual B like environment to Forth, the extended postfix used in Forth forces us to distinguish more clearly between programming and mathematical notations. Forth has a finer grained semantics, where an expression is defined as a sequence of operations, rather than in the mathematical notation of an expression sub-language. This additional detail can be captured in two ways by the semantics we investigate here. Either we can translate postfix expressions to infix in order to describe their effect (and this might require us to write our Forth in a particular way, and might be particularly useful in analysing the output of a compiler for our backtracking language bGSL [DFM⁺23]), or we can process them at the level of the individual Forth operations of which they are comprised. In both cases we can include the effect of stack manipulations in our analysis.

The mathematical expression of nothing as the constant *null* plays a key role in our semantics. We also illustrate a new form of function application, in which a mathematical function application can yield *null* to indicate that the described object does not exist, with the matching computational interpretation being that such an application triggers backtracking.

We have shown how prospective value semantics provides a description of stack based operations, but a full description of Forth semantics, covering interpretation and compilation, memory access, and the definition of defining words, is beyond the theory presented here. Nevertheless we can extend this investigation to provide a semantics that is *usable*) for developing Forth applications, and we hope to report on the best way to do this in our future work.

References

- [Abr96] J-R Abrial. The B Book. Cambridge University Press, 1996.
- [DFM⁺23] S E Dunne, J F Ferreira, A Mendes, C Ritchie, W J Stoddart, and F Zeyda. bGSL: An imperative language for specification and refinement of backtracking programs. Journal of Logical and Algebraic Methods in Programming, 130, 2023.
- [Heh93] E C R Hehner. A Practical Theory of Programming. Springer Verlag, 1993. 2023 edition available on-line.
- [RS10] C Ritchie and W J Stoddart. A compiler which creates type tagged parse trees and executes them as Forth programs. In 26th EuroForth Conference Proceedings, 2010.
- [SZL10] W J Stoddart, F Zeyda, and A R Lynas. A virtual machine for supporting reversible probabilistic guarded command languages. ENTCS, 253, 2010.