A Formal Semantics for Befunge

Cameron Wong

Abstract

There exist a host of modern static program analysis techniques used to prove programs correct such as Hoare Logic, dataflow analysis, and symbolic execution. However, current state of the art is focused around linear and multi-linear parallel program flow. This precludes these techniques from being useful when analyzing multidimensional semantics, particularly the so-called Fungeoid languages. In this paper, we introduce and explore a denotation for for an idealized Befunge-93.

1 Introduction

Many hours have gone into developing "visual" programming languages such as Scratch, mBlock or MIT's App Inventor. Being able to visualize control flow as a series of discretized blocks that can be rearranged visually is believed to aid in productivity and comprehension of many advanced programming constructs. However, these languages often lack expressiveness, and fail to break down the "tree-like" structure of many programs. For example, the body of a "while" loop in Scratch is considered a "child block" of the loop, instead of keeping a fully visual representation of the loop itself.

Befunge is a visual programming language that purports to address these issues. By arranging the program itself into a two-dimensional grid that is navigated via "arrow" primitives, all sorts of complex control flow can be both expressed and visualized without needing to speak of "loop bodies", "child nodes", or anything of the sort.

2 Specifics

It is known that any of the following extensions to Befunge-93 are sufficient to be Turing complete:

- The addressable Funge space is infinite (that is, the `g` and `p` commands can address an infinite amount of space)
- The Funge stack can be arbitrarily large
- The Funge stack can hold integers of arbitrary size

We will concern ourselves with only the second and third conditions. While it is possible to have an infinite Funge space while maintaining the "wraparound" property, we will not do so here.

Let the language $\mathbf{B}_{n,m}$ represent this idealized Befunge-93 with arbitrary precision integers, indexed by the integers modulo n and m, and $\mathbf{B}$ be $\mathbf{B}_{n,m}$ for an arbitrary choice of n and m that is "large enough".

3 Syntax

It would be quite difficult to construct a proper formal grammar for Befunge (nor would it be particularly useful), so we will leave this as an exercise for the reader, should they determine it necessary. We will instead relate the concrete syntax atoms (single characters) with their abstract comamnds.

$$
\begin{array}{llcllr}
\mathsf{Cmd} & C & ::= & + & \mathsf{plus} & \\
 & & | & \ldots & & \\
 & & | & : & \mathsf{clone} & \\
 & & | & \backslash & \mathsf{swap} & \\
 & & | & \$ & \mathsf{pop} & \\
 & & | & . & \mathsf{print} & \text{(integers)} \\
 & & | & \ldots & & \\
 & & | & \mathrm{g} & \mathsf{get} & \\
 & & | & @ & \mathsf{end} & \\
 & & | & \texttt{"} & \mathsf{str} & \text{(Toggle string mode)} \\
 & & | & \hat{} & \uparrow & \\
 & & | & \ldots & & \\
 & & | & n & n & (n \in [0..9]) \\
 & & | & c & \mathsf{chr}(c) & (c \in CHARACTERS)
\end{array}
$$

(the full chart is included in Appendix A)

Note that the final syntax rule serves to bring all characters, even those not corresponding to a Funge command, into the concrete syntax of $\mathbf{B}$. This is so we can properly represent strings in $\mathbf{B}$; it is only relevant when in *string mode.*

4 Intuitive Semantics

The "natural" way to express the semantics of $\mathbf{B}$ are those suggested by the original Befunge specification, which is that of treating the addressable program space ("Funge space") and the abstract program as a single grid F, accompanied by the abstracted Funge stack S. We must also take care to also consider the *execution direction* d so we can determine the next command to execute.

A Funge space indexed over sets A and B, then, is a grid of instructions:

$$P ::= A \times B \to \mathsf{Cmd}$$

4.1 Static Semantics

We will say a $\mathbf{B}$ program is well-formed when

1. When encountering a div command, the top of the stack is not 0.

2. No chr(c) command is executed outside of string mode.

While we could choose stricter criteria, such as disallowing the reflective instruction put to write over any executable path, we would be doing so at the cost of algorithmic expressiveness (for example, we might use put to change a directional instruction, allowing us to store and branch on up to two bits of information without ever needing to store them on the stack).

The first property is quite uninteresting:

Theorem 1. *Property 1 is undecideable.*

Proof. By application of Rice's Theorem. $\square$

As it so happens, the second property is *also* undecidable. In the absence of the put instruction, it is trivially possible to check that no path encounters an invalid character. Once we include it, however, we find that

Theorem 2. *Property 2 is undecidable for large enough n and m.*

Proof. Suppose we had some function $W : \mathbf{B} \to bool$ that returns true if and only if the input program satisfies property 2. Also let RUN be the function that runs a **B** program on some input and T be the function transforming **B** programs into Turing machines.

It is possible to write a Brainf*ck interpreter in **B** that does not violate property 2[1]. Let F be this program and F' be F with all end instructions replaced by the invalid character a. Brainf*ck is known to be Turing complete, so there must exist some computable function $B : TURING \to BRAINF*CK$ that transforms TMs into equivalent Brainf*ck programs. This begets the following reduction from $HALTS$:

```
def HALTS(M):
  def HELP(M):
    return RUN(F', B(M))
  return not W(T(HELP(M)))
```

If M halts on the empty string, then running F on $B(M)$ will also halt. This means that running F' on $B(M)$ will execute the invalid character a, so $W(T(HELP(M)))$ will return true.

If M does not halt on the emtpy string, then running F on $B(M)$ will also fail to terminate one way or another. As F does not execute any invalid instructions and F' only replaces the end instructions (which are known not to be executed, as M fails to terminate), $W(T(HELP(M)))$ must return false. $\square$

This means that well-formed checks must be done at runtime, giving the following well-formed rules over a program P:

$$\frac{}{P \ \mathsf{valid}}$$

[1] http://www.echochamber.me/viewtopic.php?t=43912

4.2 Dynamic Semantics

We will define our "program state" $\mathbf{St}$ to be the set $\mathsf{Prg} \times \mathsf{Point} \times \mathsf{Stack} \times \mathsf{Dir} \times \mathsf{Bool}$, where:

$$\begin{aligned} \mathsf{Point} &\triangleq \mathbb{Z}_n \times \mathbb{Z}_m \\ \mathsf{Prg} &\triangleq \mathsf{Point} \to \mathsf{Cmd} \\ \mathsf{Dir} &\triangleq \{\uparrow, \downarrow, \rightarrow, \leftarrow\} \end{aligned}$$

The semantic domain of $\mathbf{B}$ will be partial functions $\mathbf{St} \rightharpoonup \mathcal{P}(\mathbf{St} \times \mathsf{String})$, where an undefined result will represent an exceptional state (division by zero or an infinite loop). The associated string is the output of the overall program as a printed string of characters. Finally, because $\mathbf{B}$ has nondeterminism in the ? operator, we must consider all possible program results. We will not differentiate between nonterminating programs outputting different strings – printing "aaaaaaaaaaaaaaaaaaaaaaaa..." vs "aaaaaaaaaaaaaaAaaaaaa..." will both be undefined states.

We could choose to model user input as a sequence associated with the state, but we will instead denote a program dependent on such as being nondeterministic over all possible user inputs.

4.3 Auxiliary Functions

To ease notation, we define the function $\mathsf{next} : \mathsf{Point} \times \mathsf{Dir} \to \mathsf{Point}$ as follows:

$$\mathsf{next}(d, (x, y)) = \begin{cases} (x +_n 1, y) & d = \rightarrow \\ (x -_n 1, y) & d = \leftarrow \\ (x, y +_m 1) & d = \uparrow \\ (x, y -_m 1) & d = \downarrow \end{cases}$$

where $+_n$ and $-_n$ are addition/subtraction modulo n.

Next, let $\mathsf{ord} : CHARACTERS \to \mathbb{N}$ be the ascii representation of a character c, and $\mathsf{chr} : \mathbb{N} \to \mathsf{String}$ be the character corresponding to ascii ordinal n. $\bar{\cdot} : \mathbb{N} \to \mathsf{String}$ is the function mapping numbers to their decimal representations, and $\mathsf{parse} : \mathbb{N} \to \mathsf{Cmd}$ maps numbers to the $\mathbf{B}$ command corresponding to the ascii ordinal n.

Finally, let $\bullet : \mathsf{String} \times \mathcal{P}(\mathbf{St} \times \mathsf{String}) \to \mathcal{P}(\mathbf{St} \times \mathsf{String})$ be the function prepending a character to the output streams, in particular

$$s \bullet R = \{(\sigma, ss') \mid (\sigma, s') \in R\}$$

4.4 Rules

In all of the following, $r_d = \mathsf{next}(d, r)$.

$$
\begin{array}{lcr}
[c](P, r, S, d, \mathsf{true}) & = & [P(r_d)](P, r_d, (ord(c), S), d, \mathsf{true}) \\
[+](P, r, (y, x, S), s, \mathsf{false}) & = & [P(r_d)](P, r_d, (x + y, S), d, \mathsf{false}) \\
\ldots & & \\
[/](P, r, (0, x, S), d, \mathsf{false}) & = & undefined \\
[!](P, r, (x, S), d, \mathsf{false}) & = & \begin{cases} [P(r_d)](P, r_d, (1, S), d, \mathsf{false}) & x = 0 \\ [P(r_d)](P, r_d, (0, S), d, \mathsf{false}) & otherwise \end{cases} \\
[`](P, r, (y, x, S), d, \mathsf{false}) & = & \begin{cases} [P(r_d)](P, r_d, (1, S), d, \mathsf{false}) & x > y \\ [P(r_d)](P, r_d, (0, S), d, \mathsf{false}) & otherwise \end{cases} \\
[:](P, r, (x, S), d, \mathsf{false}) & = & [P(r_d)](P, r_d, (x, x, S), d, \mathsf{false}) \\
[\backslash](P, r, (y, x, S), d, \mathsf{false}) & = & [P(r_d)](P, r_d, (x, y, S), d, \mathsf{false}) \\
[\$](P, r, (x, S), d, \mathsf{false}) & = & [P(r_d)](P, r_d, S, d, \mathsf{false}) \\
[.](P, r, (n, S), d, \mathsf{false}) & = & \overline{n} \bullet [P(r_d)](P, r_d, S, d, \mathsf{false}) \\
[,](P, r, (n, S), d, \mathsf{false}) & = & \mathsf{chr}(n) \bullet [P(r_d)](P, r_d, S, d, \mathsf{false}) \\
[\#](P, r, S, d, \mathsf{false}) & = & [P(\mathsf{next}(d, r_d))](P, \mathsf{next}(d, r_d), S, d, \mathsf{false}) \\
[\&](P, r, S, d, \mathsf{false}) & = & \bigcup_{n \in \mathbb{N}} ([P(r_d)](P, r_d, (n, S), d, \mathsf{false})) \\
[\sim](P, r, S, d, \mathsf{false}) & = & \bigcup_{c} ([P(r_d)](P, r_d, (\mathsf{ord}(c), S), d, \mathsf{false})) \\
[\mathsf{g}](P, r, (y, x, S), d, \mathsf{false}) & = & [P(r_d)](P, r_d, (P(x, y), S), d, \mathsf{false}) \\
[\mathsf{p}](P, r, (y, x, v, S), d, \mathsf{false}) & = & [P'(r_d)](P', r_d, S, d, \mathsf{false}) \\
 & & (\text{where } P' = [P \mid (x, y) : \mathsf{parse}(v)]) \\
["](P, r, S, d, b) & = & [P(r_d)](P, r_d, S, d, \neg b) \\
[@](\sigma) & = & \sigma \\
[\hat{}](P, r, S, d, \mathsf{false}) & = & [P(r_\uparrow)](P, r_\uparrow, S, \uparrow, \mathsf{false}) \\
\ldots & & \\
[?](P, r, S, d, \mathsf{false}) & = & \bigcup_{d' \in \mathsf{Dir}} ([P(r_{d'})](P, r_{d'}, S, d', \mathsf{false})) \\
[n](P, r, S, d, \mathsf{false}) & = & [P(r_d)](P, r_d, (n, S), d, b, \mathsf{false}) \\
[c](P, r, S, d, \mathsf{false}) & = & undefined
\end{array}
$$

5 The Future

We hope to extend this technique to be used on further Fungeoid languages. We believe that this work will have great impact on the development of... well, we're not sure, but surely *something* will come of this.

A Full Syntax Chart

Cmd	C	::=	$+$	plus	
		|	*	times	
		|	-	minus	
		|	$/$	div	
		|	%	mod	
		|	!	not	
		|	‘	gt	
		|	:	clone	
		|	$\backslash$	swap	
		|	$	pop	
		|	.	print	(integers)
		|	,	print	(chars)
		|	#	skip	
		|	g	get	
		|	p	put	
		|	&	inp	(integers)
		|	$\sim$	inp	(chars)
		|	@	end	
		|	"	str	(Toggle string mode)
		|	^	$\uparrow$	
		|	v	$\downarrow$	
		|	$<$	$\leftarrow$	
		|	$>$	$\rightarrow$	
		|	?	rand	
		|	n	n	($n \in [0..9]$)
		|	c	chr(c)	($c \in CHARACTERS$)