Java Framework for Static Analysis of Forth Programs

Jaanus Pöial

The Estonian Information Technology College
e-mail: jaanus.poial@itcollege.ee

Abstract. In [2] author introduces theoretical background for static analysis of Forth programs (definitions, basic operations, typing rules, etc.). This paper is direct continuation of the topic and describes implementation of basic blocks for writing software tools to support type checking of Forth programs. On small examples we try to explain problems and possible solutions. Author hopes that these ideas help to develop some useful tools. Prototype is written in Java that is quite universal and widespread object oriented platform for software development.

Keywords: *Type Systems, Forth, Program Analysis*

1 Introduction

In [1] we first defined formal stack effects of Forth words. This definition and theory of stack effect calculus have been developed for a long time and in [2] we introduced several operations on effects to perform static type analysis of Forth texts. Basic program constructs covered so far are sequence, choice and iteration. For each of these we have corresponding operation in our calculus.

Let us start with a few informal examples. Data item on Forth stack generally does not have any run-time type but the programmer usually has some static type information in mind when composing a program. This information may be more or less exact, e.g.
`a-addr < c-addr < addr < u < x` (`x` is the least exact)
and for example word `@` is specified as `(a-addr -- x)`.

At the same time many operations manipulate stack (e.g. `SWAP, DUP, ROT,` etc.) without changing types of data items. To cover this aspect we introduced position indices to type symbols:
word `SWAP` has effect `(x[2] x[1] -- x[1] x[2])` rather than `(x x -- x x)`.
Program `DUP @` has stack effects `(x[1] -- x[1] x[1])(a-addr -- x)` and should evaluate into `(a-addr[1] -- a-addr[1] x)` rather than `(x -- x x)`.

Sequences can be longer than two words, e.g. `SWAP DUP @` gives:
`(x[2] x[1] -- x[1] x[2])(x[1] -- x[1] x[1])(a-addr -- x)`.
When two type symbols with locally defined indices must match in the process of evaluating a sequence they produce a new type symbol that has minimal (most

* Supported by Estonian Science Foundation grant no. 6713

exact) type and new "fresh" index. This new symbol replaces both matching symbols in the sequence:
`(x[2] x[1] -- x[1] x[2])(x[1] -- x[1] x[1])(a-addr -- x)`
`(x[3] x[1] -- x[1] x[3])(x[3] -- x[3] x[3])(a-addr -- x)`
`(a-addr[4] x[1] -- x[1] a-addr[4])(a-addr[4] -- a-addr[4] a-addr[4])`
`(a-addr[4] -- x)`
Let us rename (to delete unused) indices where possible:
`(a-addr[2] x[1] -- x[1] a-addr[2])(a-addr[2] -- a-addr[2] a-addr[2])`
`(a-addr[2] -- x)`
and the final evaluation result for sequence `SWAP DUP @` is
`(a-addr[2] x[1] -- x[1] a-addr[2] x)`.

On this small example we see that evaluation has to preserve information both on types and positions of data items. When type symbols do not match we have to produce some useful error information. This is one of the reasons to have evaluation of sequence as a basic block in our framework rather than composition of two effects.

Choice between two branches of a program in our framework forces these branches to have "the same" effect. Operation *glb* (greatest lower bound) of two effects tries to match all corresponding symbols and replace these with new most exact "fresh" symbols (like for composition above).

Program `IF ! ELSE C! THEN` has two branches and we calculate
glb(`(x a-addr -- ),(char c-addr --)`*)* as `(char a-addr -- )`.

Program `IF OVER ELSE @ DP @ THEN` produces
glb(`(x[2] x[1] -- x[2] x[1] x[2]),(a-addr -- x x)`*)* that evaluates into `(x[2] a-addr[1] -- x[2] a-addr[1] x[2])`.

From these examples we conclude that *glb* calculates longest type lists with most exact types.

Iteration in this framework forces the loop body not to change the stack state (loop body has "idempotent" effect: e = ee).
Effects with equal type lists on both sides are idempotents: (*list* `--` *list*).
To calculate the effect of a loop we find the most precise idempotent by matching left and right sides of the effect that describes the loop body (it is possible only if both sides have the same length).

Program `BEGIN @ AGAIN` iterates the word `@` endlessly. The loop body has effect `(a-addr -- x)` and loop (as a whole) has effect
`(a-addr[1] -- a-addr[1])`.
More complicated loop `BEGIN SWAP OVER WHILE NOT REPEAT` falls into two pieces:
`(x[2] x[1] -- x[1] x[2])(x[2] x[1] -- x[2] x[1] x[2])(flag -- )` and `(x -- x)`.
First loop body has effect `(x[2] flag[1] -- flag[1] x[2])` and the loop has effect `(flag[1] flag[1] -- flag[1] flag[1])`. Composed by the second loop effect `(x -- x)` we still have `(flag[1] flag[1] -- flag[1] flag[1])` but we also know that `NOT` operates on `flag[1]`.

If the loop body has effect *e* we can calculate loop effect as *glb(e, ee)*.

These examples are not complex enough to cover real programs but hopefully give some ideas how to evaluate sequences, choices and iterations.

2 Java framework

Package `evaluator` consists of several classes and probably will grow depending on tools we intend to develop. Let us summarize the basic blocks in this framework.

Class `TypeSymbol` defines symbolic type of a stack item (like `a-addr, flag, char, ...`) together with position index (used by stack manipulation words like `SWAP, OVER, ROT, ...`). Type names must be known by current typesystem. Usually there are more names than actual types (synonyms are allowed for convenience). Position indices are integers (when "fresh" symbol is created during the match operation this index increases, index 0 is used if position is not important).

Class `TypeSystem` is used to define and query subtyping relations between types. Type name is used as a key to access matrix of relations. Relations are `"incompatible", "subtype", "supertype", "synonym"`. Typesystem is static, once created it does not change much during evaluation process. But we keep possibility to use different typesystems to analyze the same program open (e.g. to see more or less details).

Class `Tvector` describes the stack state (top right) and each vector consists of typesymbols. Substitution of one symbol by another is defined in this class.

Class `Spec` describes the stack effect (specification). It consists of two vectors (left side - stack state before execution, right side - stack state after execution) and additional workfields (e.g. string read by scanner words like `."` or `( `). Major operations of the framework (like greatest lower bound or finding idempotent for the loop body) are defined in this class:

`spec1.`**`glb`**`(spec2, typesystem, specset) returns spec`
`bodyspec.`**`idemp`**`(typesystem, specset) returns loopspec`

Class `SpecSet` describes a mapping from Forth words to stack effects. This mapping is dynamic - all new words defined in the program must be added. Once again, we may use different specsets for the same program text to analyze different aspects (e.g. run-time stack vs. compile-time stack).

Class `SpecList` describes a linear sequence of stack effects and implements evaluation of this list against given typesystem and given specset:

`speclist.`**`evaluate`**`(typesystem, specset) returns spec`

Composition of stack effects is a particular case of evaluation.

Class `ProgText` is inner representation of the Forth program we want to analyze. Currently only linear sequence of words is implemented.

Class `Evaluator` contains the main method and a small demo that adds annotations (comments about stack state) to the linear program text.

3 Further work

All classes described above are prototypes and need to be implemented fully to develop any useful tools. Extensible nature of Forth demands that our framework is also extensible (for example, we cannot just fix control structures or data definition words). It seems to be easier to re-write this package in Forth to achieve full extensibility.

Stack effect calculus might help programmers when integrated into Forth IDE (e.g. editor that shows current stack state symbolically, evaluates selected text, etc.). To create such an editor we need these (or similar) basic blocks plus a lot of other components. Maybe it is reasonable to use some existing IDE platform (like Eclipse - `www.eclipse.org`) and develop a Forth plugin for Eclipse. Then probably we lose in extensibility but we can still work with some subset of Forth. The following is a very small but useful subset that we would like to cover next:

```
PROG = ELEM / PROG ELEM .
ELEM = SIMPLE / DEFINITION .
SIMPLE = WORD / PARSER / CONSTANT .
WORD = <word> .
PARSER = PARSER<delim> / COMMENT .
PARSER<delim> = WORD <string><delim> .
COMMENT = <comment> .
CONSTANT = <constant> .
DEFINITION = VARDEF / CONSTDEF / COLONDEF .
VARDEF = 'VARIABLE' NAME / 'CREATE' NAME .
CONSTDEF = SIMPLIST 'CONSTANT' NAME .
COLONDEF = ':' NAME CONTENT ';' /
':' NAME CONTENT 'CREATE' CONTENT 'DOES>' CONTENT ';' .
NAME = <word> .
CONTENT = CELEM / CONTENT CELEM / .
CELEM = SIMPLE / STRUCTURE .
STRUCTURE = 'IF' CONTENT 'THEN' /
'IF' CONTENT 'ELSE' CONTENT 'THEN' /
'BEGIN' CONTENT 'WHILE' CONTENT 'REPEAT' /
'[' SIMPLIST ']' .
SIMPLIST = SIMPLE / SIMPLIST SIMPLE .
```

References

1. Pöial J., "Algebraic Specifications of Stack Effects for Forth Programs," *1990 FORML Conference Proceedings, EuroFORML'90 Conference, Oct 12 – 14, 1990, Ampfield, Nr Romsey, Hampshire, UK,* Forth Interest Group, Inc., San Jose, USA, p. 282 – 290, 1991.
2. Pöial J. "Typing Tools for Typeless Stack Languages," *Proc. 22-th EuroForth Conference, September 15 – 17, 2006, Cambridge* p. 40 – 46, 2006.