Program Analysis for Stack Based Languages

Jaanus Pöial

Institute of Computer Science
University of Tartu, Estonia
e-mail: `jaanus@cs.ut.ee`

Abstract. Static program analysis is used to discover certain run-time properties of a computer program without actually executing the program. For example, data flow analysis keeps track on usage of variables (assignments, expressions, etc.), so we can find uninitialized variables, move constant expressions out from loop bodies etc. Program analysis can be used both for finding errors and performing useful transformations on programs (optimization, parallelization, etc.).
Many things from program analysis classics could be adapted to the Forth language without any changes if we forget about specific features of stack based languages. In case of Forth-like languages we have slightly different questions to ask about programs: for example, storing data to a variable is rather an exception than a usual way to compute something in Forth. We are more interested in what is happening to the data stack, to the dictionaries, to the return stack etc. Program state in Forth is more settled in stacks than in variables. Most difficulties and errors in Forth programs are related to the stack usage and this is a reasonable starting point for the stack-specific program analysis.
In this paper some rules to perform the so called "must"-analysis are described to find places in Forth programs where the strong stack discipline is violated. Even more important is an attempt to introduce some discussion in Forth community - what kind of questions we want to ask about Forth programs and what are these new specific types of analyses that better suite for the stack based programming languages.

Keywords: *Stack Based Languages, Program Analysis*

1 Introduction

Program analysis became popular in the world of embedded systems and safety critical applications where more resources are used to avoid software errors than in usual "office software" business. Many run-time properties of a program can be estimated statically using some kind of abstract interpretation [1]. Good analysis produces reasonable amount of warnings about suspicious passages in the program, so human programmer can check these places and make improvements to the software.

⋆ Supported by Estonian Science Foundation grant no. 5279 and Estonian Information Technology Foundation

Unfortunately, analysis can be very resource-consuming, in some cases even small pieces of software embedded in some device take a lot of computing power to analyze. Number of program states to explore grows very fast for precise analysis, to keep it under control some approximation is needed to glue similar states into single one. On the other hand, the analysis still has to produce valuable results.

The so called control flow graph of a program describes all possible execution paths as a finite structure. The program state is coupled with the node (sometime to the edge) of the control flow graph, the typical question is "What is known about ... in program point ... ?". There are two different kinds of statements: first, when a property must hold for all possible execution paths, and second, when a property may hold for some particular execution (there is no guarantee that it does not hold). Sometimes the "must"-analysis finds less properties guaranteed than there actually exist, similarly the "may"-analysis sometimes finds more properties than these that actually might hold. It is important to use safe, conservative approximations, because a precise result in this area is usually hard or impossible to compute.

Classical data flow analysis concentrates on memory - program state is described via set of variables and analysis keeps track on variable usage and variable updates. We can find out uninitialized variables, "live" variables, available expressions, reaching definitions, "very busy" expressions, etc. Good introduction to program analysis is made in book [2].

In case of stack based languages the memory state is a secondary issue, it is more important to check the usage of stack(s). For example, a common mistake is to write alternative program branches with different stack effects (it is not easy to discover this bug if some branch is hardly ever executed).

In this paper we introduce some new ideas on static analysis of stacks, these ideas are partially implemented as a set of Java classes. Java is used as an available multi-platform tool, we intend to use existing Java API to produce some Forth-targeted tools (like validator and editor that supports strong stack discipline).

The formalism is mainly used to give a precise definition to the rules that Forth programmers know intuitively. On the other hand, it is a short way to explain these more than thousand lines of code written to implement the basic operations.

2 Stack analysis framework

Original stack effect calculus is introduced in [3], related work by Bill Stoddart and Peter Knaggs is published in [6], few other works are referred in [5]. From the viewpoint of program analysis it is important to mention an attempt to formalize multiple stack effects for control structures in [4]. This approach did not lead to implementation of practical analysis tools, mainly because sets of stack effects grew fast and were costly to manage. What is the use of knowing that a program may produce a large set of stack effects? It is important to ask the right question about the program: not "What this program might do?" (interesting, but costly and impracticable) but "Why this program does not do what it has to do?" (locating a suspicious passage).

The following framework is oriented to the must-analysis. There are theoretical considerations to restrict ourselves to this type of analysis: the set of stack effects is a semilattice (each subset has a greatest lower bound (*glb*) but does not necessarily have a least upper bound). Only the subset of idempotents is a lattice (e is an idempotent iff $e = e \cdot e$).

In this paper the derivation rules are used to express the composition and *glb* of stack effects. There are two main constructs and one strong assumption:
1) Composition (multiplication) of stack effects describes a linear segment of a program.
2) Greatest lower bound of stack effects describes union of alternative branches of a program.
3) Body of a program cycle is described by an idempotent stack effect (composition of an idempotent element to itself gives back the same element, consequently, the stack state does not change).

Let us introduce some notation for stack effects.

$t, u, ...$ - possible types of data stack items.

$t \leq u$ - t is subtype of u (t is more exact) or equal to u
(subtype relation is transitive).

$t \perp u$ - t and u are incompatible types.

t^i - type symbol with "wild-card" index
(wild-card index i is unique for elements of "the same type").

$a, b, c, d, ...$ - type lists that represent the stack state (top right).

$s = (a \to b)$ - stack effect (a - stack state before the operation, b - after).

$\mathbf{1}$ - empty effect (no inputs, no outputs), top of lattice of idempotents.

$\mathbf{0}$ - zero effect (error, type conflict), bottom of lattice of idempotents.

$(a \to b) \cdot (c \to d)$ - composition of two stack effects (defined later).

$x, y, ...$ - sequences of stack effects.

$y,\ where\ u^j := t^k$ - substitution of u^j to t^k
(all occurrences of u^j in all type lists of sequence y are replaced by t^k)
k is unique index over y.

$(a \to b) \sqcap (c \to d)$ - *glb* of two stack effects (defined later).

$r = \sqcap^* s$ - greatest idempotent r smaller or equal to s, zero is allowed
($r \cdot r = r$ and $r \preceq s$, meaning of $\preceq$ appears later).

$\alpha, \beta, ...$ - sequences of operations (linear programs).

$s(\alpha)$ - stack effect of sequence α.

Rules for composition

These rules describe evaluation of sequence of stack effects. Whenever a type clash occurs the result is zero. When two types (coming from different contexts) for the same stack item are compared the more exact type "wins" and this information is spread to whole evaluated part of the sequence (denoted by x).

$$\frac{x \cdot \mathbf{0}}{\mathbf{0}} \qquad \frac{\mathbf{0} \cdot y}{\mathbf{0}} \qquad \frac{x \cdot (a \to bt) \cdot (cu \to d), \textit{ where } t \perp u}{\mathbf{0}}$$

$$\frac{x \cdot (a \to b) \cdot (\to d)}{x \cdot (a \to bd)} \qquad \frac{x \cdot (a \to) \cdot (c \to d)}{x \cdot (ca \to d)}$$

$$\frac{x \cdot (a \to bt^i) \cdot (cu^j \to d), \textit{ where } t \leq u}{x \cdot (a \to b) \cdot (c \to d), \textit{ where } t^i := t^k \textit{ and } u^j := t^k}$$

$$\frac{x \cdot (a \to bt^i) \cdot (cu^j \to d), \textit{ where } u \leq t}{x \cdot (a \to b) \cdot (c \to d), \textit{ where } t^i := u^k \textit{ and } u^j := u^k}$$

Example

Let us have the following toy type system:

```
a-addr < c-addr < addr < x
flag < x
char < n < x
```

Using these types and "wild-cards" we can introduce hypothetical stack effects:

```
DUP  ( x[1] -- x[1] x[1] )
DROP ( x -- )
SWAP ( x[2] x[1] -- x[1] x[2] )
ROT  ( x[3] x[2] x[1] -- x[2] x[1] x[3] )
OVER ( x[2] x[1] -- x[2] x[1] x[2] )
PLUS ( x[1] x[1] -- x[1] )
```

polymorphic "plus", arguments have to have the same type

```
+    ( x x -- x )
@    ( a-addr -- x )
!    ( x a-addr -- )
C@   ( c-addr -- char )
C!   ( char c-addr -- )
DP   ( -- a-addr )
0=   ( n -- flag )
NOT  ( x -- x )
```

Now let us apply the rules to some example programs

```
OVER OVER PLUS ROT ROT PLUS !
```

evaluates to `( a-addr[1] a-addr[1] -- )`

On the other hand, the following program has type conflict in it
`OVER OVER PLUS ROT ROT PLUS C!`

It is suggested to play with some more examples to understand how the rules work (author also has an implementation for this set of stack effects).

Rules for greatest lower bound

To join the type information from different alternative branches of a program we need an operation $\sqcup$ of finding the least upper bound of finite set of effects. As mentioned before, this approach does not work well. Instead, we formulate a different problem - what are the weakest conditions to make all branches equal? This problem can be solved using greatest lower bound operation $\sqcap$. We approximate the branching control structure as a whole by *glb* of all the branches.

$$\frac{s \sqcap \mathbf{0}}{\mathbf{0}} \qquad \frac{r \sqcap s}{s \sqcap r}$$

If there exist type lists $a_1, a_2, a_3, b_1, b_2, b_3, c_1, c_2, c_3$ such that for all elements of the lists these subtyping relations hold element-wise
$a_3 = min(a_1, a_2)$
$b_3 = min(b_1, b_2)$
$c_3 = min(c_1, c_2)$
then the following rule is applicable, in all other cases the result is zero.

$$\frac{(c_1 a_1 \to c_2 b_1) \sqcap (a_2 \to b_2)}{(c_3 a_3 \to c_3 b_3)}$$

If a set of effects has a non-zero *glb* r then all effects in this set "do the same thing", r is just the most exact description of it (having longest lists and most exact types). In case it is impossible to force effects to be comparable (in sense of finding a common predecessor for them) the *glb* is zero (zero is less or equal to any stack effect).

We also introduce the following notation that is useful for cycles:
$\sqcap^* s = s \sqcap (s \cdot s)$
The result of this operation is an idempotent element that most precisely describes the loop body s.

Example

`ROT` and `@` from the previous example have *glb*
`( a-addr[1] a-addr[1] a-addr[1] -- a-addr[1] a-addr[1] a-addr[1] )`

`C@` and `@` have *glb*
`( a-addr -- char )`

Rules for control structures

In [4] we introduced some rules for "may"-analysis like the following (we do not reproduce all the rules here but just two most characteristic examples):

$$\frac{s(\texttt{ IF } \alpha \texttt{ ELSE } \beta \texttt{ THEN })}{[(\texttt{true} \rightarrow) \cdot s(\alpha)] \sqcup [(\texttt{false} \rightarrow) \cdot s(\beta)]}$$

$$\frac{s(\texttt{ BEGIN } \alpha \texttt{ WHILE } \beta \texttt{ REPEAT })}{\sqcup^{*}[s(\alpha) \cdot (\texttt{true} \rightarrow) \cdot s(\beta)] \cdot s(\alpha) \cdot (\texttt{false} \rightarrow)}$$

These rules describe the nature (semantics) of control structures but are less useful for practical analysis. Let us introduce some new less exact rules in "must"-analysis style.

$$\frac{s(\texttt{ IF } \alpha \texttt{ ELSE } \beta \texttt{ THEN })}{(\texttt{flag} \rightarrow) \cdot [s(\alpha) \sqcap s(\beta)]}$$

$$\frac{s(\texttt{ BEGIN } \alpha \texttt{ WHILE } \beta \texttt{ REPEAT })}{\sqcap^{*}[s(\alpha) \cdot (\texttt{flag} \rightarrow)] \cdot \sqcap^{*} s(\beta)}$$

It is quite obvious how to introduce rules for all other control structures.

Example

A good exercise is to think about the program:

```
: test IF ROT ELSE @ THEN ;
```

What is the "right" analysis for this program? Is this program "correct"? Hint: we already know the *glb* (`ROT`, `@`) from the previous example.

Another good example from [4] uses a while-cycle:

```
: test2 BEGIN SWAP OVER WHILE NOT REPEAT ;
```

`test2` may loop forever in "integer" world, in "Boolean" world it is nearly equivalent to

```
: test3 OR FALSE SWAP ;
```

This leads us to the problem of program transformations that is another important application area for the program analysis.

3 Conclusion

Forth is used in embedded and safety critical system engineering where the software testing often incorporates tools for program analysis. Classical tools cover only some aspects of Forth, the stack based nature of this language induces the need for specific stack analysis methods.

This paper creates a connection between authors earlier work on stack effect calculus and the field of classical program analysis. Still it is not quite clear what are the most important questions that the Forth-specific program analysis must answer. The rules introduced above allow finding such conditions that guarantee certain behaviour of the program when hold, but probably these conditions force too strong stack discipline (no words with multiple stack effects, no branches with different stack effects, no loops that grow or shrink the stack). On the other hand, pointing to the spots where this discipline is violated might help a lot. We already started a pilot project on implementation of this analysis but to proceed we need some feedback from practitioners. It will help us to understand the relationship between program analysis and stack based languages more deeply.

References

1. Cousot P., Cousot R., "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," *4th POPL, Los Angeles, CA* , ACM Press, p. 238 – 252, 1977.
2. Nielson F., Nielson H.-R., Hankin C., "Principles of Program Analysis," Springer-Verlag, 450 pp., 1999.
3. 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.
4. Pöial J. "Multiple Stack-effects of Forth Programs," *1991 FORML Conference Proceedings, euroFORML'91 Conference, Oct 11 – 13, 1991, Marianske Lazne, Czechoslovakia,* Forth Interest Group, Inc., Oakland, USA, p. 400 – 406, 1992.
5. Pöial J. "Stack Effect Calculus with Typed Wildcards, Polymorphism and Inheritance," *Proc. 18-th EuroForth Conference, Sept. 6-8, 2002, TU Wien, Vienna, Austria,* p. 38, 2002.
6. Bill Stoddart, Peter J. Knaggs: "Type Interference in Stack Based Languages," *Formal Aspects of Computing 5(4): 289-298 (1993).*