MULTIPLE STACK-EFFECTS OF FORTH-PROGRAMS

Jaanus Pöial

Tartu University, Department of Computer Science

Tartu Estonia

The most important quality of the Forth-word is its stack-effect. There are some good tools to trace the program, but in the complicated environment it is an inconvenient task to trace all branches of the program. The main idea of this work is to introduce a formalism which allows to check the stack-effects according to the program text. In [P90] an attempt was made to handle linear programs with the unique stack-effect. Recent work tries to investigate multiple stack-effects and programs with the control structures. Our attention is concentrated only to the aspect of parameter passing through the stack excluding memory handling, I/O, e.t.c.

Each Forth-word has an informal specification of its stack-effect given in the form:

input parameter types -- output parameter types

The type lists given above are ordered, the end of the list corresponds to the top of the stack. This specification does not say anything about the essence of the operation.

Our further investigation is based on the theory of semigroups. In [NP70] M. Nivat and J.F. Perrot

introduced a 0-bisimple inverse semigroup called polycyclic monoid. We need some notations to express the main ideas:

A - an alphabet (finite set of type names),

A^* - the set of strings over A (set of type lists),

Λ - the empty string ($\Lambda \in A^*$ for arbitrary A),

ab - the concatenation of strings a and b,

0 - the *null element*, specifies the type conflict in sense of parameter passing.

The *set of stack-effects* over A is the union:

$$\Phi(A) = (A^* \times A^*) \cup [0].$$

Let $[s_1 \text{ --- } s_2]$ denote a pair $(s_1, s_2) \in A^* \times A^*$.

Here s_1 is the list of input parameters and s_2 is the list of output parameters as above. If there is no need to emphasize the alphabet we use Φ instead of $\Phi(A)$.

The pair $(\Lambda, \Lambda) = [\text{ --- }]$ is denoted by 1.

We may define the *product of stack-effects* as follows:

1) $\forall s \in \Phi : \; s0 = 0s = 0$,

2) $\forall s, t \in \Phi \setminus [0] : \; st = [s_1 \text{ --- } s_2] [t_1 \text{ --- } t_2] =$

$$= \begin{cases} [as_1 \text{ --- } t_2] , & \text{if } t_1 = as_2 , \\ [s_1 \text{ --- } bt_2] , & \text{if } s_2 = bt_1 , \\ 0 , & \text{otherwise.} \end{cases}$$

The set Φ is (isomorphic to) a polycyclic monoid (proof in [NP70]). Consequently:

1) $\forall s, t \in \Phi : \; st \in \Phi$,

2) $\forall r, s, t \in \Phi : \; (rs)t = r(st)$,

3) $\forall s \in \Phi : \; s1 = 1s = s$,

4) $\forall s \in \Phi : \; s0 = 0s = 0$.

It is important to mention that Φ is an inverse semigroup, i.e. for each element s there exists an unique element t so that $sts = s$ and $tst = t$. This property is used in case of Forth-programs are generated by the syntax directed translation scheme (see [P90]).

Each Forth-word may (on principle) have more than one stack-effect. Let us have a look at set of all subsets of Φ, which we denote by 2^{Φ}. We define the product on the set 2^{Φ} via previous definition for Φ in the following way:

$$\forall\, \gamma, \delta \in 2^{\Phi}: \gamma\delta = \{\, uv \in \Phi \mid (u \in \gamma) \,\&\, (v \in \delta) \,\} \setminus \{\, 0 \,\}.$$

An empty set $\emptyset$ is the null element of 2^{Φ} and the set $\{\, 1 \,\}$ is the unit of 2^{Φ}.

Addition on the set 2^{Φ} is defined as the set union.

Unfortunately, 2^{Φ} is not inverse semigroup (it is easy to give a contra-example). It seems to be useful to find a subset $M \subset 2^{\Phi}$ which is sufficiently good for practical use and which is inverse semigroup. Let p be a predicate on the set 2^{Φ} so that $M = \{\, \gamma \in 2^{\Phi} \mid p(\gamma) \,\}$. The predicate p has to hold the following conditions (γ, δ, x, e, f $\in$ M):

1) $p(\gamma)$ & $p(\delta)$ => $p(\gamma\delta)$ (subsemigroup condition),

2) $p(\gamma)$ => $\exists$ x : $p(x)$ & ($\gamma x \gamma = \gamma$) (regularity condition),

3) p(e) & p(f) & ee=e & ff=f => ef=fe (commutativity condition for idempotents).

A trivial example of predicate p which gives $M \cong \Phi$ is $p(\gamma) = |\gamma| \leq 1$, i.e. γ contains one element of Φ or γ is the empty set.

Let Δ denote a set of considered stack operations. Set of Forth-programs is Δ^* (program is a list of operations here).

Program specification is defined as a set of stack-effects by the mapping $s : \Delta^* \longrightarrow 2^{\Phi}$:

1) $\forall \Pi \in \Delta : s(\Pi) \in 2^{\Phi} \setminus \{ 0 \} \setminus \emptyset$ is a given specification of stack operation Π (non-empty set of non-zero stack-effects),

2) $s(\Lambda) = \{ 1 \}$ (specification for the empty program),

3) $\forall \omega \in \Delta^*, \forall \Pi \in \Delta : s(\omega\Pi) = s(\omega)s(\Pi)$ (product in 2^{Φ}).

The program $\omega \in \Delta^*$ is said to be *correct*, if $s(\omega) \neq \emptyset$, and *closed*, if $s(\omega) = \{ 1 \}$.

A set of correct programs is defined as

$$CORRECT(\Delta, s) = \{ \omega \in \Delta^* \mid s(\omega) \neq \emptyset \}$$

and a set of closed programs as

$$CLOSED(\Delta, s) = \{ \omega \in \Delta^* \mid s(\omega) = \{ 1 \} \}.$$

Obviously

$$\{ \Lambda \} \subset CLOSED \subset CORRECT \subset \Delta^*.$$

These sets are algorithmically solvable, because we can calculate stack-effects of the program proceeding from the stack-effects of given operations (up to now this is true for the linear programs).

We must add a restriction to our correctness definition in case of control structures are used - $s(\omega)$ has to be a finite non-empty set.

To handle the control structures we introduce the notion of closure of a subset of stack-effects. Let $\gamma \in 2^{\Phi}$.

$\gamma^0 = \{ 1 \}$

$\gamma^1 = \gamma$

$\gamma^{n+1} = \gamma^n \gamma$ (product in 2^{Φ})

$\gamma^+ = \bigcup_{i=1}^{\infty} \gamma^i$ (transitive closure)

$\gamma^* = \bigcup_{i=0}^{\infty} \gamma^i$ (reflexive transitive closure)

The mapping s for the control structures is given by the formulas:

ω = BEGIN α WHILE β REPEAT , where $\alpha, \beta \in \Delta^*$

$s(\omega) = [\ s(\alpha)\ \{[\text{true} \text{ --- }]\}\ s(\beta)\ ]^*\ s(\alpha)\ \{[\text{false} \text{ --- }]\}$

ω = BEGIN α UNTIL , where $\alpha \in \Delta^*$

$s(\omega) = [\ s(\alpha)\ \{[\text{false} \text{ --- }]\}\ ]^*\ s(\alpha)\ \{[\text{true} \text{ --- }]\}$

ω = IF α THEN , where $\alpha \in \Delta^*$

$s(\omega) = \{[\text{false} \text{ --- }]\} + \{[\text{true} \text{ --- }]\}\ s(\alpha)$

ω = IF α ELSE β THEN , where $\alpha, \beta \in \Delta^*$

$s(\omega) = \{[\text{true} \text{ --- }]\}\ s(\alpha) + \{[\text{false} \text{ --- }]\}\ s(\beta)$

ω = DO α LOOP , where $\alpha \in \Delta^*$ and $h > 1$

$s(\omega) = \{[h\ 1 \text{ --- }]\}\ [s(\alpha)]^{h-1}$ for known h and 1,

$s(\omega) = \{[\ **\ *\ \text{ --- }]\}\ [s(\alpha)]^+$ for dynamical case.

Symbols true and false might be renamed proceeding from the actual type system (it is the implementation level problem). Constants h and 1 are usually unknown in sense of type system - it leads us to introduce free ("wildcard") types * and **.

Example. Let us have a Forth-program

BEGIN SWAP OVER WHILE NOT REPEAT

We use the following alphabet and specifications:

$$A = \{ T, F \}$$

$$s(SWAP) = [\ **\ *\ ---\ *\ **\ ] = \left\{ \begin{array}{l} [\ F\ F\ ---\ F\ F\ ] \\ [\ F\ T\ ---\ T\ F\ ] \\ [\ T\ F\ ---\ F\ T\ ] \\ [\ T\ T\ ---\ T\ T\ ] \end{array} \right\}$$

$$s(OVER) = [\ **\ *\ ---\ **\ *\ **\ ] = \left\{ \begin{array}{l} [\ F\ F\ ---\ F\ F\ F\ ] \\ [\ F\ T\ ---\ F\ T\ F\ ] \\ [\ T\ F\ ---\ T\ F\ T\ ] \\ [\ T\ T\ ---\ T\ T\ T\ ] \end{array} \right\}$$

$$s(NOT) = \left\{ \begin{array}{l} [\ F\ ---\ T\ ] \\ [\ T\ ---\ F\ ] \end{array} \right\}$$

ω = BEGIN SWAP OVER WHILE NOT REPEAT

To apply the formula for the WHILE-cycle it is useful to pre-calculate some products :

$$s(SWAP\ OVER) = s(SWAP)\ s(OVER) = \left\{ \begin{array}{l} [\ F\ F\ ---\ F\ F\ F\ ] \\ [\ F\ T\ ---\ T\ F\ T\ ] \\ [\ T\ F\ ---\ F\ T\ F\ ] \\ [\ T\ T\ ---\ T\ T\ T\ ] \end{array} \right\}$$

$$s(SWAP\ OVER)\ ([\ T\ ---\ ]) = \left\{ \begin{array}{l} [\ F\ T\ ---\ T\ F\ ] \\ [\ T\ T\ ---\ T\ T\ ] \end{array} \right\}$$

$$s(SWAP\ OVER)\ ([\ T\ ---\ ])\ s(NOT) = \left\{ \begin{array}{l} [\ F\ T\ ---\ T\ T\ ] \\ [\ T\ T\ ---\ T\ F\ ] \end{array} \right\}$$

$$s(SWAP\ OVER)\ ([\ F\ ---\ ]) = \left\{ \begin{array}{l} [\ F\ F\ ---\ F\ F\ ] \\ [\ T\ F\ ---\ F\ T\ ] \end{array} \right\}$$

Now we may calculate the program specification $s(\omega)$:

$$s(\omega) = [\ s(SWAP\ OVER)\ ([\ T\ ---\ ])\ s(NOT)\ ]^*\ s(SWAP\ OVER) \cdot$$

$$\cdot([\ F\ ---\ ]) = \left\{ \begin{array}{l} [\ F\ T\ ---\ T\ T\ ] \\ [\ T\ T\ ---\ T\ F\ ] \end{array} \right\}^* \cdot \left\{ \begin{array}{l} [\ F\ F\ ---\ F\ F\ ] \\ [\ T\ F\ ---\ F\ T\ ] \end{array} \right\} =$$

$$= \left(1 + \left\{ \begin{bmatrix} F & T & --- & T & T \\ T & T & --- & T & F \end{bmatrix} \right\}^{+} \right) \cdot \left\{ \begin{bmatrix} F & F & --- & F & F \\ T & F & --- & F & T \end{bmatrix} \right\} =$$

$$= \left\{ \begin{bmatrix} --- \\ F & T & --- & T & T \\ F & T & --- & T & F \\ T & T & --- & T & F \end{bmatrix} \right\} \cdot \left\{ \begin{bmatrix} F & F & --- & F & F \\ T & F & --- & F & T \end{bmatrix} \right\} =$$

$$= \left\{ \begin{bmatrix} F & F & --- & F & F \\ F & T & --- & F & T \\ T & F & --- & F & T \\ T & T & --- & F & T \end{bmatrix} \right\}$$

Immediate check gives just the same result.

It may happen that some closure has infinite number of elements. In that case the program is not correct in sense of our definitions. At the same time an algorithm to detect this kind of incorrectness has to finish its work.

References

[NP70] Nivat M., Perrot J.F. Une généralisation du monoïde bicyclique. - *C. R. Acad. Sci. Paris*, 271A, 1970, p.824-827.

[P90] Pöial J. Algebraic specifications of stack-effects for Forth-programs. - *EuroFORML'90, Oct.12-14, 1990, Proceedings*. Ampfield, U.K., 1990, 8 pp.