

# Feasible C

(long abstract)

Salvatore Caporaso

Dipartimento di Informatica dell'Università di Bari

caporaso@di.uniba.it

## Abstract

Sometimes ICC is criticized for its limited role in real life programming. I introduce here a language FC (*feasible C*), which is obtained by means of a few restrictions to C. Its syntax only allows polytime programs; so feasibility is controlled at compile time. Many C programs can be transformed uniformly into FC programs, by simply prefixing some directives (pre-processing instructions). This has been checked on most examples and exercises contained in the celebrated textbook by Kernighan & Ritchie, and, in particular, to all recursion schemes they code. However, we cannot decide at compile time the feasibility of *all* recursion schemes. The price of extending FC to more schemes is that, in these cases, we can only supply a runtime warning, at certain critical points, where feasibility appears to be at risk. To handle this I apply my recent revisitation in C of the Kleene-Rogers theorem (arXiv:0712.1279 v1 [cs.LO]).

**1 Notation**  $A, \dots, Q$  are (parts of) C programs. A *configuration*  $K$  is a set of assigned C variables.  $|K|$  and  $|P|$  are the sizes of  $P$  and  $K$ .

$$|P| \leq |K| + \varphi(n)$$

means that  $P$  *adds* at most  $\varphi(n)$  characters to any  $K$  of length  $n$ . A function definition is  $m$ -adic if  $m$  calls to the function being defined occur in it. The (*syntactic*) *complexity*  $co(P)$  of  $P$  is  $(f, w, u, p)$  if it is built-up by means of: **f** for, **w** while, **u** monadic and **p** polyadic function definitions.  $w \dots$  are identifiers reserved for certain variables, called *watchdogs*, whose lengths don't contribute to the current  $|K|$ .

**2 Runtime of un-nested for** Assume  $cmpl(P)=(1,0,0,0)$ . We then have  $P = A; \text{for}(B; T; S) I; Z$ . Assume further  $A, Z$  absent. We obviously have

$$|K| [P] |K| + |P|p$$

where  $p$  is the number of times the *invariant*  $I$  is repeated. Now, define  $Q=$

```
(1)   for(B, w1 = w0; T && w1; S, w1--) I;
```

and assume that the input length  $|n|$  is assigned to  $w0$ . In this case,  $I$  is repeated at most  $n$  times (since, in the worst case, the *test*  $T, \&\& w1$  becomes false after  $n$  steps  $S, w1--$ ). So, we have

$$|K| [P] |K| + |P|n$$

The  $+$  occurring in this rather obvious statement plays an essential role in my discourse.

**3 Feasible C** In FC all input is carried out by introducing one string at a time from the *stdin* by means of the library functions `scanf`, `strlen`, and is in the form

```
scanf("%s\n", s); w0 += strlen(s);
```

All `for` constructs are in the form (1). One easily proves the

**4 Theorem** In FC  $cmpl(P)=(c,0,0,0)$  implies

$$|K| [P] |K| + |P|n^c.$$

**5 Repetitions and monadic definitions** `while` constructs and unary function definitions are in the form

```
w1=w0; while(T && w1--) I;
w1=w0; fn(...){if(w1) then w1--; else return; ...}
```

By essentially the same proof one shows that  $cmpl(P)=(c,d,u,0)$  implies

$$|K| [P] |K| + |P|n^{c+d+u}.$$

For example, we have  $cmpl(IQ)=(1,1,0,0)$  for iterative *quicksort* in FC.

**6 Note** Taking any C program  $P$  ( $cmpl(P)=(c,w,u,0)$ ) into a FC program of the same complexity can be carried-out uniformly, by adding a few directives to  $P$ . If  $P$  is polytime, then its translation in FC behaves like  $P$ .

**7 A better estimate** of runtime is obtained by considering the nesting depth of the constructs instead than their overall number. By updating the watchdog with the current configuration length, instead of the input length one gets simpler programs at the price of the worst estimate

$$|K| [P] |K| + 2^{|P|} n^{2^{|P|}}.$$

**8 Simple polyadic definitions** Two polyadic recursion schemes can be handled without difficulty. A  $k$ -adic recursion by *segments* is in the form ( $s$  is a string; the  $a$ 's and  $b$ 's are pointers)

```
fn(...){... fn(... a1,b1);... fn(a2,b2);... fn(ak,bk);...}
```

For example, in recursive quicksort one writes

```
qs(first,last){... qs(first,i--);... qs(i++,last);}
```

This can be handled by applying to the *intermediate parameters* some min / max operators, ensuring that all calls are not overlapping, and, therefore, they do not, so to say, *create new resources*. (`qs` is already ok because `i--` is less than `i++`.)

Course-of-values recursion on trees is frequently used, for example by parsers. It keeps feasible insofar as the sub-trees passed as parameters do not overlap. This can be ensured by strictly syntactical checks. Recall those fun Lisp names like `caaddr`, `caadr`, `caddadr`, etc. The first two are not *safe* because either of them is a prefix of the other, and, therefore, an overlapping occurs.

**9 Feasibility of *all* recursion schemes** cannot be checked at compile time. By recording in a stack all calls, one may issue a runtime warning when a potentially exponential duplication of resources is detected. To implement this, rather than relying on the compiler's way of dealing with recursive definitions, I am using a revisitation in C of the proof of the Kleene-Rogers theorem  $\phi_n = \phi_{\phi_x(n)}$  (arXiv:0712.1279 v1 [cs.LO]).