

Feasible ICC C and Other ICC C's

(si six scies scient six cyprès six cents six scient six
scient six cents six cyprès)

Salvatore Caporaso & Vittoria Cozza

(http://www.di.uniba.it/~scaporaso/Feasible_C)

- Humans build-up hard concepts from *names*

Pink Whales Riding Jellow Submarines

Brutes do not

- C performs hard tasks with *pointers*

```
for ( a ; a ; a -- ) I
```

is PR with parameter substitution

is exp-time in $|x|$

hence feasible if a pointer

Pascal does not

PRN (notations) is not true life programming

PR is

Syntax of Data and Expressions in Feasible C

Types scalars: `char`, `unsigned`, pointer to

arrays: of scalars and of strings

unlike C, potentially infinite arrays and pointers

I/O `scanfn(s, a)` is a built-in equivalent to

```
scanf("%s", s); a=strlen(s);
```

`printf` as usual

expr `++`, `--`, `+`, `-`, `*`, `/`, `%` on `int`

`-`, `++`, `--` on pointers

`-`, `+`, `/`, `%` on pointer (1st) and `int` (2d)

all like in C, but `--` and `-` are cut-off

Loops in FC

```
for(a,B; a && T; a--,S) I
```

Symmetrically

```
for(a,B; a<= b&&T ;a++,S) I
```

Moreover

```
while(a){I;a--;} I
```

```
while(a<=b){I;a++;} I
```

a and b are the **main** variable(s).

$f(a, b, \dots)$ is defined by

m -fold **recursion** ($m \geq 1$) with parameter substitution
in the *main* variables a, b with k parameters \vec{p}
and in the (not recursive) substitution functions \vec{g}

$$\begin{aligned} f(a, b, \vec{p}) \{ \\ & \dots \\ & \vec{q} = \vec{g}(\vec{p}); \\ & a_1 = a + e_1 + 1; \\ & a_2 = a_1 + e_2 + 1; \\ & \dots \\ & a_{2m} = a_{2m-1} + e_{2m}; \\ & \text{if}(a_{2m} > b) \text{return ERR_REC_CALL}; \\ & f(a_1, a_2, \vec{q}); \\ & \dots \\ & f(a_{2m-1}, a_{2m}, \vec{q}); \\ \} \end{aligned}$$

for $m = 1$, we may put either $a_1 = a$ or $a_2 = b$ (PR)

for $m > 2$, we may put both equalities

quicksort

For $m = 2, k = 0, a_1 = a; a_4 = b$, define

```
qs(left, right){  
    the same invariant as in K&R, then  
    a1 = left; a2 = a1 + i - 1;  
    a3 = a2 + 2; a4 = right;  
    if(a4 > right) return ERR_REC_CALL;  
    qs(a1, a2);  
    qs(a3, a4);  
}
```

Safety Restriction (**not surprising**)

A (part of) program is **flat**

if it doesn't occur in any iteration or function

Thou shalt assign main variables at flat places only

Validation All examples and all exercises in K&R can be reduced to FC by trivial changes.

One might even think about a translation from C to FC by pre-processing directives.

One proves by straightforward induction on $|P|$ the

Theorem Runtime R of any P on any x

$$R \leq qn^q \quad (q = 2^p; p = |P|; n = |x|)$$

Moreover:

for an appropriate definition of *loop nesting* $\nu(P)$

if the form of P is $Q \ R$ where Q is flat and

R is a loop possibly with other loops in its invariant

$$R \leq pn^c \quad (c = \nu(|P|); n = |x|)$$

Because if P is flat it may **add** at most $p = |P|$ char's within p steps. Hence n repetitions add np within np . So two nested `for` add pn^2 and so on. For

$$f(a, b) \{ \dots f(a_1, a_2); f(a_3, a_4); \}$$

a slightly less trivial proof by induction on $|f|$ and n using the fact that in FC a loop *adds* information that does not contribute to the iterations number

Elementary and Primitive Recursive C

With one offence to the **Law** you are deported to

$$\mathcal{E} = \mathcal{E}_3.$$

Assume that Q is flat and assigns variable c . Define

$$q = |Q|, n = |s| \text{ and}$$

```
P = scanfn(s, a);
    b = c = a;
    for1(a; a; a --;) {
        for2(b; b; b --;)
            Q;
        b = c; }    inside loop 1
```

for $a = n$ for₂ repeats Q for n times adding $\leq qn$;

for $a = n - 1$ for₂ repeats Q for qn times adding $\leq q^2$;

for $a = 1$ for₂ repeats Q for $\leq q^n$ times.

with k un-nested offences you keep in $D\text{TIME}(n_k(n^c))$

because

$$(n_a)^{n_b} \quad \text{grows like} \quad n_{1+\max(a,b)}$$

k nested offences precipitate you **down to** \mathcal{E}_{k+2}

```
P = scanfn(s, a);
    b1 = b2 = ... = bk = a;
    for(a; a; a --; ){
        for1(b1; b1; b1 --; ){
            ...
            fork(bk; bk; bk --; )
                Q;
                bk = c; }    inside a loop
            ...
            b1 = c; }    inside a loop
```

Provable C - Position of the Problem

Cobham's polytime =

closure under PRN_{\leq} (**limited** PR on notations) and sbst
of triv. init. funct's plus *ad hoc* funct'n **smash** #

Transfinite Grzegorzcyk **class of functions** \mathcal{E}_{α} introduced
first defining a transfin. sequence of **hierarchy functions**
like the Wainer-Schwichtenber *fast hierarchy*

$$F_{\alpha+1}(n) = F_{\alpha}^{n+1}(n) \quad F_{\lambda}(n) = F_{\lambda(n)}(n)$$

second closure under PR_{\leq} and sbst

This conflicts with almost all ICC principles

- ad hoc initial functions;
- limited operators;
- membership undecidable from the syntax;
- coarse and greedy to the point of *eating* the time complexity classes

Moreover $F_{\lambda}(n) = F_{\lambda(n)}$ is def'n — not algorithm

In the ϵ_0 -type hierarchy of C fragments we now introduce

- each class is defined by **unlimited** operators.

So, **no** special function and no limited operators.

Hierarchy functions are just a scale against which we measure **a posteriori** our classes

- time and provable hierarchies are **harmonized** by assigning low ordinals to poly- and exp-time

Syntax of Provable C

Define a **coding** α^* of the ordinals α in the strings by

$$0^* = Z \quad (\alpha + \beta)^* = P\alpha^*\beta^* \quad (\omega^\alpha)^* = 0\alpha^*$$

(Polish prefix with $+^* = P^{(2)}$; $0^{(1)} = \omega^*$ and $Z^{(0)} = 0^*$)

$$\omega^{\omega^2} + \omega^{\omega^2} + 2^* = +00 + 0Z0Z + 0 + 0Z0Z + 0Z0Z$$

Assume defined a **library** `provable.h` that includes

- a typedef for strings `ord_t` coding the ordinals
- functions `lim`, `fs`, `scc`, `prd` for the properties and operations on the ordinals
they name suggest

f is defined by ordinal-while scheme at λ in Q, R

if its form is

```
fn_(a) {
  ord_t s =  $\lambda^*$ ;
  while(!s) {
    if (lim(s)) s = fs(s, a); Q
    else (scc(s)) {s = prd(s, a); R}
  }
}
```

where

Q, R are programs in which calls to f may occur
submitted to some restrict's (feasible, elem., PR ...)

EXAMPLE

if Q identity and R is $a++$

$f(a)$ computes $H_\lambda(a)$ (fast growing Hardy)

if R is $f(a)++$ then

$f(a)$ computes $G_\lambda(a)$ (slow growing hierarchy)

To get the Wainer-Schwichtenberg fast hier. take as R

$b = a; \text{ for } (b; b; b--)\ a = f(a);$

To measure the syntactical complexity of \mathbb{P}
 assume \mathbb{P} defined by one of our schemes Σ in \mathbb{Q}
 define the *ordinal* $o(\mathbb{P})$ of \mathbb{P} by $o(\mathbb{P}) =$

$$\left\{ \begin{array}{ll} 1 & \mathbb{P} \text{ is flat} \\ 1 + o(\mathbb{Q}) & \text{if } \Sigma \text{ is a feasible scheme} \\ \omega + o(\mathbb{Q}) & \text{if } \Sigma \text{ is an elementary scheme} \\ \beta + \omega^{\gamma+1} & \Sigma \text{ is PR and } o(\mathbb{Q}) = \beta + \omega^{\gamma} \\ \lambda & \text{if } \Sigma \text{ is } \lambda\text{-while, with } \mathbb{Q}, \mathbb{R} \text{ flat} \end{array} \right.$$

Define a hier. of (funct's computed by) \mathbb{C} fragments

$$\mathcal{C}_{\alpha} = \text{funct's computed by some } \mathbb{f} \text{ s.t. } o(\mathbb{f}) = \alpha$$

Theorem (Harmonizing the time and provable hierarchies)

1. $\text{DTIME}(n^c) = \mathcal{C}_c$
2. $\text{DTIME}(n_c) = \mathcal{C}_{\omega c}$
3. $\text{DTIME}(F_{\alpha}(n)) = \mathcal{C}_{\omega^{\alpha}}$

to believe part 3.

assume nxt_M is flat simulation of one step by M

assume a points to tape T

define

```
sim_M(a) {  
  ord_t s =  $\lambda^*$ ;  
  while(!s) {  
    if (lim(s)) s = fs(s, a);  
    else {  
      s = prd(s, a);  $\text{nxt\_M}$ ; a++;  
    }  
  }  
}
```

One sees that sim_M

simulates $H_\lambda(|T|)$ steps by M on T ,

and we have $H_{\omega^\alpha} = F_\alpha$

parts 1. and 2. follow by applying to nxt_M

some of our previous arguments