Certifying and reasoning about cost annotations of functional programs

Roberto M. Amadio

Université Paris Diderot

Joint work with Yann Régis-Gianas,
in the framework of the EU Project CerCo,
2010-2013, U. Bologna, Edinburgh, and Paris.

Motivating question for (half of) this audience

How could ICC have an impact on programming practice?

For the other half, think of the question:

> How could computable analysis have an impact on numerical computation?

Requirements

1. An area where people **care** about resource bounds.

2. Bounds should be provided for the **standard** programming environment (for the area).

3. Bounds should be **concrete and certified** and possibly **automated**.

A success story: WCET for embedded programs

- Nowadays programs running on embedded processors are often written in C and then compiled to assembly.
- **Standard technology** estimated WCET by testing.
- **New technology** (*e.g.* AbsInt for EADS) builds an abstract model of the processor that predicts WCET.
- Validation is peformed by extensive **testing**: reject models which fail one test or are too far from reality.

CerCo's goals

- Build a **certified compiler** for C that given a program produces valid (and precise) upper bounds on the **cost** of executing each **block** (a loop free fragment of the code) on a given architecture.

- Use the derived information on the cost of the blocks to produce a **synthetic and certified** assertion on the **cost of executing the program**.

This talk

- Recall results obtained for a C compiler.
- Discuss how these results could be lifted to a compiler for a **higher-order functional language**.

Recap previous work on C

(FMICS 2012, with N. Ayache and Y. Régis-Gianas)

A compilation chain from C to binary

$$
\begin{array}{ccccccccccc}
 & & \mathsf{C} & \rightarrow & \mathsf{Clight} & \rightarrow & \mathsf{Cminor} & \rightarrow & \mathsf{RTLAbs} & & \text{(front end)} \\
 & & & & & & & & \downarrow & & \\
\mathsf{8051, Mips} & \leftarrow & \mathsf{LIN} & \leftarrow & \mathsf{LTL} & \leftarrow & \mathsf{ERTL} & \leftarrow & \mathsf{RTL} & & \text{(back-end)}
\end{array}
$$

- Moderate optimisations: instruction selection, register allocation, dead-code elimination,... (a bit more efficient than gcc0 for Mips).
- Architecture is quite close to and inspired by CompCert.
- Prototype compiler used in a master-level course on Compilation.

Main steps

- All languages are enriched with **labelled instructions** generating **labelled transitions**.
- The **erasure functions** are just functions that remove all labelling instructions.
- The compilation functions are **extended** to the labelled languages.

$$
\begin{array}{ccc}
 & \mathcal{C} & \\
\mathsf{C}^{\ell} & \rightarrow & \mathsf{Binary}^{\ell} \\
\downarrow er & & \downarrow er \\
\mathsf{C} & \rightarrow & \mathsf{Binary} \\
 & \mathcal{C} &
\end{array}
$$

> Prove that the compiled code **simulates** the source code (with labelled transitions) and that the compilation **commutes** with the label erasure.

- A **labelling** of the source language is just a function $\mathcal{L}$ which is the right inverse of the erasure function $er \circ \mathcal{L} = id$.

  A **good labelling** is a labelling that guarantees that we can associate a cost $\mathsf{cost}(\ell)$ to each label ℓ so that:

> If $\mathcal{C}(\mathcal{L}(P)) \Downarrow \ell_1 \cdots \ell_n$ then the cost of running $\mathcal{C}(P)$ is $\mathsf{cost}(\ell_1) + \cdots + \mathsf{cost}(\ell_n)$.

$$
\begin{array}{ccc}
 & \mathcal{C} & \\
\mathsf{C}^{\ell} & \rightarrow & \mathsf{Binary}^{\ell} \\
\mathcal{L} \uparrow \downarrow er & & \downarrow er \\
\mathsf{C} & \rightarrow & \mathsf{Binary} \\
 & \mathcal{C} &
\end{array}
$$

- Define an **instrumentation function** $\mathcal{I}$ that replaces each label ℓ with an increment $\mathsf{cost} = \mathsf{cost} + \mathsf{cost}(\ell)$ of a global variable. Then we have:

If $\mathcal{I}(\mathcal{L}(P)) \Downarrow c$ then the cost of running P is (bounded by) c.

$$
\begin{array}{ccc}
\mathsf{C} & & \\
\uparrow \mathcal{I} & \mathcal{C} & \\
\mathsf{C}^{\ell} & \rightarrow & \mathsf{Binary}^{\ell} \\
\mathcal{L} \uparrow \downarrow er & & \downarrow er \\
\mathsf{C} & \rightarrow & \mathsf{Binary} \\
& \mathcal{C} &
\end{array}
$$

NB Because $\mathcal{I}(\mathcal{L}(P))$ is again a program in the source language (C), we can use tools built for C to reason on the cost of the program (in our case Frama-C, CEA).

- We developed heuristics to **generate automatically annotations** on the cost of running a C function as a function of the size of the input.
- The generated annotations are then passed to Frama – C that tries to prove them. The process is completely automatic for **simple programs**:
  - C programs generated by a Lustre compiler (no loops!).
  - Functions with simple loops (sorting, basic cryptographic functions).

  NB If the automatic mechanism fails, the user can still rely on a general purpose environment to reason about the complexity of the compiled program.

Cost plugin in action (1/2)

```
int is_sorted (int *tab, int size) {
  int i, res = 1;
  for (i = 0 ; i < size-1 ; i++)
    if (tab[i] > tab[i+1])
      res = 0;
  return res;
}
```

Cost plugin in action (2/2)

```
int _cost = 0;
/*@ ensures (_cost ≤ \old(_cost)+(101+(0<size-1?(size-1)*195:0))); */
int is_sorted (int *tab, int size) {
  int i, res = 1, _cost_tmp0;
  _cost += 97; _cost_tmp0 = _cost;
  /* @ loop invariant (0 < size-1) ⇒ (i ≤ size-1);
     @ loop invariant (0 ≥ size-1) ⇒ (i ≡ 0);
     @ loop invariant (_cost ≤ _cost_tmp0 + i * 195);
     @ loop variant (size-1)-i; */
  for (i = 0; i < size-1; i++) {
    _cost += 91;
    if (tab[i] > tab[i+1]) { _cost += 104; res = 0; }
    else _cost += 84; }
  _cost += 4;    return res;   }
```

Some experiments

File	Type	Description	LOC	VCs
`3-way.c`	C	Three way block cipher	144	34
`a5.c`	C	A5 stream cipher, used in GSM cellular	226	18
`array_sum.c`	S	Sums the elements of an integer array	15	9
`fact.c`	S	Factorial function, imperative implementation	12	9
`is_sorted.c`	S	Sorting verification of an array	8	8
`LFSR.c`	C	32-bit linear-feedback shift register	47	3
`minus.c`	L	Two modes button	193	8
`mmb.c`	C	Modular multiplication-based block cipher	124	6
`parity.lus`	L	Parity bit of a boolean array	359	12
`random.c`	C	Random number generator	146	3
S: standard algorithm C: cryptographic function L: C generated from a Lustre file				

Additional work

- Partially (!) **machine-checked proof** with Matita (a variant of Coq) of the whole compilation chain (Bologna+Edinburgh).
- Adapt the methodology to more advanced **loop optimisations** that duplicate code and thus labels (P. Tranquilli).
- Handle **programs with pointers** (F. Bobot).

The whole approach relies on the hypothesis that the WCET of a block is rather precisely **predictable**.

Looking at functional programs

Extended abstract in FOPARA 2012

Long version available in HAL and to appear in

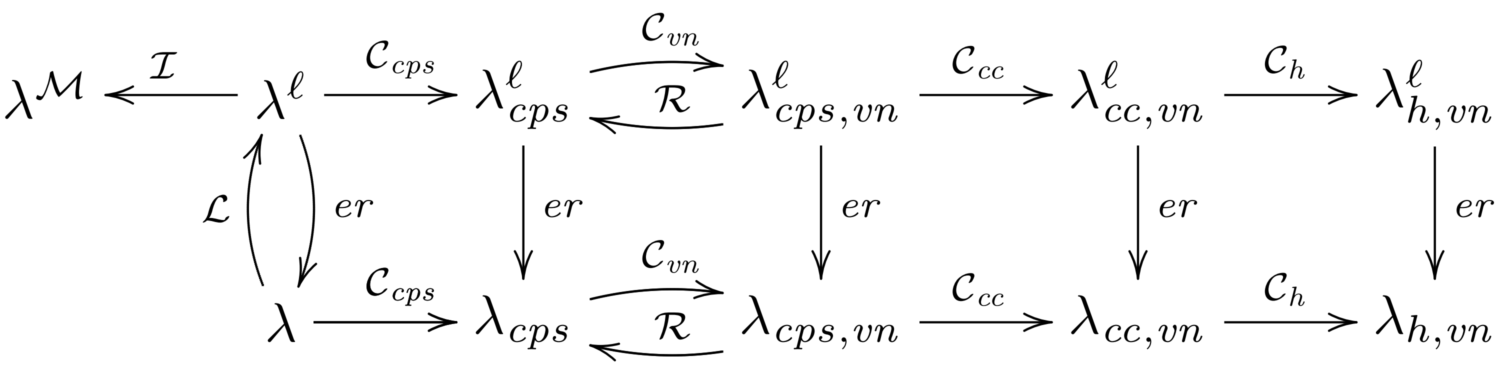
Higher order and symbolic computation.

Common wisdom

A Lisp programmer knows the **value of everything**,
but the **cost of nothing**.
A. Perlis

We **question** this common wisdom following the approach described for C. So far a **thought experiment** not targeting any particular application scenario.

Overall picture



The target language is essentially isomorphic to the RTLAbs language considered in the C compiler. We'll call it RTL λ-calculus for short. Starting from there we rely on the back-end of the C compiler.

NB Similar compilation chains studied by Morriset *et al.* 1999 (typing preservation) and Chlipala 2010 (simulation proofs in Coq).

Main issues

1. What is a **good labelling** for programs?
2. How do we **instrument** programs?
3. How do we **reason** on the instrumentation?
4. How do we account for the **cost of heap management**? (something we did not do for C).

Good labelling

- What is the source labelled language?
- Where do we put the labels?

Explication **by example**. . .

Source code: function composition

```
fun (f,g) ->
  fun (x) ->
    f(g(x))
```

CPS code

```
halt (fun (f,g,k) ->                    (* halt initial continuation *)
  k(fun (x,k) ->
    g(x,(fun x ->
      f(x,k)))))
```

CPS value named code

```
let x_1 = fun (f,g,k) ->                (* a name for each value *)
  let x_2 = fun (x,k) ->
    let x_3 = fun (x) ->  f (x,k)       (* tcall 3 *)
    in g (x,x_3)                        (* tcall 2 *)
  in k(x_2)                             (* tcall 1 *)
in halt(x_1)                            (* main *)
```

Closure conversion

```
let c_1 = fun (e_1,f,g,k) ->          (* fun 1, fv = empty *)
   let c_2 = fun (e_2,x,k) ->         (* fun 2, fv = {f,g} *)
      let (f,g)= e_2 in
      let c_3 = fun (e_3,x) ->        (* fun 3, fv = {f,k} *)
         let (f,k) =  e_3 in
         let (c,e) = f in
         c(e,x,k)  in                 (* tcall 3 *)
      let e_3 = (f,k) in
      let x_3 = (c_3,e_3) in
      let (c,e)= g in
      c(e,x,x_3)  in                  (* tcall 2 *)
   let e_2 = (f,g) in
   let x_2 = (c_2,e_2) in
   let (c,e)= k in
   c(e,x_2)  in                       (* tcall 1 *)
let e_1 = () in                       (* main *)
let x_1 = (c_1,e_1) in
halt(x_1)
```

Hoisted code (RTL level)

```
let c_3 = fun (e_3,x) ->              (* fun 3 *)
   let (f,k) =  e_3 in
   let (c,e) =  f in
   c(e,x,k)  in
let c_2 = fun (e_2,x,k) ->            (* fun 2 *)
   let (f,g) = e_2 in
   let e_3 = (f,k) in
   let x_3 = (c_3,e_3) in
   let (c,e) = g in
   c(e,x,x_3)
let c_1 = fun (e_1,f,g,k) ->          (* fun 1 *)
   let e_2 = (f,g) in
   let x_2 = (c_2,e_2) in
   let (c,e) = k in
   c(e,x_2)  in
let e_1 = () in                       (* main *)
let x_1 = (c_1,e_1) in
halt(x_1)
```

Labelled hoisted code (RTL level)

```
let c_3 = fun (e_3,x) ->  LAB3>    (* fun 3 *)
   let (f,k) = e_3 in
   let (c,e) = f in
   c(e,x,k)  in
let c_2 = fun (e_2,x,k) -> LAB2>   (* fun 2 *)
   let (f,g) = e_2 in
   let e_3 = (f,k) in
   let x_3 = (c_3,e_3) in
   let (c,e) =  g in
   c(e,x,x_3)
let c_1 = fun (e_1,f,g,k) -> LAB1> (* fun 1 *)
   let e_2 = (f,g) in
   let x_2 = (c_2,e_2) in
   let (c,e) =  k in
   c(e,x_2)  in
LAB0 > let e_1 = () in             (* main *)
let x_1 = (c_1,e_1) in
halt(x_1)
```

Back to labelled CPS

```
LAB0> halt (fun (f,g,k) -> LAB1>
  k(fun (x,k) ->  LAB2>
    g(x,(fun x -> LAB3>
      f(x,k)))))
```

And to labelled source

```
LAB0>fun (f,g) -> LAB1>
      fun (x) ->  LAB2>
      f(g(x)> LAB3)                (* post-labelling *)
```

The good initial labelling

The source language has **two labelling instructions**:

- $\ell > M$: emits ℓ before reducing M (**pre-labelling**)
- $M > \ell$: reduces M to a value and then emits ℓ (**post-labelling**).

The good initial labelling associates a **distinct**:

- **pre-labelling** to every function abstraction.
- **post-labelling** to every application which is not immediately sourrounded by an abstraction.

The 'post-labelling' takes care of the **functions created by the CPS translation** while ensuring the **commutation property** (which would fail if we considered $M > \ell$ as syntactic sugar for $(\lambda x.\ell > x)M$).

Instrumentation

In C we add a 'cost variable', but we would rather stay in the **functional world**. We rely on a simple **monadic transformation** (Gurr).

$$
\begin{array}{lcl}
\psi(x) & = & x \\
\psi(\lambda x.M) & = & \lambda x.\mathcal{I}(M) \\
 & & \\
\mathcal{I}(V) & = & (\mathbf{0}, \psi(V)) \\
\mathcal{I}(@(M,N)) & = & \mathsf{let}\ (m_0, x_0) = \mathcal{I}(M), (m_1, x_1) = \mathcal{I}(N), (m_2, x_2) = @(x_0, x_1)\ \mathsf{in} \\
 & & (m_0 \oplus m_1 \oplus m_2, x_{n+1}) \\
\mathcal{I}(\ell > M) & = & \mathsf{let}\ (m, x) = \mathcal{I}(M)\ \mathsf{in}\ (m_\ell \oplus m, x) \\
\mathcal{I}(M > \ell) & = & \mathsf{let}\ (m, x) = \mathcal{I}(M)\ \mathsf{in}\ (m \oplus m_\ell, x)
\end{array}
$$

If $\pi_1(\mathcal{I}(\mathcal{L}(M))) \Downarrow m$ then m is the cost of running M.

Reasoning on the instrumentation

We rely on a higher-order Hoare logic (Régis-Gianas & Pottier 2008).

1. **Annotate** the functional program with logic assertions.
2. **Compute** a set of proof obligations implying the validity of these assertions.
3. **Prove** these proof obligations.

Reasoning, in practice

The monadic interpretation of the functional program is **not** human-friendly.

- Logic assertions are written **directly on source code** as if the program *was* in monadic form.
- An implicit variable `cost` is automatically added to the logical environment.
- The monadic transformation is applied just before the Verification Condition Generator.

The cost of a higher-order function

```
type list = Nil | Cons (nat, list)          type bool = BTrue | BFalse
logic {
 Definition bound (p : nat --> (nat * bool)) (k : nat) : Prop :=
    forall x m: nat, forall r: bool, post p x (m, r) => m <= k.
 Definition k0 := costof_lm + costof_lnil.
 Definition k1 := costof_lm + costof_lp + costof_lc + costof_lf + costof_lr.
}
let rec pexists (p : nat -> bool, l: list) { forall x, pre p x } : bool {
 ((result = BTrue) <=> (exists x c: nat, mem x l /\ post p x (c, BTrue))) /\
 (forall k: nat, bound p k /\
                (result = BFalse) => cost <= k0 + (k + k1) * length (l))
} = _lm> match l with
  | Nil -> _lnil> BFalse
  | Cons (x, xs) -> _lc> match p (x) > _lp with
                   | BTrue -> BTrue
                   | BFalse -> _lf> (pexists (p, xs) > _lr)
```

Of 53 proof obligations, 46 are discharged automatically and 7 proved in Coq.

Account for the cost of heap management

Non-solution 'Real-time' GC (see Bacon *et al.* 2003).

How do you go from an **experimental and amortized O(1) cost** to a **proved and useful O(1) WCET cost**?

Chosen approach **Type and effect** system to guarantee **safe deallocation** in **constant time**.

A very important property of our implementation scheme is that programs are executed 'as they are written', with no additional costs of unbounded size (...). The memory management directives which are inserted are each constant time operations.
Tofte and Talpin 1997.

This amounts to add **one more step** to a **typed** compilation chain.

Typing of the compilation chain

- Typing of **CPS** is preserved by a standard **double negation translation**.
- Typing of the **closure conversion** relies on the introduction of **existential types** to hide the details of the environment representation (Hannah, Minamide *et al.* 95-96).
- **Value naming** and **hosting transformations** do not affect the typing.

A λ-term typed with **simple types** compiles to a
RTL λ-term typed with **simple and existential types**.

A region enriched RTL λ-calculus

Additional operations:

- Allocate a region.
- Allocate a value to a region.
- Dispose a region (with all the values allocated there).
- Region abstraction and application.

These operations correspond to simple sequences of instructions which are inserted by the compiler. The labelling technology takes their cost into account automatically.

A type and effect system

In the region enriched RTL λ-calculus **types depend on regions** (and effects).

RTL types	Regions enriched RTL types
$\mathbf{1}$	$\mathbf{1}$
$A \to R$	$\forall r_1, \ldots, r_n . A \xrightarrow{e} R$
$A \times B$	$(A \times B)\mathsf{at}(r)$
$\exists t.A$	$(\exists t.A)\mathsf{at}(r)$

The type and effect system guarantees that when disposing a region r: (i) no value allocated in r is accessed and (ii) no further disposal of the region r occurs in the rest of the computation.

Compilation as type inference

- The last compilation step amounts to **infer region allocations and deallocations** which are legal according to the **type and effect** system.
- A trivial solution is **always possible**.
- We rely on previous work (Aiken *et al.* in particular, PDLI 2005), for effective methods based on constraint solving to find **more interesting solution**.

Summary for the functional case

Good labelling Done.

Instrumentation Done.

Reasoning Requires (more) user interaction.

Cost of heap management Region-based so far.

Tentative conclusion

- The approach developed for C can be lifted to a ML-like language.
- It remains to be seen whether there is any **real interest** in bounding the resources of an ML-like language (our requirement 1). So far this is a **thought experiment**.
- Going on with the thought experiment, there should be a **connection with ICC** (seen as a **provider** of synthetic bounds for (higher-order) functional programs).
- Notice that the integration of ICC insights rises **non-trivial problems**. Bounds should be:
  - **tight and concrete**.
  - **robust** enough to be propagated down the compilation chain and be machine checked.