Certifying and reasoning about cost annotations of functional programs

Roberto M. Amadio Y. Régis-Gianas

Université Paris Diderot

Project FP7-ICT-2009-C-243881 CerCo

References

Extended abstract in FOPARA 2012

Long version (60 pages) available in `arXiv`.

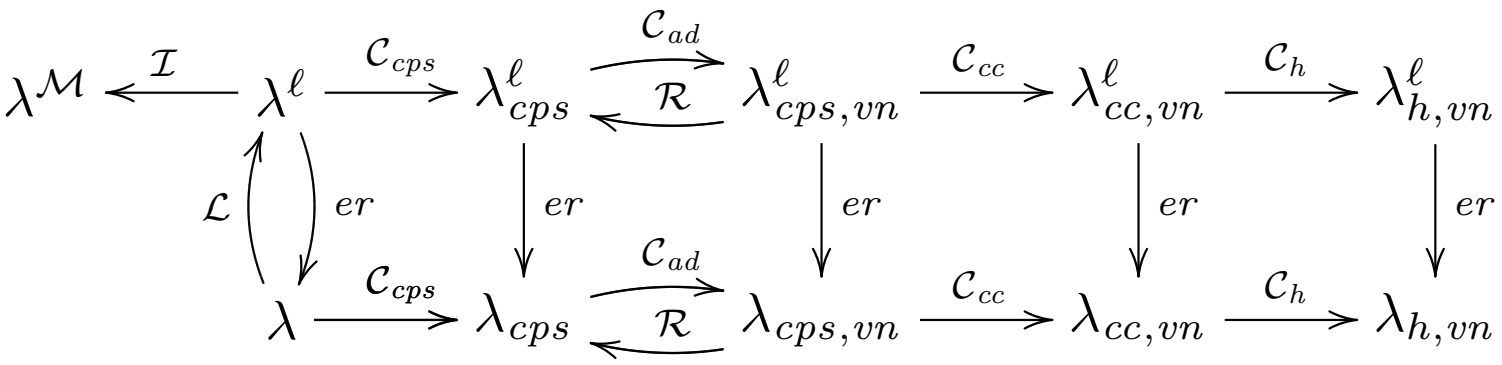
Software:
`http://www.pps.univ-paris-diderot.fr/~yrg/fun-cca/`

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).

$$
\begin{array}{lcl}
\psi(x) & = & x \\
\psi(\lambda x.M) & = & \lambda x.\mathcal{I}(M)
\end{array}
$$

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

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.

Related work: Hume (IFL 2006)

Closest work in the area of **higher-order** functional programming.

- Different compilation chain: Hume programs are compiled to the instructions of a Hume abstract machine (think of SECD) which is **implemented in** C and then compiled with standard compilers gcc.
- The main **experimental result** is an estimation of the cost of executing the instructions of the abstract machine on a simple processor (Resenas) using the AbsInt WCET tool.
- **Not covered:** certification of the cost annotations, reasoning about the cost annotations, and the cost of heap management.

Problems/Under-developed areas/Perspectives

- As for C, we needs tools for **assume-guarantee reasoning on WCET** of complex processors.
- Region-inference. Very little is known on the **complexity of region-inference** or the definition/existence of **optimal solutions**.
- Connection with **implicit complexity** for **higher-order functional languages** (light lambda calculi).