

Séminaire ENSTA

Certifying cost annotations in compilers

Nicolas Ayache

Post-doctorate at PPS — Paris 7

13th december 2011

The CerCo project

Certified Complexity

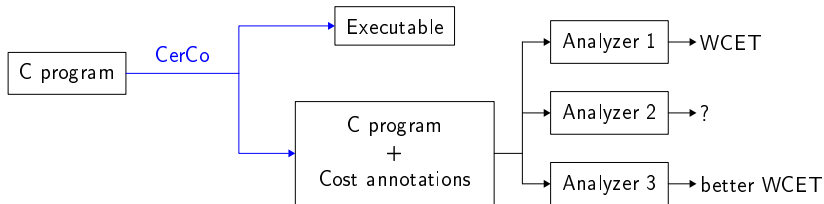
3 years European project (FP7)

Laboratories

- ▶ University of Bologna — Claudio Sacerdoti Cohen
- ▶ University of Edinburgh — Ian Stark
- ▶ **University of Paris Diderot (PPS)**
 - Roberto Amadio (site leader)
 - Yann Régis-Gianas
 - Nicolas Ayache

Goal

Formally sound, cost annotating compiler



✓ Overapproximated concrete complexity

CerCo's approach: problematic

What is the cost of evaluating `tab[i]+1` ?

Depends on:

- ▶ The variable final locations
- ▶ The way memory accesses are compiled
- ▶ The way operations are compiled

⇒ Depends on the compilation process

CerCo's approach: solution

Bad solution

Consider worst case scenarios

- ✗ Too imprecise
- ✗ Optimizations lost
- ✗ Not modular

CerCo's approach: solution

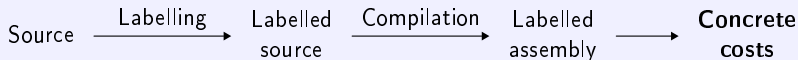
Bad solution

Consider worst case scenarios

- ✗ Too imprecise
- ✗ Optimizations lost
- ✗ Not modular

CerCo's solution

Symbolic cost update: label



CerCo's approach: common considerations

Operational semantics

- ▶ *Without labels:* $\mathcal{P}(\text{Prog} \times \text{State} \times \text{State})$
- ▶ *With labels:* $\mathcal{P}(\text{Prog} \times \text{State} \times \text{label trace} \times \text{State})$

CerCo's approach: common considerations

Operational semantics

- ▶ *Without labels*: $\mathcal{P}(\text{Prog} \times \text{State} \times \text{State})$
- ▶ *With labels*: $\mathcal{P}(\text{Prog} \times \text{State} \times \text{label trace} \times \text{State})$

Annotation semantics

- ✓ Annotation = Instruction of the source language
- ▶ Explicit cost annotations
- ▶ Analysis tools for cost synthesis

CerCo's approach: common considerations

Operational semantics

- ▶ *Without labels*: $\mathcal{P}(\text{Prog} \times \text{State} \times \text{State})$
- ▶ *With labels*: $\mathcal{P}(\text{Prog} \times \text{State} \times \text{label trace} \times \text{State})$

Annotation semantics

- ✓ Annotation = Instruction of the source language
- ▶ Explicit cost annotations
- ▶ Analysis tools for cost synthesis

Demo

Outline

- 1 Toy compiler
 - Languages
 - Compilation
 - Labelling
 - Annotation
- 2 Realistic C compiler
 - Architecture
 - Labelling
- 3 Experiments
 - Cost synthesis
 - Lustre case study

Outline

- 1 Toy compiler
 - Languages
 - Compilation
 - Labelling
 - Annotation
- 2 Realistic C compiler
 - Architecture
 - Labelling
- 3 Experiments
 - Cost synthesis
 - Lustre case study

Overview

Imp $\longrightarrow$ **VM** $\longrightarrow$ **ASM**

- ▶ **Imp**: simple while language
- ▶ **VM**: virtual machine (stack operations)
- ▶ **ASM**: assembly

Overview

Imp $\longrightarrow$ **VM** $\longrightarrow$ **ASM**

- ▶ **Imp**: simple while language
- ▶ **VM**: virtual machine (stack operations)
- ▶ **ASM**: assembly

In the following...

- ▶ How to label the languages?
- ▶ How to adapt the proofs?
- ▶ How to keep the proofs modular?

Syntax

Imp

$$\begin{aligned} e &::= id \mid n \in \mathbb{N} \mid e + e & b &::= e < e \\ S &::= \text{skip} \mid id := e \mid S ; S \mid \text{if } b \text{ then } S \text{ else } S \mid \text{while } b \text{ do } S \\ P &::= \text{prog } S \end{aligned}$$

VM

$$\begin{aligned} \mathcal{I} &::= \text{cst}(n) \mid \text{var}(id) \mid \text{setvar}(id) \mid \text{add} \mid \text{branch}(k) \mid \text{bge}(k) \mid \text{halt} \\ P &::= \mathcal{I} \text{ list} \end{aligned}$$

ASM

$$\begin{aligned} \mathcal{I} &::= \text{loadi } R, n \mid \text{load } R, \text{addr} \mid \text{store } R, \text{addr} \mid \text{add } R, R, R \\ &\quad \mid \text{branch } k \mid \text{bge } R, R, k \mid \text{halt} \\ P &::= \mathcal{I} \text{ list} \end{aligned}$$

Labelled syntax

 $\text{Imp}^{\mathcal{L}}$

$e ::= id \mid n \in \mathbb{N} \mid e + e \quad b ::= e < e$
 $S ::= \text{skip} \mid id := e \mid S ; S \mid \text{if } b \text{ then } S \text{ else } S \mid \text{while } b \text{ do } S \mid \ell : S$
 $P ::= \text{prog } S$

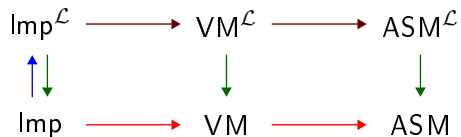
 $\text{VM}^{\mathcal{L}}$

$\mathcal{I} ::= \text{cst}(n) \mid \text{var}(id) \mid \text{setvar}(id) \mid \text{add} \mid \text{branch}(k) \mid \text{bge}(k) \mid \text{halt} \mid \text{emit}(\ell)$
 $P ::= \mathcal{I} \text{ list}$

 $\text{ASM}^{\mathcal{L}}$

$\mathcal{I} ::= \text{loadi } R, n \mid \text{load } R, \text{addr} \mid \text{store } R, \text{addr} \mid \text{add } R, R, R$
 $\quad \mid \text{branch } k \mid \text{bge } R, R, k \mid \text{halt} \mid \text{emit } \ell$
 $P ::= \mathcal{I} \text{ list}$

Simulation



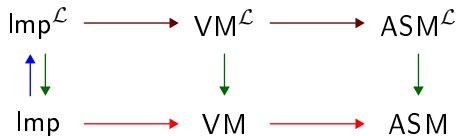
Labelling

Erasure

Compilation

Labelled compilation

Simulation



Labelling

Erasure

Compilation

Labelled compilation

Theorem: diagram commutativity

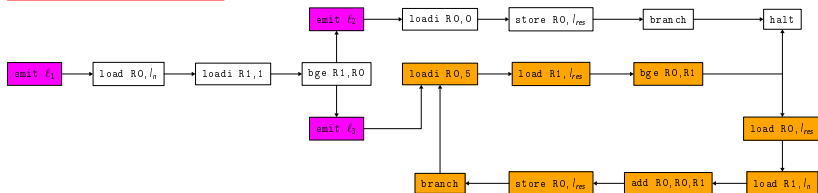
$$\left. \begin{array}{l}
 \mathcal{E}_{\text{Imp}} \circ \mathcal{L}_{\text{Imp}} = \text{Id}_{\text{Imp}} \\
 \mathcal{C}_{\text{Imp}} \circ \mathcal{E}_{\text{Imp}} = \mathcal{E}_{\text{VM}} \circ \mathcal{C}_{\text{Imp}}^{\mathcal{L}} \\
 \mathcal{C}_{\text{VM}} \circ \mathcal{E}_{\text{VM}} = \mathcal{E}_{\text{ASM}} \circ \mathcal{C}_{\text{VM}}^{\mathcal{L}}
 \end{array} \right\} \Rightarrow \mathcal{E}_{\text{ASM}} \circ \mathcal{C}^{\mathcal{L}} \circ \mathcal{L}_{\text{Imp}} = \mathcal{C}$$

Example: soundness

```

prog
ℓ1:
if n < 1 then
  ℓ2: res := 0
else
  ℓ3:
    while res < 5 do
      res := res + n

```

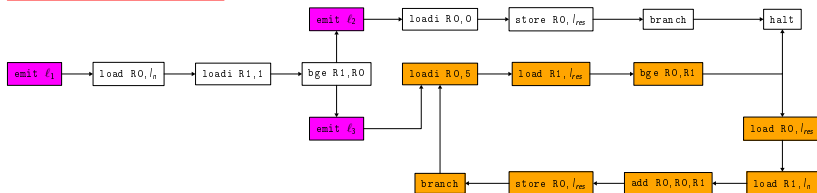


Example: soundness

```

prog
 $\ell_1$ :
if n < 1 then
   $\ell_2$ : res := 0
else
   $\ell_3$ :
    while res < 5 do
      res := res + n

```



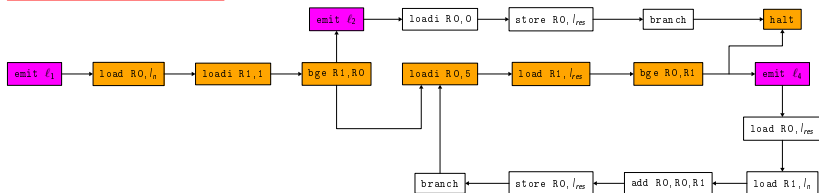
Loop with no label $\Rightarrow$ not constant cost code

Example: precision

```

prog
l1:
if n < 1 then
  l2: res := 0
else
  while res < 5 do
    l4: res := res + n

```

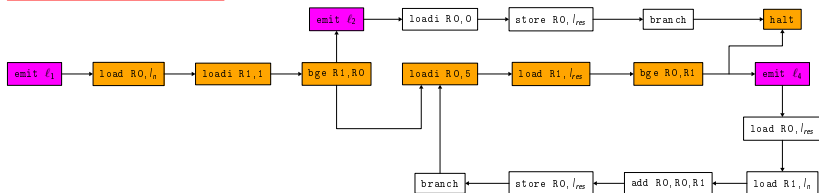


Example: precision

```

prog
l1:
if n < 1 then
  l2: res := 0
else
  while res < 5 do
    l4: res := res + n

```



From emit l_1 : three paths with different costs $\Rightarrow$ imprecision

Labelling criteria

Soundness

Every reachable code is in the scope of a label.
At least one label inside each loop.

Precision

Two different paths to the next labels have the same cost.

Criteria

The labelling must be **sound** and **precise**.
(This can be syntatically checked on the assembly code.)

✓ Nice plus is a reasonable **economy**: not too many labels.

Instrumentation

Cost deduction

Given: $\phi : \text{ASM Instruction list} \rightarrow \mathbb{N}$ $\left(\begin{array}{c} \text{loop free} \\ \text{may overapproximate} \end{array} \right)$

Deduced: $\kappa : \mathcal{L} \rightarrow \mathbb{N}$

Instrumentation

Cost deduction

Given: $\phi : \text{ASM Instruction list} \rightarrow \mathbb{N}$ $\left(\begin{array}{c} \text{loop free} \\ \text{may overapproximate} \end{array} \right)$

Deduced: $\kappa : \mathcal{L} \rightarrow \mathbb{N}$

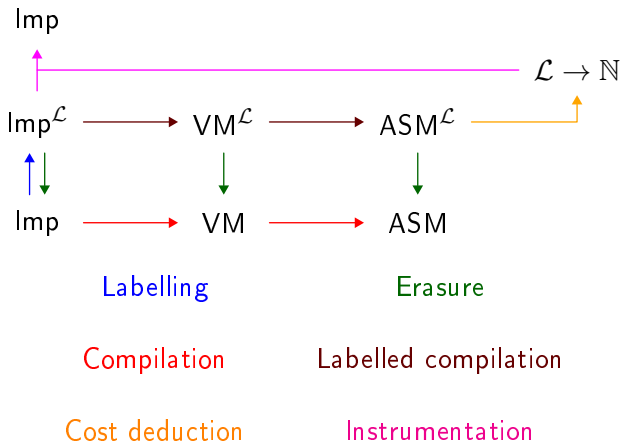
Instrumentation

- ▶ Use a **fresh variable**
- ▶ Initialize it to **0**
- ▶ Replace **labels** with **increments** (following κ)

<pre> prog l₁: while res < 5 do l₂: res := res + n </pre>	→	<pre> prog _cost := 0; _cost := _cost + 2; while res < 5 do _cost := _cost + 4; res := res + n </pre>
--	---	--

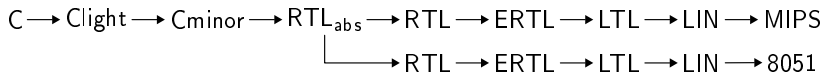
Annotation

$$\text{Annotation} = \text{Instrumentation} \circ \text{Labelling}$$



Outline

- 1 Toy compiler
 - Languages
 - Compilation
 - Labelling
 - Annotation
- 2 Realistic C compiler
 - Architecture
 - Labelling
- 3 Experiments
 - Cost synthesis
 - Lustre case study



C to Cminor

- Adapted from **CompCert**
(GNU GPL and INRIA Non-Commercial)

RTL_{abs}

- ▶ Retargeting simplified
 - ✓ Some common optimizations
 - ✗ Some optimizations lost

Back-ends

- Adapted from pedagogical Pseudo-Pascal compiler (François Pottier, Creative Commons)

Restrictions



Standalone files

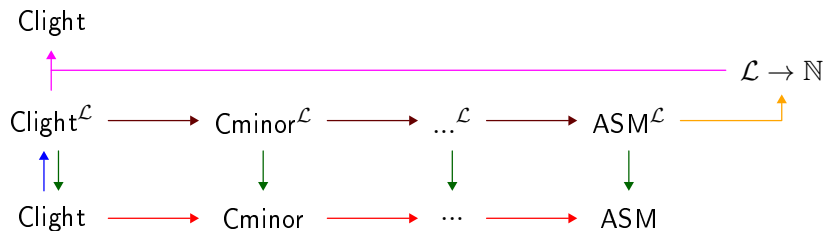
Clight

- ✗ long long and long double types
- ✗ longjmp and setjmp instructions
- ✗ Unreasonable forms of switch
- ✗ Unprototyped and variable-arity functions

RTL

- ✗ float

Overview



Labelling

Erasure

Compilation

Labelled compilation

Cost deduction

Instrumentation

Differences Imp / C

Side effect expressions `y = x++;`

Ternary expressions `x ? y+2*z : z`

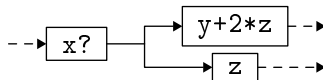
Labels and Gotos `lbl: ... goto lbl;`

Function calls `f(&x);`

Differences Imp / C

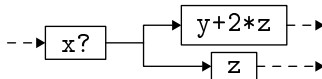
Side effect expressions`y = x++;`**eliminated by CIL****→**`_tmp = x; x = _tmp+1; y = _tmp;`**Ternary expressions**`x ? y+2*z : z`**Labels and Gotos**`lbl: ... goto lbl;`**Function calls**`f(&x);`

Ternary expressions

$$x? (y+2*z) : z$$


Branching $\Rightarrow$ 1 label per branch for precision

Ternary expressions

$$x? (y+2*z) : z$$


Branching $\Rightarrow$ 1 label per branch for precision

Labelled expressions

$$x? (\ell_1 : y+2*z) : (\ell_2 : z)$$

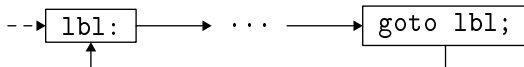
Instrumentation

- 1) Side effects inside expressions 2) Elimination by CIL

$$C \xrightarrow{\text{CIL}} \text{Clight} \xrightarrow{\text{Instrument}} C \xrightarrow{\text{CIL}} \text{Clight}$$

Labels and Gotos

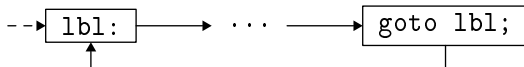
```
lbl: ... goto lbl;
```



Label $\Rightarrow$ potential loop $\Rightarrow$ 1 cost label needed

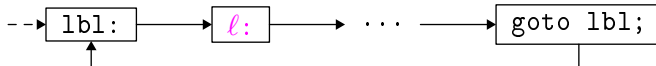
Labels and Gotos

```
lbl: ... goto lbl;
```



Label $\Rightarrow$ potential loop $\Rightarrow$ 1 cost label needed

```
lbl: l: ... goto lbl;
```



Function calls

Function call: sequential instruction

```
x++;  
f(&x);  
y = x;
```

Diagram illustrating a function call: sequential instruction. The code shows a sequence of instructions: `x++;`, `f(&x);`, and `y = x;`. The function `f` is defined as `void f(int* x) { ... return; }`. An arrow points from the argument `&x` in the function call to the parameter `x` in the function definition, indicating the argument passed to the function.

Function calls

Function call: sequential instruction

```
scope1 [ x++;  
        f(&x);  
        y = x;  
scope2 [ void f(int* x) {  
        ...  
        return; } ]
```

Function calls

Function call: sequential instruction

```

scope1 [ x++;
        f(&x);
        y = x;
      ]
      void f(int* x) {
        ...
        return;
      } scope2

```

Function pointer: statically unresolvable destination

✓ Each function handles its cost

```

scope1 [ x++;
        f(&x);
        y = x;
      ]
      void f(int* x) {
        ...
        return;
      } scope2

```

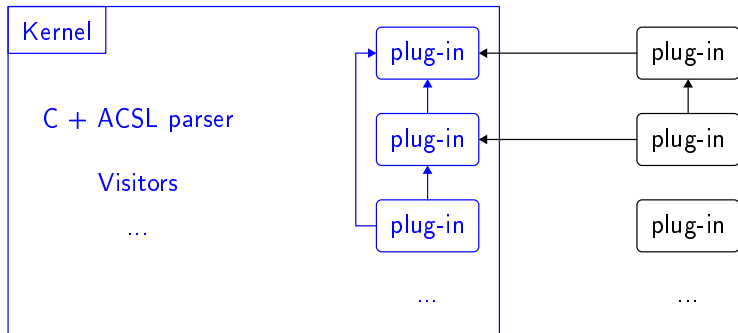
Outline

- 1 Toy compiler
 - Languages
 - Compilation
 - Labelling
 - Annotation
- 2 Realistic C compiler
 - Architecture
 - Labelling
- 3 Experiments
 - Cost synthesis
 - Lustre case study

Frama-C

CEA - LSL

Tool for collaborative and modular analyzers for C



Jessie plug-in

ACSL

Specification language for C programs

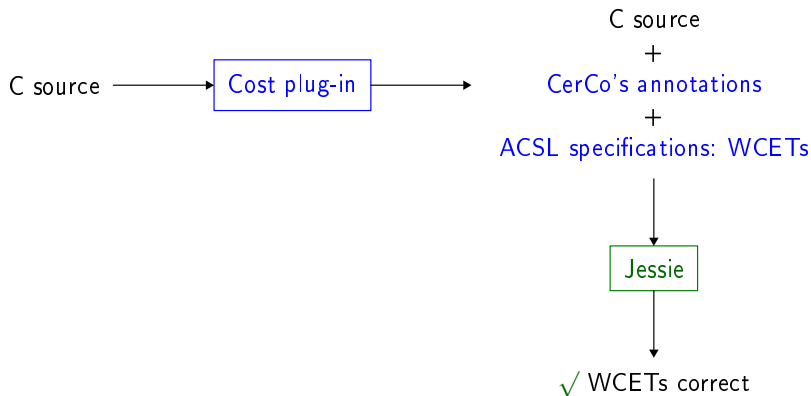
- ▶ Based on Hoare logic
- ▶ C Comments
- ▶ First-order logical definitions and propositions

Jessie

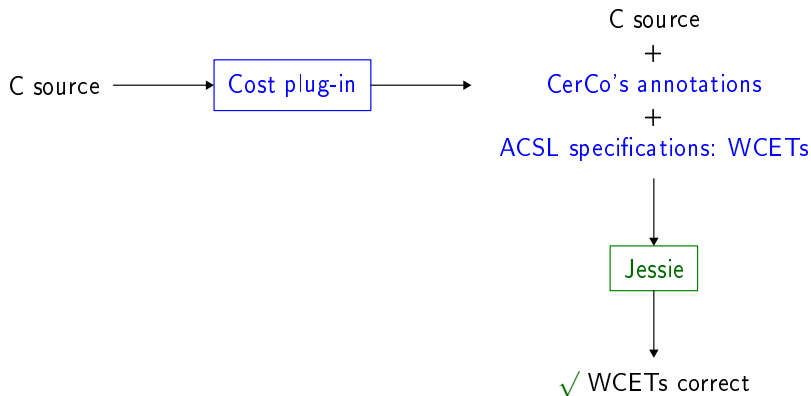
Translator to Why (LRI - ProVal)

- ▶ Weakest pre-condition calculus
- ▶ Generates proof obligations
- ▶ Output to various provers (automatic or interactive)

Cost plug-in



Cost plug-in



Demo

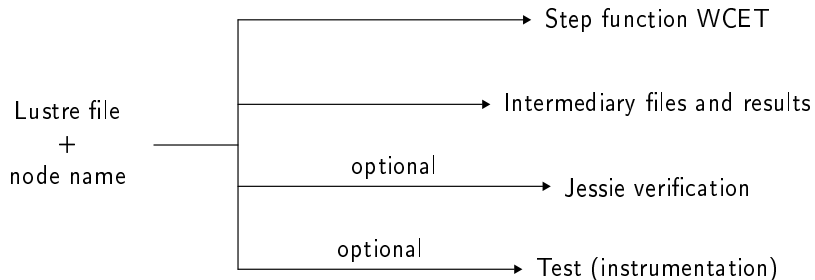
Lustre

- ▶ Synchronous language
- ▶ Node = flow
- ▶ `lus2c`: node $\rightarrow$ C step function (cycle)
- ▶ Step function WCET = component reaction time

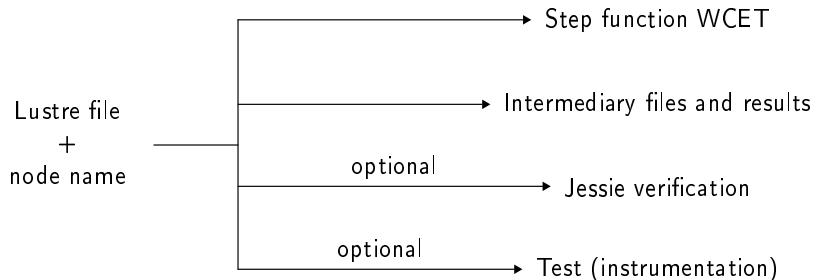
Lustre C specificities

- ▶ No arrays
- ▶ No loop
- ▶ Boolean variables identified

Wrapper for Lustre



Wrapper for Lustre



Demo

CerCo

- ▶ C sound compiler (proofs in progress in Matita)
- ▶ Correct cost annotations (overapproximation)
- ▶ C analyzers for WCET
- ▶ Symbolic cost labels
- ▶ Modular proofs

Thank you for your attention!

Questions?