# Certifying and reasoning about cost annotations of C programs


CerCo

This work was (partially) supported by the Information and Communication Technologies (ICT) Programme as Project FP7-ICT-2009-C-243881 CerCo.

# Concrete and certified complexity

```
int summul (int n) {
  int accu = 1;
  int stop = 1;
  int k = n;
  while (k >= stop) {
    accu += accu * k;
    k -= 1;
  }
  return (accu);
}
```

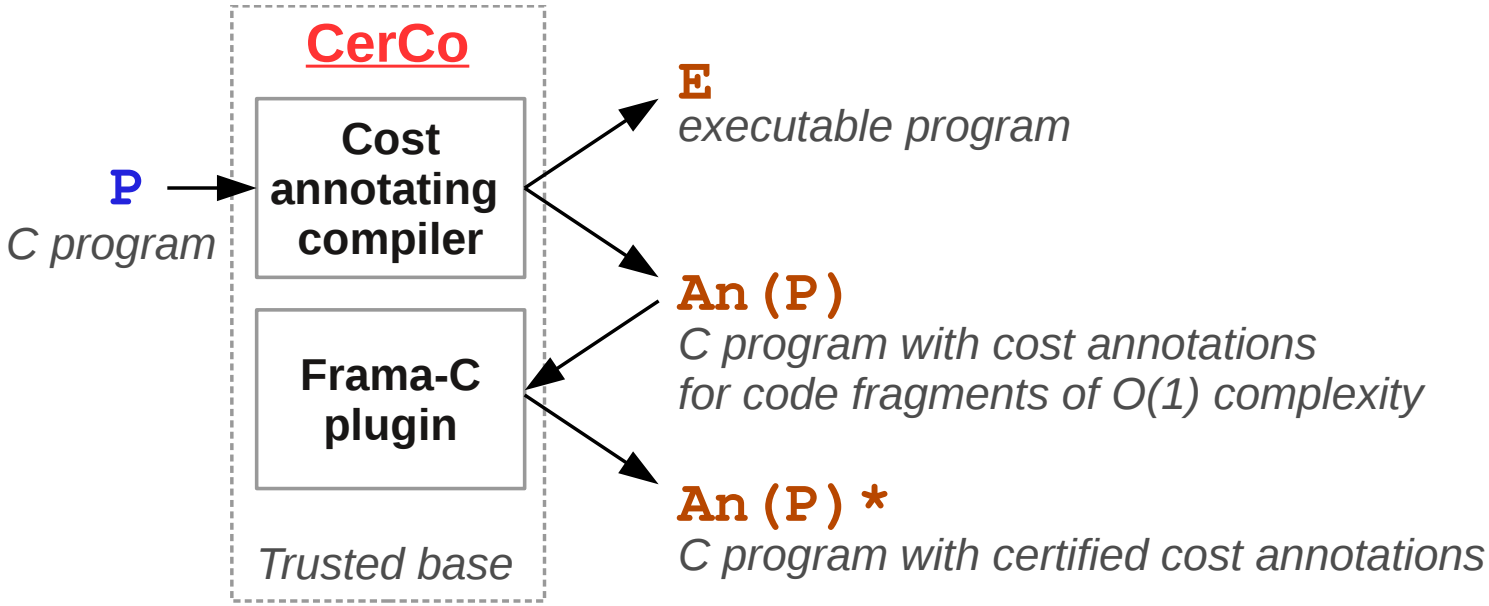
$\Leftrightarrow$

```
$v0 ← 1
$a1 ← 1
summul10:
$a3 ← 0
$a2 ← $a0 ≥ $a1
$a2 ← $a2 == $zero
$a2 == $a3 ⇒ summul6
jr $ra
summul6:
$a2 ← $v0 × $a0
$v0 ← $v0 + $a2
$a2 ← 1
$a0 ← $a0 − $a2
j summul10
```

- Reasoning about the complexity is rather made on the source.
- Concrete execution costs are better guessed on the binary code.

How can we *lift* in a provably correct way
information on the execution cost of the binary code
to cost annotations on the source code?

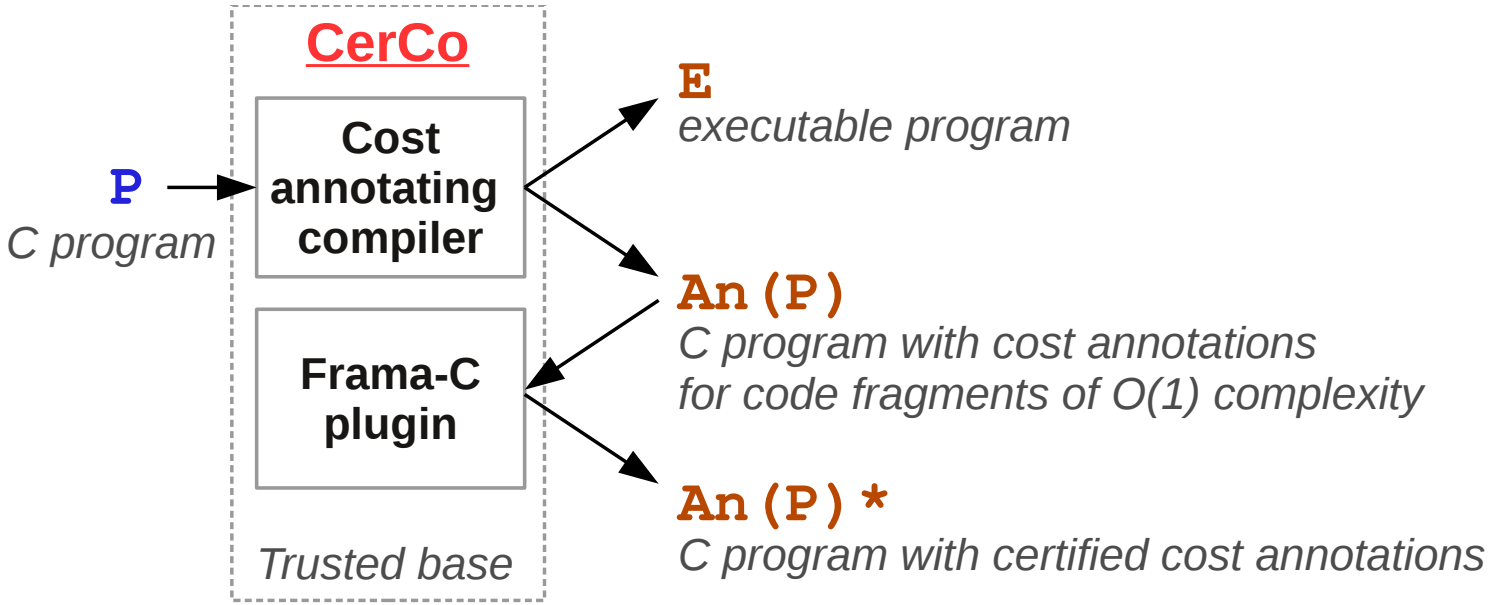
# Cost annotating compilation




## This talk

1. How can we extend existing proof techniques for semantic preservation from compilers to cost annotating compilers?
2. Do these proof techniques scale to the programming languages and hardware architectures typically used in embedded systems?

# Cost annotating compilation




## Key technical points

(i) What meaning for the cost annotations in **An(P)**?

(ii) How to provide them being sound and precise?

(iii) How to *compose* the proofs?

# What meaning for the cost annotations An(P)?

**An(P)** must behave as **P** while **self-monitoring** the execution cost.

## Example

**P**

```
int summul (int n) {
  int accu = 1;
  int stop = 1;
  int k = n;
  while (k >= stop) {
    accu += accu * k;
    k -= 1;
  }
  return (accu);
}
```

**E**

```
$v0 ← 1
$a1 ← 1
summul10:
$a3 ← 0
$a2 ← $a0 ≥ $a1
$a2 ← $a2 == $zero
$a2 == $a3 ⇒ summul6
jr $ra
summul6:
$a2 ← $v0 × $a0
$v0 ← $v0 + $a2
$a2 ← 1
$a0 ← $a0 − $a2
j summul10
```

**An(P)**

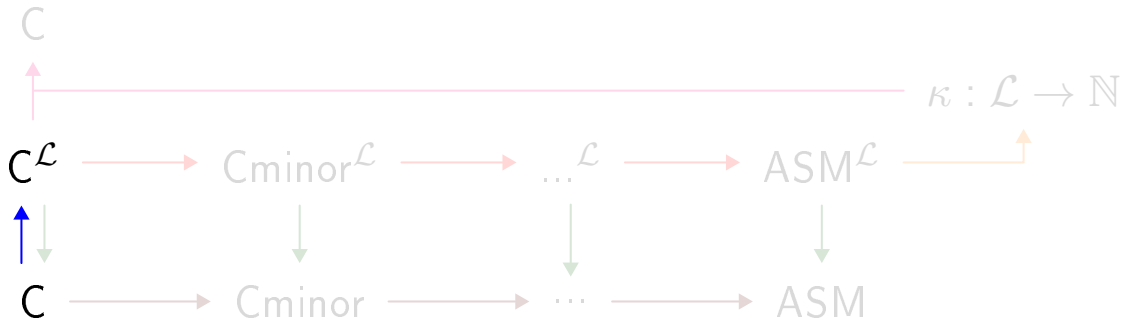
```
int summul (int n) {
  int accu = 1;
  int stop = 1;
  int k = n;
  _cost += 2;
  while (k >= stop) {
    _cost += 9;
    accu += accu * k;
    k -= 1; }
  _cost += 1;
  return (accu);
}
```

# How to provide sound and precise cost annotations? (1/3)

## Definitions

- A cost annotation is sound if the predicted cost is an upper bound of the real execution cost.
- A cost annotation is precise if the difference between the predicted cost and the real cost is bounded by a constant $\delta$ that only depends on the program (not the input).

# How to provide sound and precise cost annotations? (2/3)




Labels in the source code represents symbolic cost updates.

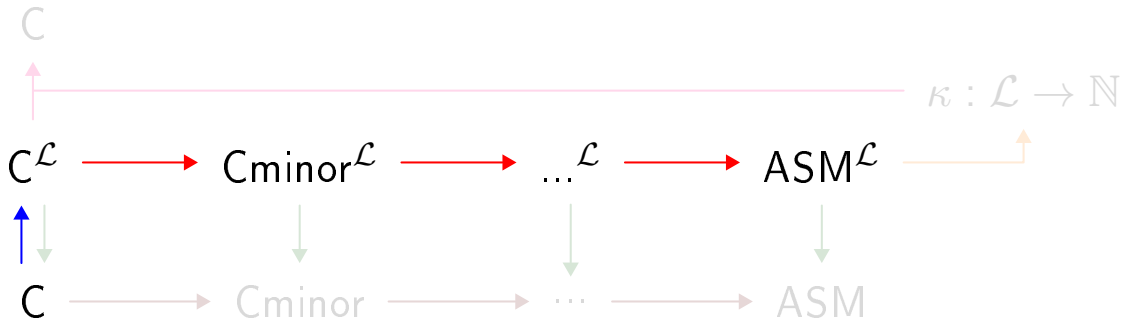
(We will come back later on the strategic locations where to put the labels.)

```
int summul (int n) {
  int accu = 1;
  int stop = 1;
  int k = n;
  while (k >= stop) {
    accu += accu * k;
    k -= 1;
  }
  return (accu);
}
```

$\mathcal{L}(P) \Longrightarrow$

```
int summul (int n) {
_cost2:
  int accu = 1;
  int stop = 1;
  int k = n;
  while (k >= stop) {
    _cost3:
    accu += accu * k;
    k -= 1; }
  _cost4:
  return (accu);
}
```

# How to provide sound and precise cost annotations? (2/3)



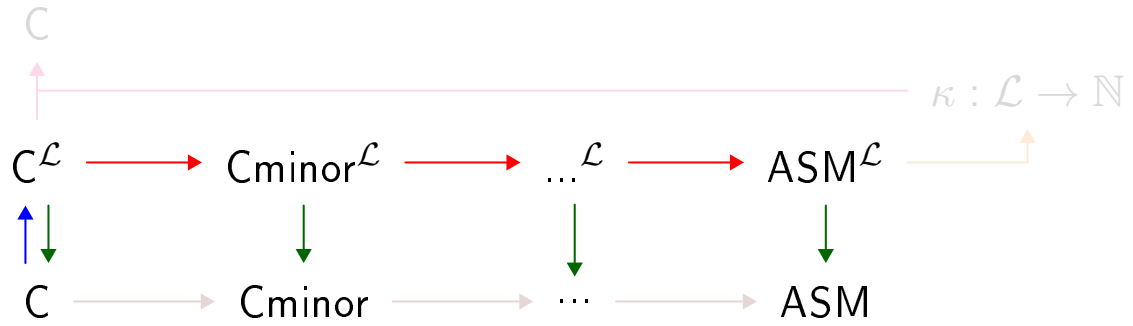

Labels are conveyed through the compilation, down to the binary.

```
int summul (int n) {
_cost2:
  int accu = 1;
  int stop = 1;
  int k = n;
  while (k >= stop) {
    _cost3:
    accu += accu * k;
    k -= 1; }
  _cost4:
  return (accu);
}
```

$$\mathcal{C}_{\mathcal{L}}(\mathcal{L}(P)) \Longrightarrow$$

```
emit _cost2
$v0 ← 1
$a1 ← 1
summul10:
$a3 ← 0
$a2 ← $a0 ≥ $a1
$a2 ← $a2 == $zero
$a2 == $a3 ⇒ summul6
emit _cost4
jr $ra
summul6:
emit _cost3
$a2 ← $v0 × $a0
$v0 ← $v0 + $a2
$a2 ← 1
$a0 ← $a0 − $a2
j summul10
```

# How to provide sound and precise cost annotations? (2/3)



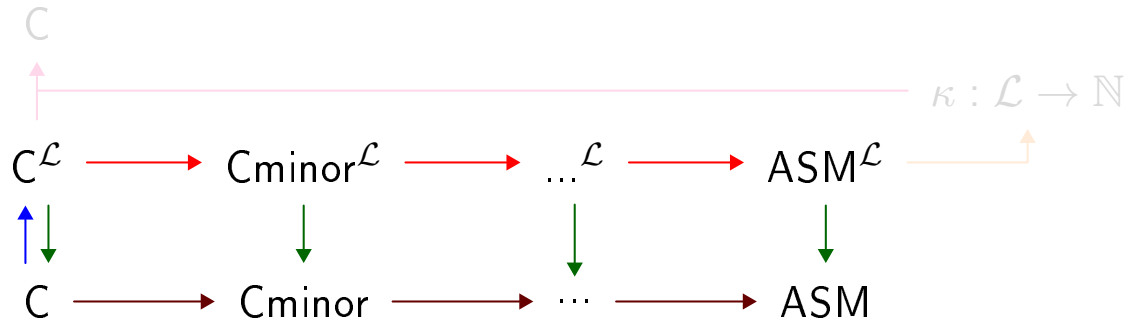

There are erasure functions $\mathcal{E}$ from labelled to unlabelled languages.

```
int summul (int n) {
_cost2:
  int accu = 1;
  int stop = 1;
  int k = n;
  while (k >= stop) {
    _cost3:
    accu += accu * k;
    k -= 1; }
  _cost4:
  return (accu);
}
```

$$\mathcal{E}(\mathcal{C}_{\mathcal{L}}(\mathcal{L}(P))) \Rightarrow$$

```
emit _cost2
$v0 ← 1
$a1 ← 1
summul10:
$a3 ← 0
$a2 ← $a0 ≥ $a1
$a2 ← $a2 == $zero
$a2 == $a3 ⇒ summul6
emit _cost4
jr $ra
summul6:
emit _cost3
$a2 ← $v0 × $a0
$v0 ← $v0 + $a2
$a2 ← 1
$a0 ← $a0 − $a2
j summul10
```

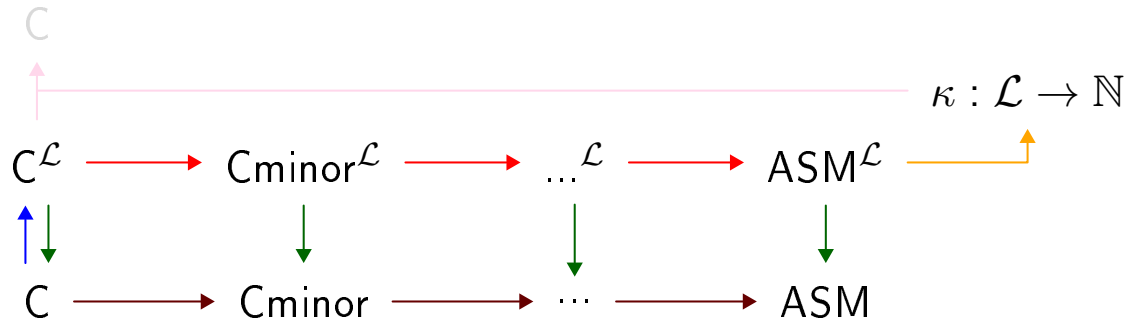
# How to provide sound and precise cost annotations? (2/3)




Labelled compilation extends the compilation in a modular way.

$$\mathcal{E}_{ASM}(\mathcal{C}_{\mathcal{L}}(\mathcal{L}(P))) = \mathcal{C}(P)$$

# How to provide sound and precise cost annotations? (2/3)



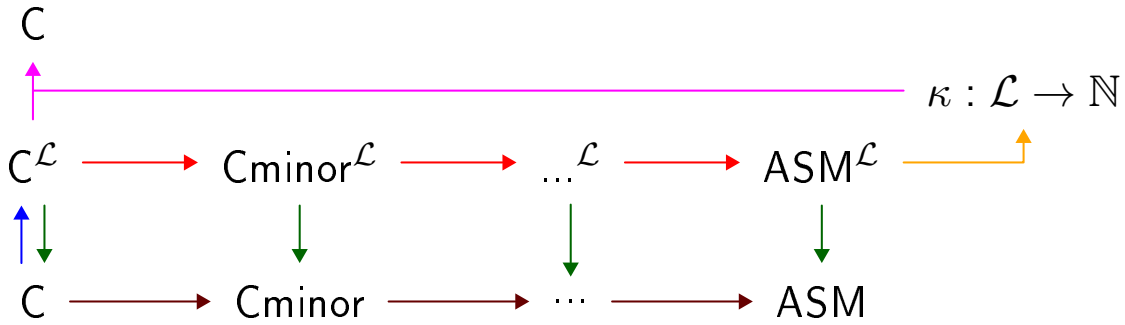

The cost of a label is deduced from the instructions in its scope.

```
  emit _cost2
  $v0 ← 1
  $a1 ← 1
  summul10:
  $a3 ← 0
  $a2 ← $a0 ≥ $a1
  $a2 ← $a2 == $zero
  $a2 == $a3 ⇒ summul6
  emit _cost4
  jr $ra
  summul6:
  emit _cost3
  $a2 ← $v0 × $a0
  $v0 ← $v0 + $a2
  $a2 ← 1
  $a0 ← $a0 − $a2
  j summul10
```

⇒

| $\ell \in \mathcal{L}$ | $\kappa(\ell)$ |
|---|---|
| _cost2 | 6 |
| _cost3 | 9 |
| _cost4 | 1 |

# How to provide sound and precise cost annotations? (2/3)

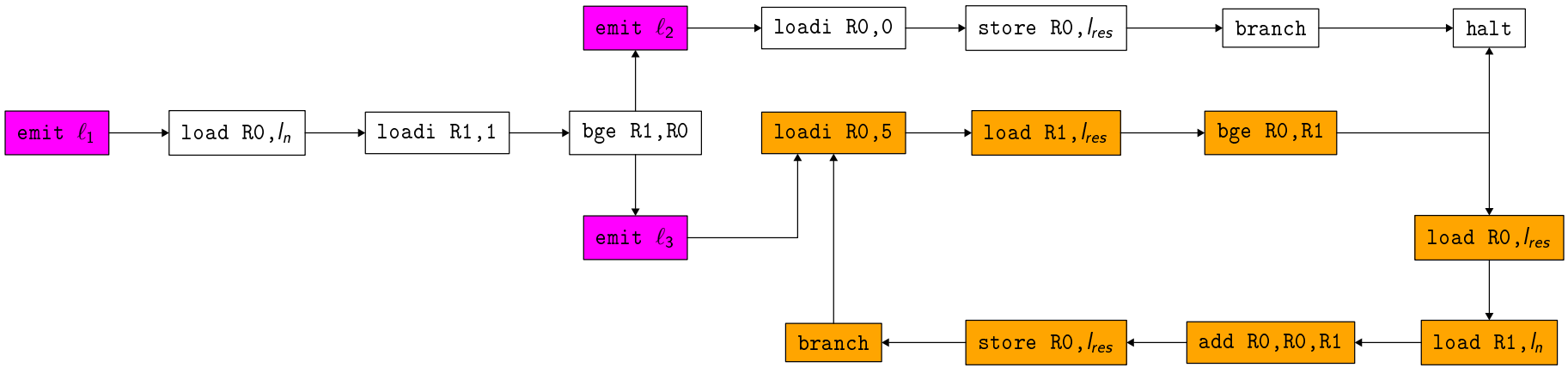



Instrumentation injects $\kappa$ in the source to reason about the complexity.

```
int summul (int n) {
  int accu = 1;
  int stop = 1;
  int k = n;
  _cost += 2;
  while (k >= stop) {
    _cost += 9;
    accu += accu * k;
    k -= 1; }
  _cost += 1;
  return (accu);
}
```

# How to provide sound and precise cost annotations? (3/3)

```
ℓ1 :
if (n < 1)
  ℓ2 :  res = 0
else
  ℓ3 :
  while (res < 5)
        res = res + n
```

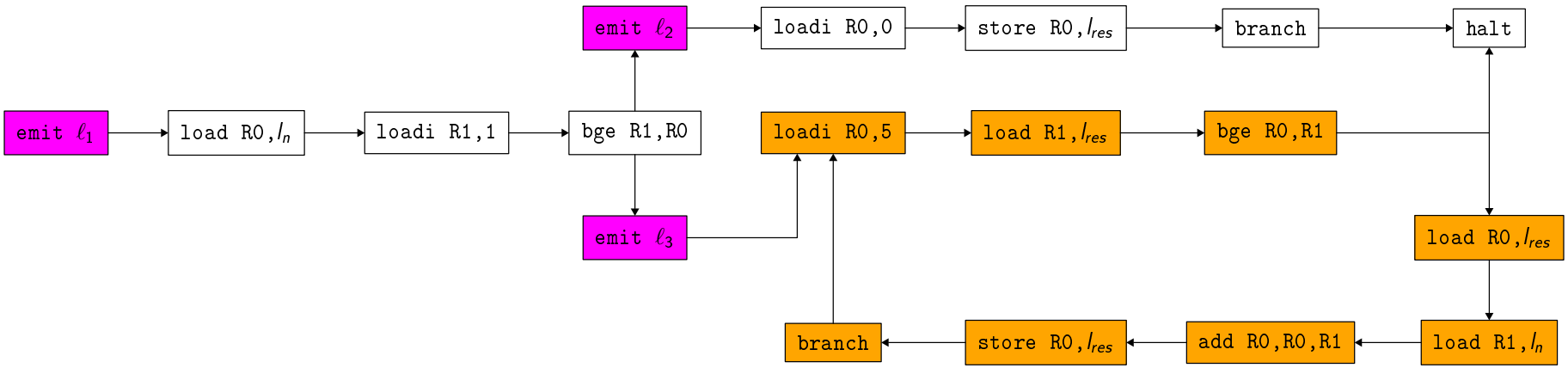



## Sufficient conditions on labelling for **soundness**

- There must be a label inside each loop.
- Every reachable code must be in the scope of a label.

# How to provide sound and precise cost annotations? (3/3)

```
ℓ1 :
if (n < 1)
  ℓ2 :  res = 0
else
  ℓ3 :
  while (res < 5)
        res = res + n
```

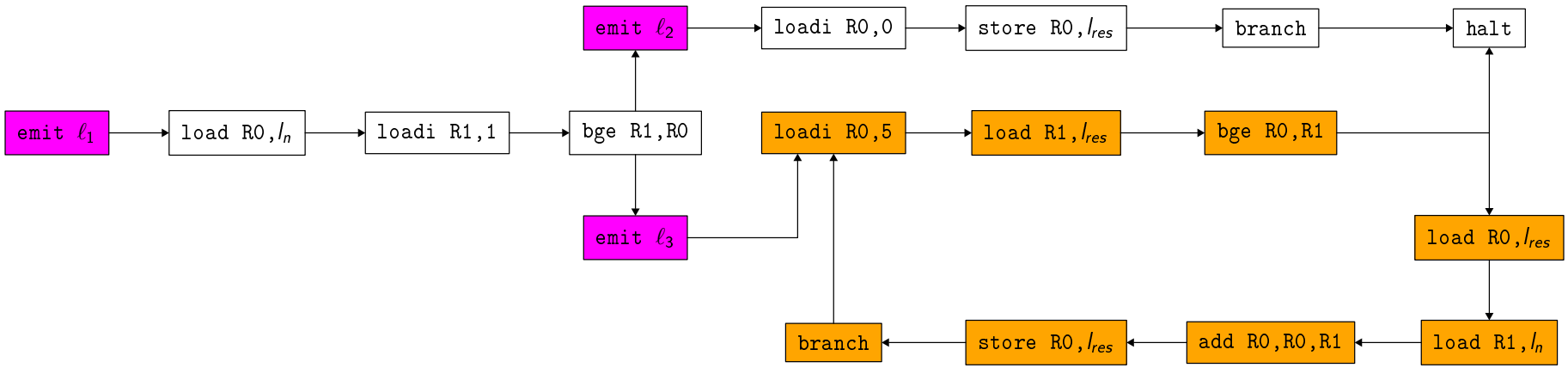



## Sufficient conditions on labelling for **soundness**

- There must be a label inside each loop.
- Every reachable code must be in the scope of a label.

# How to provide sound and precise cost annotations? (3/3)

```
ℓ1 :
if (n < 1)
  ℓ2 :  res = 0
else
  ℓ3 :
  while (res < 5)
        res = res + n
```

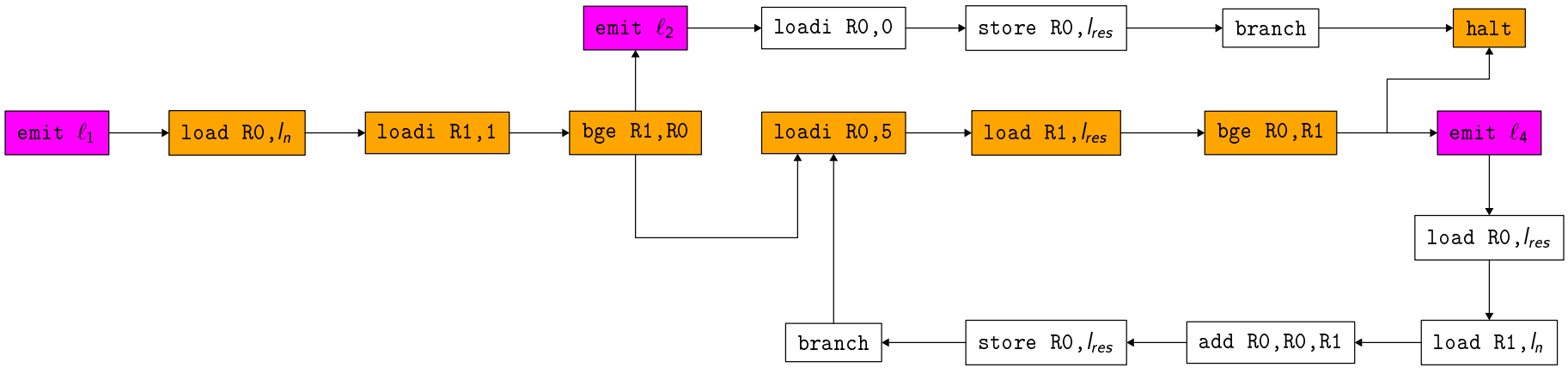



## Sufficient conditions on labelling for **soundness**

- There must be a label inside each loop.
- Every reachable code must be in the scope of a label.

# How to provide sound and precise cost annotations? (3/3)

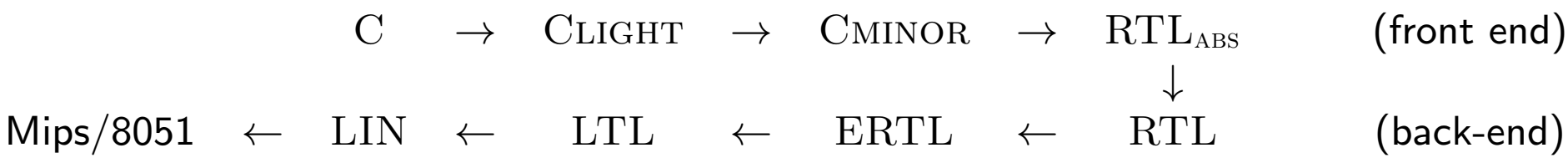
```
ℓ1:
if (n < 1)
  ℓ2:  res = 0
else

  while (res < 5)
    ℓ4:  res = res + n
```




## Sufficient conditions on labelling for **precision**

Two different paths from the same label must have the same cost.

# A cost annotating compiler for the C language




- Untrusted compiler prototype written in OCaml.
- Inspired from CompCert, a C compiler certified in Coq.
- Handle a large subset of C.
- Optimize moderately.
- The Intel 8051 microprocessor has a specified cost model.

# Demo

# A tool to reason about the complexity of C programs

- Frama-C, a set of analysers for C programs.
- Cost, a plugin for Frama-C that:
  - (i) takes C program $P$ as input ;
  - (ii) applies the CerCo compiler to enrich $P$ with cost annotations ;
  - (iii) applies some heuristics to produce a tentative bound on the cost of executing the C functions of the program as a function of the value of their parameters.
- The deductive tool Jessie is used to discharge the proof obligations.

# 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;
 }
```

# Demo

# Focus on an experiment

A certified bound on the reaction time of a LUSTRE program

## The specificities of synchronous programs

- LUSTRE is a data-flow language to program synchronous systems.
- The language is usually compiled to C as a `step` function.
- This function only has assignment and conditional statements.
- In that setting, the assertion inference is fully automatic.
- Once this assertion is proved, we get a certified bound on the reaction time of the system.

# 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<br>L: C generated from a LUSTRE file | | | | |

# Limitations

- The labelling approach depends on the possibility of obtaining accurate information on the execution cost of relatively short sequences of binary instructions.
- This seems beyond the scope of current Worst-Case Execution Time (WCET) tools such as AbsInt or Chronos5 which do not support a compositional analysis of WCET.
- An important characteristic of the 8051 processor is that its cost model is additive: the cost of a sequence of instructions is exactly the sum of the costs of each instruction.

# Conclusion, ongoing and further work

- The labelling method allows a modular extension of a compiler and its proof of correctness to produce sound and precise cost annotations.
- It scales to functional programming languages.
- It is being generalized to handle more agressive optimizations.
- The CerCo compiler is being certified in the Matita proof assistant.

Thanks for your attention!
Any questions?

Get related (free) software here:

`http://www.pps.univ-paris-diderot.fr/cerco`