Automated Approximate Recurrence Solving applied to Static Analysis of Energy Consumption

Louis Rustenholz, supervised by
Manuel Hermenegildo, Pedro López-García and José F. Morales,
CLIP Lab, IMDEA Software Institute, Madrid

6 September 2022

Introduction

Team

IMDEA Software Institute, Madrid

Manuel Hermenegildo

Pedro López-García

José Francisco Morales

Why Energy?

IT's share of global carbon emissions has grown from 2.5% to around 5% in the last ten years.

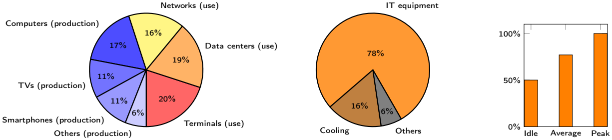
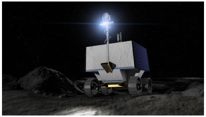
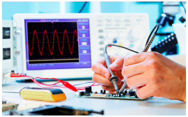


Figure: Energy Usage in IT. By subfield, and case studies on a data center.

- Energy consumption of programs is relevant and understudied by carbon audit experts.
- Also, potential applications to verification of embedded software or against side-channel attacks.





- Applicable to other resources than energy: time, memory, number of communications, ...

Why Recurrence Equations?

- There are both software and hardware components to such line of work. In this internship, we focused more on the problem of control flow analysis than on energy models.
- The hard software problem is recursivity.
  Use Horn Clauses as Intermediate Representation.

  > "Systems of recurrence equations may be seen as programs stripped from information irrelevant to cost analysis"

- Here, we don't really care about exact solutions of equations: bounds on the solutions are satisfactory.
  This is a research opportunity.

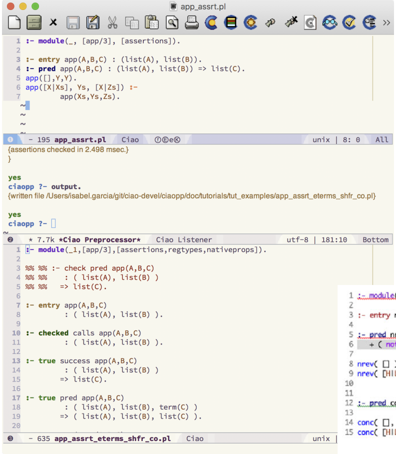
Contents of this talk

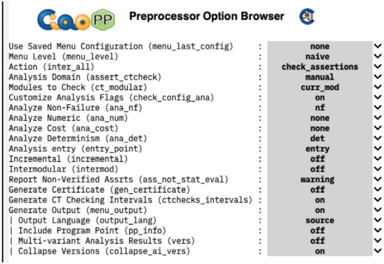
1. Background (Ciao/CiaoPP, Logic Programming, Abstract Interpretation)
2. Current pipeline for energy analysis of imperative programs
3. Implementation of classical recurrence solving techniques
4. Proposal of new order-theoretical recurrence solving techniques

Background

Ciao / CiaoPP

An extensible logic programming language making full use of analysis/verification/optimisation.





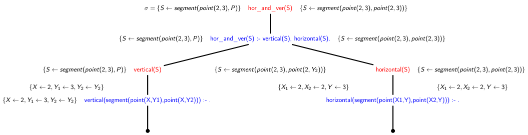
Logic Programming

- Fact, rules and queries.
- Programing with Relations: nondeterminism and no fixed input/output status.
- Execution $\leftrightarrow$ automated proof search. Computation happens by unification.

- Using Logic Programs as (Horn Clause) Intermediate Representation of imperative programs,

  the only control structure is function call.

- AND-OR trees, sets of substitutions, ...




```
vertical  (segment(point(X,_),point(X,_))).
horizontal(segment(point(_,Y),point(_,Y))).

hor_and_ver(S) :- vertical(S), horizontal(S).

hor_or_ver(S) :- vertical(S).
hor_or_ver(S) :- horizontal(S).

?- hor_and_ver(segment(point(2,3), P)).
   P = point(2, _1658)
   P = point(_1364, 3)
```

Abstract Interpretation

- A general theory of sound abstractions of program semantics, based on order theory and Galois connections between lattices called *concrete* and *abstract domains*.

Figure: Abstract Interpretation was first developed by Patrick and Radhia Cousot in the 70's.

- Observation: program semantics can be viewed as the *least fixed point* of some monotone operator.

$$
\begin{tikzcd}
\text{Usual Syntax} \arrow[r, "\text{rewrite}"] \arrow[d, "S\ \text{(execute on all inputs)}"'] & \begin{array}{c}\text{Syntax for}\\ \text{modified program}\end{array} \arrow[d, "S^\sharp\ \text{(execute abstract program)}"] \\
\begin{array}{c}\text{Usual Semantic}\\ \text{e.g. } \mathcal{P}(\Sigma^*)\end{array} \arrow[r, "\text{extract (forget)}"] & \begin{array}{c}\text{Abstract Semantic}\\ \text{i.e. desired information}\end{array}
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \text{Usual Syntax} \arrow[dl, "S"'] \arrow[dr, "S^\sharp\ \text{(abstract execution)}"] & \\
\begin{array}{c}\text{Usual Semantic}\\ \text{e.g. } \mathcal{P}(\Sigma^*)\end{array} \arrow[rr, "\text{extract (forget)}"] & & \begin{array}{c}\text{Abstract Semantic}\\ \text{i.e. desired information}\end{array}
\end{tikzcd}
$$

Figure: "Execute then Abstract" or "Abstract then Execute", or even "Execute Abstractly".

Pipeline

Pipeline

$$
\begin{tikzcd}
\mathtt{C} \arrow[rr] \arrow[d] & & \mathtt{HCIR} \arrow[r, dashed, "\alpha_{\mathrm{Sz}}"] \arrow[d] \arrow[dr] & \mathtt{HCIR}^{(\mathtt{Size})} \arrow[r, "\text{extract}"] \arrow[d] & \mathtt{RecEq}(\mathrm{Sz} + \mathrm{Cost}) \arrow[dd, "\text{solve}"] \\
\mathcal{T}_{races}(\mathrm{Mem}^{(C)}, \mathrm{Cost}) \arrow[drr] & \approx & \begin{array}{c} \mathcal{T}_{rees}(\mathrm{Mem}^{(HC)}) \\ + \text{ Cost model for builtins} \end{array} \arrow[r, dashed, "\overline{\alpha_{\mathrm{Sz}}}"] \arrow[d] & \begin{array}{c} \mathcal{T}_{rees}(\mathrm{Mem}^{(HC)}_{\mathrm{Size}}) \\ + \text{ Cost model for builtins} \end{array} \arrow[d] & \\
 & & \mathcal{F}_{\mathcal{P}} + \mathrm{Cost}_{\mathcal{P}} \arrow[r, "\alpha^{\circ}_{\mathrm{Sz}}"] & \mathrm{Sz}_{\mathcal{P}} + \mathrm{Cost}_{\mathrm{Sz},\mathcal{P}} \arrow[r, "\alpha^{\circ}_{\mathcal{I}}"] & \mathrm{Sz}_{\mathcal{I}} + \mathrm{Cost}_{\mathrm{Sz},\mathcal{I}}
\end{tikzcd}
$$

Pipeline

$$
\begin{tikzcd}
\texttt{C} \arrow[rr] \arrow[d, red] & & \texttt{HCIR} \arrow[r, dashed, "\alpha_{\mathrm{Sz}}"] \arrow[d] \arrow[dr] & \texttt{HCIR}^{(\texttt{Size})} \arrow[r, "\text{extract}"] \arrow[d] & \texttt{RecEq}(\mathrm{Sz} + \mathrm{Cost}) \arrow[dd, "\text{solve}"] \\
\mathcal{T}_{races}(\mathrm{Mem}^{(C)}, \mathrm{Cost}) \arrow[drr, red] & \approx & \begin{array}{c} \mathcal{T}_{rees}(\mathrm{Mem}^{(HC)}) \\ + \text{ Cost model for builtins} \end{array} \arrow[r, dashed, "\overline{\alpha_{\mathrm{Sz}}}"] \arrow[d] & \begin{array}{c} \mathcal{T}_{rees}(\mathrm{Mem}^{(HC)}_{\mathrm{Size}}) \\ + \text{ Cost model for builtins} \end{array} \arrow[d] & \\
& & \mathcal{F}_{\mathcal{P}} + \mathrm{Cost}_{\mathcal{P}} \arrow[r, "\alpha^{\circ}_{\mathrm{Sz}}"] & \mathrm{Sz}_{\mathcal{P}} + \mathrm{Cost}_{\mathrm{Sz},\mathcal{P}} \arrow[r, "\alpha^{\circ}_{I}"] & \mathrm{Sz}_{\mathcal{I}} + \mathrm{Cost}_{\mathrm{Sz},\mathcal{I}}
\end{tikzcd}
$$

Pipeline

$$
\begin{tikzcd}
\texttt{C} \arrow[r] \arrow[d, red] & \texttt{HCIR} \arrow[r, dashed, "\alpha_{\mathrm{Sz}}"] \arrow[d] \arrow[dr] & \texttt{HCIR}^{(\texttt{Size})} \arrow[r, "\text{extract}"] \arrow[d] & \texttt{RecEq}(\mathrm{Sz} + \mathrm{Cost}) \arrow[dd, "\text{solve}"] \\
\mathcal{T}_{races}(\mathrm{Mem}^{(C)}, \textcolor{green}{\mathrm{Cost}}) \arrow[r, phantom, "\approx"] \arrow[dr, red] & \begin{array}{c} \mathcal{T}_{rees}(\mathrm{Mem}^{(HC)}) \\ + \text{ Cost model for builtins} \end{array} \arrow[r, dashed, "\overline{\alpha_{\mathrm{Sz}}}"] \arrow[d] & \begin{array}{c} \mathcal{T}_{rees}(\mathrm{Mem}^{(HC)}_{\mathrm{Size}}) \\ + \text{ Cost model for builtins} \end{array} \arrow[d] & \\
& \mathcal{F}_{\mathcal{P}} + \textcolor{green}{\mathrm{Cost}_{\mathcal{P}}} \arrow[r, "\alpha^{\circ}_{\mathrm{Sz}}"] & \mathrm{Sz}_{\mathcal{P}} + \mathrm{Cost}_{\mathrm{Sz},\mathcal{P}} \arrow[r, "\alpha^{\circ}_{I}"] & \mathrm{Sz}_{\mathcal{I}} + \mathrm{Cost}_{\mathrm{Sz},\mathcal{I}}
\end{tikzcd}
$$

Pipeline

$$
\begin{tikzcd}
\texttt{C} \arrow[rr] \arrow[d] & & \texttt{HCIR} \arrow[r, dashed, "\alpha_{\mathrm{Sz}}"] \arrow[d] \arrow[dr] & \texttt{HCIR}^{(\texttt{Size})} \arrow[r, "\text{extract}"] \arrow[d] & \texttt{RecEq}(\mathrm{Sz} + \mathrm{Cost}) \arrow[dd, "\text{solve}"] \\
\mathcal{T}_{races}(\mathrm{Mem}^{(C)}, \mathrm{Cost}) \arrow[drr] & \approx & \begin{array}{c} \mathcal{T}_{rees}(\mathrm{Mem}^{(HC)}) \\ + \text{ Cost model for builtins} \end{array} \arrow[r, dashed, "\overline{\alpha_{\mathrm{Sz}}}"] \arrow[d] & \begin{array}{c} \mathcal{T}_{rees}(\mathrm{Mem}^{(HC)}_{\mathrm{Size}}) \\ + \text{ Cost model for builtins} \end{array} \arrow[d] & \\
 & & \mathcal{F}_{\mathcal{P}} + \mathrm{Cost}_{\mathcal{P}} \arrow[r, "\alpha^{\circ}_{\mathrm{Sz}}"] & \mathrm{Sz}_{\mathcal{P}} + \mathrm{Cost}_{\mathrm{Sz},\mathcal{P}} \arrow[r, "\alpha^{\circ}_{I}"] & \mathrm{Sz}_{\mathcal{I}} + \mathrm{Cost}_{\mathrm{Sz},\mathcal{I}}
\end{tikzcd}
$$

Code transformation (1)

```
int fact(int n){
    if(n <= 0)
        return 1;
    return n*fact(n-1);
}
```

```
<fact>:
001: entsp   0x2
002: stw     r0, sp[0x1]
003: ldw     r1, sp[0x1]
004: ldc     r0, 0x0
005: lss     r0, r1
006: bf      <008>

007: bu      <010>
010: ldw     r0, sp[0x1]
011: sub     r0, r0, 0x1
012: bl      <fact>

013: ldw     r1, sp[0x1]
014: mul     r0, r1, r0
015: retsp   0x2

008: mkmsk   r0, 0x1
009: retsp   0x2
```

```
fact(R0, R0_3) :-
    entsp(0x2),
    stw(R0, Sp0x1),
    ldw(R1, Sp0x1),
    ldc(R0_1, 0x0),
    lss(R0_2, R0_1, R1),
    bf(R0_2, 0x8) ,
    fact_aux(R0_2, Sp0x1, R0_3, R1_1).

fact_aux(1, Sp0x1, R0_4, R1) :-
    bu(0x0A),
    ldw(R0_1, Sp0x1),
    sub(R0_2, R0_1, 0x1),
    bl(fact),
    fact(R0_2, R0_3),
    ldw(R1, Sp0x1),
    mul(R0_4, R1, R0_3),
    retsp(0x2).

fact_aux(0, Sp0x1, R0, R1) :-
    mkmsk(R0, 0x1),
```

Figure: C program (left) translated into ISA level (middle) and HCIR from ISA (right).

Code transformation (2)

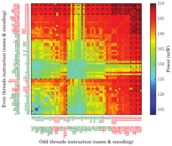
```
int fact(int n){
    if(n <= 0)
        return 1;
    return n*fact(n-1);
}
```

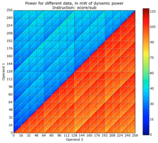
$$\Longrightarrow \quad \begin{cases} \mathrm{Sz}_{\mathtt{fact}}(n) = 1 & \text{when } n \leq 0, \\ \mathrm{Sz}_{\mathtt{fact}}(n) = n \times \mathrm{Sz}_{\mathtt{fact}}(n-1) & \text{otherwise.} \end{cases}$$

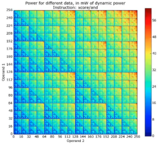
- The recurrence structure of programs appears in the corresponding equations.
- This is a simple case. In general, size measure may introduce abstractions, and we use both size and cost functions.
- When translating imperative loops into recursive programs, ranking functions may have to be inferred.

The energy model problem

- Need to create models by measurements or simulation.
- A simple model might assign a *constant consumption* to *each type of instruction*, or a reasonably tight *interval*.
- More complex models might care about *history* of previous instructions, e.g. *pairs of instructions*, or about value of operands (*data-dependent consumption*).







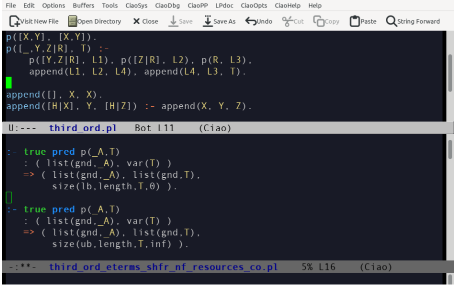
- Can be important to deal with *"hardware's runtime policies"*, e.g. *cache behaviour*. Static analysis techniques exist, but it is hard.
- Choosing the right level of granularity is hard.
  May have to do compromises depending on the application.

Implementation

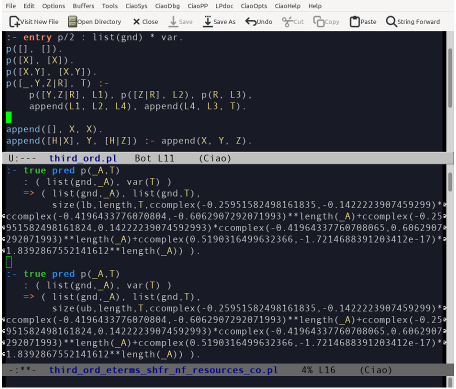
- Adding some classical techniques to the recurrence solver
  - "Dictionary lookup"
  - Rewriting

Adding classical recurrence solving techniques

- Second and third order linear recurrence equations with constant coefficients and a few options for the affine term. Classical method with particular solution + homogeneous solution using roots of characteristic polynomial.
- Required to add complex numbers to CiaoPP (and to its numerical expressions).



Before



After

Irrelevant variables analysis – An analyse + rewrite pass

Example from cost analysis of a `fact` program with accumulator.

$$\begin{cases} f(n,a) = 1 + f(n-1,(n-1)\times a) & \text{if } n > 0, \\ f(n,a) = 0 & \text{if } n \leq 0. \end{cases}$$

Irrelevant variables analysis – An analyse + rewrite pass

Example from cost analysis of a `fact` program with accumulator.

$$\begin{cases} f(n,a) = 1 + f(n-1,(n-1)\times a) & \text{if } n > 0, \\ f(n,a) = 0 & \text{if } n \leq 0. \end{cases}$$

$$\Downarrow$$

$$\begin{cases} \tilde{f}(n) = 1 + \tilde{f}(n-1) & \text{if } n > 0, \\ \tilde{f}(n) = 0 & \text{if } n \leq 0. \end{cases}$$

Irrelevant variables analysis – Fixpoint formulation

$$\begin{cases} f(n,a) = 1 + f(n-1,(n-1)\times a) & \text{if } n > 0, \\ f(n,a) = 0 & \text{if } n \leq 0. \end{cases}$$

- In general, we do this by overapproximating the set of *relevant indices*, as the lfp of the following operator.

$$\begin{aligned} F : \mathcal{P}(\llbracket 1,k \rrbracket) &\to \mathcal{P}(\llbracket 1,k \rrbracket) \\ I &\mapsto I \cup \{ i \mid n_i \text{ appears in a condition } \phi_j \} \\ &\qquad \cup \left\{ i \;\middle|\; \begin{array}{l} n_i \text{ appears in an expression } \Psi_j \\ \text{via a path going only through } f \text{ via indices } i' \in I \end{array} \right\} \end{aligned}$$

Irrelevant variables analysis – Fixpoint formulation

$$\begin{cases} f(n,a) = 1 + f(n-1,(n-1)\times a) & \text{if } n > 0, \\ f(n,a) = 0 & \text{if } n \leq 0. \end{cases}$$

- In general, we do this by overapproximating the set of *relevant indices*, as the lfp of the following operator.

$$\begin{aligned} F : \mathcal{P}(\llbracket 1,k \rrbracket) &\to \mathcal{P}(\llbracket 1,k \rrbracket) \\ I &\mapsto I \cup \{i \mid n_i \text{ appears in a condition } \phi_j\} \\ &\quad \cup \left\{ i \;\middle|\; \begin{array}{l} n_i \text{ appears in an expression } \Psi_j \\ \text{via a path going only through } f \text{ via indices } i' \in I \end{array} \right\} \end{aligned}$$

- In this example, only n is added, using the boundary conditions:

$$\mathrm{lfp}\, F = \{1\}.$$

Irrelevant variables analysis – Fixpoint formulation (2)

$$\begin{cases} f(n_1, n_2, n_3, n_4, n_5) = 0 & \text{if } n_1 \leq 0 \\ f(n_1, n_2, n_3, n_4, n_5) = 1 + f(n_1 - n_3 - 1, n_1 \times n_2, n_2 \times n_3, n_3 \times n_4, n_4 \times n_5) & \text{if } n_1 > 0. \end{cases}$$

- Again, we overapproximate the set of *relevant indices*, as the lfp of F.

$$\begin{aligned} F : \mathcal{P}(\llbracket 1, k \rrbracket) &\to \mathcal{P}(\llbracket 1, k \rrbracket) \\ I &\mapsto I \cup \{ i \mid n_i \text{ appears in a condition } \phi_j \} \\ &\qquad \cup \left\{ i \;\middle|\; \begin{array}{l} n_i \text{ appears in an expression } \Psi_j \\ \text{via a path going only through } f \text{ via indices } i' \in I \end{array} \right\} \end{aligned}$$

Irrelevant variables analysis – Fixpoint formulation (2)

$$\begin{cases} f(n_1, n_2, n_3, n_4, n_5) = 0 & \text{if } n_1 \leq 0 \\ f(n_1, n_2, n_3, n_4, n_5) = 1 + f(n_1 - n_3 - 1, n_1 \times n_2, n_2 \times n_3, n_3 \times n_4, n_4 \times n_5) & \text{if } n_1 > 0. \end{cases}$$

- Again, we overapproximate the set of *relevant indices*, as the lfp of F.

$$\begin{aligned} F : \mathcal{P}(\llbracket 1, k \rrbracket) &\to \mathcal{P}(\llbracket 1, k \rrbracket) \\ I &\mapsto I \cup \{ i \mid n_i \text{ appears in a condition } \phi_j \} \\ &\quad \cup \left\{ i \;\middle|\; \begin{array}{l} n_i \text{ appears in an expression } \Psi_j \\ \text{via a path going only through } f \text{ via indices } i' \in I \end{array} \right\} \end{aligned}$$

- First, n_1 is relevant because of the boundary conditions.

Irrelevant variables analysis – Fixpoint formulation (2)

$$\begin{cases} f(n_1, n_2, n_3, n_4, n_5) = 0 & \text{if } n_1 \leq 0 \\ f(n_1, n_2, n_3, n_4, n_5) = 1 + f(n_1 - n_3 - 1, n_1 \times n_2, n_2 \times n_3, n_3 \times n_4, n_4 \times n_5) & \text{if } n_1 > 0. \end{cases}$$

- Again, we overapproximate the set of *relevant indices*, as the lfp of F.

$$\begin{aligned} F : \mathcal{P}(\llbracket 1, k \rrbracket) &\to \mathcal{P}(\llbracket 1, k \rrbracket) \\ I &\mapsto I \cup \{ i \mid n_i \text{ appears in a condition } \phi_j \} \\ &\quad \cup \left\{ i \;\middle|\; \begin{array}{l} n_i \text{ appears in an expression } \Psi_j \\ \text{via a path going only through } f \text{ via indices } i' \in I \end{array} \right\} \end{aligned}$$

- First, n_1 is relevant because of the boundary conditions.
- To compute n_1, we need n_3, .

Irrelevant variables analysis – Fixpoint formulation (2)

$$\begin{cases} f(n_1, n_2, n_3, n_4, n_5) = 0 & \text{if } n_1 \leq 0 \\ f(n_1, n_2, n_3, n_4, n_5) = 1 + f(n_1 - n_3 - 1, n_1 \times n_2, n_2 \times n_3, n_3 \times n_4, n_4 \times n_5) & \text{if } n_1 > 0. \end{cases}$$

- Again, we overapproximate the set of *relevant indices*, as the lfp of F.

$$\begin{aligned} F : \mathcal{P}(\llbracket 1, k \rrbracket) &\to \mathcal{P}(\llbracket 1, k \rrbracket) \\ I &\mapsto I \cup \{ i \mid n_i \text{ appears in a condition } \phi_j \} \\ &\qquad \cup \left\{ i \;\middle|\; \begin{array}{l} n_i \text{ appears in an expression } \Psi_j \\ \text{via a path going only through } f \text{ via indices } i' \in I \end{array} \right\} \end{aligned}$$

- First, n_1 is relevant because of the boundary conditions.
- To compute n_1, we need n_3, and thus also n_2.

Irrelevant variables analysis – Fixpoint formulation (2)

$$\begin{cases} f(n_1, n_2, n_3, n_4, n_5) = 0 & \text{if } n_1 \leq 0 \\ f(n_1, n_2, n_3, n_4, n_5) = 1 + f(n_1 - n_3 - 1, n_1 \times n_2, n_2 \times n_3, n_3 \times n_4, n_4 \times n_5) & \text{if } n_1 > 0. \end{cases}$$

- Again, we overapproximate the set of *relevant indices*, as the lfp of F.

$$\begin{aligned} F : \mathcal{P}(\llbracket 1, k \rrbracket) &\to \mathcal{P}(\llbracket 1, k \rrbracket) \\ I &\mapsto I \cup \{ i \mid n_i \text{ appears in a condition } \phi_j \} \\ &\quad \cup \left\{ i \;\middle|\; \begin{array}{l} n_i \text{ appears in an expression } \Psi_j \\ \text{via a path going only through } f \text{ via indices } i' \in I \end{array} \right\} \end{aligned}$$

- First, n_1 is relevant because of the boundary conditions.
- To compute n_1, we need n_3, and thus also n_2.
- $\mathrm{lfp}\, F = \{1, 2, 3\}$, we can rewrite the equation to

$$\begin{cases} \tilde{f}(n_1, n_2, n_3) = 0 & \text{if } n_1 \leq 0 \\ \tilde{f}(n_1, n_2, n_3) = 1 + \tilde{f}(n_1 - n_3 - 1, n_1 \times n_2, n_2 \times n_3) & \text{if } n_1 > 0. \end{cases}.$$

Irrelevant variables analysis – Before/After

File Edit Options Buffers Tools CiaoSys CiaoDbg CiaoPP LPdoc CiaoOpts CiaoHelp Help

Visit New File | Open Directory | Close | Save | Save As | Undo | Cut | Copy | Paste | String Forward | Print Buffer | Analyze Buffer

```
:- module(_,[fact/2],[assertions, regtypes, nativeprops, resdefs]).

:- resource fuel.
:- head_cost(ub, fuel, 0).
:- literal_cost(ub, fuel, 0).
:- default_cost(ub,fuel,0).
:- trust_default+cost(ub,fuel,0).

:- entry fact(N,R) : num * var.
fact(N,R) :-
    A is 1,
    fact_aux(N,A,R).

fact_aux(N,A,R) :-
    N > 0,
    mul(A,N,A1),
    N1 is N - 1,
    fact_aux(N1,A1,R).

fact_aux(0,A,R) :-
    R = A.

:- impl_defined([mul/3]).

:- trust pred mul(X,Y,Z)
    : num * num * var
    => num * num * num
    + (not_fails, is_det, cost(ub,fuel,1), cost(lb,fuel,1)).

mul(X,Y,Z) :-
    Z is X * Y.
```

-:**- fact.pl   All L19   (Ciao)

Mark set

```
:- true pred fact(N,R)
   : ( num(N), var(R) )
   => ( num(N), num(R),
        size(lb,int,R,inf) )
   + cost(lb,fuel,0).

:- true pred fact(N,R)
   : ( num(N), var(R) )
   => ( num(N), num(R),
        size(ub,int,R,inf) )
   + cost(ub,fuel,inf).

fact(N,R) :-
    A is 1,
    fact_aux(N,A,R).

:- true pred fact_aux(N,A,R)
   : ( num(N), num(A), term(R) )
   => ( num(N), num(A), num(R) ).

:- true pred fact_aux(N,A,R)
   : ( mshare([[R]]),
       var(R), ground([N,A]), num(N), num(A), term(R) )
   => ( ground([N,A,R]), num(N), num(A), num(R) )
   + ( not_fails, covered ).

:- true pred fact_aux(N,A,R)
   : ( num(N), num(A), var(R) )
   => ( num(N), num(A), num(R),
        size(lb,int,R,inf) )
   + cost(lb,fuel,inf).

:- true pred fact_aux(N,A,R)
   : ( num(N), num(A), var(R) )
   => ( num(N), num(A), num(R),
        size(ub,int,R,inf) )
   + cost(ub,fuel,inf).
```

-:--- fact_eterms_nf_resources_co.pl   17% L65   (Ciao)

Irrelevant variables analysis – Before/After

File Edit Options Buffers Tools CiaoSys CiaoDbg CiaoPP LPdoc CiaoOpts CiaoHelp Help

Visit New File | Open Directory | Close | Save | Save As | Undo | Cut | Copy | Paste | String Forward | Print Buffer | Analyze Buffer

```prolog
:- module(_,[fact/2],[assertions, regtypes, nativeprops, resdefs]).

:- resource fuel.
:- head_cost(ub, fuel, 0).
:- literal_cost(ub, fuel, 0).
:- default_cost(ub,fuel,0).
:- trust_default+cost(ub,fuel,0).

:- entry fact(N,R) : num * var.
fact(N,R) :-
    A is 1,
    fact_aux(N,A,R).

fact_aux(N,A,R) :-
    N > 0,
    mul(A,N,A1),
    N1 is N - 1,
    fact_aux(N1,A1,R).

fact_aux(0,A,R) :-
    R = A.

:- impl_defined([mul/3]).

:- trust pred mul(X,Y,Z)
    : num * num * var
    => num * num * num
    + (not_fails, is_det, cost(ub,fuel,1), cost(lb,fuel,1)).

mul(X,Y,Z) :-
    Z is X * Y.
```

-:--- fact.pl All L19 (Ciao)

```prolog
:- true pred fact(N,R)
   : ( num(N), var(R) )
   => ( num(N), num(R),
        size(lb,int,R,0) )
   + cost(lb,fuel,0).

:- true pred fact(N,R)
   : ( num(N), var(R) )
   => ( num(N), num(R),
        size(ub,int,R,inf) )
   + cost(ub,fuel,int(N)).

fact(N,R) :-
    A is 1,
    fact_aux(N,A,R).

:- true pred fact_aux(N,A,R)
   : ( num(N), num(A), term(R) )
   => ( num(N), num(A), num(R) ).

:- true pred fact_aux(N,A,R)
   : ( mshare([[R]]),
       var(R), ground([N,A]), num(N), num(A), term(R) )
   => ( ground([N,A,R]), num(N), num(A), num(R) )
   + ( possibly_fails, possibly_not_covered ).

:- true pred fact_aux(N,A,R)
   : ( num(N), num(A), var(R) )
   => ( num(N), num(A), num(R),
        size(lb,int,R,0) )
   + cost(lb,fuel,0).

:- true pred fact_aux(N,A,R)
   : ( num(N), num(A), var(R) )
   => ( num(N), num(A), num(R),
        size(ub,int,R,inf) )
   + cost(ub,fuel,int(N)).
```

-:--- fact_eterms_nf_resources_co.pl 17% L46 (Ciao)

Mark set

Irrelevant variables analysis – Before/After

File Edit Options Buffers Tools CiaoSys CiaoDbg CiaoPP LPdoc CiaoOpts CiaoHelp Help

Visit New File | Open Directory | Close | Save | Save As | Undo | Cut | Copy | Paste | String Forward | Print Buffer | Analyze Buffer

```
:- module(_,[fact/2],[assertions, regtypes, nativeprops, resdefs]).

:- resource fuel.
:- head_cost(ub, fuel, 0).
:- literal_cost(ub, fuel, 0).
:- default_cost(ub,fuel,0).
:- trust_default+cost(ub,fuel,0).

:- entry fact(N,R) : num * var.
fact(N,R) :-
    A is 1,
    fact_aux(N,A,R).

fact_aux(N,A,R) :-
    N > 0,
    mul(A,N,A1),
    N1 is N - 1,
    fact_aux(N1,A1,R).

fact_aux(0,A,R) :-
    R = A.

:- impl_defined([mul/3]).

:- trust pred mul(X,Y,Z)
    : num * num * var
    => num * num * num
    + (not_fails, is_det, cost(ub,fuel,1), cost(lb,fuel,1)).

mul(X,Y,Z) :-
    Z is X * Y.
```

-:--- fact.pl All L19 (Ciao)

Mark set

```
:- true pred fact(N,R)
   : ( num(N), var(R) )
   => ( num(N), num(R),
        size(lb,int,R,0) )
   + cost(lb,fuel,0).

:- true pred fact(N,R)
   : ( num(N), var(R) )
   => ( num(N), num(R),
        size(ub,int,R,inf) )
   + cost(ub,fuel,int(N)).

fact(N,R) :-
    A is 1,
    fact_aux(N,A,R).

:- true pred fact_aux(N,A,R)
   : ( num(N), num(A), term(R) )
   => ( num(N), num(A), num(R) ).

:- true pred fact_aux(N,A,R)
   : ( mshare([[R]]),
       var(R), ground([N,A]), num(N), num(A), term(R) )
   => ( ground([N,A,R]), num(N), num(A), num(R) )
   + ( possibly_fails, possibly_not_covered ).

:- true pred fact_aux(N,A,R)
   : ( num(N), num(A), var(R) )
   => ( num(N), num(A), num(R),
        size(lb,int,R,0) )
   + cost(lb,fuel,0).

:- true pred fact_aux(N,A,R)
   : ( num(N), num(A), var(R) )
   => ( num(N), num(A), num(R),
        size(ub,int,R,inf) )
   + cost(ub,fuel,int(N)).
```

-:--- fact_eterms_nf_resources_co.pl 17% L46 (Ciao)

Other analyse + rewrite passes could be implemented,
e.g. using change of variables or usage of inferred ranking functions.

Order

Automated Approximate Recurrence Solving

- Design a *generic*, *approximate* recurrence solver.

Automated Approximate Recurrence Solving

- Design a *generic*, *approximate* recurrence solver.
- Observation: just like program semantics, solutions of recurrence equations may be seen as (least) fixed points of a well-chosen operator.

$$\begin{cases} f(0) = a \\ f(n) = f(f(n-1)) + 1, \ \forall n \in \mathbb{N}^* \end{cases} \quad \Longleftrightarrow \quad \boxed{\begin{aligned} &S : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \\ &\qquad g \mapsto \left(n \mapsto \begin{cases} a & \text{if } n = 0 \\ g(g(n-1)) + 1 & \text{otherwise} \end{cases} \right) \end{aligned}}$$

Automated Approximate Recurrence Solving

- Design a *generic*, *approximate* recurrence solver.
- Observation: just like program semantics, solutions of recurrence equations may be seen as (least) fixed points of a well-chosen operator.

$$\begin{cases} f(0) = a \\ f(n) = f(f(n-1)) + 1, \ \forall n \in \mathbb{N}^* \end{cases} \quad \Longleftrightarrow \quad \boxed{\begin{aligned} S : (\mathbb{N} \to \mathbb{N}) &\to (\mathbb{N} \to \mathbb{N}) \\ g &\mapsto \left(n \mapsto \begin{cases} a & \text{if } n = 0 \\ g(g(n-1)) + 1 & \text{otherwise} \end{cases} \right) \end{aligned}}$$

- To make the theory work well, restrict ourselves to *monotone* recurrence equations, and use $D \triangleq \mathbb{N}_\infty \to \mathbb{N}_\infty$, which is a *complete lattice*, as the *domain of sequences*.

First idea: Abstract interpretation of Recurrence equations

- With $S : D \to D$ an equation/operator on the concrete domain of sequences, $f_{\mathrm{sol}} = \mathrm{lfp}\, S = \sup_{\alpha \in \mathbf{Ord}} S^{(\alpha)}(\bot)$.
- Use an abstract domain $D^\sharp$ and abstract operator $S^\sharp$, $f_{\mathrm{sol}} \leq \gamma(\mathrm{lfp}\, S^\sharp)$.
- Using subsets of D (e.g. all affine sequences) as $D^\sharp$ doesn't work well, so we use the power trick and use the domain of *sets of abstract bounds*.

Definition (Domain of abstract bounds)

Let A be a domain of abstract sequences, e.g.
$A =$ all affine sequences $\cong \mathbb{N}_\infty \times \mathbb{N} + \{\top_A\}$, with $\phi : A \to D$ a concretisation.
We set $D^\sharp \triangleq (\mathcal{P}(A), \supseteq)$.

Proposition

We have a Galois connection $D \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} D^\sharp$, defined by

$$\begin{aligned} \alpha : D &\to D^\sharp & \gamma : D^\sharp &\to D \\ f &\mapsto \left\{ f^\sharp \in A \,\middle|\, f \leq_D \phi(f^\sharp) \right\}, & U &\mapsto \left(n \mapsto \min_{f^\sharp \in U} \phi(f^\sharp)(n)\right). \end{aligned}$$

Moreover, for any U, $\gamma \circ \alpha(U) = \uparrow U$. In particular, $\alpha(U) = \alpha(\uparrow U)$.

First idea: Abstract interpretation of Recurrence equations

In the ideal case, for an equation like $f(0) = 0$, $f(n) = f(f(n-1)) + 1$ when $n > 0$, we could do

$$
\begin{tikzcd}
D^\sharp \arrow[r, bend left, dotted, "S \circ \phi \text{ on extremal bounds}"] & \mathcal{P}(D) \arrow[l, bend left, dotted, "\text{union of } \alpha\text{'s}"]
\end{tikzcd}
$$

$$
\begin{tikzcd}
\bot = \uparrow\{0n+0\} \arrow[r] \arrow[d, dotted, "\alpha \circ S \circ \gamma \sqsubseteq"] & \{01^*\} \arrow[dl] \\
\uparrow\{1n+0,\ 0n+1\} \arrow[r] & \{1n+0,\ 02^*\} \arrow[dl] \\
\uparrow\left\{ \begin{array}{l} 1n+0, \\ 2n+0,\ 0n+2 \end{array} \right\} \arrow[r] & \{1n+0,\ 03^*\} \arrow[dl] \\
\uparrow\left\{ \begin{array}{l} 1n+0, \\ 3n+0,\ 0n+3 \end{array} \right\} \arrow[d, dashed] & \\
\uparrow\{1n+0,\ \top\} = \uparrow\{1n+0\} &
\end{tikzcd}
$$

First idea: Abstract interpretation of Recurrence equations

In practice, we don't have direct access to S, γ and α, so we need to design *transfer functions*.
An abstract semantic $[\![\cdot]\!]^\sharp : \texttt{Eqs} \to (A \to \mathcal{P}(A))$ is defined and extended to $\texttt{Eqs} \to (\mathcal{P}(A) \to \mathcal{P}(A))$.
For example, with affine functions,

$$[\![\mathrm{Cst}\ c]\!]^\sharp(f^\sharp) = \{(0, c)\}$$
$$[\![\mathrm{n}]\!]^\sharp(f^\sharp) = \{(1, 0)\}$$
$$[\![\mathrm{f}]\!]^\sharp(f^\sharp) = \{f^\sharp\}$$

$$[\![\Diamond]\!]^\sharp(\top_A, g^\sharp) = [\![\Diamond]\!]^\sharp(f^\sharp, \top_A) = \{\top_A\} \quad \text{for } \Diamond \in \{+, \times, \circ, -\}$$
$$[\![+]\!]^\sharp((a_1, b_1), (a_2, b_2)) = \{(a_1 + a_2, b_1 + b_2)\}$$
$$[\![\times]\!]^\sharp((a_1, b_1), (a_2, b_2)) = \{(a_1 a_2 \infty + a_1 b_2 + a_2 b_1, b_1 b_2)\} \quad \text{where we set } 0 \times \infty = \infty$$
$$[\![\circ]\!]^\sharp((a_1, b_1), (a_2, b_2)) = \{(a_1 a_2, a_1 b_2 + b_1)\}$$
$$[\![-]\!]^\sharp((a_1, b_1), (a_2, b_2)) = \{(a_1 - a_2, b_1 - b_2)\}$$

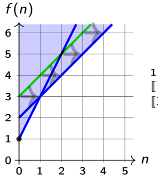
$$[\![\mathrm{Mult}_{in}\ c]\!]^\sharp((a, b)) = \{(ca, b)\}$$
$$[\![\mathrm{Div}_{in}\ c]\!]^\sharp((a, b)) = \left\{\left(\left\lceil \frac{a}{c} \right\rceil, b\right)\right\}$$

$$[\![\mathrm{Push}\ c]\!]^\sharp(f^\sharp) = \begin{cases} \{(\infty, c)\} & \text{if } f^\sharp = \top_A \\ \{(a, c)\} & \text{if } f^\sharp = (a, b) \text{ and } b \leq a + c \\ \{(a, b - a), (b - c, c)\} & \text{if } f^\sharp = (a, b) \text{ and } b > a + c \end{cases}$$

$$[\![\mathrm{Pop}]\!]^\sharp((a, b)) = \begin{cases} \{\top_A\} & \text{if } f^\sharp = \top_A \\ \{(a, b + a)\} & \text{if } f^\sharp = (a, b) \end{cases}$$

$$[\![\mathrm{Shift}\ \delta]\!]^\sharp(f^\sharp) = \begin{cases} ([\![\mathrm{Push}\ 0]\!]^\sharp)^\delta(f^\sharp) & \text{if } \delta \geq 0 \\ \{\top_A\} & \text{if } \delta \leq 0 \text{ and } f^\sharp = \top_A \\ \{(a, b + |\delta| a)\} & \text{if } \delta \leq 0 \text{ and } f^\sharp = (a, b) \end{cases}$$

$$[\![\mathrm{Set}_0\ c]\!]^\sharp(f^\sharp) = [\![\mathrm{Push}\ c]\!]^\sharp \circ [\![\mathrm{Pop}]\!]^\sharp(f^\sharp)$$
$$= \begin{cases} \{(\infty, c)\} & \text{if } f^\sharp = \top_A \\ \{(a, c)\} & \text{if } f^\sharp = (a, b) \text{ and } b \leq c \\ \{(a, b), ((a + b) - c, c)\} & \text{if } f^\sharp = (a, b) \text{ and } b > c \end{cases}$$



$1n + 3$ (green)
$[\![\mathrm{Push}\ 1]\!](1n + 3)$ (frontier)
$[\![\mathrm{Push}\ 1]\!]^\sharp(1n + 3)$ (blue)

Second idea: Pre-/Post-fixpoints as easily checkable bounds

- By the properties of lattices, for a monotone equation/operator S, if $f \geq S(f)$ (prefix), then $f \geq \mathrm{lfp}\, S$. Similarly, if $f \leq S(f)$ (postfix), then $f \leq \mathrm{gfp}\, S$.
- In particular, when we can prove that there is only one solution to $f = S(f)$, we get $f_{\mathrm{sol}} \leq f$ for any prefixpoint and $f \leq f_{\mathrm{sol}}$ for any postfixpoint.
- This gives a new simple proof method for complexity analysis equations.

Example

Consider the following equation, written as a sequence operator. Such equation may arise while doing worst-case analysis of a quicksort program. S is indeed monotone.

$$S : f \mapsto n \mapsto \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \max\limits_{1 \leq k \leq n-1} f(k) + f(n-k) + n & \text{otherwise} \end{cases}$$

- $g : n \mapsto n^2$?

Second idea: Pre-/Post-fixpoints as easily checkable bounds

- By the properties of lattices, for a monotone equation/operator S, if $f \geq S(f)$ (prefix), then $f \geq \operatorname{lfp} S$.
  Similarly, if $f \leq S(f)$ (postfix), then $f \leq \operatorname{gfp} S$.
- In particular, when we can prove that there is only one solution to $f = S(f)$, we get $f_{\mathrm{sol}} \leq f$ for any prefixpoint and $f \leq f_{\mathrm{sol}}$ for any postfixpoint.
- This gives a new simple proof method for complexity analysis equations.

Example

Consider the following equation, written as a sequence operator. Such equation may arise while doing worst-case analysis of a quicksort program. S is indeed monotone.

$$S : f \mapsto n \mapsto \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \max\limits_{1 \leq k \leq n-1} f(k) + f(n-k) + n & \text{otherwise} \end{cases}$$

- $g : n \mapsto n^2$?
- $(Sg)(0) = 0$, $(Sg)(1) = 1$, and $(Sg)(n) = 1^2 + (n-1)^2 + n = n^2 - n + 2$ when $n \geq 2$, thus $Sg \leq g$, thus $f_{\mathrm{sol}}(n) \leq n^2$.

Second idea: Pre-/Post-fixpoints as easily checkable bounds

- By the properties of lattices, for a monotone equation/operator S, if $f \geq S(f)$ (prefix), then $f \geq \operatorname{lfp} S$. Similarly, if $f \leq S(f)$ (postfix), then $f \leq \operatorname{gfp} S$.
- In particular, when we can prove that there is only one solution to $f = S(f)$, we get $f_{\text{sol}} \leq f$ for any prefixpoint and $f \leq f_{\text{sol}}$ for any postfixpoint.
- This gives a new simple proof method for complexity analysis equations.

Example

Consider the following equation, written as a sequence operator. Such equation may arise while doing worst-case analysis of a quicksort program. S is indeed monotone.

$$S : f \mapsto n \mapsto \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \max_{1 \leq k \leq n-1} f(k) + f(n-k) + n & \text{otherwise} \end{cases}$$

- $f_{\text{sol}}(n) \leq n^2$.

Second idea: Pre-/Post-fixpoints as easily checkable bounds

- By the properties of lattices, for a monotone equation/operator S, if $f \geq S(f)$ (prefix), then $f \geq \mathrm{lfp}\, S$. Similarly, if $f \leq S(f)$ (postfix), then $f \leq \mathrm{gfp}\, S$.
- In particular, when we can prove that there is only one solution to $f = S(f)$, we get $f_{\mathrm{sol}} \leq f$ for any prefixpoint and $f \leq f_{\mathrm{sol}}$ for any postfixpoint.
- This gives a new simple proof method for complexity analysis equations.

Example

Consider the following equation, written as a sequence operator. Such equation may arise while doing worst-case analysis of a quicksort program. S is indeed monotone.

$$S : f \mapsto n \mapsto \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \max_{1 \leq k \leq n-1} f(k) + f(n-k) + n & \text{otherwise} \end{cases}$$

- $f_{\mathrm{sol}}(n) \leq n^2$.
- $g : n \mapsto \frac{1}{2}n^2$?

Second idea: Pre-/Post-fixpoints as easily checkable bounds

- By the properties of lattices, for a monotone equation/operator S, if $f \geq S(f)$ (prefix), then $f \geq \operatorname{lfp} S$. Similarly, if $f \leq S(f)$ (postfix), then $f \leq \operatorname{gfp} S$.
- In particular, when we can prove that there is only one solution to $f = S(f)$, we get $f_{\mathrm{sol}} \leq f$ for any prefixpoint and $f \leq f_{\mathrm{sol}}$ for any postfixpoint.
- This gives a new simple proof method for complexity analysis equations.

Example

Consider the following equation, written as a sequence operator. Such equation may arise while doing worst-case analysis of a quicksort program. S is indeed monotone.

$$S : f \mapsto n \mapsto \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \max\limits_{1 \leq k \leq n-1} f(k) + f(n-k) + n & \text{otherwise} \end{cases}$$

- $f_{\mathrm{sol}}(n) \leq n^2$.
- $g : n \mapsto \frac{1}{2}n^2$?
- $(Sg)(0) \geq g(0)$, $(Sg)(1) \geq g(1)$ and $(Sg)(n) = \frac{1}{2}(1^2 + (n-1)^2) + n = \frac{1}{2}n^2 + 1 \geq g(n)$, thus $g \leq Sg$, thus $\frac{1}{2}n^2 \leq f_{\mathrm{sol}}(n)$.

Second idea: Pre-/Post-fixpoints as easily checkable bounds

- By the properties of lattices, if we can prove that there is only one solution to $f = S(f)$, we get $f_{\mathrm{sol}} \leq f$ for any prefixpoint and $f \leq f_{\mathrm{sol}}$ for any postfixpoint.
- This gives a new simple proof method for complexity analysis equations.
- Because it is simple, it is amenable to automation.
  - Use some sort of divide-and-conquer heuristics ?
    Recent papers have improved dichotomy to deal with the lattice $(\llbracket 1, n \rrbracket)^d$ in logarithmic time, but their technique to deal with the case $f \not\lesseqgtr Sf$ isn't easy to translate to general lattices.

  - Use some some of guess-and-check method ?
    A current technique (Lasso + SMT) tries to *fit* the *exact* solution using common complexity functions, and then check equality.
    Guessing *bounds* and then checking *inequality* seems more likely to suceed.
  - Note that automatic check of $f_{\mathrm{sol}} \leq g$ is much harder than $f_{\mathrm{sol}} = g$ given only the equation, and our method provides a way to do so in many cases.

Conclusion

Summary and Next Steps

Contributions

- Implementation – Classical Recurrence Solver
  - Small additions to the "table lookup" side of the solver,
  - New analyse + rewrite pass for irrelevant variable analysis,
  - Various small bugfixes.
- Theory – New order-theoretical point of view
  - Abstract iteration, with a few domains,
  - Pre-/Post-fixpoints as "easily checkable bounds".

Possible Next Steps

- Implementation of algorithms, with their extensions.
- Design of new domains, e.g. using results of experiments with SMT solvers.
- Exploration of the new abstract point of views on "ideal size measures" as optimal abstractions, on $\mathit{HCIR}^{\mathrm{Sz}} \leftrightarrow \texttt{RecEq}$ for rewritings, etc.

- Finish benchmark work on "the recurrence structure of real programs".

- Improvement in translation of imperative programs to equations (inference of ranking functions, output/output size relations, preservation of type properties, ...).
- Usage of analysis for optimisation.
- Improvement in energy models (VHDL, low-level runtime policies, etc.).

- Go further in contacts with industry/associations.
- At some point, include our cost analysis as a plugin of other tools? (Clang, Frama-C, ...).

Thank you !