Horn Clause-based Program Analysis and Verification with CiaoPP

Manuel Hermenegildo[1,2] P. López-García[1,3] J. Morales[1]

I. García-Contreras[1] M. Klemen[1] N. Stulova[1]

[1]IMDEA Software Institute

[2]T. U. of Madrid (UPM)

[3]Spanish Research Council (CSIC)

DPA Workshop @ ECOOP/ISSTA — Jul 18, 2018, Amsterdam

Outline:

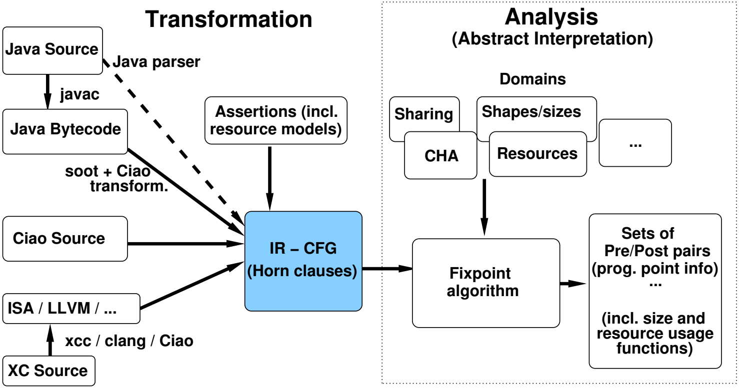
- The CiaoPP Horn clause analyzer.

Some recent results:

- Combining the incremental and the modular fixpoints.
- Energy analysis.
- Static guarantees on run-time checks.

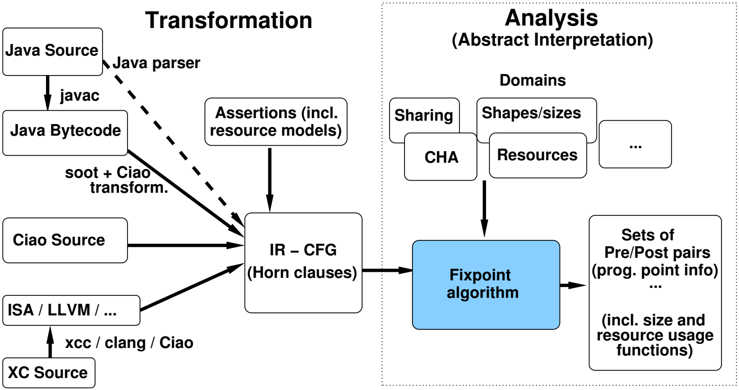
Intermediate Repr.: (Constraint) Horn Clauses (CiaoPP)

[LOPSTR'07]



- Transformation:
  - Source: Program P in L_P + (possibly abstract) Semantics of L_P
  - Target: A (C) Horn Clause program capturing $[\![P]\!]$ (or, possibly, $[\![P]\!]^\alpha$)
- Block-based CFG. Each block represented as a *Horn clause*.
- Used for all analyses: aliasing, CHA/shape/types, data sizes, resources, etc.
- Allows supporting multiple languages.

Analysis: CiaoPP Parametric AI Framework



- Analysis *parametric* w.r.t. abstractions, resources, ... (and languages).
- Efficient fixpoint algorithm for (C)HC IR.

[JLP'92, POPL'94, TOPLAS'99, SAS'96, TOPLAS'00, FTfJP'07, ICLP'18]

[NACLP'89, ICLP'91, ICLP'97, SAS'02, FLOPS'04, LOPSTR'04, PADL'06, PASTE'07]

[VMCAI'08, LCPC'08, PASTE'08, CC'08, ISMM'09, NGC'10, LCPC'08]

Efficient, Parametric Fixpoint Algorithm

- *Generic framework* for implementing HC-based analyses:
  given P (as a set of HCs) and abstract domain(s),
  computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

→ Essentially efficient, incremental, abstract OLDT resolution of HC's.
"Top-down driven, bottom-up computation" (related to magic sets)

Efficient, Parametric Fixpoint Algorithm

- *Generic framework* for implementing HC-based analyses:
  given P (as a set of HCs) and abstract domain(s),
  computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

→ Essentially efficient, incremental, abstract OLDT resolution of HC's.
  "Top-down driven, bottom-up computation" (related to magic sets)

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - ⋆ Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.

Efficient, Parametric Fixpoint Algorithm

- *Generic framework* for implementing HC-based analyses:
  given P (as a set of HCs) and abstract domain(s),
  computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

$\rightarrow$ Essentially efficient, incremental, abstract OLDT resolution of HC's.
"Top-down driven, bottom-up computation" (related to magic sets)

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$:
      (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
    - $Dep(B : \lambda_{inB})$ = the set of entries depending on $B : \lambda_{inB}$.

Efficient, Parametric Fixpoint Algorithm

- *Generic framework* for implementing HC-based analyses:
  given P (as a set of HCs) and abstract domain(s),
  computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

→ Essentially efficient, incremental, abstract OLDT resolution of HC's.
  "Top-down driven, bottom-up computation" (related to magic sets)

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$:
      (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
    - $Dep(B : \lambda_{inB})$ = the set of entries depending on $B : \lambda_{inB}$.

- Characteristics:
  - **Precision:** context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, widening, ...
  - Handles mutually recursive methods.
  - Handles library calls, externals, ...
  - Modular and *incremental*

Efficient, Parametric Fixpoint Algorithm

- *Generic framework* for implementing HC-based analyses:
  given P (as a set of HCs) and abstract domain(s),
  computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

$\rightarrow$ Essentially efficient, incremental, abstract OLDT resolution of HC's.
"Top-down driven, bottom-up computation" (related to magic sets)

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
    - $Dep(B : \lambda_{inB})$ = the set of entries depending on $B : \lambda_{inB}$.

- Characteristics:
  - **Precision:** context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, widening, ...
  - Handles mutually recursive methods.
  - Handles library calls, externals, ...
  - Modular and *incremental* $\rightarrow \rightarrow$ recently combined! $\rightarrow \rightarrow$

Combining the incremental and the modular fixpoints

Analysis running continuously in the background

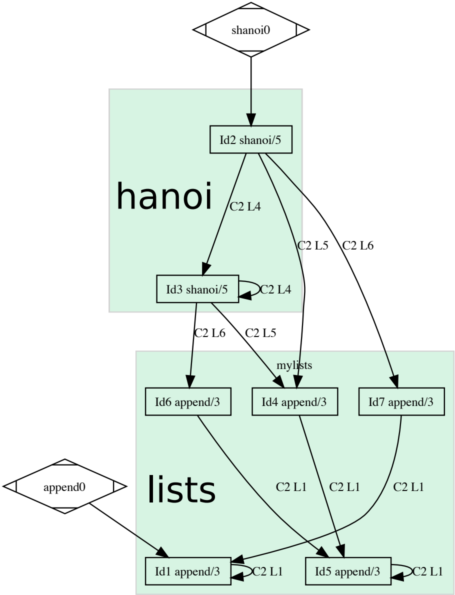
1. We take *"snapshots"* of the program sources (e.g., at each editor save/pause/... while developing).
2. We *detect the changes* w.r.t. the previous snapshot and *reanalyze*:
   - Annotate and remove potentially *outdated information*.
   - (Re-)Analyze *incrementally* (i.e., only parts needed) module by module until an intermodular fixpoint is reached again.

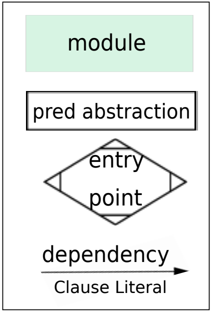
Our previous work:

- *Fine-grain* (block-level) incremental analysis for *non-modular* programs [SAS'96, TOPLAS'00].
- *Coarse-grain* (module level) incremental analysis for *modular* programs [ENTCS'00, LOPSTR'01].

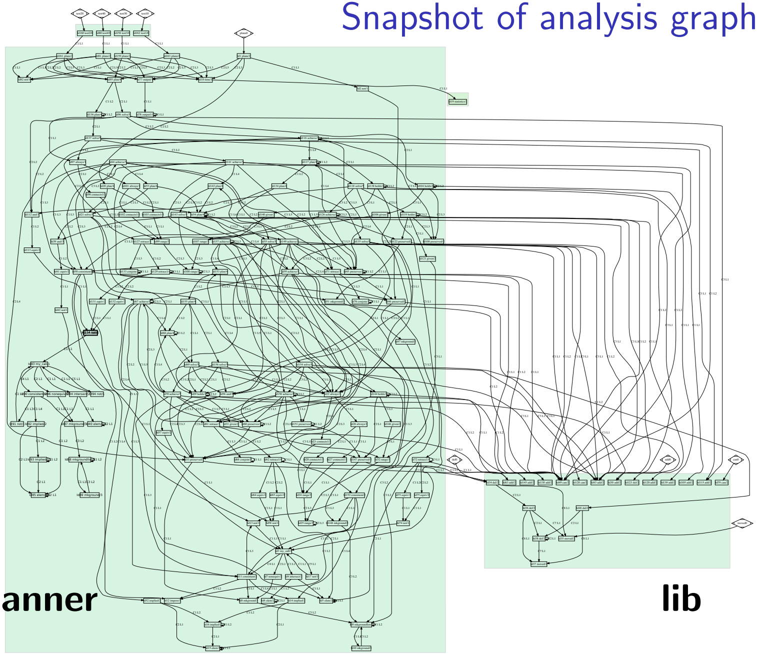
Recent work [ICLP'18]: combine (non-trivial).

Analysis result example





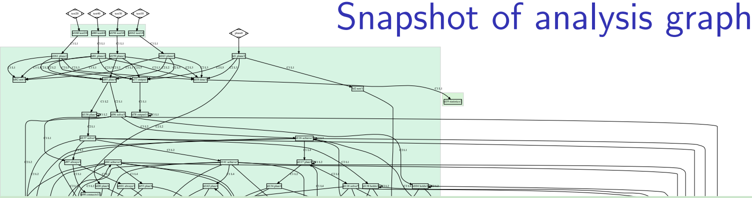
Snapshot of analysis graphs



planner

lib

Snapshot of analysis graphs



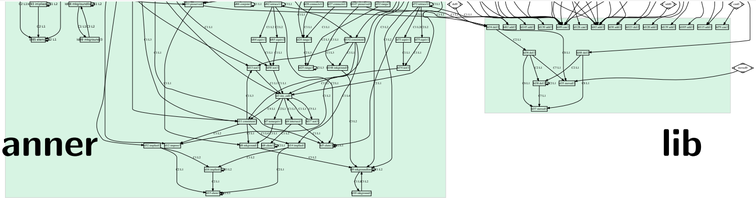
Changes detected!

planner.pl

```
%%
- explore(P,Map,[P|Map]) :-
-     safe(P).
%%
```

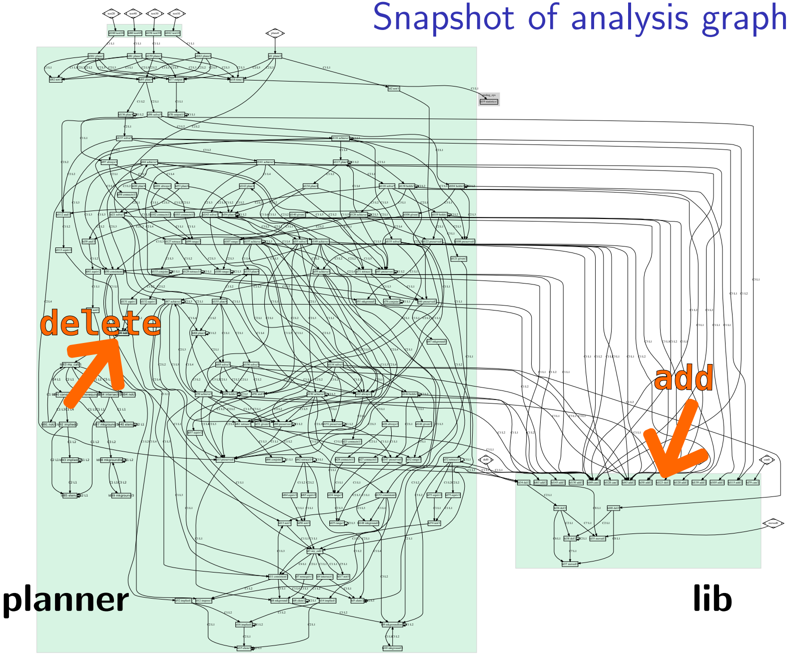
lib.pl

```
%%
+ add(Node,Graph) :-
+       %% implementation
+       %% implementation
%%
```

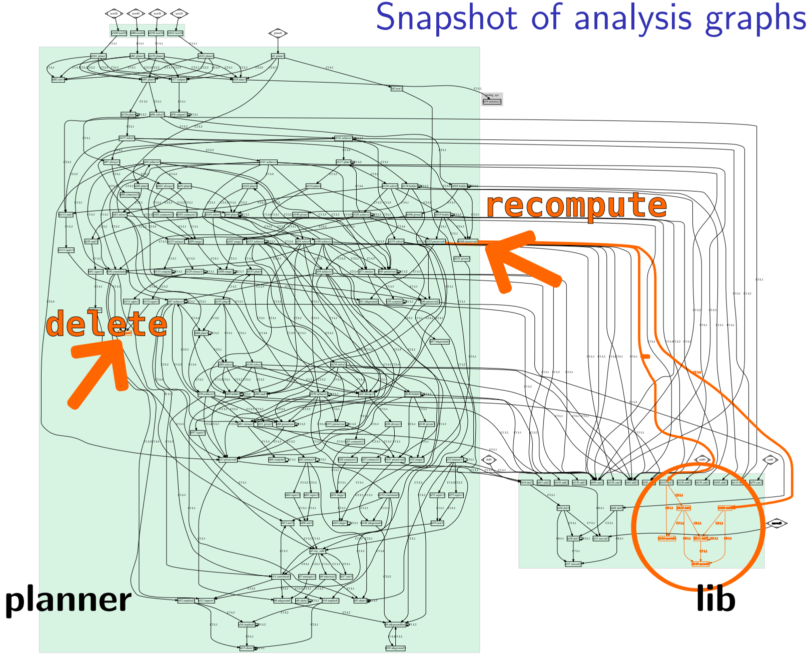


planner **lib**

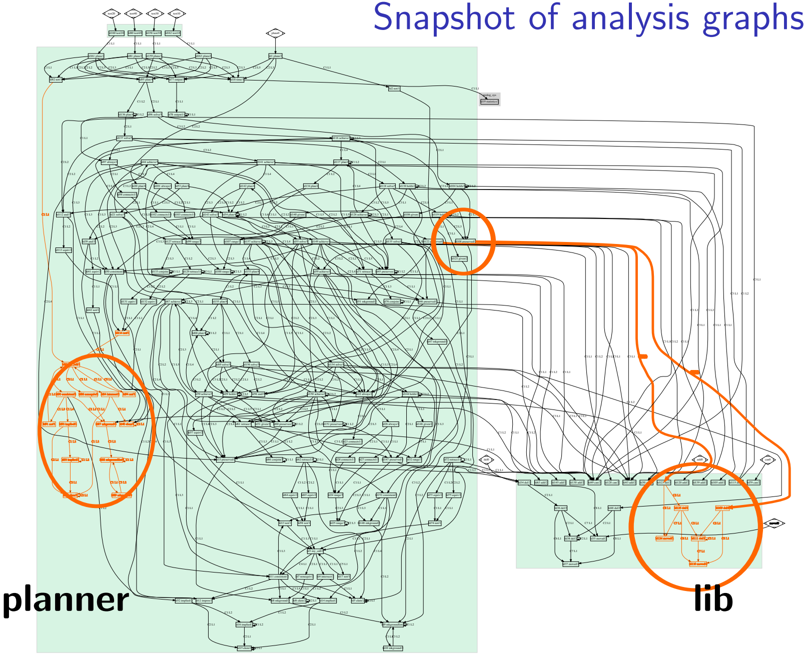
Snapshot of analysis graphs



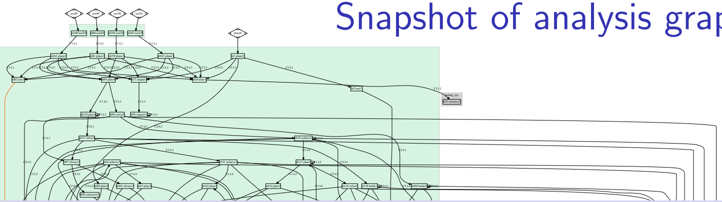
Snapshot of analysis graphs



Snapshot of analysis graphs

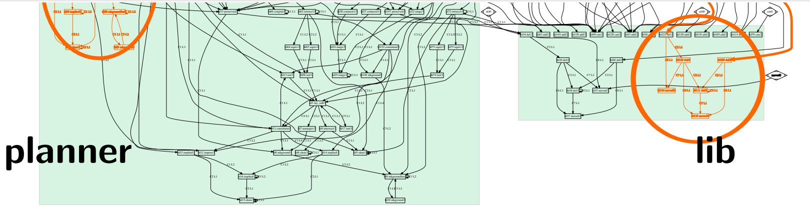


Snapshot of analysis graphs



The algorithm:

- Maintains local and global tables of call/success pairs of the predicates *and their dependencies*.
- Deals incrementally with *additions, deletions*.
- Localizes as possible fixpoint (re)computation inside modules to minimize context swaps.



Fundamental results

Theorem 1 (Base PLAI analysis from scratch) *For a program P and initial λ^cs Es, the PLAI algorithm returns an AT and a DT which represents the* least *program analysis graph of P and Es.*

Proposition 1 (Analyzing a module from scratch) *If module M is analyzed for entries Es within the incremental modular analysis algorithm from scratch (i.e., with no previous information available):*

$$\mathscr{L}^M = \textsc{LocIncAnalyze}(M, Es, \mathcal{G}, (\emptyset,\emptyset), (\emptyset,\emptyset))$$

$\mathscr{L}^M$ will represent the least *module analysis graph of M and Es, assuming $\mathcal{G}$.*

Proposition 2 (Adding clauses to a module) *Given M and M' s.t., $M' = M \cup C_i$,*
$\mathscr{L}^M = \textsc{LocIncAnalyze}(M, Es, \mathcal{G}, (\emptyset,\emptyset), (\emptyset,\emptyset))$, *then*
$\textsc{LocIncAnalyze}(M', Es, \mathcal{G}, (\emptyset,\emptyset), (\emptyset,\emptyset)) =$
$\textsc{LocIncAnalyze}(M, Es, \mathcal{G}, \mathscr{L}^M, (C_i,\emptyset))$

Proposition 3 (Removing clauses from a module) *Given M and M' s.t. $M' = M \setminus C_i$,*
$\mathscr{L}^M = \textsc{LocIncAnalyze}(M, Es, \mathcal{G}, (\emptyset,\emptyset), (\emptyset,\emptyset))$, *then*
$\textsc{LocIncAnalyze}(M', Es, \mathcal{G}, (\emptyset,\emptyset), (\emptyset,\emptyset)) =$
$\textsc{LocIncAnalyze}(M, Es, \mathcal{G}, \mathscr{L}^M, (\emptyset, C_i))$

Proposition 4 (Updating the $\mathscr{L}$) *Given $\mathscr{L}^M = \textsc{LocIncAnalyze}(M, Es, \mathcal{G}, (\emptyset,\emptyset), (\emptyset,\emptyset))$ if $\mathcal{G}$ changes to $\mathcal{G}'$:*
$\textsc{LocIncAnalyze}(M, Es, \mathcal{G}', (\emptyset,\emptyset), (\emptyset,\emptyset)) =$
$\textsc{LocIncAnalyze}(M, Es, \mathcal{G}', \mathscr{L}^M, (\emptyset,\emptyset))$

Proposition 5 (Analyzing modular programs from scratch) *If program P is analyzed for entries Es by the incremental modular analysis algorithm from scratch (with no previous information available):*

$$\mathcal{G} = \textsc{ModIncAnalyze}(P, Es, (\emptyset,\emptyset), (\emptyset,\emptyset))$$

$\mathcal{G}$ will represent the least *modular program analysis graph of* exports(M)*, s.t. $M \in P$.*

Theorem 2 (Modular incremental analysis) *Given modular programs P, P' s.t. $\Delta P = (C_i, C_j)$, $P' = (P \cup C_i) \setminus C_j$, entries Es, and $\mathcal{G} = \textsc{ModIncAnalyze}(P, Es, (\emptyset,\emptyset), (\emptyset,\emptyset))$:*
$\textsc{ModIncAnalyze}(P', Es, \emptyset, (\emptyset,\emptyset)) =$
$\textsc{ModIncAnalyze}(P, Es, \mathcal{G}, \Delta P')$

Fundamental results

Theorem 1 (Base PLAI analysis from scratch)
For a program P and initial λ^cs Es, the PLAI algorithm returns an AT and a DT which represents the least *program analysis graph of P and Es.*

Proposition 1 (Analyzing a module from scratch)

What it means

The results from our incremental, modular analysis are:

- *Correct over-approximations*.
- The most *accurate* (lfp).

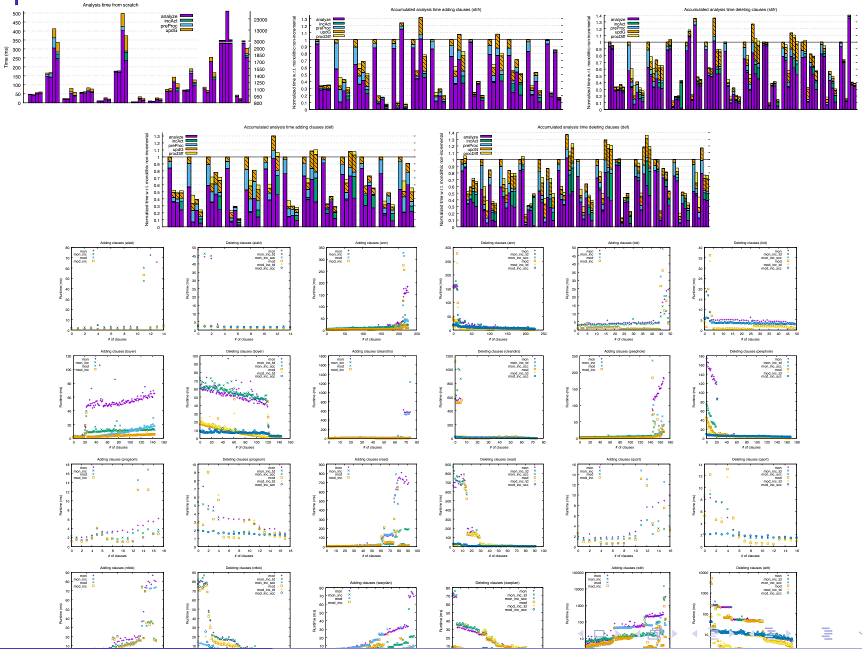
M *and* M' *s.t.,* $M' = M \cup C_i$,
$\mathscr{L}^M = \text{LocIncAnalyze}(M, Es, \mathcal{G}, (\emptyset,\emptyset), (\emptyset,\emptyset))$, *then*
$\text{LocIncAnalyze}(M', Es, \mathcal{G}, (\emptyset,\emptyset), (\emptyset,\emptyset)) =$
$\text{LocIncAnalyze}(M, Es, \mathcal{G}, \mathscr{L}^M, (C_i, \emptyset))$

Proposition 3 (Removing clauses from a module)
Given M *and* M' *s.t.* $M' = M \setminus C_i$,
$\mathscr{L}^M = \text{LocIncAnalyze}(M, Es, \mathcal{G}, (\emptyset,\emptyset), (\emptyset,\emptyset))$, *then*
$\text{LocIncAnalyze}(M', Es, \mathcal{G}, (\emptyset,\emptyset), (\emptyset,\emptyset)) =$
$\text{LocIncAnalyze}(M, Es, \mathcal{G}, \mathscr{L}^M, (\emptyset, C_i))$

Proposition 4 (Updating the $\mathscr{L}$)
Given $\mathscr{L}^M = \text{LocIncAnalyze}(M, Es, \mathcal{G}, (\emptyset,\emptyset), (\emptyset,\emptyset))$ *if* $\mathcal{G}$ *changes to* $\mathcal{G}'$:
$\text{LocIncAnalyze}(M, Es, \mathcal{G}', (\emptyset,\emptyset), (\emptyset,\emptyset)) =$
$\text{LocIncAnalyze}(M, Es, \mathcal{G}', \mathscr{L}^M, (\emptyset,\emptyset))$

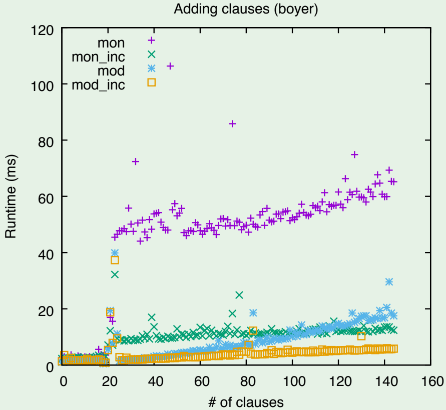
Theorem 2 (Modular incremental analysis)
Given modular programs P, P' *s.t.* $\Delta P = (C_i, C_j)$, $P' = (P \cup C_i) \setminus C_j$, *entries* Es, *and* $\mathcal{G} = \text{ModIncAnalyze}(P, Es, (\emptyset,\emptyset), (\emptyset,\emptyset))$:
$\text{ModIncAnalyze}(P', Es, \emptyset, (\emptyset,\emptyset)) =$
$\text{ModIncAnalyze}(P, Es, \mathcal{G}, \Delta P')$

Experimental results



Experimental results

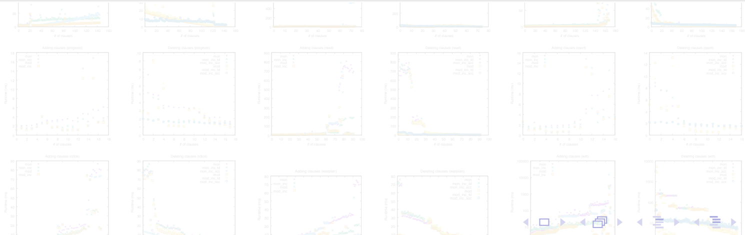
Addition experiment



Experimental results

To take home:

- *Modular Incremental analysis works!* – Up to 60× speedup.
- *Modular analysis* from scratch is *improved* (up to 9×).
- Keeping structures for incrementality produces *small overhead*.
- Using the analyzer *interactively* becomes quite feasible, even for complex abstract domains.



Energy analysis

Energy Consumption Analysis – Approach

Requires low-level modeling – approach: [NASA FM'08]

- Specialize our parametric resource analysis with instruction-level models:
  - Provide energy and data size assertions for each individual instruction. (Energy and data sizes can be constants or *functions*.)
- CiaoPP then generates statically safe upper- and lower-bound energy consumption functions.

⇒ Addressed recently: [LOPSTR'13, FOPARA'15, HIP3ES'16]

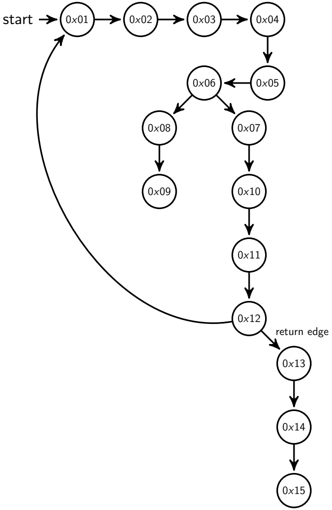
- Analysis of (embedded) programs written in XC, on XMOS processors.
- Using more sophisticated *ISA- and LLVM-level energy models* for XMOS XS1 (Bristol & XMOS).
- Comparing to measured energy consumption.

ENTRA Whole-Systems Energy Transparency · XMOS · Universitas Roskildensis · University of Bristol · imdea software

Transformation example - binaries

Xcore ISA Example: Control Flow Graph (CFG)

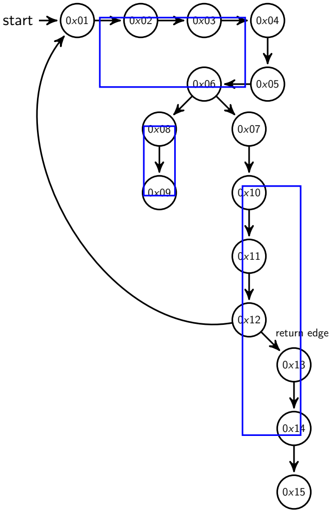
```
<fact>:
0x01: entsp (u6)    0x2
0x02: stw   (ru6)   r0, sp[0x1]
0x03: ldw   (ru6)   r1, sp[0x1]
0x04: ldc   (ru6)   r0, 0x0
0x05: lss   (3r)    r0, r0, r1
0x06: bf    (ru6)   r0, 0x1 <0x08>
0x07: bu    (u6)    0x2 <0x10>
0x08: mkmsk (rus)   r0, 0x1
0x09: retsp (u6)    0x2
0x10: ldw   (ru6)   r0, sp[0x1]
0x11: sub   (2rus)  r0, r0, 0x1
0x12: bl    (u10)   -0xc <fact>
0x13: ldw   (ru6)   r1, sp[0x1]
0x14: mul   (l3r)   r0, r1, r0
0x15: retsp (u6)    0x2
```



Transformation example - binaries

Xcore ISA Example: Block Representation

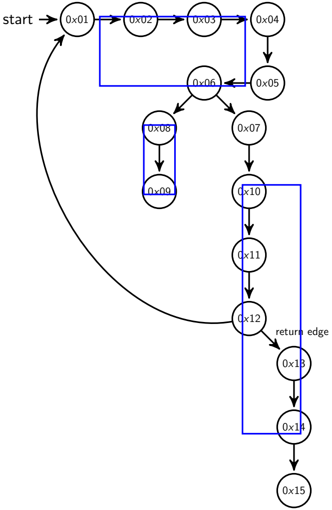
```
<fact>
0x01: entsp (u6)    0x2
0x02: stw  (ru6)    r0, sp[0x1]
0x03: ldw  (ru6)    r1, sp[0x1]
0x04: ldc  (ru6)    r0, 0x0
0x05: lss  (3r)     r0, r0, r1
0x06: bf  (ru6)     r0, 0x1 <0x08>

0x07: bu  (u6)      0x2 <0x10>
0x10: ldw  (ru6)    r0, sp[0x1]
0x11: sub  (2rus)   r0, r0, 0x1
0x12: bl  (u10)     -0xc <fact>
0x13: ldw  (ru6)    r1, sp[0x1]
0x14: mul  (l3r)    r0, r1, r0
0x15: retsp (u6)    0x2

0x08: mkmsk (rus)   r0, 0x1
0x09: retsp (u6)    0x2
```



Transformation example - binaries

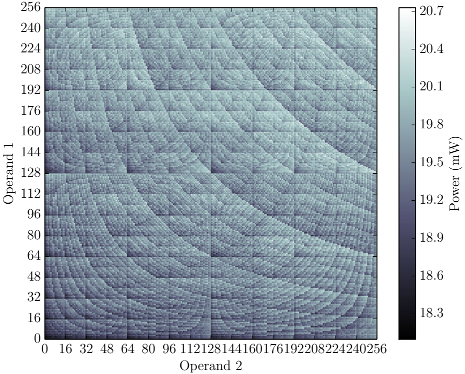
Xcore ISA Example: Constrained Horn Clauses IR

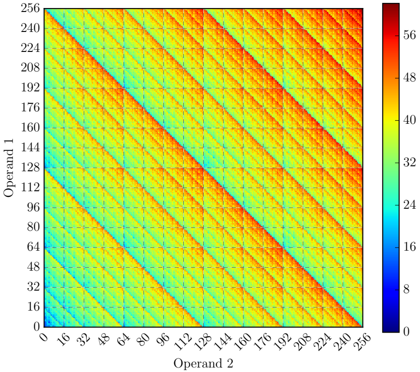
```
:- entry fact/2.
fact(R0,R0_3):-
      entsp(_),
      stw(R0,Sp0x1),
      ldw(R1,Sp0x1),
      ldc(R0_1,0x0),
      lss(R0_2,R0_1,R1),
      bf(R0_2,_),
      bf01(R0_2,Sp0x1,R0_3,R1_1).

bf01(1,Sp0x1,R0_4,R1):-
      bu(_),
      ldw(R0_1,Sp0x1),
      sub(R0_2,R0_1,0x1),
      bl(_),
      fact(R0_2,R0_3),
      ldw(R1,Sp0x1),
      mul(R0_4,R1,R0_3),
      retsp(_).

bf01(0,Sp0x1,R0,R1):-
      mkmsk(R0,0x1),
      retsp(_).
```



Low-level ISA characterization – operand size

Obtaining the cost model: energy consumption/instruction; operand size.

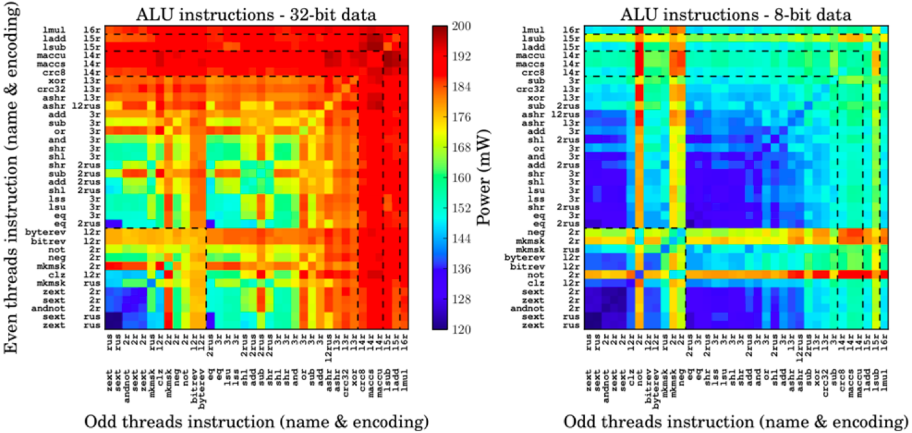




Eder, Kerrison – Bristol U / XMOS.

Low-level ISA characterization – interference

Obtaining the cost model: energy consumption/instruction; interference.



Eder, Kerrison – Bristol U / XMOS.

Energy model, expressed in the Ciao assertion language

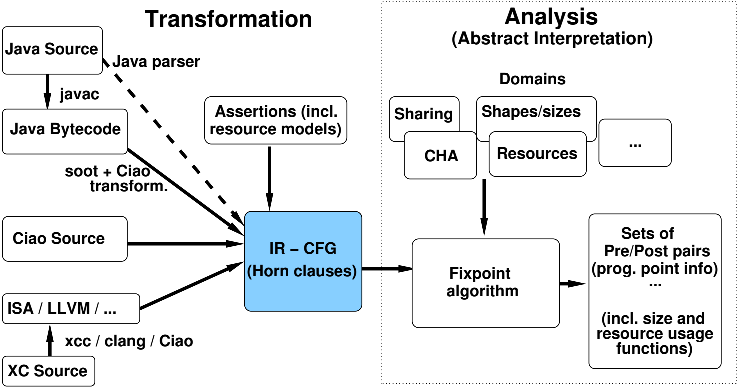
```
:- package(energy).
:- use_package(library(resources(definition))).
:- load_resource_definition(ciaopp(xcore(model(res_energy)))).

:- trust pred mkmsk_rus2(X)
       : var(X) => (num(X), rsize(X,num(A,B)))
       + ( resource(energy, 1112656, 1112656) ).

:- trust pred add_2rus2(X)
       : var(X) => (num(X), rsize(X,num(A,B)))
       + ( resource(energy, 1147788, 1147788) ).

:- trust pred add_3r2(X)
       : var(X) => (num(X), rsize(X,num(A,B)))
       + ( resource(energy, 1215439, 1215439 )).

:- trust pred sub_2rus2(X)
       : var(X) => (num(X), rsize(X, num(A,B)))
       + ( resource(energy, 1150574, 1150574)).

:- trust pred sub_3r2(X)
       : var(X) => (num(X), rsize(X,num(A,B)))
       + ( resource(energy, 1210759, 1210759 )).

:- trust pred ashr_l2rus2(X)
       : var(X) => (num(X), rsize(X,num(A,B)))
       + ( resource(energy, 1219682, 1219682) ).

--:---  energy.pl      Top L1     (Ciao)----------------------------------------
```

Very simple model depicted (constant cost) but real models can include:

- Data properties: operand sizes or other (e.g., number of 1's, bits changing, ...).
- External parameters (voltage, clock, ...).
- List of previous instructions, pipeline state, cache state, etc.

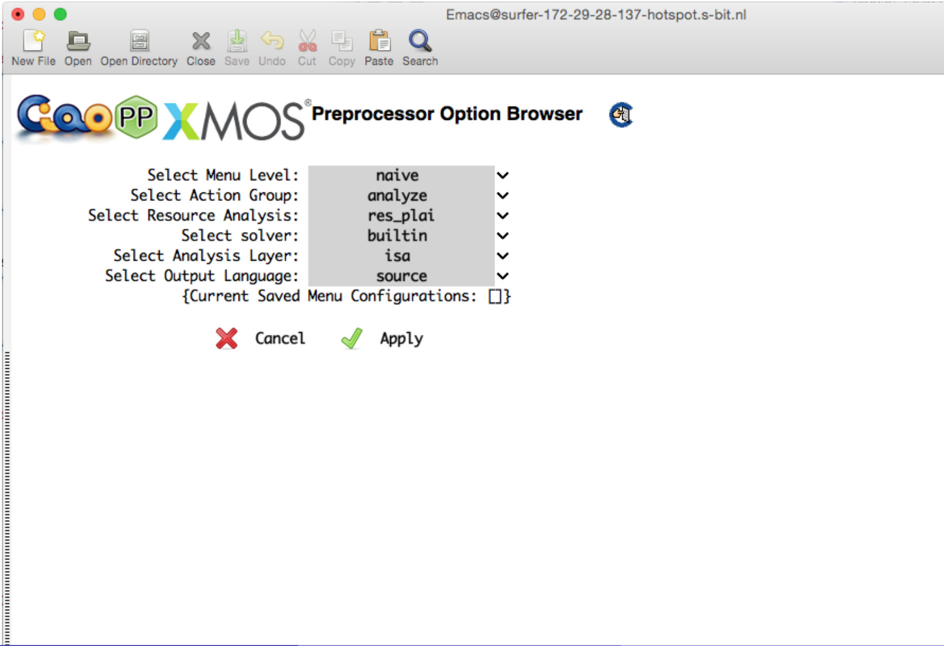
Intermediate Repr.: (Constraint) Horn Clauses (CiaoPP)

[LOPSTR'07]



- Transformation:
  - Source: Program P in L_P + (possibly abstract) Semantics of L_P
  - Target: A (C) Horn Clause program capturing $[\![P]\!]$ (or, possibly, $[\![P]\!]^\alpha$)
- Block-based CFG. Each block represented as a *Horn clause*.
- Used for all analyses: aliasing, CHA/shape/types, data sizes, resources, etc.
- Allows supporting multiple languages.

CiaoPP Menu


Emacs@surfer-172-29-28-137-hotspot.s-bit.nl
New File
Open
Open Directory
Close
Save
Undo
Cut
Copy
Paste
Search
Preprocessor Option Browser
Select Menu Level: naive
Select Action Group: analyze
Select Resource Analysis: res_plai
Select solver: builtin
Select Analysis Layer: isa
Select Output Language: source
{Current Saved Menu Configurations: []}
Cancel
Apply

Analysis Results

fact_results.pl

```
:- module(_,[fact/2],[ciaopp(xcore(model(instructions))),ciaopp(xcore(model(energy))),assertions]).

:- true pred fact(X,Y)
         : ( num(X), var(Y) )
        => ( num(X), num(Y), rsize(X,num(A,B)), rsize(Y,num('Factorial'(A),'Factorial'(B))) )
         + ( resource(energy, 6439360, 21469718 * B + 16420396) ).

fact(X,Y) :-
        entsp_u62(_3459),
        _3467 is X,
        stw_ru62(_3476),
        _3484 is X,
        stw_ru62(_3493),
        _3501 is _3467,
        ldw_ru62(_3510),
        _3518 is 0,
        ldc_ru62(_3527),
        _3518<_3501,
        lss_3r2(_3544),
        bt_ru62(_3552),
        1\=0,
        _3569 is _3467,
        ldw_ru62(_3578),
        _3586 is _3569-1,
        sub_2rus2(_3598),
        _3606 is _3569,
        stw_ru62(_3615),
        _3623 is _3586+0,
--:---  fact_results.pl    Top L11    (Ciao)----------------------------------------
```

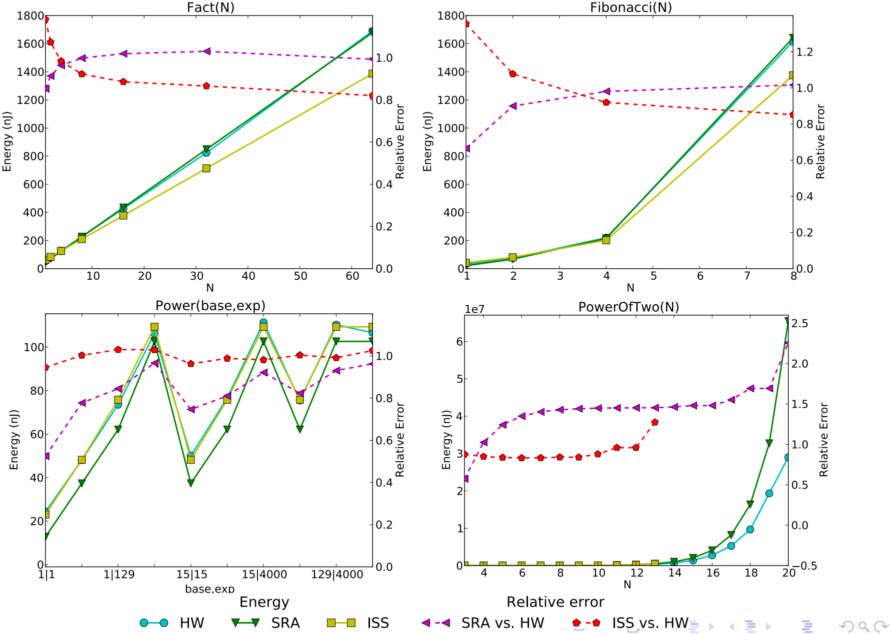
Analysis Output

Emacs@surfer-172-29-28-137-hotspot.s-bit.nl

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

```
#include "fact.h"

#pragma true fact(A) ==> (energy <= 2845229*A+1940746)

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

Some Results [LOPSTR'13]



XC Analysis Results (FIR Filter, LLVM IR level)

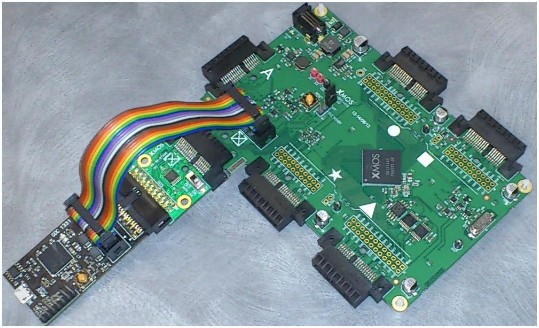
```
int fir(int xn, int coeffs[], int state[], int ELEMENTS)
{
  unsigned int ynl; int ynh;
  ynl = (1<<23); ynh = 0;
  for(int j=ELEMENTS-1; j!=0; j--) {
      state[j] = state[j-1];
      {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl);
  }
  state[0] = xn;
  {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
  if (sext(ynh,24) == ynh) {
      ynh = (ynh << 8) | (((unsigned) ynl) >> 24);}
  else if (ynh < 0) { ynh = 0x80000000; }
  else { ynh = 0x7fffffff; }
  return ynh;
}
```

XC Analysis Results (FIR Filter, LLVM IR level)

```
#pragma true fir(xn, coeffs, state, N) :
              (3347178*N + 13967829 <= energy &&
               energy <= 3347178*N + 14417829)

int fir(int xn, int coeffs[], int state[], int ELEMENTS)
{
  unsigned int ynl; int ynh;
  ynl = (1<<23); ynh = 0;
  for(int j=ELEMENTS-1; j!=0; j--) {
      state[j] = state[j-1];
      {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl);
  }
  state[0] = xn;
  {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
  if (sext(ynh,24) == ynh) {
      ynh = (ynh << 8) | (((unsigned) ynl) >> 24);}
  else if (ynh < 0) { ynh = 0x80000000; }
  else { ynh = 0x7fffffff; }
  return ynh;
}
```

Measuring Power Consumption on the Hardware

- XMOS XTAG3 measurement circuit.
- Plugs into XMOS XS1 board.



We compare these HW measurements with:

- Static Resource Analysis (SRA).
- Instruction Set Simulation (ISS).

Accuracy vs. HW measurements (ISA and LLVMIR)

[FOPARA'15]

Program	Error vs. HW		ISA/LLVMIR
	isa	llvmir	
fact(N)	2.86%	4.50%	0.94
fibonacci(N)	5.41%	11.94%	0.92
sqr(N)	1.49%	9.31%	0.91
power_of_two(N)	4.26%	11.15%	0.93
Average	**3.50%**	**9.20%**	**0.92**
reverse(N,M)	N/A	2.18%	N/A
concat(N,M)	N/A	8.71%	N/A
mat_mult(N,M)	N/A	1.47%	N/A
sum_facts(N,M)	N/A	2.42%	N/A
fir(N)	N/A	0.63%	N/A
biquad(N)	N/A	2.34%	N/A
Average	**N/A**	**3.0%**	**N/A**
Gobal Avg.	**3.50%**	**5.48%**	**0.92**

Accuracy vs. HW measurements (ISA and LLVMIR)

[FOPARA'15]

- ISA analysis estimations are reasonably accurate.
- ISA estimations are more accurate than LLVM estimations.
- LLVM estimations are close to ISA estimations.
- Some programs cannot be analysed at the ISA level but can be analyzed at the LLVM level.

XC Program (FIR Filter) w/Energy Specification [HIP3ES'15]

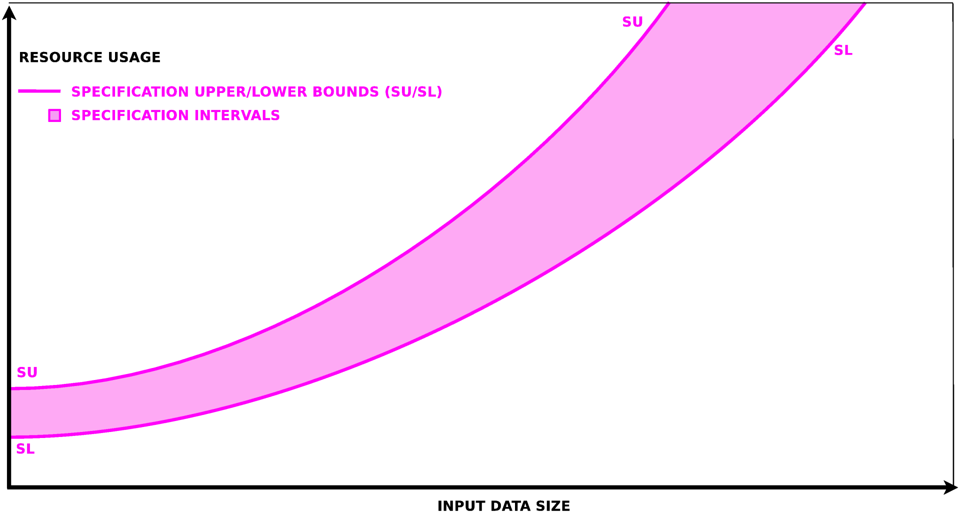
```
#pragma check fir(xn, coeffs, state, N) :
          (1 <= N) ==> (energy <= 416079189)
```

```
int fir(int xn, int coeffs[], int state[], int ELEMENTS)
{
  unsigned int ynl; int ynh;
  ynl = (1<<23); ynh = 0;
  for(int j=ELEMENTS-1; j!=0; j--) {
      state[j] = state[j-1];
      {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl);
  }
  state[0] = xn;
  {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
  if (sext(ynh,24) == ynh) {
      ynh = (ynh << 8) | (((unsigned) ynl) >> 24);}
  else if (ynh < 0) { ynh = 0x80000000; }
  else { ynh = 0x7fffffff; }
  return ynh;
}
```

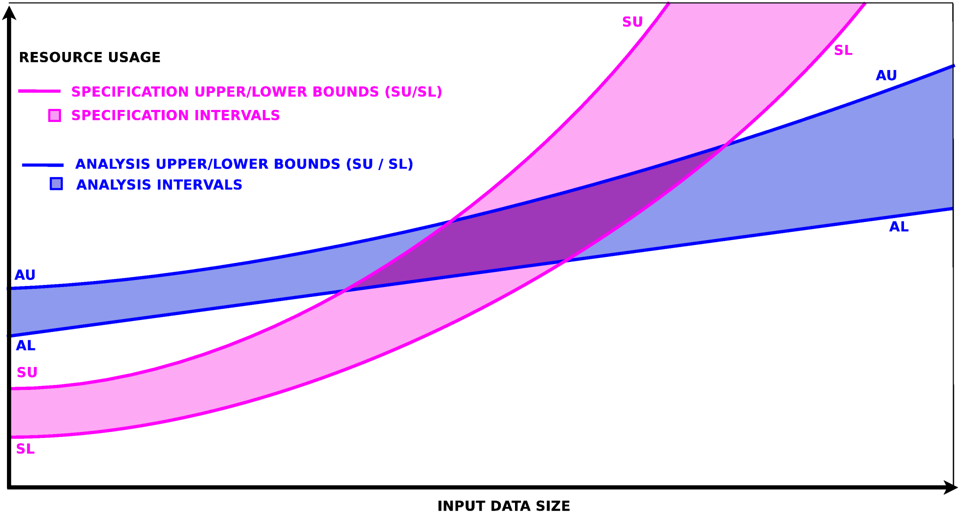
XC Program (FIR Filter) w/Energy Specification [HIP3ES'15]

```
#pragma check fir(xn, coeffs, state, N) :
          (1 <= N) ==> (energy <= 416079189)

#pragma true fir(xn, coeffs, state, N) :
             (3347178*N + 13967829 <= energy &&
              energy <= 3347178*N + 14417829)

#pragma checked fir(xn, coeffs, state, N) :
             (1 <= N && N <= 120) ==> (energy <= 416079189)

#pragma false fir(xn, coeffs, state, N) :
             (121 <= N) ==> (energy <= 416079189)

int fir(int xn, int coeffs[], int state[], int ELEMENTS)
{
  unsigned int ynl; int ynh;
  ynl = (1<<23); ynh = 0;
  for(int j=ELEMENTS-1; j!=0; j--) {
      state[j] = state[j-1];
      {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl);
  }
  state[0] = xn;
  {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
  if (sext(ynh,24) == ynh) {
      ynh = (ynh << 8) | (((unsigned) ynl) >> 24);}
  else if (ynh < 0) { ynh = 0x80000000; }
  else { ynh = 0x7fffffff; }
  return ynh;
}
```

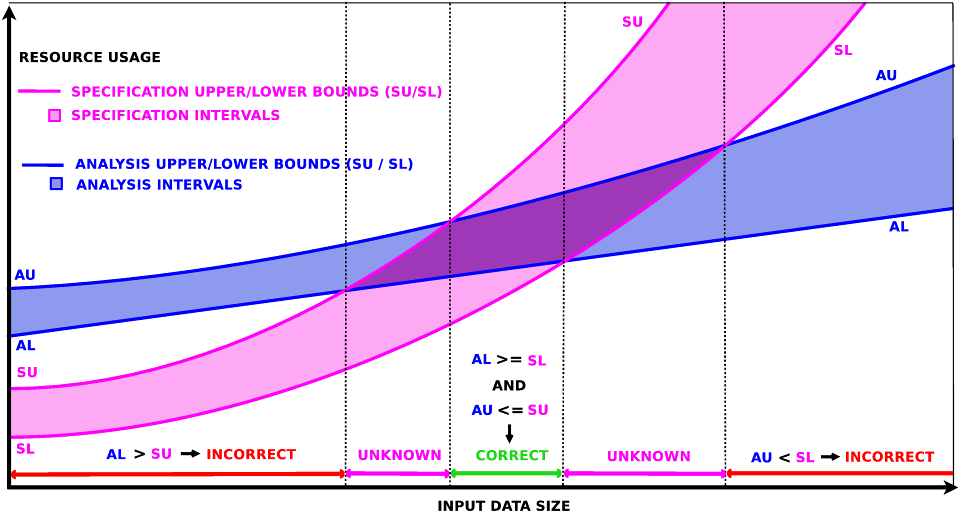
Resource Usage Verification – Function Comparisons
[ICLP'10, FOPARA'12]



Resource Usage Verification – Function Comparisons [ICLP'10, FOPARA'12]



Resource Usage Verification – Function Comparisons
[ICLP'10, FOPARA'12]



Static performance guarantees for programs with run-time checks

Our Static Cost Analysis (SCA) [PLDI'90, SAS'94, ILPS'97, ICLP'07, TPLP'14, TPLP'16]

Example

Consider the following predicate (`rev/2`) for reversing a list of terms.

```
:- pred rev/2:list*var.
rev([], []).
rev([X|Xs], Y):-
    rev(Xs, Ys),
    app1(Ys,X,Y).
```

```
app1([],X,[X]).
app1([E|Y],X,[E|T]):-
    app1(Y,X,T).
```

Our Static Cost Analysis (SCA) [PLDI'90, SAS'94, ILPS'97, ICLP'07, TPLP'14, TPLP'16]

Example

Consider the following predicate (`rev/2`) for reversing a list of terms.

```
:- pred rev/2:list*var.
rev([], []).
rev([X|Xs], Y):-
    rev(Xs, Ys),
    app1(Ys,X,Y).
```

```
app1([],X,[X]).
app1([E|Y],X,[E|T]):-
    app1(Y,X,T).
```

Result of SCA:

```
:- true pred rev(X,Y)
   : (list(X),var(Y),length(X,L))
  => (list(Y), length(Y,L))
   + cost(exact(1/2 L^2 + 3/2 L + 1)).
```

$$\texttt{cost(exact(}\tfrac{1}{2}L^2 + \tfrac{3}{2}L + 1\texttt{))}$$

```
:- true pred app1(X,Y,Z)
   : (list(X),var(Z),length(X,L))
  => (list(Z), length(Z,L+1))
   + cost(exact(L)).
```

Our Static Cost Analysis (SCA) [PLDI'90, SAS'94, ILPS'97, ICLP'07, TPLP'14, TPLP'16]

Example

$$\texttt{cost(exact(}\frac{1}{2}L^2 + \frac{3}{2}L + 1\texttt{))}$$

$$\texttt{cost(exact(}L\texttt{))}$$

```
rev([X|Xs], Y) :-
    rev(Xs, Ys),
    app1(Ys,X,Y).
```

Result of SCA:

```
:- true pred rev(X,Y)
   : (list(X),var(Y),length(X,L))
  => (list(Y), length(Y,L))
   + cost(exact(1/2 L^2 + 3/2 L + 1)).
```

$\texttt{cost(exact(}\frac{1}{2}L^2 + \frac{3}{2}L + 1\texttt{))}$

```
:- true pred app1(X,Y,Z)
   : (list(X),var(Z),length(X,L))
  => (list(Z), length(Z,L+1))
   + cost(exact(L)).
```

$\texttt{cost(exact(}L\texttt{))}$

Our Static Cost Analysis (SCA) [PLDI'90, SAS'94, ILPS'97, ICLP'07, TPLP'14, TPLP'16]

Example

Co length(X,L)) v/2) for reversing a list of terms.

```
rev([], []).                          app1([],X,[X]).
rev([X|Xs], Y):-      length(X,L))    [E|T]):-
    rev(Xs, Ys),                      T).
    app1(Ys,X,Y).
```

Result of SCA:

```
:- true pred rev(X,Y)                     :- true pred app1(X,Y,Z)
   : (list(X),var(Y),length(X,L))            : (list(X),var(Z),length(X,L))
  => (list(Y), length(Y,L))                 => (list(Z), length(Z,L+1))
   + cost(exact(1/2 L^2 + 3/2 L + 1)).       + cost(exact(L)).
```

$$\text{cost}(\text{exact}(\tfrac{1}{2}L^2 + \tfrac{3}{2}L + 1)) \qquad \text{cost}(\text{exact}(L))$$

Run-time Checks - Assertions and Admissible Overhead

check assertions specify pre- and post-conditions for calls to a given predicate.

Example (contd.)

```
:- check pred rev/2
   : list*var => list*list.
:- check pred app1/3
   : list*term*var => list*term*list.
rev([], []).
rev([X|Xs], Y):-
   rev(Xs, Ys), app1(Ys,X,Y).
app1([],X,[X]).
app1([E|Y],X,[E|T]):- app1(Y,X,T).
```

Run-time Checks - Assertions and Admissible Overhead

check assertions specify pre- and post-conditions for calls to a given predicate.

Example (contd.)

```
:- check pred rev/2
   : list*var => list*list.
:- check pred app1/3
   : list*term*var => list*term*list.
rev([], []).
rev([X|Xs], Y):-
    rev(Xs, Ys), app1(Ys,X,Y).
app1([],X,[X]).
app1([E|Y],X,[E|T]):- app1(Y,X,T).
```

Program code

Run-time Checks - Assertions and Admissible Overhead

check assertions specify pre- and post-conditions for calls to a given predicate.

Example (contd.)

Assertions

```
:- check pred rev/2
    : list*var => list*list.
:- check pred app1/3
    : list*term*var => list*term*list.
```

Program code

```
rev([], []).
rev([X|Xs], Y):-
    rev(Xs, Ys), app1(Ys,X,Y).
app1([],X,[X]).
app1([E|Y],X,[E|T]):- app1(Y,X,T).
```

Run-time Checks - Instrumentation

Program instrumented with run-time checking code (assuming no analysis, i.e., full RT checks).

```
rev(A,B) :-
    revC(A,B,C),
    rev_(A,B),
    revS(A,B,C).

revC(A,B,E) :-
    reify_check(list(A),C),
    reify_check(var(B), D),
    E is C/\D,
    warn_if_false(E,calls).

revS(A,B,E) :-
     reify_check(list(A),C),
     reify_check(list(B),D),
     F is C/\D,G is (E#1)\/F,
     warn_if_false(G,success).

 rev_([],[]).
 rev_([X|Xs],Y) :-
     rev(Xs,Ys),
     app1(Ys,X,Y).
```

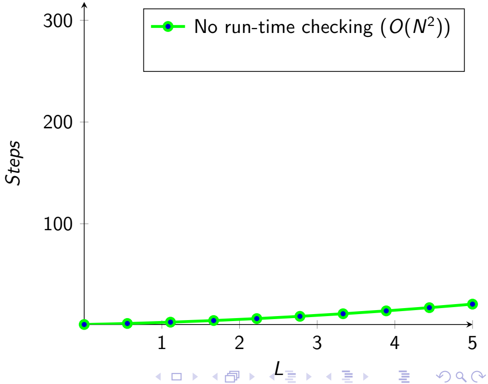
```
app1(A,B,C) :-
    app1C(A,B,C,D),
    app1_(A,B,C),
    app1S(A,B,C,D).

app1C(A,B,C,G) :-
    reify_check(list(A),D),
    reify_check(term(B),E),
    reify_check(var(C),F),
    G is D/\(E/\F),
    warn_if_false(G,calls).

app1S(A,B,C,G) :-
    reify_check(list(A),D),
    reify_check(term(B),E),
    reify_check(list(C),F),
    H is D/\E/\F,K is (G#1)\/H,
    warn_if_false(K,success).

app1_([],X,[X]).
app1_([E|Y],X,[E|T]) :-
    app1(Y,X,T).
```

Run-time Checks - Analysis Results (1)

Our Static Cost Analysis analyzes both the original and the instrumented version .

[PLDI'90, SAS'94, ILPS'97, ICLP'07, TPLP'14, TPLP'16]

```
:- true pred rev(X,Y)
 : (list(X),var(Y),length(X,L))
 =>(list(Y), length(Y,L))
 + cost(exact(1/2 L^2 + 3/2 L + 1)).
```



Run-time Checks - Analysis Results (1)

Our Static Cost Analysis analyzes both the original and the instrumented version.

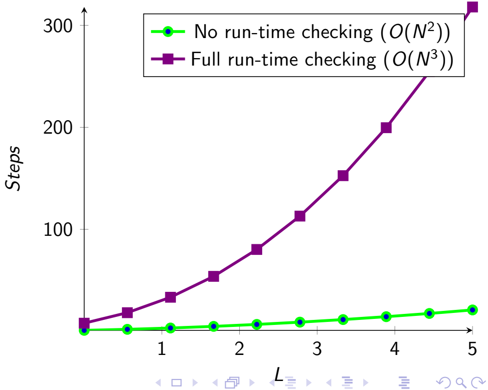
[PLDI'90, SAS'94, ILPS'97, ICLP'07, TPLP'14, TPLP'16] [PPDP'18]

```
:- true pred rev(X,Y)
 : (list(X),var(Y),length(X,L))
 =>(list(Y), length(Y,L))
 + cost(exact(1/2 L^2 + 3/2 L + 1)).
```

$\frac{1}{2}L^2 + \frac{3}{2}L + 1$

```
:- true pred rev(X,Y)
 : (list(X),var(Y),length(X,L))
 =>(list(Y), length(Y,L))
 + cost(exact(1/2 L^3 + 7L^2 + 29/2 L + 8)).
```

$\frac{1}{2}L^3 + 7L^2 + \frac{29}{2}L + 8$



Run-time Checks - Instrumentation

Program instrumented with run-time checking code (assuming no analysis, i.e., full RT checks).

```
rev(A,B) :-
    revC(A,B,C),
    rev_(A,B),
    revS(A,B,C).

revC(A,B,E) :-
    reify_check(list(A),C),
    reify_check(var(B), D),
    E is C/\D,
    warn_if_false(E,calls).

revS(A,B,E) :-
     reify_check(list(A),C),
     reify_check(list(B),D),
     F is C/\D,G is (E#1)\/F,
     warn_if_false(G,success).

 rev_([],[]).
 rev_([X|Xs],Y) :-
     rev(Xs,Ys),
     app1(Ys,X,Y).
```

```
app1(A,B,C) :-
    app1C(A,B,C,D),
    app1_(A,B,C),
    app1S(A,B,C,D).

app1C(A,B,C,G) :-
    reify_check(list(A),D),
    reify_check(term(B),E),
    reify_check(var(C),F),
    G is D/\(E/\F),
    warn_if_false(G,calls).

app1S(A,B,C,G) :-
    reify_check(list(A),D),
    reify_check(term(B),E),
    reify_check(list(C),F),
    H is D/\E/\F,K is (G#1)\/H,
    warn_if_false(K,success).

app1_([],X,[X]).
app1_([E|Y],X,[E|T]) :-
    app1(Y,X,T).
```

Run-time Checks - Instrumentation

Program instrumented with run-time checking code (assuming no analysis, i.e., full RT checks).

```
rev(A,B) :-
    revC(A,B,C),
    rev_(A,B),
    revS(A,B,C).

revC(A,B,E) :-
    reify_check(list(A),C),
    reify_check(var(B), D),
    E is C/\D,
    warn_if_false(E,calls).

revS(A,B,E) :-
     reify_check(list(A),C),
     reify_check(list(B),D),
     F is C/\D,G is (E#1)\/F,
     warn_if_false(G,success).

 rev_([],[]).
 rev_([X|Xs],Y) :-
      rev(Xs,Ys),
      app1(Ys,X,Y).
```

```
app1(A,B,C) :-
    app1C(A,B,C,D),
                 ,
                D).

                 -
                list(A),D),
                term(B),E),
                var(C),F),
                F),
                e(G,calls).

                 -
                list(A),D),
                term(B),E),
                (list(C),F),
    H is D/\E/\F,K is (G#1)\/H,
    warn_if_false(K,success).

app1_([],X,[X]).
app1_([E|Y],X,[E|T]) :-
     app1(Y,X,T).
```

list/1

```
:- true pred list(X)
   : length(X,L)
   + cost(exact(L+1)).

:- regtype list/1.
list([]).
list([_|T]):-
   list(T).
```

Run-time Checks - Instrumentation

Program instrumented with run-time checking code (assuming no analysis, i.e., full RT checks).

```
rev(A,B) :-
    revC(A,B,C),
    rev_(A,B),
    revS(A,B,C).

revC(A,B,E) :-
    reify_check(list(A),C),
    reify_check(var(B), D),
    E is C/\D,
    warn_if_false(E,calls).

revS(A,B,E) :-
     reify_check(list(A),C),
     reify_check(list(B),D),
     F is C/\D,G is (E#1)\/F,
     warn_if_false(G,success).

 rev_([],[]).
 rev_([X|Xs],Y) :-
      rev(Xs,Ys),
      app1(Ys,X,Y).
```

```
app1(A,B,C) :-
    app1C(A,B,C,D),
    app1_(A,B,C),
    app1S(A,B,C,D).

app1C(A,B,C,G) :-
    reify_check(list(A),D),
    reify_check(term(B),E),
    reify_check(var(C),F),
    G is D/\(E/\F),
    warn_if_false(G,calls).

app1S(A,B,C,G) :-
    reify_check(list(A),D),
    reify_check(term(B),E),
    reify_check(list(C),F),
    H is D/\E/\F,K is (G#1)\/H,
    warn_if_false(K,success).

app1_([],X,[X]).
app1_([E|Y],X,[E|T]) :-
     app1(Y,X,T).
```

Run-time Checks - Assertions and Admissible Overhead

We can also specify the admissible run-time overhead for a set of predicates.

Example (contd.)

```
:- check pred *   % Applies to all preds
   + cost(so_ub(constant),[steps,rtc_ratio]).
```

Assertions

```
:- check pred rev/2
   : list*var => list*list.
:- check pred app1/3
   : list*term*var => list*term*list.
```

Program code

```
rev([], []).
rev([X|Xs], Y):-
    rev(Xs, Ys), app1(Ys,X,Y).
app1([],X,[X]).
app1([E|Y],X,[E|T]):- app1(Y,X,T).
```

Run-time Checks - Assertions and Admissible Overhead

We can also specify the admissible run-time overhead for a set of predicates.

Example (contd.)

Admissible RT Overhead

```
:- check pred *  % Applies to all preds
   + cost(so_ub(constant),[steps,rtc_ratio]).
```

Assertions

```
:- check pred rev/2
   : list*var => list*list.
:- check pred app1/3
   : list*term*var => list*term*list.
```

Program code

```
rev([], []).
rev([X|Xs], Y):-
    rev(Xs, Ys), app1(Ys,X,Y).
app1([],X,[X]).
app1([E|Y],X,[E|T]):- app1(Y,X,T).
```

Run-time Checks - Analysis Results (1 contd.)

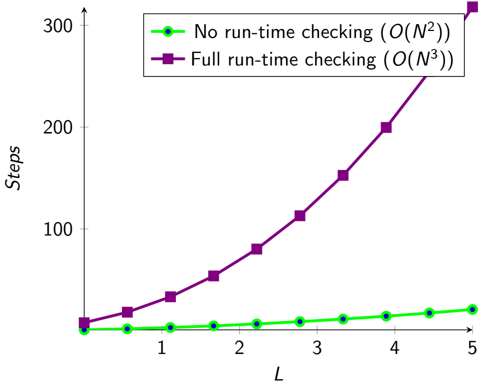
Given an admissible run-time checking overhead specification, our system automatically verifies whether it is met or not.

```
:- true pred rev(X,Y)
 : (list(X),var(Y),length(X,L))
 =>(list(Y), length(Y,L))
 + cost(exact(½L² + 3/2 L + 1)).
```

$\frac{1}{2}L^2 + \frac{3}{2}L + 1$

```
:- true pred rev(X,Y)
 : (list(X),var(Y),length(X,L))
 =>(list(Y), length(Y,L))
 + cost(exact(½L³ + 7L² + 29/2 L + 8)).
```

$\frac{1}{2}L^3 + 7L^2 + \frac{29}{2}L + 8$



Run-time Checks - Analysis Results (1 contd.)

Given an admissible run-time checking overhead specification, our system automatically verifies whether it is met or not.

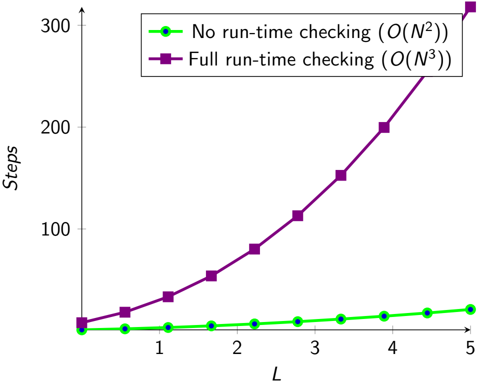
```
:- true pred rev(X,Y)
 : (list(X),var(Y),length(X,L))
 =>(list(Y), length(Y,L))
 + cost(exact(½L² + 3/2 L + 1)).
```

$$\text{cost(exact(}\frac{1}{2}L^2 + \frac{3}{2}L + 1\text{))}$$

```
:- true pred rev(
 : (list(
 =>
 +                        + 29/2 L + 8)).
```

NOT ADMISSIBLE

$$\frac{L^3}{L^2} = L > 1$$



Run-time Checks - Optimizing using Static Analysis

Static analysis can be applied to prove some run-time assertions, reducing the generated run-time code. [AADEBUG'97, LOPSTR'99, LPAR'06, SAS'03, PPDP'16]

```
 rev(A,B) :-
    revC(A,B,C),
    rev_(A,B).

revC(A,B,E) :-
    reify_check(list(A),C),
    reify_check(var(B), D),
    E is C/\D,
    warn_if_false(E,calls).

rev_(A,B) :- rev_i(A,B).

rev_i([],[]).
rev_i([X|Xs],Y) :-
    rev_i(Xs,Ys),app1(Ys,X,Y).

app1([],X,[X]).
app1([E|Y],X,[E|T]):-
    app1(Y,X,T).
```

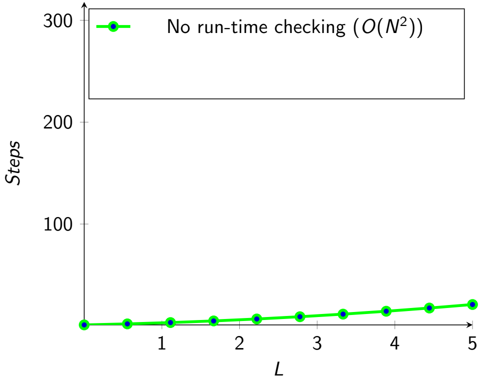
Static Analysis reduces the necessity for instrumentation (overhead), after proving the correctness of some assertions statically.
Here: **postcondition** *check eliminated by SA.*

However, some run-time checking may still remain.
Here: **precondition** *check left.*

Run-time Checks - Analysis Results (2)

```
:- true pred rev(X,Y)
 : (list(X),var(Y),length(X,L))
 =>(list(Y), length(Y,L))
 + cost(exact(½L² + 3/2 L + 1)).
```

$$\text{cost}(\text{exact}(\tfrac{1}{2}L^2 + \tfrac{3}{2}L + 1))$$



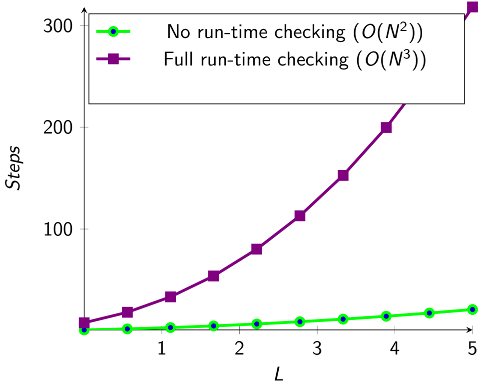
Run-time Checks - Analysis Results (2)

```
:- true pred rev(X,Y)
 : (list(X),var(Y),length(X,L))
 =>(list(Y), length(Y,L))
 + cost(exact(½L² + 3/2 L + 1)).
```

$$\texttt{+ cost(exact(}\tfrac{1}{2}L^2 + \tfrac{3}{2}L + 1\texttt{))}.$$

```
:- true pred rev(X,Y)
 : (list(X),var(Y),length(X,L))
 =>(list(Y), length(Y,L))
 + cost(exact(½L³ + 7L² + 29/2 L + 8)).
```

$$\texttt{+ cost(exact(}\tfrac{1}{2}L^3 + 7L^2 + \tfrac{29}{2}L + 8\texttt{))}.$$



Run-time Checks - Analysis Results (2)

```
:- true pred rev(X,Y)
 : (list(X),var(Y),length(X,L))
 =>(list(Y), length(Y,L))
 + cost(exact(1/2 L^2 + 3/2 L + 1)).
```

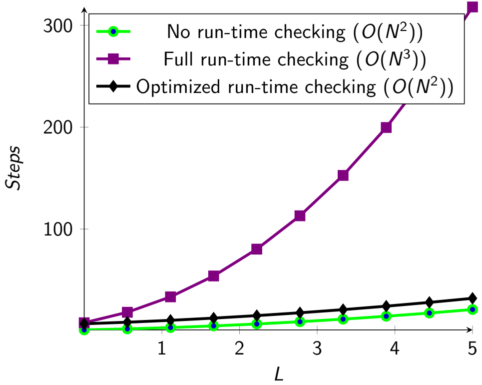
$$\frac{1}{2}L^2 + \frac{3}{2}L + 1$$

```
:- true pred rev(X,Y)
 : (list(X),var(Y),length(X,L))
 =>(list(Y), length(Y,L))
 + cost(exact(1/2 L^3 + 7L^2 + 29/2 L + 8)).
```

$$\frac{1}{2}L^3 + 7L^2 + \frac{29}{2}L + 8$$

```
:- true pred rev(X,Y)
 : (list(X),var(Y),length(X,L))
 =>(list(Y), length(Y,L))
 + cost(exact(1/2 L^2 + 5/2 L + 7)).
```

$$\frac{1}{2}L^2 + \frac{5}{2}L + 7$$



Run-time Checks - Analysis Results (2)

```
:- true pred rev(X,Y)
 : (list(X),var(Y),length(X,L))
 =>(list(Y), length(Y,L))
 + cost(exact(½L² + 3/2 L + 1)).
```

```
:- true pred r
 : (li
   ... L² + 29/2 L + 8)).
```

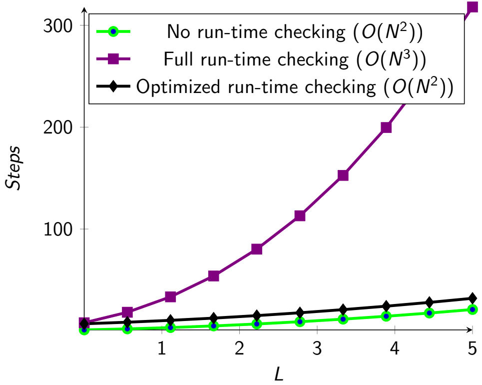
NOT ADMISSIBLE

$\frac{L^3}{L^2} = L > 1$

```
:- true pred r
 : (li
 =
 +          2 L + 7)).
```

OK

$\frac{L^2}{L^2} = 1$



Experimental Results - Verifying Admissible Overhead

The experimental evaluation suggests that our method is feasible and promising.

Bench	RTC	Bound Inferred	%D	T_A(ms)	Ovhd	Verif.
`app1(A,B,_)`	off	$l_A + 1$	0.0	98.13		
	full	$l_A{}^2 + 6 \cdot l_A \cdot l_B + 17 \cdot l_A + 6 \cdot l_B + 8$	0.0	521.18	$l_A + l_B$	false
	opt	$3 \cdot l_A + 2 \cdot l_B + 8$	0.0	311.98	$\frac{l_B}{l_A} + 1$	false
`nrev(L,_)`	off	$\frac{1}{2} \cdot l_L{}^2 + \frac{3}{2} \cdot l_L + 1$	0.0	218.15		
	full	$\frac{1}{2} \cdot l_L{}^3 + 7 \cdot l_L{}^2 + \frac{29}{2} \cdot l_L + 8$	0.0	885.08	l_L	false
	opt	$\frac{1}{2} \cdot l_L{}^2 + \frac{5}{2} \cdot l_L + 7$	0.0	756.82	1	checked
`sift(A,_)`	off	$\frac{1}{2} \cdot l_A{}^2 + \frac{3}{2} \cdot l_A + 1$	0.0	255.55		
	full	$\frac{2}{3} \cdot l_A{}^3 + \frac{15}{2} \cdot l_A{}^2 + \frac{95}{6} \cdot l_A + 7$	0.0	980.63	l_A	false
	opt	$\frac{1}{2} \cdot l_A{}^2 + \frac{7}{2} \cdot l_A + 7$	0.0	521.65	1	checked
`pfxsum(A,_)`	off	$l_A + 2$	0.0	146.98		
	full	$2 \cdot l_A{}^2 + 12 \cdot l_A + 14$	0.0	749.94	l_A	false
	opt	$3 \cdot l_A + 10$	0.0	469.71	1	checked

Experimental Results - Verifying Admissible Overhead

The experimental evaluation suggests that our method is feasible and promising.

Bench	RTC	Bound Inferred	%D	T_A(ms)	Ovhd	Verif.
`oins(E,L,_)`	off	$l_L + 2$	0.09	142.55		
	full	$\frac{1}{3} \cdot {l_L}^3 + \frac{9}{2} \cdot {l_L}^2 - \frac{5}{2} \cdot l_L + \frac{11}{3}$	99.93	917.39	${l_L}^2$	false
	opt*	$\frac{3}{2} \cdot l_L + 6$	50.14	340.15	1	checked
`mmtx(A,B,_)`	off	$r_A \cdot c_A \cdot c_B + 3 \cdot r_A \cdot c_B + 2 \cdot r_A - 2 \cdot c_B$	7.58	460.21		
	full	$4 \cdot {r_A}^2 c_A \cdot c_B + 4 \cdot {r_A}^2 \cdot c_A + 4 \cdot {r_A}^2 \cdot c_B + 4 \cdot {r_A}^2 + r_A \cdot {c_A}^2 \cdot c_B + 4 \cdot r_A \cdot {c_A}^2 + 2 \cdot r_A \cdot c_A \cdot {c_B}^2 + 11 \cdot r_A \cdot c_A \cdot c_B + 20 \cdot r_A \cdot c_A + 15 \cdot r_A + 7$	0.0	1682.54	$N^\dagger$	false
	opt	$r_A \cdot c_A \cdot c_B + 2 \cdot c_A \cdot c_B + 2 \cdot r_A \cdot c_A + 4 \cdot r_A \cdot c_A + 6 \cdot r_A + 2 \cdot c_A + 11$	0.0	1120.23	1	checked
`ldiff(A,B,_)`	off	$l_A \cdot l_B + 2 \cdot l_A + 1$	2.06	786.22		
	full	${l_A}^2 + 3 \cdot l_A \cdot l_B + 10 \cdot l_A + 2 \cdot l_B + 7$	0.27	1769.22	$\frac{l_A}{l_B} + 1$	false
	opt	$l_A \cdot l_B + 5 \cdot l_A + 2 \cdot l_B + 8$	0.0	1226.15	1	checked
`bsts(N,T)`	off	$d_T + 3$	0.1	714.83		
	full	$3 \cdot 2^{(d_T+2)} + 3 \cdot 2^{(d_T+1)} + 3 \cdot 2^{(d_T-1)} + 3 \cdot 2^{d_T} + \frac{3}{2} \cdot (d_T - 1)^2 + \frac{47}{2} \cdot (d_T + 2) - \frac{27}{2}$	1.19	438.72	$\frac{2^{d_T}}{d_T}$	false
	opt*	$3 \cdot 2^{(d_T+1)} + 4 \cdot d_T + 14$	4.01	245.09	$\frac{2^{d_T}}{d_T}$	false

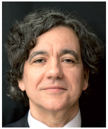
$\dagger N = max(r_A, c_A, c_B)$

Demo!

Please see examples in the CiaoPP playground.

(`http://play.ciao-lang.org`)

The Team


Manuel Hermenegildo


Pedro López-García


José-Francisco Morales


Maximiliano Klemen


Umer Liqat


Isabel García-Contreras


Nataliia Stulova

- Previous main contributors to CiaoPP:

Saumya Debray	Nai-wei Lin	Jorge Navas	Alejandro Serrano
Mario Méndez-Lojo	Edison Mera	Francisco Bueno	M. Ga-de-la-Banda
Claudio Vaucheret	Germán Puebla	Jesús Correas	Elvira Albert
Pawel Pietrzak	Claudio Ochoa	John Gallagher	Peter Stuckey

Work currently at: IMDEA Software Institute, T.U. Madrid (UPM).

And previously at: U. T. Austin, MCC, U. of Arizona, U. of New Mexico.

Playground at: `http://play.ciao-lang.org`

Thank you!

Selected Bibliography on CiaoPP

CiaoPP References – Horn Clauses as Intermediate Representation / Multi-Language Support

[LOPSTR'07] M. Méndez-Lojo, J. Navas, and M. Hermenegildo.
A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs.
In *17th Intl. Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, number 4915 in LNCS, pages 154–168. Springer-Verlag, August 2007.

CiaoPP References – Inferring/Reducing Run-time Checking Overhead

[PPDP'18] M. Klemen, N. Stulova, P. Lopez-Garcia, J. F. Morales, and M. V. Hermenegildo.
Static Performance Guarantees for Programs with Run-time Checks.
In *20th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'18)*. ACM Press, September 2018.

[PADL'18] N. Stulova, J. F. Morales, and M. V. Hermenegildo.
Exploiting Term Hiding to Reduce Run-time Checking Overhead.
20th International Symposium on Practical Aspects of Declarative Languages (PADL 2018), number 10702 in LNCS, pages 99–115. Springer-Verlag, January 2018.

[PPDP'16] N. Stulova, J. F. Morales, and M. V. Hermenegildo.
Reducing the Overhead of Assertion Run-time Checks via static analysis.
In *18th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'16)*, pages 90–103. ACM Press, September 2016.

[TPLP'15] N. Stulova, J. F. Morales, and M. V. Hermenegildo.
Practical Run-time Checking via Unobtrusive Property Caching.
Theory and Practice of Logic Programming, 31st Int'l. Conference on Logic Programming (ICLP'15) Special Issue, 15(04-05):726–741, September 2015.

CiaoPP References – The Ciao Debugging and Verification Model

[ICLP'09] E. Mera, P. López-García, and M. Hermenegildo.
Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework.
In *25th Intl. Conference on Logic Programming (ICLP'09)*, number 5649 in LNCS, pages 281–295. Springer-Verlag, July 2009.

[SCP'05] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García.
Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation
Science of Computer Programming, 58(1–2), 2005.

[SAS'03] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García.
Program Development Using Abstract Interpretation (and The Ciao System Preprocessor).
In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

[PBH00a] G. Puebla, F. Bueno, and M. Hermenegildo.
A Generic Preprocessor for Program Validation and Debugging.
In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, Sept. 2000.

[LOPSTR'99] G. Puebla, F. Bueno, and M. Hermenegildo.
Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs.
In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.

[LNCS'99] M. Hermenegildo, G. Puebla, and F. Bueno.
Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging.
In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[AADEBUG'97] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla.
On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs.
In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

CiaoPP References – Analysis and Verification of Energy

[TPLP'18] P. Lopez-Garcia, L. Darmawan, M. Klemen, U. Liqat, F. Bueno, and M. V. Hermenegildo.
Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption.
Theory and Practice of Logic Programming, Special Issue on Computational Logic for Verification, 18:167–223, March 2018. arXiv:1803.04451.

[HIP3ES'16] U. Liqat, Z. Banković, P. Lopez-Garcia, and M. V. Hermenegildo.
Inferring Energy Bounds Statically by Evolutionary Analysis of Basic Blocks.
In *(HIP3ES'16)*, 2016. arXiv:1601.02800.

[FOPARA'15] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder.
Inferring Parametric Energy Consumption Functions at Different SW Levels: ISA vs. LLVM IR.
In *Foundational and Practical Aspects of Resource Analysis: 4th Intl. Workshop, FOPARA 2015, Revised Selected Papers*, volume 9964 LNCS, pages 81–100. Springer, 2016.

[HIP3ES'15] P. Lopez-Garcia, R. Haemmerlé, M. Klemen, U. Liqat, and M. V. Hermenegildo
Towards Energy Consumption Verification via Static Analysis.
In *HIPEAC Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES 2015)*, Amsterdam.

[LOPSTR'13] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. López-Garcia, N. Grech, M.V. Hermenegildo, and K. Eder.
Energy Consumption Analysis of Programs based on XMOS ISA-Level Models.
In *Pre-proceedings of the 23rd Intl. Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*, LNCS, Springer, September 2013.

[NASA FM'08] J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications.
In *6th NASA Langley Formal Methods Workshop (LFM 08)*, April 2008. Extended Abstract.

CiaoPP References – Analysis and Verification of Resources in General

[TPLP'16] P. Lopez-Garcia, M. Klemen, U. Liqat, and M.V. Hermenegildo.
A General Framework for Static Profiling of Parametric Resource Usage.
Theory and Practice of Logic Programming, 32nd Int'l. Conference on Logic Programming (ICLP'16) Special Issue, 16(5-6):849–865, October 2016.

[FLOPS'16] R. Haemmerlé, P. Lopez-Garcia, U. Liqat, M. Klemen, J. P. Gallagher, and M. V. Hermenegildo.
A Transformational Approach to Parametric Accumulated-cost Static Profiling.
In *FLOPS'16*, volume 9613 of *LNCS*, pages 163–180. Springer, 2016.

[TPLP'14] A. Serrano, P. López-Garcia, and M. Hermenegildo.
Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types.
In *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue*, Vol. 14, Num. 4-5, pages 739-754, Cambridge U. Press, 2014.

[ICLP'13] A. Serrano, P. López-Garcia, F. Bueno, and M. Hermenegildo.
Sized Type Analysis of Logic Programs (Technical Communication).
In *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement,* pages 1–14, Cambridge U. Press, August 2013.

[FOPARA'12] P. Lopez-Garcia, L. Darmawan, F. Bueno, and M. Hermenegildo
Interval-Based Resource Usage Verification: Formalization and Prototype
In *Foundational and Practical Aspects of Resource Analysis. Second International Workshop FOPARA 2011, Revised Selected Papers*. Lecture Notes in Computer Science, 2012, 7177, 54–71, Springer.

[ICLP'10] P. López-García, L. Darmawan, and F. Bueno.
A Framework for Verification and Debugging of Resource Usage Properties.
In *Technical Communications of the 26th ICLP. Leibniz Int'l. Proc. in Informatics (LIPIcs)*, Vol. 7, pages 104–113, Dagstuhl, Germany, July 2010.

[Bytecode'09] J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
User-Definable Resource Usage Bounds Analysis for Java Bytecode.
In *Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *ENTCS*, pages 6–86. Elsevier, March 2009.

[PPDP'08] E. Mera, P. López-García, M. Carro, and M. Hermenegildo.
Towards Execution Time Estimation in Abstract Machine-Based Languages.
In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 174–184. ACM Press, July 2008.

[ICLP'07] J. Navas, E. Mera, P. López-García, and M. Hermenegildo.
User-Definable Resource Bounds Analysis for Logic Programs.
In *23rd International Conference on Logic Programming (ICLP'07)*, LNCS Vol. 4670. Springer, 2007. **ICLP 2017 10-year Test of Time Award.**

[CLEI'06] H. Soza, M. Carro, and P. López-García.
Probabilistic Cost Analysis of Logic Programs: A First Case Study.
In *XXXII Latin-American Conference on Informatics* (CLEI 2006), August 2006.

[ILPS'97] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin.
Lower Bound Cost Estimation for Logic Programs.
In *1997 Intl. Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[SAS'94] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin.
Estimating the Computational Cost of Logic Programs.
In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Sep 1994. Springer.

[ICLP'95] P. López-García and M. Hermenegildo.
Efficient Term Size Computation for Granularity Control.
In *International Conference on Logic Programming*, pages 647–661, Cambridge, MA, June 1995. MIT Press, Cambridge, MA.

[JSC'96] P. López-García, M. Hermenegildo, and S. K. Debray.
A Methodology for Granularity Based Control of Parallelism in Logic Programs.
Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation, 21(4–6):715–734, 1996.

[PASCO'94] P. López-García, M. Hermenegildo, and S.K. Debray.
Towards Granularity Based Control of Parallelism in Logic Programs.
In Hoon Hong, editor, *Proc. of First Intl. Symposium on Parallel Symbolic Computation, PASCO'94*, pages 133–144. World Scientific, September 1994.

[PLDI'90] S. K. Debray, N.-W. Lin, and M. Hermenegildo.
Task Granularity Analysis in Logic Programs.
In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 174–188. ACM Press, June 1990.

CiaoPP References – Assertion Language

[PPDP'14] N. Stulova, J. F. Morales, M. V. Hermenegildo.
Assertion-based Debugging of Higher-Order (C)LP Programs.
In *16th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'14)*, ACM Press, September 2014.

[ICLP'09] E. Mera, P. López-García, and M. Hermenegildo.
Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework.
In *25th Intl. Conference on Logic Programming (ICLP'09)*, number 5649 in LNCS, pages 281–295. Springer-Verlag, July 2009.

[LNCS'00] G. Puebla, F. Bueno, and M. Hermenegildo.
An Assertion Language for Constraint Logic Programs.
In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[ILPS-WS'97] G. Puebla, F. Bueno, and M. Hermenegildo.
An Assertion Language for Debugging of Constraint Logic Programs.
In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997. Available from `ftp://cliplab.org/pub/papers/assert_lang_tr_discipldeliv.ps.gz` as tech. report CLIP2/97.1.

CiaoPP References – Semantic Code Search

[ICLP'16] I. Garcia-Contreras, J. F. Morales, and M. V. Hermenegildo.
Semantic Code Browsing.
Theory and Practice of Logic Programming, 32nd Int'l. Conference on Logic Programming (ICLP'16) Special Issue, 16(5-6):721–737, October 2016.

CiaoPP References – Abstraction-Carrying Code

[ICLP'06] E. Albert, P. Arenas, G. Puebla, and M. Hermenegildo.
Reduced Certificates for Abstraction-Carrying Code.
In *22nd Intl. Conference on Logic Programming (ICLP 2006)*, number 4079 in LNCS, pages 163–178. Springer-Verlag, August 2006.

[PPDP'05] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla.
Abstraction Carrying Code and Resource-Awareness.
In *PPDP*. ACM Press, 2005.

[LPAR'04] E. Albert, G. Puebla, and M. Hermenegildo.
Abstraction-Carrying Code.
In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.

CiaoPP References – Basic Analysis Framework (Abstract Interpreter)

[FTfJP'07] J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
An Efficient, Context and Path Sensitive Analysis Framework for Java Programs.
In *9th Workshop on Formal Techniques for Java-like Programs FTfJP 2007*, July 2007.

[TOPLAS'00] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey.
Incremental Analysis of Constraint Logic Programs.
ACM Transactions on Programming Languages and Systems, 22(2):187–223, March 2000.

[SAS'96] G. Puebla and M. Hermenegildo.
Optimized Algorithms for the Incremental Analysis of Logic Programs.
In *Intl. Static Analysis Symposium (SAS 1996)*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[POPL'94] K. Marriott, M. García de la Banda, and M. Hermenegildo.
Analyzing Logic Programs with Dynamic Scheduling.
In *20th. Annual ACM Conf. on Principles of Programming Languages*, pages 240–254. ACM, January 1994.

[JLP'92] K. Muthukumar and M. Hermenegildo.
Compile-time Derivation of Variable Dependency Using Abstract Interpretation.
Journal of Logic Programming, 13(2/3):315–347, July 1992.

[ICLP'88] R. Warren, M. Hermenegildo, and S. K. Debray
On the Practicality of Global Flow Analysis of Logic Programs
In *5th Intl. Conf. and Symp. on Logic Programming*, pp. 684–699, MIT Press, August 1988.

CiaoPP References – Modular Analysis, Specialization, Verification

[ICLP'18] I. Garcia-Contreras, J. F. Morales, and M. V. Hermenegildo.
Towards Incremental and Modular Context-sensitive Analysis.
In *Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018)*, OpenAccess Series in Informatics (OASIcs), July 2018.

[PEPM'08] P. Pietrzak, J. Correas, G. Puebla, and M. Hermenegildo.
A Practical Type Analysis for Verification of Modular Prolog Programs.
In *PEPM'08*, pages 61–70. ACM Press, January 2008.

[LPAR'06] P. Pietrzak, J. Correas, G. Puebla, and M. Hermenegildo.
Context-Sensitive Multivariant Assertion Checking in Modular Programs.
In *LPAR'06*, number 4246 in LNCS, pages 392–406. Springer-Verlag, November 2006.

[LOPSTR'01] F. Bueno, M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey.
A Model for Inter-module Analysis and Optimizing Compilation.
In *Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, March 2001.

[ENTCS'00] G. Puebla and M. Hermenegildo.
Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs.
In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[ESOP'96] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla.
Global Analysis of Standard Prolog Programs.
In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

CiaoPP References – Abstract Domains: Sharing/Aliasing

[ISMM'09] M. Marron, D. Kapur, and M. Hermenegildo.
Identification of Logically Related Heap Regions.
In *ISMM'09: Proceedings of the 8th international symposium on Memory management*, New York, NY, USA, June 2009. ACM Press.

[LCPC'08] M. Méndez-Lojo, O. Lhoták, and M. Hermenegildo.
Efficient Set Sharing using ZBDDs.
In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS. Springer-Verlag, August 2008.

[LCPC'08] M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo.
Identification of Heap-Carried Data Dependence Via Explicit Store Heap Models.
In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS. Springer-Verlag, August 2008.

[PASTE'08] M. Marron, M. Méndez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur.
Sharing Analysis of Arrays, Collections, and Recursive Structures.
In *ACM WS on Program Analysis for SW Tools and Engineering (PASTE'08)*. ACM, November 2008.

[VMCAI'08] M. Méndez-Lojo and M. Hermenegildo.
Precise Set Sharing Analysis for Java-style Programs.
In *9th Intl. Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, number 4905 in LNCS, pages 172–187. Springer-Verlag, January 2008.

[PADL'06] J. Navas, F. Bueno, and M. Hermenegildo.
Efficient top-down set-sharing analysis using cliques.
In *Eight Intl. Symposium on Practical Aspects of Declarative Languages*, number 2819 in LNCS, pages 183–198. Springer-Verlag, January 2006.

[TOPLAS'99] F. Bueno, M. García de la Banda, and M. Hermenegildo.
Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming.
ACM Transactions on Programming Languages and Systems, 21(2):189–238, March 1999.

[NACLP'89] K. Muthukumar and M. Hermenegildo.
Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation.
In *1989 N. American Conf. on Logic Programming*, pages 166–189. MIT Press, October 1989.

CiaoPP References – Abstract Domains: Shape/Type Analysis

[ICLP'13] A. Serrano, P. López-Garcia, F. Bueno, and M. Hermenegildo.
Sized Type Analysis of Logic Programs (Technical Communication).
In *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement*, pages 1–14, Cambridge U. Press, August 2013.

[CC'08] M. Marron, M. Hermenegildo, D. Kapur, and D. Stefanovic.
Efficient context-sensitive shape analysis with graph-based heap models.
In Laurie Hendren, editor, *Intl. Conference on Compiler Construction (CC 2008)*, Lecture Notes in Computer Science. Springer, April 2008.

[PASTE'07] M. Marron, D. Stefanovic, M. Hermenegildo, and D. Kapur.
Heap Analysis in the Presence of Collection Libraries.
In *ACM WS on Program Analysis for SW Tools and Engineering (PASTE'07)*. ACM, June 2007.

[SAS'02] C. Vaucheret and F. Bueno.
More Precise yet Efficient Type Inference for Logic Programs.
In *Intl. Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.

CiaoPP References – Abstract Domains: Non-failure, Determinacy

[NGC'10] P. López-García, F. Bueno, and M. Hermenegildo.
Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information.
New Generation Computing, 28(2):117–206, 2010.

[LOPSTR'04] P. López-García, F. Bueno, and M. Hermenegildo.
Determinacy Analysis for Logic Programs Using Mode and Type Information.
In *Proceedings of the 14th Intl. Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 19–35. Springer-Verlag, August 2005.

[FLOPS'04] F. Bueno, P. López-García, and M. Hermenegildo.
Multivariant Non-Failure Analysis via Standard Abstract Interpretation.
In *7th Intl. Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany, April 2004. Springer-Verlag.

[ICLP'97] S.K. Debray, P. López-García, and M. Hermenegildo.
Non-Failure Analysis for Logic Programs.
In *1997 Intl. Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

CiaoPP References – Automatic Parallelization, (Abstract) Partial Evaluation, Other Optimizations

[LCPC'08] M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo.
Identification of Heap-Carried Data Dependence Via Explicit Store Heap Models.
In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS. Springer-Verlag, August 2008.

[ICLP'08] A. Casas, M. Carro, and M. Hermenegildo.
A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism.
In M. García de la Banda and E. Pontelli, editors, *24th Intl. Conference on Logic Programming (ICLP'08)*, volume 5366 of *LNCS*, pages 651–666. Springer-Verlag, December 2008.

[CASES'06] M. Carro, J. Morales, H.L. Muller, G. Puebla, and M. Hermenegildo.
High-Level Languages for Small Devices: A Case Study.
In Krisztian Flautner and Taewhan Kim, editors, *Compilers, Architecture, and Synthesis for Embedded Systems*, pages 271–281. ACM Press / Sheridan, October 2006.

[SAS'06] G. Puebla, E. Albert, and M. Hermenegildo.
Abstract Interpretation with Specialized Definitions.
In *SAS'06*, number 4134 in LNCS, pages 107–126. Springer-Verlag, 2006.

[PADL'04] J. Morales, M. Carro, and M. Hermenegildo.
Improving the Compilation of Prolog to C Using Moded Types and Determinism Information.
In *PADL'04*, number 3057 in LNCS, pages 86–103. Springer-Verlag, June 2004.

[PEPM'03] G. Puebla and M. Hermenegildo.
Abstract Specialization and its Applications.
In *ACM Partial Evaluation and Semantics based Program Manipulation (PEPM'03)*, pages 29–43. ACM Press, June 2003.
Invited talk.

[PEPM'99] G. Puebla, M. Hermenegildo, and J. Gallagher.
An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework.
In O Danvy, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, number NS-99-1 in BRISC Series, pages 75–85. University of Aarhus, Denmark, January 1999.

[ICLP'91] K. Muthukumar and M. Hermenegildo.
Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation.
In *Intl. Conference on Logic Programming (ICLP 1991)*, pages 49–63. MIT Press, June 1991.

[JLP'99] G. Puebla and M. Hermenegildo.
Abstract Multiple Specialization and its Application to Program Parallelization.
J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs, 41(2&3):279–316, November 1999.

[JLP'99] K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo.
Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism.
Journal of Logic Programming, 38(2):165–218, February 1999.

[LOPSTR'97] G. Puebla and M. Hermenegildo.
Abstract Specialization and its Application to Program Parallelization.
In J. Gallagher, editor, *Logic Program Synthesis and Transformation*, number 1207 in LNCS, pages 169–186. Springer-Verlag, 1997.

[PLILP'91] F. Giannotti and M. Hermenegildo.
A Technique for Recursive Invariance Detection and Selective Program Specialization.
In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 323–335. Springer-Verlag, August 1991.