Horn Clause-based Program Analysis and Verification with CiaoPP

Manuel Hermenegildo\textsuperscript{1,2}, P. López-García\textsuperscript{1,3}, J. Morales\textsuperscript{1}, I. García-Contreras\textsuperscript{1}, M. Klemen\textsuperscript{1}, N. Stulova\textsuperscript{1}

\textsuperscript{1}IMDEA Software Institute
\textsuperscript{2}T. U. of Madrid (UPM)
\textsuperscript{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]**

**[NAACL'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 $\text{lfp}(S^\alpha_P) = [P]_\alpha$, 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)

- 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

[NACLP'89, JLP'92, POPL'94, SAS'96, TOPLAS'00, FTfJP'07]

Hermenegildo, Lopez-Garcia, Morales, . . .
Efficient, Parametric Fixpoint Algorithm

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

→ 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 $\{\text{block} : \lambda_{\text{in}} \mapsto \lambda_{\text{out}}\}$.
    * Exit states for calls to $\text{block}$ satisfying precond $\lambda_{\text{in}}$ meet postcond $\lambda_{\text{out}}$.
Efficient, Parametric Fixpoint Algorithm

- **Generic framework** for implementing HC-based analyses:
  - given \( P \) (as a set of HC's) and abstract domain(s),
  - computes \( \text{lfp}(S^\alpha_P) = \llbracket P \rrbracket^\alpha \), s.t. \( \llbracket P \rrbracket^\alpha \) safely approximates \( \llbracket P \rrbracket \).

→ 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 \( \{\text{block} : \lambda_{in} \mapsto \lambda_{out}\} \).
    - Exit states for calls to block satisfying precond \( \lambda_{in} \) meet postcond \( \lambda_{out} \).
  - **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).
    - \( \text{Dep}(B : \lambda_{inB}) = \) 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 $\text{lfp}(S_P^\alpha) = [P]_\alpha$, s.t. $[P]_\alpha$ safely approximates $[P]$.

→ Essentially efficient, incremental, abstract OLDT resolution of HC’s.
  - “Top-down efficient, 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 $\lambda_{in}$ meet postcond $\lambda_{out}$.
  - 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).
    - $\text{Dep}(B : \lambda_{inB}) = \text{the set of entries depending on } B : \lambda_{inB}$.

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 $\text{lfp}(S_P^\alpha) = \llbracket P \rrbracket_\alpha$, s.t. $\llbracket P \rrbracket_\alpha$ safely approximates $\llbracket P \rrbracket$.

→ 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 $\{\text{block} : \lambda_{\text{in}} \mapsto \lambda_{\text{out}}\}$.
    - Exit states for calls to $\text{block}$ satisfying precond $\lambda_{\text{in}}$ meet postcond $\lambda_{\text{out}}$.
  - **A dependency arc table**: $\{A : \lambda_{\text{in}A} \Rightarrow B : \lambda_{\text{in}B}\}$.
    - Answers for call $A : \lambda_{\text{in}A}$ depend on the answers for $B : \lambda_{\text{in}B}$:
      - (if exit for $B : \lambda_{\text{in}B}$ changes, exit for $A : \lambda_{\text{in}A}$ possibly also changes).
    - $\text{Dep}(B : \lambda_{\text{in}B}) = \text{the set of entries depending on } B : \lambda_{\text{in}B}$.

- 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** → recently combined! 

---

Hermenegildo, Lopez-Garcia, Morales, …
Combining the incremental and the modular fixpoints
Analysis running continuously in the background

We take “snapshots” of the program sources (e.g., at each editor save/pause/... while developing).

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

```
module
pred abstraction
entry point
dependency
Clause Literal
```

```
shanoi0

Id2 shanoi/5
Id3 shanoi/5
C2 L4
Id7 append/3
C2 L6
Id4 append/3
C2 L5
C2 L4
C2 L5
Id6 append/3
C2 L6
C2 L5

Id1 append/3
Id5 append/3
C2 L1
C2 L1
C2 L1
```

```
mylists

Id6 append/3
Id4 append/3
Id7 append/3
```

```
append0

Id1 append/3
```

```
hanoi

hanoi
```

```
lists
```

```
Hermenegildo, Lopez-Garcia, Morales, ...
HC-based Analysis and Verification with CiaoPP
DPA WS @ECOOP/ISSTA 18/07/18, Amsterdam
```
Snapshot of analysis graphs
Changes detected!

```
planner.pl

100 \%
101 \- explore(P, Map, [P|Map]) :-
102 \- safe(P).
103 \%

lib.pl

41 \%
42 + add(Node, Graph) :-
43 + \%
44 + \%
45 \%
```
Snapshot of analysis graphs

delete
recompute
planner
lib
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.
Theorem 1 (Base PLAI analysis from scratch)
For a program $P$ and initial $\lambda^c$'s $Es$, 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):

$$L^M = \text{LocIncAnalyze}(M, Es, G, (\emptyset, \emptyset))$$

$L^M$ will represent the least module analysis graph of $M$ and $Es$, assuming $G$.

Proposition 2 (Adding clauses to a module) Given $M$ and $M'$ s.t., $M' = M \cup C_i$,

$$L^M = \text{LocIncAnalyze}(M, Es, G, (\emptyset, \emptyset), (\emptyset, \emptyset))$$

Then

$$\text{LocIncAnalyze}(M', Es, G, (\emptyset, \emptyset), (\emptyset, \emptyset)) =$$

$$\text{LocIncAnalyze}(M, Es, G, L^M, (C_i, \emptyset))$$

Proposition 3 (Removing clauses from a module) Given $M$ and $M'$ s.t. $M' = M \setminus C_i$,

$$L^M = \text{LocIncAnalyze}(M, Es, G, (\emptyset, \emptyset), (\emptyset, \emptyset))$$

Then

$$\text{LocIncAnalyze}(M', Es, G, (\emptyset, \emptyset), (\emptyset, \emptyset)) =$$

$$\text{LocIncAnalyze}(M, Es, G, L^M, (\emptyset, C_i))$$

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

$$\text{LocIncAnalyze}(M, Es, G', (\emptyset, \emptyset), (\emptyset, \emptyset)) =$$

$$\text{LocIncAnalyze}(M, Es, G', L^M, (\emptyset, \emptyset))$$

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):

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

$G$ 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 $G = \text{ModIncAnalyze}(P, Es, (\emptyset, \emptyset), (\emptyset, \emptyset))$:

$$\text{ModIncAnalyze}(P', Es, (\emptyset, \emptyset)) =$$

$$\text{ModIncAnalyze}(P, Es, G, \Delta P')$$
Theorem 1 (Base PLAI analysis from scratch)
For a program $P$ and initial $\lambda^{c}$s $Es$, 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):

$$L_{M} = \text{LocIncAnalyze}(M, Es, G, (\emptyset, \emptyset), (\emptyset, \emptyset))$$

$L_{M}$ will represent the least module analysis graph of $M$ and $Es$, assuming $G$.

Proposition 2 (Adding clauses to a module)
Given $M$ and $M'$ s.t., $M' = M \cup C_{i}$,

$$L_{M} = \text{LocIncAnalyze}(M, Es, G, (\emptyset, \emptyset), (\emptyset, \emptyset))$$

LocIncAnalyze($M'$, $Es$, $G$, $(\emptyset, \emptyset)$) =

LocIncAnalyze($M$, $Es$, $G$, $L_{M}$, $(\emptyset, C_{i})$)

Proposition 3 (Removing clauses from a module)
Given $M$ and $M'$ s.t. $M' = M \setminus C_{i}$,

$$L_{M} = \text{LocIncAnalyze}(M, Es, G, (\emptyset, \emptyset), (\emptyset, \emptyset))$$

LocIncAnalyze($M'$, $Es$, $G$, $(\emptyset, \emptyset)$) =

LocIncAnalyze($M$, $Es$, $G$, $L_{M}$, $(\emptyset, C_{i})$)

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 $G = \text{ModIncAnalyze}(P, Es, (\emptyset, \emptyset), (\emptyset, \emptyset))$:

ModIncAnalyze($P'$, $Es$, $(\emptyset, \emptyset)$) =

ModIncAnalyze($P$, $Es$, $G$, $\Delta P'$)

Fundamental results

What it means

The results from our incremental, modular analysis are:

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

$M$ and $M'$ s.t., $M' = M \cup C_{i}$,

$$L_{M} = \text{LocIncAnalyze}(M, Es, G, (\emptyset, \emptyset), (\emptyset, \emptyset))$$

LocIncAnalyze($M'$, $Es$, $G$, $(\emptyset, \emptyset)$) =

LocIncAnalyze($M$, $Es$, $G$, $L_{M}$, $(\emptyset, C_{i})$)
Experimental results
Experimental results

Addition experiment
Experimental results

To take home:

- **Modular Incremental analysis works!** – Up to $60\times$ speedup.
- **Modular analysis** from scratch is improved (up to $9\times$).
- 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.
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>
0x08: mkmsk (rus) r0, 0x1
0x09: retsp (u6) 0x2
0x010: ldw (ru6) r0, sp[0x1]
0x011: sub (2rus) r0, r0, 0x1
0x012: bl (u10) -0xc <fact>
0x013: ldw (ru6) r1, sp[0x1]
0x014: mul (l3r) r0, r1, r0
0x015: retsp (u6) 0x2

return edge
Xcore ISA Example: Constrained Horn Clauses IR

```prolog
:- 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(_).
```

Transformation example - binaries
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

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)

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 (Abstract Interpretation):
- Sharing
- Shapes/sizes
- Resources
- Sets of Pre/Post pairs (prog. point info)
- (incl. size and resource usage functions)
Analysis Results

:- 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,
#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]

- **Fact(N)**
  - Energy vs. N
  - Relative Error vs. N

- **Fibonacci(N)**
  - Energy vs. N
  - Relative Error vs. N

- **Power(base, exp)**
  - Energy vs. base, exp
  - Relative Error vs. base, exp

- **PowerOfTwo(N)**
  - Energy vs. N
  - Relative Error vs. N
XC Analysis Results (FIR Filter, LLVM IR level)

```c
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)

```c
#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]

<table>
<thead>
<tr>
<th>Program</th>
<th>Error vs. HW</th>
<th>ISA/LLVMIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>fact(N)</td>
<td>2.86%</td>
<td>4.50%</td>
</tr>
<tr>
<td>fibonacci(N)</td>
<td>5.41%</td>
<td>11.94%</td>
</tr>
<tr>
<td>sqr(N)</td>
<td>1.49%</td>
<td>9.31%</td>
</tr>
<tr>
<td>power_of_two(N)</td>
<td>4.26%</td>
<td>11.15%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>3.50%</strong></td>
<td><strong>9.20%</strong></td>
</tr>
<tr>
<td>reverse(N,M)</td>
<td>N/A</td>
<td>2.18%</td>
</tr>
<tr>
<td>concat(N,M)</td>
<td>N/A</td>
<td>8.71%</td>
</tr>
<tr>
<td>mat_mult(N,M)</td>
<td>N/A</td>
<td>1.47%</td>
</tr>
<tr>
<td>sum_facts(N,M)</td>
<td>N/A</td>
<td>2.42%</td>
</tr>
<tr>
<td>fir(N)</td>
<td>N/A</td>
<td>0.63%</td>
</tr>
<tr>
<td>biquad(N)</td>
<td>N/A</td>
<td>2.34%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>N/A</strong></td>
<td><strong>3.0%</strong></td>
</tr>
<tr>
<td><strong>Gobal Avg.</strong></td>
<td><strong>3.50%</strong></td>
<td><strong>5.48%</strong></td>
</tr>
</tbody>
</table>
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)

#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;
}
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

HC-based Analysis and Verification with CiaoPP

DPA WS @ECOOP/ISSTA 18/07/18, Amsterdam
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
Example

Consider the following predicate (\texttt{rev/2}) for reversing a list of terms.

\begin{verbatim}
:- 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).
\end{verbatim}
Our Static Cost Analysis (SCA)

Example

Consider the following predicate \(\text{rev}/2\) for reversing a list of terms.

\[
\begin{align*}
\text{:- pred rev}/2: &\text{list*var}. \\
\text{rev}([], []) &. \\
\text{rev}([X|Xs], Y):- \\
& \text{rev}(Xs, Ys), \\
& \text{app1}(Ys, X, Y).
\end{align*}
\]

Result of SCA:

\[
\begin{align*}
\text{:- true pred rev}(X, Y) & : (\text{list}(X), \text{var}(Y), \text{length}(X, L)) \\
& => (\text{list}(Y), \text{length}(Y, L)) \\
& + \text{cost}(\text{exact}(\frac{1}{2}L^2 + \frac{3}{2}L + 1)).
\end{align*}
\]

\[
\begin{align*}
\text{:- true pred app1}(X, Y, Z) & : (\text{list}(X), \text{var}(Z), \text{length}(X, L)) \\
& => (\text{list}(Z), \text{length}(Z, L + 1)) \\
& + \text{cost}(\text{exact}(L)).
\end{align*}
\]
Our Static Cost Analysis (SCA)

Example

Consider the following predicate (\texttt{rev/2}) for reversing a list of terms.

\begin{verbatim}
:- 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).
\end{verbatim}

Result of SCA:

\begin{verbatim}
:- true pred rev(X, Y)
    : (list(X), var(Y), length(X, L))
=> (list(Y), length(Y, L))
cost(exact(\frac{1}{2}L^2 + \frac{3}{2}L + 1))
\end{verbatim}

\begin{verbatim}
:- true pred app1(X, Y, Z)
    : (list(X), var(Z), length(X, L))
=> (list(Z), length(Z, L + 1))
cost(exact(L))
\end{verbatim}
Our Static Cost Analysis (SCA) [PLDI’90, SAS’94, ILPS’97, ICLP’07, TPLP’14, TPLP’16]

Example

Consider the following predicate \( \texttt{rev/2} \) for reversing a list of terms.

\[
\begin{align*}
\text{:- pred rev/2: list-var.} \\
\text{rev([], []).} \\
\text{rev([X|Xs], Y):-} \\
\quad \text{rev(Xs, Ys),} \\
\quad \text{app1(Ys, X, Y).}
\end{align*}
\]

Result of SCA:

\[
\begin{align*}
\text{:- true pred rev(X, Y)} \\
\quad : (\text{list}(X), \text{var}(Y), \text{length}(X, L)) \\
\quad \Rightarrow (\text{list}(Y), \text{length}(Y, L)) \\
\quad + \text{cost(exact}(\frac{1}{2}L^2 + \frac{3}{2}L + 1)).
\end{align*}
\]

\[
\begin{align*}
\text{:- true pred app1(X, Y, Z)} \\
\quad : (\text{list}(X), \text{var}(Z), \text{length}(X, L)) \\
\quad \Rightarrow (\text{list}(Z), \text{length}(Z, L + 1)) \\
\quad + \text{cost(exact}(L)).
\end{align*}
\]
check: assertions specify pre- and post-conditions for calls to a given predicate.

Example (contd.)

```prolog
:- 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).
```
check assertions specify pre- and post-conditions for calls to a given predicate.

Example (contd.)

```prolog
:- 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 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),
    appl(Ys,X,Y).

appl(A,B,C) :-
    applC(A,B,C,D),
    appl_(A,B,C),
    applS(A,B,C,D).

applC(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(K,success).

applS(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).

appl_([],X,[X]).
appl_([E|Y],X,[E|T]) :-
    appl(Y,X,T).
```
Our Static Cost Analysis analyzes both the original and the instrumented version.

\[ \text{:- true pred } \text{rev}(X, Y) \]
\[ : (\text{list}(X), \text{var}(Y), \text{length}(X, L)) \]
\[ => (\text{list}(Y), \text{length}(Y, L)) \]
\[ + \text{cost}(\text{exact}(\frac{1}{2}L^2 + \frac{3}{2}L + 1)). \]

### Graph

- **No run-time checking \((O(N^2))\)**

---

Hermenegildo, Lopez-Garcia, Morales, ...  
HC-based Analysis and Verification with CiaoPP  
DPA WS @ECOOP/ISSTA 18/07/18, Amsterdam
Our **Static Cost Analysis** analyzes both the original and the instrumented version.

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

```prolog
:- true pred rev(X,Y)
  : (list(X), var(Y), length(X,L))
=> (list(Y), length(Y,L))
+ cost(exact(\(\frac{1}{2}L^3 + 7L^2 + \frac{29}{2}L + 8\))).
```
Program instrumented with run-time checking code (assuming no analysis, i.e., full RT checks).

```prolog
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),
    appl(Ys,X,Y).

appl(A,B,C) :-
    applC(A,B,C,D),
    appl_(A,B,C),
    applS(A,B,C,D).

applC(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(K,success).

applS(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).

appl_([],X,[X]).
appl_([E|Y],X,[E|T]) :-
    appl(Y,X,T).
```
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),
  appl(Ys,X,Y).

appl(A,B,C,D) :-
  applC(A,B,C,D),
  appl_(A,B,C,D).

applC(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).

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

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

```
list/1
:- true pred list(X) :
  length(X,L) + cost(exact(L+1)).
:- regtype list/1.
list([]).
list([_|T]) :-
  list(T).
list([X|T]) :-
  list([X|T]),
  appl([X],X,[X]).
```
**Run-time Checks - Instrumentation**

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

<table>
<thead>
<tr>
<th>rev(A,B) :-</th>
</tr>
</thead>
<tbody>
<tr>
<td>revC(A,B,C),</td>
</tr>
<tr>
<td>rev_(A,B),</td>
</tr>
<tr>
<td>revS(A,B,C).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>app1(A,B,C) :-</th>
</tr>
</thead>
<tbody>
<tr>
<td>app1C(A,B,C,D),</td>
</tr>
<tr>
<td>app1_(A,B,C),</td>
</tr>
<tr>
<td>app1S(A,B,C,D).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>revC(A,B,E) :-</th>
</tr>
</thead>
<tbody>
<tr>
<td>reify_check(list(A),C),</td>
</tr>
<tr>
<td>reify_check(var(B), D),</td>
</tr>
<tr>
<td>E is C\D,</td>
</tr>
<tr>
<td>warn_if_false(E,calls).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>app1C(A,B,C,G) :-</th>
</tr>
</thead>
<tbody>
<tr>
<td>reify_check(list(A),D),</td>
</tr>
<tr>
<td>reify_check(term(B),E),</td>
</tr>
<tr>
<td>reify_check(var(C),F),</td>
</tr>
<tr>
<td>G is D(E\F),</td>
</tr>
<tr>
<td>warn_if_false(G,calls).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>revS(A,B,E) :-</th>
</tr>
</thead>
<tbody>
<tr>
<td>reify_check(list(A),C),</td>
</tr>
<tr>
<td>reify_check(list(B),D),</td>
</tr>
<tr>
<td>F is C\D,G is (E#1)\F,</td>
</tr>
<tr>
<td>warn_if_false(G,success).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>app1S(A,B,C,G) :-</th>
</tr>
</thead>
<tbody>
<tr>
<td>reify_check(list(A),D),</td>
</tr>
<tr>
<td>reify_check(term(B),E),</td>
</tr>
<tr>
<td>reify_check(list(C),F),</td>
</tr>
<tr>
<td>H is D\E\F,K is (G#1)\H,</td>
</tr>
<tr>
<td>warn_if_false(K,success).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rev_([],[]).</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>app1_([],X,[X]).</th>
</tr>
</thead>
</table>

| 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.)**

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

:- 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).
```

** Assertions**

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

**Example (contd.)**

Admissible RT Overhead:

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

Assertions:

```prolog
:- check pred rev/2
  : list*var => list*list.
```

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

Program code:

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

\[
\text{:- true pred } \text{rev}(X,Y) \\
: (\text{list}(X),\text{var}(Y),\text{length}(X,L)) \\
\Rightarrow (\text{list}(Y),\text{length}(Y,L)) \\
+ \text{cost(exact(} \frac{1}{2}L^2 + \frac{3}{2}L + 1)) .
\]

\[
\text{:- true pred } \text{rev}(X,Y) \\
: (\text{list}(X),\text{var}(Y),\text{length}(X,L)) \\
\Rightarrow (\text{list}(Y),\text{length}(Y,L)) \\
+ \text{cost(exact(} \frac{1}{2}L^3 + 7L^2 + \frac{29}{2}L + 8)) .
\]
Given an admissible run-time checking overhead specification, our system automatically verifies whether it is met or not.

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

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

The ratio \(\frac{L^3}{L^2}\) = \(L > 1\) is NOT ADMISSIBLE.
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]

```prolog
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(\(\frac{1}{2}L^2 + \frac{3}{2}L + 1\)).

---

No run-time checking \(O(N^2)\)

Steps

\(L\)

1 2 3 4 5

300

200

100
:- true pred rev(X,Y)
  : (list(X), var(Y), length(X,L))
=>(list(Y), length(Y,L))
+ cost(exact(\(\frac{1}{2}L^2 + \frac{3}{2}L + 1\))).

:- true pred rev(X,Y)
  : (list(X), var(Y), length(X,L))
=>(list(Y), length(Y,L))
+ cost(exact(\(\frac{1}{2}L^3 + 7L^2 + \frac{29}{2}L + 8\))).
:- true pred rev(X, Y)
  : (list(X), var(Y), length(X, L))
=> (list(Y), length(Y, L))
+ cost(exact($\frac{1}{2}L^2 + \frac{3}{2}L + 1$)).

No run-time checking ($O(N^2)$)
Full run-time checking ($O(N^3)$)
Optimized run-time checking ($O(N^2)$)

:- true pred rev(X, Y)
  : (list(X), var(Y), length(X, L))
=> (list(Y), length(Y, L))
+ cost(exact($\frac{1}{2}L^3 + 7L^2 + \frac{29}{2}L + 8$)).

:- true pred rev(X, Y)
  : (list(X), var(Y), length(X, L))
=> (list(Y), length(Y, L))
+ cost(exact($\frac{1}{2}L^2 + \frac{5}{2}L + 7$)).
Run-time Checks - Analysis Results (2)

:\- true pred \text{rev}(X, Y);
:: (\text{list}(X), \text{var}(Y), \text{length}(X, L))
=> (\text{list}(Y), \text{length}(Y, L))
+ \text{cost}(\text{exact}(\frac{1}{2}L^2 + \frac{3}{2}L + 1))
.

:\- true pred \text{rev}(X, Y);
:: (\text{list}(X), \text{var}(Y), \text{length}(X, L))
=> (\text{list}(Y), \text{length}(Y, L))
+ \text{cost}(\text{exact}(\frac{1}{2}L^2 + \frac{3}{2}L + 1))
.

:\- true pred \text{rev}(X, Y);
:: (\text{list}(X), \text{var}(Y), \text{length}(X, L))
=> (\text{list}(Y), \text{length}(Y, L))
+ \text{cost}(\text{exact}(\frac{1}{2}L^2 + \frac{3}{2}L + 1))
.

\text{NOT ADMISSIONAL}
\frac{L^3}{L^2} = L > 1
\frac{L^2}{L^2} = 1
\frac{L^2}{L^2} = 1
\frac{L^2}{L^2} = 1

\text{Steps}
\text{No run-time checking \((O(N^2))\)}
\text{Full run-time checking \((O(N^3))\)}
\text{Optimized run-time checking \((O(N^2))\)}

\text{Hermenegildo, Lopez-Garcia, Morales, \ldots}
\text{HC-based Analysis and Verification with CiaoPP}
\text{DPA WS @ECOOP/ISSTA 18/07/18, Amsterdam}
The experimental evaluation suggests that our method is feasible and promising.

<table>
<thead>
<tr>
<th>Bench</th>
<th>RTC</th>
<th>Bound Inferred</th>
<th>%D</th>
<th>T_A (ms)</th>
<th>Ovhd</th>
<th>Verif.</th>
</tr>
</thead>
<tbody>
<tr>
<td>app1(A,B,_)</td>
<td>off</td>
<td>(l_A + 1)</td>
<td>0.0</td>
<td>98.13</td>
<td>(l_A + l_B)</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>full</td>
<td>(l_A^2 + 6 \cdot l_A \cdot l_B + 17 \cdot l_A + 6 \cdot l_B + 8)</td>
<td>0.0</td>
<td>521.18</td>
<td>(l_A + l_B)</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>opt</td>
<td>(3 \cdot l_A + 2 \cdot l_B + 8)</td>
<td>0.0</td>
<td>311.98</td>
<td>(l_B) + 1</td>
<td>false</td>
</tr>
<tr>
<td>nrev(L,_)</td>
<td>off</td>
<td>(\frac{1}{2} \cdot l_L^2 + \frac{3}{2} \cdot l_L + 1)</td>
<td>0.0</td>
<td>218.15</td>
<td>(l_L)</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>full</td>
<td>(\frac{1}{2} \cdot l_L^3 + 7 \cdot l_L^2 + \frac{29}{2} \cdot l_L + 8)</td>
<td>0.0</td>
<td>885.08</td>
<td>(l_L)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>opt</td>
<td>(\frac{1}{2} \cdot l_L^2 + \frac{5}{2} \cdot l_L + 7)</td>
<td>0.0</td>
<td>756.82</td>
<td>1</td>
<td>checked</td>
</tr>
<tr>
<td>sift(A,_)</td>
<td>off</td>
<td>(\frac{1}{2} \cdot l_A^2 + \frac{3}{2} \cdot l_A + 1)</td>
<td>0.0</td>
<td>255.55</td>
<td>(l_A)</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>full</td>
<td>(\frac{2}{3} \cdot l_A^3 + \frac{15}{2} \cdot l_A^2 + \frac{95}{6} \cdot l_A + 7)</td>
<td>0.0</td>
<td>980.63</td>
<td>(l_A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>opt</td>
<td>(\frac{1}{2} \cdot l_A^2 + \frac{7}{2} \cdot l_A + 7)</td>
<td>0.0</td>
<td>521.65</td>
<td>1</td>
<td>checked</td>
</tr>
<tr>
<td>pfxsum(A,_)</td>
<td>off</td>
<td>(l_A + 2)</td>
<td>0.0</td>
<td>146.98</td>
<td>(l_A)</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>full</td>
<td>(2 \cdot l_A^2 + 12 \cdot l_A + 14)</td>
<td>0.0</td>
<td>749.94</td>
<td>(l_A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>opt</td>
<td>(3 \cdot l_A + 10)</td>
<td>0.0</td>
<td>469.71</td>
<td>1</td>
<td>checked</td>
</tr>
</tbody>
</table>
# Experimental Results - Verifying Admissible Overhead

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

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

† $N = \max(r_A, c_A, c_B)$
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

A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs.

CiaoPP References – Inferring/Reducing Run-time Checking Overhead

Static Performance Guarantees for Programs with Run-time Checks.

Exploiting Term Hiding to Reduce Run-time Checking Overhead.

Reducing the Overhead of Assertion Run-time Checks via static analysis.

Practical Run-time Checking via Unobtrusive Property Caching.
CiaoPP References – The Ciao Debugging and Verification Model


Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging.

On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs.
CiaoPP References – Analysis and Verification of Energy


CiaoPP References – Analysis and Verification of Resources in General


User-Definable Resource Usage Bounds Analysis for Java Bytecode.  

Towards Execution Time Estimation in Abstract Machine-Based Languages.  

User-Definable Resource Bounds Analysis for Logic Programs.  

Probabilistic Cost Analysis of Logic Programs: A First Case Study.  
In XXXII Latin-American Conference on Informatics (CLEI 2006), August 2006.
Lower Bound Cost Estimation for Logic Programs.

Estimating the Computational Cost of Logic Programs.

[ICLP’95] P. López-García and M. Hermenegildo.
Efficient Term Size Computation for Granularity Control.

A Methodology for Granularity Based Control of Parallelism in Logic Programs.

Towards Granularity Based Control of Parallelism in Logic Programs.

Task Granularity Analysis in Logic Programs.
CiaoPP References – Assertion Language

Assertion-based Debugging of Higher-Order (C)LP Programs.

[ICLP’09]  E. Mera, P. López-García, and M. Hermenegildo.
Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework.

An Assertion Language for Constraint Logic Programs.

An Assertion Language for Debugging of Constraint Logic Programs.
CiaoPP References – Semantic Code Search

Semantic Code Browsing.

CiaoPP References – Abstraction-Carrying Code

Reduced Certificates for Abstraction-Carrying Code.

Abstraction Carrying Code and Resource-Awareness.

Abstraction-Carrying Code.
CiaoPP References – Basic Analysis Framework (Abstract Interpreter)

An Efficient, Context and Path Sensitive Analysis Framework for Java Programs.

Incremental Analysis of Constraint Logic Programs.

Optimized Algorithms for the Incremental Analysis of Logic Programs.
Springer-Verlag, September 1996.

Analyzing Logic Programs with Dynamic Scheduling.

Compile-time Derivation of Variable Dependency Using Abstract Interpretation.

On the Practicality of Global Flow Analysis of Logic Programs
CiaoPP References – Modular Analysis, Specialization, Verification

Towards Incremental and Modular Context-sensitive Analysis. 

A Practical Type Analysis for Verification of Modular Prolog Programs. 

Context-Sensitive Multivariant Assertion Checking in Modular Programs. 

A Model for Inter-module Analysis and Optimizing Compilation. 

Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. 

Global Analysis of Standard Prolog Programs. 
CiaoPP References – Abstract Domains: Sharing/Aliasing

Identification of Logically Related Heap Regions.

Efficient Set Sharing using ZBDDs.
In *21st Int’l. WS on Languages and Compilers for Parallel Computing (LCPC’08)*, LNCS. Springer-Verlag, August 2008.

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.

Sharing Analysis of Arrays, Collections, and Recursive Structures.

Precise Set Sharing Analysis for Java-style Programs.

Efficient top-down set-sharing analysis using cliques.
CiaoPP References – Abstract Domains: Shape/Type Analysis


CiaoPP References – Abstract Domains: Non-failure, Determinacy

Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information. 

Determinacy Analysis for Logic Programs Using Mode and Type Information. 

Multivariant Non-Failure Analysis via Standard Abstract Interpretation. 

Non-Failure Analysis for Logic Programs. 
CiaoPP References – Automatic Parallelization, (Abstract) Partial Evaluation, Other Optimizations


An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework.

Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation.

Abstract Multiple Specialization and its Application to Program Parallelization.

Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism.

Abstract Specialization and its Application to Program Parallelization.

A Technique for Recursive Invariance Detection and Selective Program Specialization.