Analysis and Verification “of and with” Horn Clauses (using the Coq system)

Manuel Hermenegildo ¹,² M. Carro¹,² P. López-García³,¹ U. Liqat¹
J. Morales¹ P. Chico⁴ R. Haemmerlé² A. Serrano¹

¹IMDEA Software Institute
²Technical University of Madrid (UPM)
³Spanish Research Council (CSIC)
⁴Elasticbox

COST Rich Model Toolkit Workshop, Malta, June 16-17, 2013
Outline

Part I  The Ciao approach to Analysis and verification of Constraint Logic Programs

- The programming language
- The analysis, verification, and testing model

Part II  The Ciao approach to Analysis and verification of other paradigms using Constraint Logic Programs as IR

- CLP (Horn Clauses) as intermediate representation
- User-defined resource analysis/verif. of Java bytecode
- Energy analysis/verification of (Xmos) C programs
Part I The Ciao approach to Analysis and verification of Constraint Logic Programs

- The programming language
- The analysis, verification, and testing model

Part II The Ciao approach to Analysis and verification of other paradigms using Constraint Logic Programs as IR

- CLP (Horn Clauses) as intermediate representation
- User-defined resource analysis/verif. of Java bytecode
- Energy analysis/verification of (Xmos) C programs
Logic and constraint programming: Mid-90’s:
- Prolog/CLPs (dynamic), Mercury (static), Ciao (combination).
- Static analysis (abstract interpretation) maturing (aliasing, modes, data sizes, execution cost, .... scalability, incrementality, ...)

The Ciao approach [CP’94,AADEBUG’97,ICLP’99,...]
- Start from a small, but very extensible (LP-based) kernel – a language-building language.
- Build gradually extensions in layers on top of it.
- Incorporating the most useful features from different prog. paradigms.
- Offer the best of the dynamic and static language approaches.
  - Provide the flexibility of dynamic languages,
    - Dynamic typing, dynamic load, dynamic program modification, meta-programming, top level, call (eval), scripts, ...
  - But with guaranteed safety and efficiency.
    - Assertion checking, modules, itf files, separate/incr. compilation, small executables, embeddability, high-performance, ...
- Support the programmer with a great environment.
Ciao Enablers

- Module system design:
  - Allows separating dynamic and static code.
  - Allows global analysis, separate/incremental compilation.

- Syntactic and semantic extension mechanism (*packages*):
  - All language features are in libraries (loaded, combined per module):
    - Predicates, functions, higher order, *constraints*, objects, ...
    - *Tabling*, other search rules, ASP, ... concurrency, parallelism.
    - Full ISO-Prolog support – also via a library.

- The Ciao **assertions** model
  - Optional assertions, expressing rich (possibly undecidable) properties.
  - Integrated verification/certification, testing, diagnosis (in comp. loop).
  - Use throughout of *safe* approx. (abstract interpretation), “best effort.”

- Compile-time and run-time technology:
  - Analysis, partial evaluation, profiling, ...
  - Several back ends (*including Javascript*)
  - Also bytecode (abstract machine written in Ciao dialect, specializable)

*High performance* through optimization, not language restriction.
Extension: Constraint Logic Programming

- Natural extension of LP: very general relations between variables allowed (beyond Herbrand term equality).
- Execution inserts new constraints in the constraint store (CS).
- Constraint solver checks consistency of CS.

Example

\[
p(X, Y) :-
\begin{align*}
X & > 5, \\
X & < 2.
\end{align*}
\]

\[
p(X, Y) :-
\begin{align*}
X & \geq 2, \\
Y & \leq 2, \\
X & = Y.
\end{align*}
\]

?- p(X, Y).
X = 2, Y = 2
Extension: Tabling (OLDT resolution)

Properties:
- Conservative extension of Prolog/SLD.
- Avoids recomputations.
- Better termination properties; easier to reason about termination.
  - Ensures termination for “bounded term size” programs.
  - In other cases, less dependent on clause / subgoal order.

Applications:
- Deductive databases.
- Natural language (left recursive grammars).
- Fixpoint: program analysis, reachability analysis...
- Well Founded Semantics:
  - A predicate can be defined based on its negation.
  - Semantic web reasoning.
- ...

[Hermenegildo et al. (IMDEA, UPM, ...)]
Analysis and Verification “of and with” CLP
Rich Model–Jun 16-17, 2013
CLP+Tabling

- Early work:
  - Theoretical; deductive databases, bottom-up deduction.
- Goal-directed, top-down poses interesting questions.
  - Existing approaches in LP: XSB, TCHR, Ciao TCLP.
  - Still evolving.
- Some issues:
  - Checking applicability of calls and previous solutions: entailment (vs., e.g., call variant or call abstraction)

<table>
<thead>
<tr>
<th>Goal</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>{X &gt; 3} p(X, Y)</td>
<td>X &gt; 3 ∧ Y = 1</td>
</tr>
<tr>
<td></td>
<td>X &gt; 3 ∧ Y = 2</td>
</tr>
<tr>
<td></td>
<td>X &gt; 5 ∧ Y = 3</td>
</tr>
</tbody>
</table>

What can we say about \{X > 4\} p(X, Y)?

- Answers to new (subsumed) calls: conj. of input + answer constraints.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>{X &gt; 4} p(X, Y)</td>
<td>X &gt; 4 ∧ X &gt; 3 ∧ Y = 1 \equiv X &gt; 4 ∧ Y = 1</td>
</tr>
<tr>
<td></td>
<td>X &gt; 4 ∧ X &gt; 3 ∧ Y = 2 \equiv X &gt; 4 ∧ Y = 2</td>
</tr>
<tr>
<td></td>
<td>X &gt; 4 ∧ X &gt; 5 ∧ Y = 3 \equiv X &gt; 5 ∧ Y = 3</td>
</tr>
</tbody>
</table>

- Non subsumed calls: cannot use stored answer constraint safely.
- Useful to *project* constraint store on call variables.
Tabled CLP applications
Some Experiments with Timed Automata

- UPPAAL is a fast tool built specifically for TA verification:
  - Developed since 1999.
- Ciao is a general-purpose, multi-paradigm language.

<table>
<thead>
<tr>
<th></th>
<th>Ciao</th>
<th>TCLP</th>
<th>UPPAAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fisher 2</td>
<td>0</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>Fisher 3</td>
<td>12</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Fisher 4</td>
<td>270</td>
<td>44</td>
<td>44</td>
</tr>
<tr>
<td>Fisher 5</td>
<td>10576</td>
<td>4514</td>
<td>4514</td>
</tr>
</tbody>
</table>

- Tried to select comparable UPPAAL and Ciao options.
- Additionally: in Ciao, full programming power.
Demo: properties, types, predicates, functions, higher order, constraints, breadth-first search, tabling, ...
The Assertion Language

- Assertions optional, can be added at any time. Provide partial spec.
- Sets of pre/post/global triples (+ “status” field, documentation, ...).
- Used everywhere, for many purposes (incl. doc gen., foreign itf).
- System makes it worthwhile for the programmer to include them.
- Part of the programming language and “runnable.”

[BDD+97, PBH97, HPB99, PBH00b, MLGH09]
Modular, parametric, polyvariant abstract interpretation.

Accelerated, incremental fixpoint.

Properties:
- Shapes, data sizes, sharing/aliasing, CHA, determinacy, exceptions, termination, ...
- Resources (time, memory, energy, ...), (user-defined) resources.
Integrated Static/Dynamic Debugging and Verification

Program P

:- check
:- trust
:- test
I_α

Builtins/ Libs

PREPROCESSOR

Static Analysis

Analysis Info [[P]]_α

Comparator (Incl. VCgen)

Assertion Normalizer & Lib Itf.

RT Check

:- check
:- texec
:- false
:- checked

Possible run-time error

Verification warning

Compile-time error

Verified

Certificate (ACC)

(optimized)

Code

<table>
<thead>
<tr>
<th>Definition</th>
<th>Sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>P is prt. correct w.r.t. I_α if</td>
<td>α([[P]]) ≤ I_α</td>
</tr>
<tr>
<td>P is complete w.r.t. I_α if</td>
<td>I_α ≤ α([[P]])</td>
</tr>
<tr>
<td>P is incorrect w.r.t. I_α if</td>
<td>α([[P]]) ≤ I_α</td>
</tr>
<tr>
<td>P is incomplete w.r.t. I_α if</td>
<td>I_α ̸≤ α([[P]])</td>
</tr>
</tbody>
</table>

[BDD^+97, HPB99, PBH00c, PBH00a, HPBLG03, HALGP05, PCPH06, PCPH08, MLGH09]
Based throughout on the notion of safe approximation (abstraction).

Run-time checks generated for parts of asserts. not verified statically.

Diagnosis (for both static and dynamic errors).

Comparison not always trivial: e.g., resource debugging/certification

  - Need to compare functions.
  - “Segmented” answers.

[BDD+97, HPB99, PBH00c, PBH00a, HPBLG03, HALGP05, PCPH06, PCPH08, MLGH09]
Demo: assertions, static errors (types, data sizes, procedure cost, non-determinacy, ...), run-time check generation, certification, unit tests...
Abstraction-based Certification, Abstraction-Carrying Code

PRODUCER

\[[P]_\alpha = \text{Analysis} = \text{Ifp}(\text{analysis\_step})\]

\[
\text{Certificate} \subset [P]_\alpha
\]

Safety Policy

\[
\text{Certificate} \rightarrow \text{Checker} = \text{analysis\_step}
\]

Interesting extensions: reduced certificates, incrementality, ...

[APH05, HALGP05, AAPH06]

CONSUMER

Program P

:− check
:− trust
:− test \(I_\alpha\)

Builtins/Libs

Static Analysis

Analysis Info \([P]_\alpha\)

RT Check

Comparator (Incl. VCgen)

Unit Test

:− texec
:− check
:− false
:− checked

PREPROCESSOR

(optimized)

possible run-time error

verification warning

compile-time error

verified

Certificate (ACC) +

(optimized) code

[9x251]Abstraction-based Certification, Abstraction-Carrying Code

[208x55]Comparator

(Incl. VCgen)

Normalizer

& Lib Itf.

Assertion

Analysis

Info

\[
\llbracket P \rrbracket
\]

Program

P

:− ... error

:− check

:− false

:− checked

:− test

PREPROCESSOR

(optimized)

PRODUCER CONSUMER

\[
\llbracket P \rrbracket_\alpha = \text{Analysis} = \text{Ifp}(\text{analysis\_step})
\]

\[
\text{Certificate} \subset [P]_\alpha
\]

Safety Policy

\[
\text{Certificate} \rightarrow \text{Checker} = \text{analysis\_step}
\]

Interesting extensions: reduced certificates, incrementality, ...

[APH05, HALGP05, AAPH06]
Many interactions within the integrated framework:

- (Unit) tests are part of the assertion language:
  
  \[-\text{test Pred [+Precond] [\implies Postcond] [+CompExecProps].}\]

- Parts of unit tests that can be verified at compile-time are deleted.
- Unit testing uses the run-time assertion-checking machinery.
- Unit tests also provide test cases for the run-time checks.
  
  - Assertions checked by unit testing, even if not conceived as tests.

[MLGH09]
Optimization

Source-level optimizations:
▶ Partial evaluation, (multiple) (abstract) specialization, ...

Low-level optimizations (e.g., dynamic check elimination, unboxing):
▶ Use of specialized instructions.
▶ Optimized native code generation.
→ obtaining close-to-C performance for declarative languages (Ciao).

Parallelization. Granularity control.

[Hermenegildo et al. (IMDEA, UPM, ...) Analysis and Verification “of and with” CLP Rich Model–Jun 16-17, 2013 17 / 80]
Discussion: The Ciao Approach [AADEBUG’97, etc.]

- Approaches prior to Ciao had what we perceived as limitations:
  - limited the properties which may appear in specifications, or
  - checked specifications only at run-time or only at compile-time, or
  - were not automatic, or required assertions for all predicates, or . . .

- The Ciao approach – solution to static/dynamic conundrum, which:
  - Integrates automatic compile-time and run-time checking of assertions.
  - Allows using assertions in only some parts of the program.
  - Deals safely with complex properties (beyond, e.g., traditional types).

Allows “modern” (agile/extreme/...) programming, “Scripts to Ps:”
  - Develop program and specifications gradually, not necessarily in sync.
  - Both can be incomplete (including types).
    - Temporarily use spec (including tests) as implementation.
  - Go from types, to more complex assertions, to full specifications.

- Assertion language design is important: many roles, used throughout.
- Assertions, properties in source language; “seamless integration.”
- Performance through optimization, not language restriction.
**Discussion: Comparison with Classical Types**

<table>
<thead>
<tr>
<th>“Traditional” Types</th>
<th>Ciao Assertion-based Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Properties” limited by decidability</td>
<td>Much more general property language</td>
</tr>
<tr>
<td>May need to limit prog. lang.</td>
<td>No need to limit prog. lang.</td>
</tr>
<tr>
<td>“Untypable” programs rejected</td>
<td>Run-time checks introduced</td>
</tr>
<tr>
<td>(Almost) Decidable</td>
<td>Decidable + Undecidable(approximated)</td>
</tr>
<tr>
<td>Expressed in a different language</td>
<td>Expressed in the source language</td>
</tr>
<tr>
<td>Types must be defined</td>
<td>Types can be defined or inferred</td>
</tr>
<tr>
<td>Assertions are only of type “check”</td>
<td>“check”, “trust”, ...</td>
</tr>
<tr>
<td>Type signatures &amp; assertions different</td>
<td>Type signatures are assertions</td>
</tr>
</tbody>
</table>

- Some key issues:
  - Safe / Sound approximation
  - Abstract Interpretation
  - Suitable assertion language
  - Powerful abstract domains

- Works best when properties and assertions can be expressed in the source language (i.e., source lang. supports predicates, constraints).
Outline

Part I  The Ciao approach to Analysis and verification of Constraint Logic Programs

- The programming language
- The analysis, verification, and testing model

Part II  The Ciao approach to Analysis and verification of other paradigms using Constraint Logic Programs as IR

- CLP (Horn Clauses) as intermediate representation
- User-defined resource analysis/verif. of Java bytecode
- Energy analysis/verification of (Xmos) C programs
Allows supporting multiple languages / paradigms.

Used for all analyses: aliasing, CHA/shape/types, data sizes / resources, etc.

Based on “blocks:” each block represented as a Horn clause.
IR Issues: IR Level Trade-offs

Precision Loss

Information Loss

Layer 1

XC source code

Optimized LLVM

LLVM

Analysis

Layer 2

XC Compiler

Optimizations

Transform to Ciao IR

Analysis

Layer 3

Hardware

XC Assembly

LLVM Code Generator

Energy Model?

Transform to Ciao IR

Analysis

Energy Model

Transform to Ciao IR

Analysis

Hermenegildo et al. (IMDEA, UPM, ...) Analysis and Verification “of and with” CLP Rich Model–Jun 16-17, 2013 22 / 80
IR Issues: Approaches to Performing the Transformation

- The transformation (akin to *Abstract Compilation*):
  - **Source**: Program P in $L_P$ + (possibly abstract) Semantics of $L_P$
  - **Target**: A (C) Horn Clause program capturing the semantics of P

- Some approaches to performing the transformation:
  - Direct transformation into block-based intermediate representation.
    ★ More control but correctness proof more indirect.
    ★ Used in the following (translation to a Ciao program).
    ★ Can add assertions to help analysis (sizes, metrics, resource models, ..).
  - Partial evaluation of instrumented interpreters $+$ slicing.
    ★ Systematic construction from small- and big-step semantics.
    ★ Correctness proof more direct.
    ★ Less automatic?

Some evidence that the two approaches can produce similar results.

- Cf. John Gallagher’s talk!
Generating the Intermediate Representation

- **Specifics for Java:**
  - Control flow graph construction from bytecode representation.
  - Elimination of stack variables.
  - Conversion to three-address statements.
  - Explicit representation of this and ret as extra block parameters.

- **Specifics for XC:**
  - Control flow graph construction from ISA (or LLVM IR) representation.
  - Resolving branching to predicates with multiple clauses.
  - Inferring block parameters.

- **Some common tasks:**
  - Generation of block-based CFG.
  - SSA transformation (e.g., splitting of input/output param).
  - Conversion of loops into recursions among blocks.
  - Branching, cases, dynamic dispatch → blocks w/same signature.
  - Conversion to horn clauses.
Java Example 1: sending SMSs

```java
public class CellPhone {
    void sendSms(SmsPacket smsPk, Encoder enc, Stream stm) {
        if (smsPk != null) {
            stm.send(enc.format(smsPk.sms));
            sendSms(smsPk.next, enc, stm);
        }
    }

    class SmsPacket{
        String sms;
        SmsPacket next;
    }

    abstract class Stream{
        @Cost({" cents", "2* size(data)"})
        native void send(String data);
    }

    interface Encoder{
        String format(String data);
    }

    class TrimEncoder implements Encoder{
        @Cost({" cents", "0" })
        @Size(" size(ret) <= size(s) ")
        public String format(String s){
            return s.trim();
        }
    }

    class UnicodeEncoder implements Encoder{
        @Cost({" cents", "0" })
        @Size(" size(ret) <= 6*size(s) ")
        public String format(String s){
            return java.net.URLEncoder.encode(s);
        }
    }
}
```
Java Example 1: sending SMSs – IR

Internal representation: basic block → Horn clause.

Annotations (since Java 1.5) are preserved in the bytecode so they can be carried over to our IR.
Java Example 2: Factorial

@Resources({Resource.STEPS})
public class Fact {
    public int factorial(int n) {
        if (n == 0)
            return 1;
        else
            return n * factorial(n - 1);
    }
}

Source code → Basic blocks.
Java Example 2: Factorial

```java
Fact.factorial(Ret, This, N)

Builtin.eq(void, N, 0)
Builtin.asg_int(Ret, I3)
Fact.factorial(Ret, This, N)
Builtin.ne_int(void, N, 0)
Builtin.sub(I1, N, 1)
Fact.factorial(I2, This, I1)
Builtin.mul(I3, N, I2)
Builtin.asg_int(Ret, I3)
```

:- entry 'Fact.factorial'/3:var*atm*num.
:- resource 'STEPS'.

'Fact.factorial'(Ret, This, N):-
    eq_int(void, N, int, 0, int),
    asg_int(Ret, int, 1, int).

'Fact.factorial'(Ret, This, N):-
    ne_int(void, N, int, 0, int),
    sub(I1, int, N, int, 1, int),
    Fact.factorial(I2, This, I1),
    mul(I3, int, N, int, I2, int),
    asg_int(Ret, int, I3, int).

- Intermediate representation: basic block → Horn clause.
- Annotations (since Java 1.5) are preserved in the bytecode so they can be carried over to our IR.
Xcore 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
Basic block

A basic block is a maximal sequence $S$ of consecutive nodes $G$ in CFG, starting from node $n$ and ending in node $m$ such that:

$$(\forall k \in S / \{n, m\}. \text{outEdges}(k) = 1 \land \text{inEdges}(k) = 1) \land \text{outEdges}(n) = 1 \land \text{inEdges}(m) = 1$$

- Initial block starts from the entry node.
- Dead code elimination.
Xcore 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
0x0a: stw (ru6) r0, sp[0x1]
0x0b: sub (2rus) r0, r0, 0x1
0x0c: bl (u10) -0xc <fact>
0x0d: ldw (ru6) r1, sp[0x1]
0x0e: mul (l3r) r0, r1, r0
0x0f: retsp (u6) 0x2
0x10: retsp (u6) 0x2
0x11: retsp (u6) 0x2
0x12: retsp (u6) 0x2
0x13: retsp (u6) 0x2
0x14: retsp (u6) 0x2
0x15: retsp (u6) 0x2

Hermenegildo et al. (IMDEA, UPM, ...)
Analysis and Verification “of and with” CLP
Rich Model–Jun 16-17, 2013 31 / 80
Xcore Example: Block Representation

\[
\begin{align*}
\text{fact} & :- \\
0\times01 & : \text{entsp}(0\times2) \\
0\times02 & : \text{stw}(r0, \text{sp}[0\times1]) \\
0\times03 & : \text{ldw}(r1, \text{sp}[0\times1]) \\
0\times04 & : \text{ldc}(r0, 0\times0) \\
0\times05 & : \text{lss}(r0, r0, r1) \\
0\times06 & : \text{bf}(r0, 0\times1 <0\times08>) \\
& \text{branch}(\text{bf0}, \text{bf1}) \\
\text{bf1} & :- \\
0\times07 & : \text{bu}(0\times2 <0\times10>) \\
0\times10 & : \text{ldw}(r0, \text{sp}[0\times1]) \\
0\times11 & : \text{sub}(r0, r0, 0\times1) \\
0\times12 & : \text{bl}(-0\timesc <\text{fact}>) \\
& \text{call}(\text{fact}) \\
0\times13 & : \text{ldw}(r1, \text{sp}[0\times1]) \\
0\times14 & : \text{mul}(r0, r1, r0) \\
0\times15 & : \text{retsp}(0\times2) \\
\text{bf0} & :- \\
0\times08 & : \text{mkmsk}(r0, 0\times1) \\
0\times09 & : \text{retsp}(0\times2)
\end{align*}
\]

Figure: Block Control Flow Graph
Xcore Example: Horn Clause IR

:- entry fact/2 : int * var.

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

bf01(0,Sp0x1,R0,R1):-
  mkmsk(R0,0x1),
  retsp(0x2).
Fixpoint-based Analyzers

Transformation

Java Source

javac

Java Bytecode

soot + Ciao transform.

Java parser

Ciao Source

IR - CFG (Horn clauses)

Analysis

Fixpoint algorithm (AI-based)

Sharing

Shape

CHA

Sets of Pre/Post pairs Prog. Point Info ...

Resource Usage

Sizes and Resource Info.

MH92, BGH99, PH96, HPMS00, NMLH07
MGH94, BCPHP96, PH00, BdlBH+01, PCPH06, PCPH08
MH89, MH91, DLGH97, VB02, BLGH04, LGBH05, NBH06, MSHK07
MLH08, MKSH08, MMLH+08, MHKS08, MKH09, LGBH10, MLLH08
SLBH13, LKSGL13, SLH13

Hermenegildo et al. (IMDEA, UPM, ...)
Analysis and Verification “of and with” CLP
Rich Model–Jun 16-17, 2013 34 / 80
Efficient, Parametric Fixpoint Algorithm

- **Generic framework** for implementing analyses: given abstract domain, computes $\text{lfp}(S^\alpha_P) = \llbracket P \rrbracket^\alpha$, s.t. $\llbracket P \rrbracket^\alpha$ safely approximates $\llbracket P \rrbracket$.

- 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 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.
  - Modular and *incremental*.
  - Handles library calls, externals, ...

Essentially efficient, incremental, (abstract) OLDT resolution.
Blocks are nodes; edges are invocations.

Top-down traversal of this CFG, starting from entry point.

Within each block: sequence of builtins, handled in the domain.

Inter-block calls/edges: project, extend, etc. (next slide).

As graph is traversed, triples \((block, \lambda_{in}, \lambda_{out})\) are stored for each block in a memo table.

Memo table entries have status \(\in \{\text{fixpoint}, \text{approx.}, \text{complete}\}\).

Iterate until all complete.
Interprocedural analysis / recursion support

- **Project** the caller state over the actual parameters,
- find all the **compatible implementations** (blocks),
- **rename** to their formal parameters,

... abstractly execute each compatible block, ...

- calculate the **least upper bound** of the partial results of each block (if “monovariant on success” flag),
- **rename back** to the actual parameters and, finally
- **extend** (reconcile) return state into calling state.
Speeding up convergence

- Analyze non-recursive blocks first, use as starting $\lambda_{out}$ in recursions.
- Blocks derived from conditionals treated specially (no *project* or *extend* operations required).
- The $(block, \lambda_{in}, \lambda_{out})$ tuples act as a cache that avoids recomputation.
- Use strongly-connected components (on the fly).
Resource Analysis

Transformation

Java Source

javac

Java Bytecode

soot + Ciao
transform.

IR − CFG
(Horn clauses)

Java parser

Analysis

Sharing

Shape

CHA

Fixpoint
algorithm
(AI−based)

Sets of
Pre/Post pairs
Prog. Point Info
... 

Resource Usage

Sets of
Pre/Post pairs
Prog. Point Info
...

Sizes and
Resource Info.

[DLH90, LGHD94, LGHD96, DLGHL94, DLGHL97, NMLGH07, MLNH07, MLGCH08, NMLH08]

[NMLH09, LGDB10, SLBH13, LKSGL13, SLH13]
**Analysis/Debugging/Verification of Resources**

Automatically infer upper/lower bounds on the usage that a program makes of a general notion of various (*user-definable*) resources.

- **Examples:**
  - Memory, execution time, execution steps, data sizes.
  - Bits sent or received over a socket, SMSs sent or received, accesses to a database, calls to a procedure, files left open, money spent, ..
  - **Energy consumed**, ...

- **Approach:**
  1. Programmer defines via *assertions* resource-related properties for basic procedures (e.g., instructions, bytecodes, libraries).
  2. System infers the resource usage bounds for rest of program as functions of input data sizes.

- Involved properties normally undecidable \(\rightarrow\) approximation required (bounds that are safe and also as accurate as possible).

- Applications: performance debugging and verification, resource-oriented optimization, granularity control in parallelism, ..

[NMLGH07, NMLH09]
User-definable aspects of the analysis

- A cost model defines an upper/lower bound cost for primitive operations (e.g., methods, bytecode instructions).
  - Provided by the user, via the assertion language.
    ```java
    @Cost("cents","2*size(data)"
    public native void Stream.send(java.lang.String data);
    ```
  - Some predefined in system libraries.

For platform-dependent resources such as execution time or energy consumption model needs to consider low level factors.

- Assertions:
  - Also used to provide other inputs to the resource analysis such as argument sizes, size metrics, etc. if needed.
  - Also allow improving the accuracy and scalability of the system.
  - Output of resource analysis also expressed via assertions.
  - Used additionally to state resource-related specifications which allows finding bugs, verifying, certifying, etc.
The Assertion Language (*simplified grammar*, Java)

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>primitive_assrt</td>
<td><code>primitive_name(var*)&lt;assrt&gt;^*</code></td>
</tr>
</tbody>
</table>
| assrt | `@requires ( ⟨prop⟩^* )`
| | `@ensures ( ⟨prop⟩^* )`
| | `@cost ( ⟨resource_usage⟩^* )`
| | `@if ( ⟨prop⟩^* ) { ⟨prop⟩^* } [ cost ( ⟨resource_usage⟩^* ) ]` |
| resource_usage | `res_usage(res_name, ⟨expr⟩)` |
| prop | `type`
| | `size(var, ⟨sz_metric⟩, ⟨expr⟩)`
| | `size_metric(var, ⟨sz_metric⟩)` |
| expr | `⟨expr⟩ ⟨bin_op⟩ ⟨expr⟩ | (Σ | Π) ⟨expr⟩`
| | `⟨expr⟩ ⟨expr⟩ | log_num ⟨expr⟩ | − ⟨expr⟩`
| | `⟨expr⟩ | ∞ | num`
| | `size(⟨sz_metric⟩, arg(r num))` |
| bin_op | `+ | − | × | / | %` |
| sz_metric | `int | ref | . . .` |
Overview of the Analysis

1. Pre-analysis phase using the fixpoint analyzers:
   - Class hierarchy analysis simplifies CFG and improves overall precision.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures—currently limited to acyclic).
   - Determinacy information inferred and used to obtain tighter bounds.
   - Non-failure (no exceptions) inferred for non-trivial lower bounds.

2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine relative sizes of variable contents.
     (Size measures are derived from inferred shape information.)

3. Compute upper bounds to the solutions of these recurrence equations to obtain bounds on output argument sizes.
   - We have a simple recurrence solver, although the system can easily interface with tools like Parma, PUBS, Mathematica, Matlab, etc.

4. Use the size information to set up recurrence equations representing the computational cost of each block and compute upper bounds to their solutions to obtain resource usage.
Overview of the Analysis

1. Pre-analysis phase using the fixpoint analyzers:
   - Class hierarchy analysis simplifies CFG and improves overall precision.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures – currently limited to acyclic).
   - Determinacy information inferred and used to obtain tighter bounds.
   - Non-failure (no exceptions) inferred for non-trivial lower bounds.

2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine relative sizes of variable contents. (Size measures are derived from inferred shape information.)

3. Compute upper bounds to the solutions of these recurrence equations to obtain bounds on output argument sizes.
   - We have a simple recurrence solver, although the system can easily interface with tools like Parma, PUBS, Mathematica, Matlab, etc.

4. Use the size information to set up recurrence equations representing the computational cost of each block and compute upper bounds to their solutions to obtain resource usage.
Overview of the Analysis

1. Pre-analysis phase using the fixpoint analyzers:
   ▶ Class hierarchy analysis simplifies CFG and improves overall precision.
   ▶ Sharing analysis for correctness (conservative: only when there is no sharing among data structures — currently limited to acyclic).
   ▶ Determinacy information inferred and used to obtain tighter bounds.
   ▶ Non-failure (no exceptions) inferred for non-trivial lower bounds.

2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   ▶ Data dependency graphs determine relative sizes of variable contents.
     (Size measures are derived from inferred shape information.)

3. Compute upper bounds to the solutions of these recurrence equations to obtain bounds on output argument sizes.
   ▶ We have a simple recurrence solver, although the system can easily interface with tools like Parma, PUBS, Mathematica, Matlab, etc.

4. Use the size information to set up recurrence equations representing the computational cost of each block and compute upper bounds to their solutions to obtain resource usage.
Overview of the Analysis

1. Pre-analysis phase using the fixpoint analyzers:
   - Class hierarchy analysis simplifies CFG and improves overall precision.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures – currently limited to acyclic).
   - *Determinacy* information inferred and used to obtain tighter bounds.
   - *Non-failure* (no exceptions) inferred for non-trivial lower bounds.

2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine *relative* sizes of variable contents. (Size measures are derived from inferred shape information.)

3. Compute upper bounds to the solutions of these recurrence equations to obtain bounds on output argument *sizes*.
   - We have a simple recurrence solver, although the system can easily interface with tools like Parma, PUBS, Mathematica, Matlab, etc.

4. Use the size information to set up recurrence equations representing the computational cost of each block and compute upper bounds to their solutions to obtain *resource usage.*
public class CellPhone {
    void sendSms(SmsPacket smsPk, Encoder enc, Stream stm) {
        if (smsPk != null) {
            stm.send(
                enc.format(smsPk.sms));
            sendSms(smsPk.next, enc, stm);
        }
    }
}

class SmsPacket{
    String sms;
    SmsPacket next;
}

abstract class Stream{
    @Cost({"cents","2*size(data)"})
    native void send(String data);
}

interface Encoder{
    String format(String data);
}

class TrimEncoder implements Encoder{
    @Cost({"cents","0"})
    @Size("size(ret)<=size(s)")
    public String format(String s){
        return s.trim();
    }
}

class UnicodeEncoder implements Encoder{
    @Cost({"cents","0"})
    @Size("size(ret)<=6*size(s)")
    public String format(String s){
        return java.net.URLEncoder.encode(s);
    }
}
Example (I)

1. System takes by default size of input data: \( \text{size}(\text{smsPk}) = n. \)
   - Result will be parametric on this.

2. The number of characters sent depends on the formatting done by the different encoders:
   - The user indicates that the encoding in TrimEncoder results in a smaller or equal (output) string.
     ```java
     class TrimEncoder implements Encoder {
         @Size("size(ret)\leq size(s)")
         public String format(String s) {
     }
     ```
   - And that the result of UnicodeEncoder can be up to 6 times larger (\(\backslash uxxxx\)) than the one received.
     ```java
     class UnicodeEncoder implements Encoder {
         @Size("size(ret)\leq 6*size(s)")
         public String format(String s) {
     }
     ```
After setting up and solving the size equations the system obtains that the upper bound on the number of characters sent is:

$$\max(6, 1) \times n = 6 \times n = 6 \times \text{size(smsPk)}$$

The analysis establishes then (cost) recurrences for every method:

$$\text{Cost}_{\text{sendSms}}(r_0, 0, r_2, r_3) = 0$$
$$\text{Cost}_{\text{sendSms}}(r_0, r_1, r_2, r_3) = \text{cost of sending a char} \times \text{Cost}_{\text{sendSms}}(r_0, r_1 - 1, r_2, r_3)$$

where $r_0, r_1, r_2,$ and $r_3$ represent the size of This, SmsPk, enc, and stm, respectively.

Given that we are charged 2 cents per character sent:

```java
@Cost({"cents","2*size(data)"})
native void send(String data);
```

$$\text{Cost}_{\text{sendSms}}(r_0, 0, r_2, r_3) = 0$$
$$\text{Cost}_{\text{sendSms}}(r_0, r_1, r_2, r_3) = 2 \times 6 \times (r_1 - 1) \times \text{Cost}_{\text{sendSms}}(r_0, r_1 - 1, r_2, r_3)$$

and the total cost of the \text{sendSMS} method is $6 \times r_1^2 - 6 \times r_1$ cents.
### Some results (Java)

<table>
<thead>
<tr>
<th>Program</th>
<th>Resource(s)</th>
<th>t</th>
<th>Resource Usage Func. / Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>BST</td>
<td>Heap usage</td>
<td>367</td>
<td>( O(2^n) ) ( n \equiv \text{tree depth} )</td>
</tr>
<tr>
<td>CellPhone</td>
<td>SMS monetary cost</td>
<td>386</td>
<td>( O(n^2) ) ( n \equiv \text{packets length} )</td>
</tr>
<tr>
<td>Client</td>
<td>Bytes received and</td>
<td>527</td>
<td>( O(n) ) ( n \equiv \text{stream length} )</td>
</tr>
<tr>
<td></td>
<td>bandwidth required</td>
<td></td>
<td>( O(1) ) ( ___ )</td>
</tr>
<tr>
<td>Dhrystone</td>
<td>Energy consumption</td>
<td>759</td>
<td>( O(n) ) ( n \equiv \text{int value} )</td>
</tr>
<tr>
<td>Divbytwo</td>
<td>Stack usage</td>
<td>219</td>
<td>( O(\log_2(n)) ) ( n \equiv \text{int value} )</td>
</tr>
<tr>
<td>Files</td>
<td>Files left open and</td>
<td>649</td>
<td>( O(n) ) ( n \equiv \text{number of files} )</td>
</tr>
<tr>
<td></td>
<td>Data stored</td>
<td></td>
<td>( O(n \times m) ) ( m \equiv \text{stream length} )</td>
</tr>
<tr>
<td>Join</td>
<td>DB accesses</td>
<td>460</td>
<td>( O(n \times m) ) ( n, m \equiv \text{table records} )</td>
</tr>
<tr>
<td>Screen</td>
<td>Screen width</td>
<td>536</td>
<td>( O(n) ) ( n \equiv \text{stream length} )</td>
</tr>
</tbody>
</table>

- Different complexity functions, resources, types of loops/recursion, etc.
<table>
<thead>
<tr>
<th>Program</th>
<th>Resource</th>
<th>Usage Function</th>
<th>Metrics</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>client</td>
<td>“bits received”</td>
<td>$\lambda x.8 \cdot x$</td>
<td>length</td>
<td>186</td>
</tr>
<tr>
<td>color_map</td>
<td>“unifications”</td>
<td>39066</td>
<td>size</td>
<td>176</td>
</tr>
<tr>
<td>copy_files</td>
<td>“files left open”</td>
<td>$\lambda x.x$</td>
<td>length</td>
<td>180</td>
</tr>
<tr>
<td>eight_queen</td>
<td>“queens movements”</td>
<td>19173961</td>
<td>length</td>
<td>304</td>
</tr>
<tr>
<td>eval.polynom</td>
<td>“FPU usage”</td>
<td>$\lambda x.2.5x$</td>
<td>length</td>
<td>44</td>
</tr>
<tr>
<td>fib</td>
<td>“arith. operations”</td>
<td>$\lambda x.2.17 \cdot 1.61^x + 0.82 \cdot (-0.61)^x - 3$</td>
<td>value</td>
<td>116</td>
</tr>
<tr>
<td>grammar</td>
<td>“phrases”</td>
<td>24</td>
<td>length/size</td>
<td>227</td>
</tr>
<tr>
<td>hanoi</td>
<td>“disk movements”</td>
<td>$\lambda x.2^x - 1$</td>
<td>value</td>
<td>100</td>
</tr>
<tr>
<td>insert_stores</td>
<td>“accesses Stores”</td>
<td>$\lambda n, m.n + k$</td>
<td>length</td>
<td>292</td>
</tr>
<tr>
<td></td>
<td>“insertions Stores”</td>
<td>$\lambda n, m.n$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>perm</td>
<td>“WAM instructions”</td>
<td>$\lambda x.\left(\sum_{i=1}^{x} 18 \cdot x!\right) + \left(\sum_{i=1}^{x} 14 \cdot \frac{x!}{i!}\right) + 4 \cdot x!$</td>
<td>length</td>
<td>98</td>
</tr>
<tr>
<td>power_set</td>
<td>“output elements”</td>
<td>$\lambda x.\frac{1}{2} \cdot 2^{x+1}$</td>
<td>length</td>
<td>119</td>
</tr>
<tr>
<td>qsort</td>
<td>“lists parallelized”</td>
<td>$\lambda x.4 \cdot 2^x - 2x - 4$</td>
<td>length</td>
<td>144</td>
</tr>
<tr>
<td>send_files</td>
<td>“bytes read”</td>
<td>$\lambda x, y.x \cdot y$</td>
<td>length/size</td>
<td>179</td>
</tr>
<tr>
<td>subst_exp</td>
<td>“replacements”</td>
<td>$\lambda x, y.2xy + 2y$</td>
<td>size/length</td>
<td>153</td>
</tr>
<tr>
<td>zebra</td>
<td>“resolution steps”</td>
<td>30232844295713061</td>
<td>size</td>
<td>292</td>
</tr>
</tbody>
</table>
Interesting Resource: Execution Time

- Important: e.g., verification of real-time constraints.
- Very hard in current architectures, (e.g., worst-case cache behavior).
  - Certainly feasible in simple processors and with caches turned off.
  - Our approach is *complementary* to accurate WCET models, which consider cache behavior, pipeline state, etc. (inputs to us).

- Approach:
  - Obtain timing model of abstract machine instructions through a one-time profiling phase (results provided as assertions).
    - Includes fitting constants in a function if the execution time depends on the argument’s properties.
  - Static cost analysis phase which infers a function which returns (bounds on) the execution time of program for given input data sizes.

[MLGCH08]
First Phase Output

Cost assertions automatically generated in first phase and stored to make the instruction execution costs available to the static analyzer.

Examples

:- true pred unify_variable(A, B): int(A), int(B)
   + (cost(ub, exectime, 667.07),
      cost(lb, exectime, 667.07)).

:- true pred unify_variable(A, B): var(A), gnd(B)
   + (cost(ub, exectime, 233.3),
      cost(lb, exectime, 233.3)).

:- true pred unify_variable(A, B): list(A),list(B)
   + cost(ub, exectime, 271.58+284.34*length(A)). ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E</td>
<td>110</td>
<td>110</td>
<td>113</td>
<td>-2.4</td>
<td>-2.4</td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td>69</td>
<td>69</td>
<td>71</td>
<td>-2.3</td>
<td>-2.3</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>1525</td>
<td>1525</td>
<td>1576</td>
<td>-3.3</td>
<td>-3.3</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>1501</td>
<td>1501</td>
<td>1589</td>
<td>-5.7</td>
<td>-5.7</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>2569</td>
<td>2569</td>
<td>2638</td>
<td>-2.7</td>
<td>-2.7</td>
</tr>
<tr>
<td>6</td>
<td>E</td>
<td>1875</td>
<td>1875</td>
<td>2027</td>
<td>-7.8</td>
<td>-7.8</td>
</tr>
<tr>
<td>7</td>
<td>E</td>
<td>1868</td>
<td>1868</td>
<td>1931</td>
<td>-3.3</td>
<td>-3.3</td>
</tr>
<tr>
<td>8</td>
<td>L</td>
<td>43</td>
<td>68</td>
<td>81</td>
<td>-67.2</td>
<td>-17.8</td>
</tr>
<tr>
<td></td>
<td>U</td>
<td>3414</td>
<td>3569</td>
<td>3640</td>
<td>-6.4</td>
<td>-2.0</td>
</tr>
<tr>
<td>9</td>
<td>L</td>
<td>54</td>
<td>79</td>
<td>91</td>
<td>-54.6</td>
<td>-14.8</td>
</tr>
<tr>
<td></td>
<td>U</td>
<td>3414</td>
<td>3694</td>
<td>4011</td>
<td>-16.2</td>
<td>-8.2</td>
</tr>
<tr>
<td>10</td>
<td>L</td>
<td>135</td>
<td>142</td>
<td>124</td>
<td>8.6</td>
<td>13.7</td>
</tr>
<tr>
<td></td>
<td>U</td>
<td>7922</td>
<td>2937</td>
<td>2858</td>
<td>120.6</td>
<td>2.7</td>
</tr>
<tr>
<td>11</td>
<td>L</td>
<td>216</td>
<td>138</td>
<td>111</td>
<td>72.3</td>
<td>22.5</td>
</tr>
<tr>
<td></td>
<td>U</td>
<td>226</td>
<td>216</td>
<td>162</td>
<td>34.0</td>
<td>29.5</td>
</tr>
</tbody>
</table>
In the classical CiaoPP resource analysis the last steps (setting up and solving recurrences) were not implemented as an abstract domain.

We have now defined, implemented and integrated the resource analysis as an \textit{abstract domain} (a plugin of the generic fixpoint).

We get all the good features of the AI framework for free:

- Multivariance: e.g., separate different call patterns for same block: 
  \begin{itemize}
  \item \texttt{sort(lst(int),var)} ... \texttt{sort(lst(flt),var)} ... \texttt{sort(var,lst(int))}
  \end{itemize}

- Easier combination with other domains.

- Easier integration w/static debugging/verification and rt-checking.

- Many other engineering advantages.

New domain for size analysis (\textit{sized types}) that infers bounds on the size of data structures \textit{and substructures}.

- Size: number of rule applications in type/shape definition.

Used in the XC energy analysis.
The Sized Types Abstract Domain

[SLBL13]

**Sized types** are representations of data shape information including both lower and upper bounds on the size of the corresponding terms and their subterms at any position and depth.

- Derived from the *regular types* inferred for program variables.
- If $\tau$ is a regular type, $\text{sized}(\tau)$ is its corresponding sized type:
  
<table>
<thead>
<tr>
<th></th>
<th>sized(listnum)</th>
</tr>
</thead>
<tbody>
<tr>
<td>listnum</td>
<td>listnum$^{(\alpha,\beta)}(\text{num}^{(\gamma,\delta)})$</td>
</tr>
<tr>
<td>listnum -&gt; []</td>
<td>listnum$^{(\alpha,\beta)}(\text{num}^{(\gamma,\delta)})$</td>
</tr>
<tr>
<td>listnum -&gt; [num</td>
<td>listnum]</td>
</tr>
</tbody>
</table>

- The superscripts (*size bound variables*) express bounds on the number of rule (functor) applications.
  
  $\{ [1,2,3,4], [2,4] \} \quad \text{listnum}^{(3,5)}(\text{num}^{(1,4)})$

- Size analysis infers *relations (inequations)* among the *size bound variables* of the sized types occurring at different argument positions.
## Experimental Results

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>$\alpha$</td>
<td>$\alpha$</td>
<td>$\beta$</td>
</tr>
<tr>
<td>appAll</td>
<td>$a_1 a_2 a_3$</td>
<td>$a_1$</td>
<td>$b_1 b_2 b_3$</td>
</tr>
<tr>
<td>coupled</td>
<td>$\mu$</td>
<td>0</td>
<td>$\nu$</td>
</tr>
<tr>
<td>dyade</td>
<td>$\alpha_1 \alpha_2$</td>
<td>$\alpha_1 \alpha_2$</td>
<td>$\beta_1 \beta_2$</td>
</tr>
<tr>
<td>erathos</td>
<td>$\alpha$</td>
<td>$\alpha$</td>
<td>$\beta^2$</td>
</tr>
<tr>
<td>fib</td>
<td>$\phi^\mu$</td>
<td>$\phi^\mu$</td>
<td>$\phi^\nu$</td>
</tr>
<tr>
<td>hanoi</td>
<td>1</td>
<td>0</td>
<td>$2^\nu$</td>
</tr>
<tr>
<td>isort</td>
<td>$\alpha^2$</td>
<td>$\alpha^2$</td>
<td>$\beta^2$</td>
</tr>
<tr>
<td>isortl</td>
<td>$a_1^2$</td>
<td>$a_1^2$</td>
<td>$b_1^2 b_2$</td>
</tr>
<tr>
<td>lisfact</td>
<td>$\alpha \gamma$</td>
<td>$\alpha$</td>
<td>$\beta \delta$</td>
</tr>
<tr>
<td>listnum</td>
<td>$\mu$</td>
<td>$\mu$</td>
<td>$\nu$</td>
</tr>
<tr>
<td>minsert</td>
<td>$\alpha^2$</td>
<td>$\alpha$</td>
<td>$\beta^2$</td>
</tr>
<tr>
<td>nub</td>
<td>$a_1$</td>
<td>$a_1$</td>
<td>$b_1^2 b_2$</td>
</tr>
<tr>
<td>part</td>
<td>$\alpha$</td>
<td>$\alpha$</td>
<td>$\beta$</td>
</tr>
<tr>
<td>zip3</td>
<td>$\min(\alpha_i)$</td>
<td>0</td>
<td>$\min(\beta_i)$</td>
</tr>
</tbody>
</table>
Energy Consumption Analysis

- Specialize the generic resource analysis by encoding energy models: provide cost and size assertions for each individual instruction.

- Some energy models:
  - Java bytecode energy consumption models available for simple processors – upper bound consumption per bytecode in joules:

    | Opcode | Inst. Cost in $\mu$J | Mem. Cost in $\mu$J | Total Cost in $\mu$J |
    |--------|-----------------------|----------------------|----------------------|
    | iadd   | .957860               | 2.273580             | 3.23144              |
    | isub   | .957360               | 2.273580             | 3.23094              |
    | ...    | ...                   | ...                  | ...                  |

  - More sophisticated ISA-level energy models developed w/Bristol & XMOS (based on “Tiwari” model).

- The CiaoPP resource analysis then generates at compile time safe upper- and lower-bound energy consumption functions for given programs.

[NMLH08]
Demo: java resource analysis (including CHA, nullity, etc.);
XC energy analysis.
Low-level ISA characterization

Obtaining the cost model: energy consumption per instruction

Coll. w/Xmos and Bristol U (based on Tiwari model).
Energy Model

Expressed in the Ciao assertion language

```prolog
:- 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 ) ).
```

---

Hermenegildo et al. (IMDEA, UPM, ...)

Analysis and Verification “of and with” CLP

Rich Model–Jun 16-17, 2013
#include "fact.h"

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

```
fact:
   entsp 6
   stw r0, sp[4]
   stw r0, sp[2]
.Lxtalabel0:
   ldw r0, sp[4]
   ldc r1, 0
   lss r0, r1, r0
   bt r0, .LBB0_4
   bu .LBB0_3
.LBB0_3:
   mkmsk r0, 1
   stw r0, sp[3]
   bu .LBB0_5
.LBB0_4:
.Lxtalabel1:
   ldw r0, sp[4]
   sub r1, r0, 1
   stw r0, sp[1]
   mov r0, r1
.Lxta.call_labels0:
   bl fact
   ldw r1, sp[1]
   mul r0, r1, r0
   stw r0, sp[3]
.LBB0_5:
   ldw r0, sp[3]
   retsp 6
```
CiaoPP Menu

Preprocessor Option Browser

- Use Saved Menu Configuration: none
- Select Menu Level: naive
- Select Action Group: analyze
- Select Aliasing-Mode Analysis: none
- Select Shape-Type Analysis: res_plai
- Select Resource Analysis: res_plai
- Include Energy Model: yes
- Multivariant Success: off
- Print Program Point Info: off
- Collapse AI Info: on

[Current Saved Menu Configurations: ]

Cancel Apply
Select Resource Analysis

![Preprocessor Option Browser](image)

- Use Saved Menu Configuration: none
- Select Menu Level: naive
- Select Action Group: analyze
- Select Aliasing-Mode Analysis: none
- Select Shape-Type Analysis: none
- Select Resource Analysis: res_plai
- Include Energy Model: yes
- Multivariant Success: off
- Print Program Point Info: off
- Collapse AI Info: on

{Current Saved Menu Configurations: □}

- Cancel
- Apply

---

```
**--:**- *CiaoPP Interface*  All L16 (Fundamental)
```

---
:- 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,
Checking against actual HW energy consumption

Test programs run on *two different* HW rigs:

- ISS (Instruction Set Simulation) and
- SRA (Static Resource Analysis).
## Some Results

### Benchmarks

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
<th>Energy function</th>
</tr>
</thead>
<tbody>
<tr>
<td>fact(N)</td>
<td>Calculates $N!$</td>
<td>$26.0 \ N + 19.4$</td>
</tr>
<tr>
<td>fibonacci(N)</td>
<td>$N$th Fibonacci no.</td>
<td>$30.1 + 35.6 \ \phi^N + 11.0 \ (1 - \phi)^N$</td>
</tr>
<tr>
<td>sqr(N)</td>
<td>Computes $N^2$</td>
<td>$103.0 \ N^2 + 205.8 \ N + 188.32$</td>
</tr>
<tr>
<td>poweroftwo(N)</td>
<td>Calculates $2^N$</td>
<td>$62.4 \cdot 2^N - 312.3$</td>
</tr>
<tr>
<td>sumofdigits(N)</td>
<td>Adds all digits in $N$</td>
<td>$84.4 \left\lceil \log_{10} \ N \right\rceil - 78.7$</td>
</tr>
<tr>
<td>isprime(N)</td>
<td>Checks if $N$ is prime</td>
<td>$58.6 \ N - 35.5$</td>
</tr>
<tr>
<td>power(base,exp)</td>
<td>Calculates $base^{exp}$</td>
<td>$6.3 \ (\log_2 \ exp + 1) + 6.5$</td>
</tr>
</tbody>
</table>
Some Results

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

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

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

- **PowerOfTwo(N)**
  - Energy (nJ) vs. N
  - Relative Error
Feedback from the hardware experts (Xmos, Bristol)

- SRA provides results beyond what is possible with simulation (as test run-time increases, ISS becomes impractically long).
- SRA shows promising accuracy in comparison with ISS and the HW (at least for the simple cases studied so far).
- Simulation time limits the usefulness of ISS method, whereas equation solving limits SRA.
IR Level Trade-offs

 XC source code → XC Compiler → Optimized LLVM → LLVM Code Generator → XC Assembly → Hardware

- Layer 1: XC source code → Transform to Ciao IR → Analysis
- Layer 2: Optimized LLVM → Optimizations → Transform to Ciao IR → Analysis
- Layer 3: XC Assembly → Transform to Ciao IR → Analysis

Precision Loss

Information Loss

Energy Model?
### xC Program

<table>
<thead>
<tr>
<th>Program</th>
<th>Error vs. HW</th>
<th>ISA / LLVM IR</th>
</tr>
</thead>
<tbody>
<tr>
<td>fact</td>
<td>4.5%</td>
<td>2.86%</td>
</tr>
<tr>
<td>fibonacci</td>
<td>11.94%</td>
<td>5.41%</td>
</tr>
<tr>
<td>sqr</td>
<td>9.31%</td>
<td>1.49%</td>
</tr>
<tr>
<td>power_of_two</td>
<td>11.15%</td>
<td>4.26%</td>
</tr>
<tr>
<td>reverse</td>
<td>2.18%</td>
<td>N/A</td>
</tr>
<tr>
<td>concat</td>
<td>8.71%</td>
<td>N/A</td>
</tr>
<tr>
<td>mat_mult</td>
<td>1.47%</td>
<td>N/A</td>
</tr>
<tr>
<td>sum_facts</td>
<td>2.42%</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>6.46%</strong></td>
<td><strong>3.50%</strong></td>
</tr>
</tbody>
</table>
Energy consumption verification / debugging

:- check pred fact(A, B) : (int(A), var(B))
    + resource(energy, 0, 100).

1 Resource analysis infers upper and lower bounds for resource “energy.”
   The analysis results produced are:
   :- true pred fact(A, B)
       : (int(A), var(B))
       => (int(A), int(B), rsize(A, num(LA,UA)),
          rsize(B, num('Factorial'(LA),'Factorial'(UA))))
       + resource(energy, 21 * LA + 16, 21 * UA + 16).

2 Then, the analysis results are compared with the “check” assertion
   (the specification) and the following assertions are produced:
   :- checked pred fact(A, B)
       : (int(A), intervals(int(A), [i(0,4)]), var(B))
       + resource(energy, 0, 100).
   :- false pred fact(A, B)
       : (int(A), intervals(int(A), [i(5,inf)]), var(B))
       + resource(energy, 0, 100).
Resource Usage Verification – Function Comparisons

- **SPECIFICATION UPPER/LOWER BOUNDS (SU/SL)**
- **SPECIFICATION INTERVALS**

**RESOURCE USAGE**

**INPUT DATA SIZE**

SU and SL represent the specification upper and lower bounds, respectively.
Resource Usage Verification – Function Comparisons

```
RESOURCE USAGE
- SPECIFICATION UPPER/LOWER BOUNDS (SU/SL)
- SPECIFICATION INTERVALS
- ANALYSIS UPPER/LOWER BOUNDS (SU / SL)
- ANALYSIS INTERVALS

AL > SU → INCORRECT
AL ≥ SL AND AU ≤ SU → CORRECT
AU ≤ SL → INCORRECT

INPUT DATA SIZE
```

Hermenegildo et al. (IMDEA, UPM, ...)
Analysis and Verification “of and with” CLP
Rich Model–Jun 16-17, 2013 73 / 80
<table>
<thead>
<tr>
<th>Year</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'83</td>
<td>Parallel abstract machines → motivation: auto-parallelization.</td>
</tr>
<tr>
<td>'88</td>
<td><strong>MA3 analyzer</strong>: memo tables (cf. OLDT resolution), practicality established.</td>
</tr>
<tr>
<td>'89</td>
<td><strong>PLAI framework</strong>: accelerated fixpoint, abstract domains as plugins. Sharing analysis, side-effect analysis.</td>
</tr>
<tr>
<td>90's</td>
<td>Incremental analysis, concurrency (dynamic scheduling), automatic domain combinations, scalability, auto-parallelization, extension to constraints.</td>
</tr>
<tr>
<td>'90</td>
<td><strong>GraCos analyzer</strong>: fully automatic cost analysis (upper bounds).</td>
</tr>
<tr>
<td>early 90's</td>
<td>Automatic parallelization with task granularity control.</td>
</tr>
<tr>
<td>mid 90's</td>
<td><strong>Ciao model</strong>: <em>Integrated verification/debugging/optimization w/ assertions</em>.</td>
</tr>
<tr>
<td>'97-present</td>
<td><strong>CiaoPP tool</strong>:</td>
</tr>
<tr>
<td>'91–'06</td>
<td>Combined abstract interpretation and partial evaluation.</td>
</tr>
<tr>
<td>late 90's</td>
<td><em>Lower bound</em> cost analysis. Non-failure (no exceptions), determinacy.</td>
</tr>
<tr>
<td>'01</td>
<td><strong>Verification</strong> of cost, additional resources, ...</td>
</tr>
<tr>
<td>'03</td>
<td>Abstraction carrying code, reduced certificates.</td>
</tr>
<tr>
<td>'04</td>
<td><strong>Verification/debugging/optimization of user-defined</strong> resources.</td>
</tr>
<tr>
<td>'05</td>
<td><em>Multi-language support</em> using CLP as IR: Java, C# (shapes, resources, ...).</td>
</tr>
<tr>
<td>'08</td>
<td>Verification of exec. time. First results in energy (Java), heap models, ...</td>
</tr>
<tr>
<td>'12</td>
<td>(X)C program energy analysis/verification, ISA-level energy models.</td>
</tr>
<tr>
<td>'13</td>
<td><strong>Cost analysis as abstract interpretation</strong>: Sized shapes inference. LLVM.</td>
</tr>
</tbody>
</table>
http://www.ciao-lang.org

Provides access to:
- Ciao, CiaoPP, LPdoc, etc.
- Documentation.
- Mailing lists.
- etc.

Please contact us for **GIT access**.

Around 1,000,000 lines of (mostly Ciao/Prolog) code.
Mostly **LGPL** (some packages have some variations).
References – Overall Model

On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs.

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

Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs.

A Generic Preprocessor for Program Validation and Debugging.

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.

[MLGH09] E. Mera, P. López-García, and M. Hermenegildo.
Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework.
References – Assertion Language

An Assertion Language for Debugging of Constraint Logic Programs.

An Assertion Language for Constraint Logic Programs.

[MLGH09] E. Mera, P. López-García, and M. Hermenegildo.
Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework.

References – Horn Clauses as Intermediate Representation

A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs.
References – Abstraction Carrying Code

Abstraction-Carrying Code.

Abstraction Carrying Code and Resource-Awareness.

Reduced Certificates for Abstraction-Carrying Code.
References – Fixpoint-based Framework (Abstract Interpreters)


References – Modular Analysis, Analysis of Concurrency

Analyzing Logic Programs with Dynamic Scheduling. 

Global Analysis of Standard Prolog Programs. 

[PH00] G. Puebla and M. Hermenegildo. 
Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. 

A Model for Inter-module Analysis and Optimizing Compilation. 

Context-Sensitive Multivariant Assertion Checking in Modular Programs. 
In LPAR’06, number 4246 in LNCS, pages 392–406. Springer-Verlag, November 2006.

A Practical Type Analysis for Verification of Modular Prolog Programs. 
References – Domains: Sharing/Aliasing


References – Domains: Shape/Type Analysis


References – Domains: Non-failure, Determinacy

Non-Failure Analysis for Logic Programs.

Multivariant Non-Failure Analysis via Standard Abstract Interpretation.

Determinacy Analysis for Logic Programs Using Mode and Type Information.

Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information.
References – Analysis and Verification of Resources

Task Granularity Analysis in Logic Programs.

Towards Granularity Based Control of Parallelism in Logic Programs.

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

Estimating the Computational Cost of Logic Programs.

Lower Bound Cost Estimation for Logic Programs.

User-Definable Resource Bounds Analysis for Logic Programs.

Towards Execution Time Estimation in Abstract Machine-Based Languages.
Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications.
In The Sixth NASA Langley Formal Methods Workshop (LFM 08), April 2008.
Extended Abstract.

User-Definable Resource Usage Bounds Analysis for Java Bytecode.


Sized Type Analysis Logic Programs (Technical Communication).

Energy Consumption Analysis of Programs based on XMOS ISA-Level Models.
In Pre-proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR’13), September 2013.

Towards an Abstract Domain for Resource Analysis of Logic Programs Using Sized Types.
References – Automatic Parallelization, (Abstract) Partial Evaluation, Other Optimizations


Abstract Specialization and its Applications.
Invited talk.

Abstract Interpretation with Specialized Definitions.

A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism.

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.

Improving the Compilation of Prolog to C Using Modeled Types and Determinism Information.

High-Level Languages for Small Devices: A Case Study.
Ciao Architecture Overview

Development Environment
Emacs based, command line, top-levels (compilation, analysis)

Source (user and library)
Packages (multi-paradigm)
- fsyntax
- hiord
- clpr
...

Modules (w./wo. assertions)
- mod1
- mod2
...
- modn

Preprocessor
Analysis (types, modes, resources, . . .)
Verification (static checking of assertions)
Optimization (parallelism, specialization, . . .)

Compiler
Expand Code (Kernel Language)

Back-end Compiler (optimized from annotations)

Executable Code (bytecode, native code)

Compile-time Messages
Errors/warnings
Static Violations

Run-time Messages
Debugging
Dynamic Violations

Run-time Engine and Libs.
Multi-platform
Parallel, sequential, tabled, . . .

Documenter
(automatic documentation from programs with assertions)
The tabling algorithm via an example (OLDT resolution)

Example

```prolog
:- table reach/2.

reach(X,Y) :-
    edge(X,Z),
    reach(Z,Y).
reach(X,Y) :-
    edge(X,Y).

edge(1,2).
edge(2,1).

?- reach(1,Y).
```

<table>
<thead>
<tr>
<th>Subgoal</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. reach(1,Y)</td>
<td>10. Y = 1</td>
</tr>
<tr>
<td></td>
<td>15. Y = 2</td>
</tr>
<tr>
<td></td>
<td>18. Complete</td>
</tr>
<tr>
<td>5. reach(2,Y)</td>
<td>9. Y = 1</td>
</tr>
<tr>
<td></td>
<td>17. Y = 2</td>
</tr>
<tr>
<td></td>
<td>18. Complete</td>
</tr>
</tbody>
</table>

Z=2

Y=1

Z=1

Y=2

13. edge(1,Y).

14. edge(1,2).

11. reach(1,1).

16. reach(1,2).

12. fail
Entailment vs. Call Abstraction

- **TCHR**: implementation of CHR on top of XSB Prolog with tabling.
  - It uses call abstraction.
- **Reach**: is some graph in a node reachable within some distance?

<table>
<thead>
<tr>
<th>Reach</th>
<th>Ciao</th>
<th>TCLP</th>
<th>TCHR</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>7140</td>
<td>129 978</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>6680</td>
<td>129 876</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>5964</td>
<td>128 955</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>4316</td>
<td>129 313</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2296</td>
<td>128 994</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>427</td>
<td>129 616</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>129 472</td>
<td></td>
</tr>
</tbody>
</table>

- Constraints reduce the search space.
The Assertion Language (subset)

\[
\text{:- pred } \text{Pred} \ [\text{:Precond}] \ [\Rightarrow \text{ Postcond}] \ [+ \text{ Comp-formula }] .
\]

Each typically a “mode” of use; the set covers the valid calls.

\[
\text{:- pred quicksort}(X,Y) : \text{list}() * \text{var} \Rightarrow \text{sorted}(Y) + (\text{is\_det,not\_fails}).
\]

\[
\text{:- pred quicksort}(X,Y) : \text{var} * \text{list}(\text{int}) \Rightarrow \text{ground}(X) + \text{non\_det}.
\]

Properties; from libraries or user defined (in the source language):

\[
\text{:- regtype color := green | blue | red.}
\]

\[
\text{:- regtype list}(X) := [], [X|\text{list}]. \quad \equiv \quad \text{list}(\_,[]). \quad \text{list}(X,[H|T]) \quad \Rightarrow \quad X(H), \text{list}(X,T).
\]

\[
\text{:- prop sorted := [], [\_], [X,Y|Z] \Rightarrow X > Y, \text{sorted}([Y|Z]).}
\]

Types/shapes, cost, data sizes, aliasing, termination, determinacy, non-failure, ...

Program-point Assertions:
- Inlined with code: \(..., \text{check}(\text{int}(X), X>0, \text{mshare}([\![X]\!])), \ldots\)

Assertion Status (so far “to be checked” – check status – default)
- Also: \text{trust} (guide analyzer), \text{true/false} (analysis output), \text{test}, etc.
Verification and Error Detection using Safe Approximations

- Need to compare actual semantics $[P]$ with intended semantics $\mathcal{I}$:

<table>
<thead>
<tr>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ is partially correct w.r.t. $\mathcal{I}$ iff</td>
<td>$[P] \leq \mathcal{I}$</td>
</tr>
<tr>
<td>$P$ is complete w.r.t. $\mathcal{I}$ iff</td>
<td>$\mathcal{I} \leq [P]$</td>
</tr>
<tr>
<td>$P$ is incorrect w.r.t. $\mathcal{I}$ iff</td>
<td>$[P] \nleq \mathcal{I}$</td>
</tr>
<tr>
<td>$P$ is incomplete w.r.t. $\mathcal{I}$ iff</td>
<td>$\mathcal{I} \nleq [P]$</td>
</tr>
</tbody>
</table>

- Usually, partial descriptions of $\mathcal{I}$ available, typically as assertions.

- **Problem:** difficulty computing $[P]$ w.r.t. interesting observables.

- **Approach:** use a safe approximation of $[P] \rightarrow$ i.e., $[P]_{\alpha^+}$ or $[P]_{\alpha^-}$

- Specially attractive if compiler computes (most of) $[P]_{\alpha^+}$ anyway.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ is prt. correct w.r.t. $\mathcal{I}_\alpha$ if</td>
<td>$\alpha([P]) \leq \mathcal{I}_\alpha$</td>
</tr>
<tr>
<td>$[P]<em>{\alpha^+} \leq \mathcal{I}</em>\alpha$</td>
<td></td>
</tr>
<tr>
<td>$P$ is complete w.r.t. $\mathcal{I}_\alpha$ if</td>
<td>$\mathcal{I}_\alpha \leq \alpha([P])$</td>
</tr>
<tr>
<td>$\mathcal{I}<em>\alpha \leq [P]</em>{\alpha}$</td>
<td></td>
</tr>
<tr>
<td>$P$ is incorrect w.r.t. $\mathcal{I}_\alpha$ if</td>
<td>$\alpha([P]) \nleq \mathcal{I}_\alpha$</td>
</tr>
<tr>
<td>$[P]<em>{\alpha^+} \nleq \mathcal{I}</em>\alpha$, or $[P]_{\alpha^-}$</td>
<td></td>
</tr>
<tr>
<td>$\mathcal{I}<em>\alpha \nleq [P]</em>{\alpha^+}$</td>
<td>$\mathcal{I}<em>\alpha \nleq [P]</em>{\alpha^-}$</td>
</tr>
</tbody>
</table>

[Hermenegildo et al. (IMDEA, UPM, ...) Analysis and Verification “of and with” CLP Rich Model–Jun 16-17, 2013 80 / 80]