Computational Logic: 
(Constraint) Logic Programming
Theory, practice, and implementation

Program Analysis, Debugging, and Optimization

A Tour of ciaopp: The Ciao Prolog Preprocessor

Department of Artificial Intelligence
School of Computer Science
Technical University of Madrid
28660-Boadilla del Monte, Madrid, SPAIN

The following people have contributed to this course material:
Manuel Hermenegildo (editor), Francisco Bueno, Manuel Carro, Germán Puebla, and Pedro López Technical University of Madrid, Spain
Ciao is a next-generation (C)LP programming environment – features:

- Public domain (GNU license).
- Pure kernel (*no “built-ins”*); subsumes ISO-Prolog (transparently) via *library*.
- Designed to be extensible and analyzable.
- Support for programming *in the large*:
  * robust module/object system, separate/incremental compilation, ...
  * “industry standard” performance.
  * (semi-automatic) interfaces to other languages, databases, etc.
  * assertion language, automatic static inference and checking, autodoc, ...
- Support for programming *in the small*:
  * scripts, small (static/dynamic/lazy-load) executables, ...
- Support for several paradigms:
  * functions, higher-order, objects, constraint domains, ...
  * concurrency, parallelism, distributed execution, ...
- Advanced Emacs environment (with e.g., automatic access to documentation).
Components of the environment (independent):

- **ciaosh**: Standard top-level shell.
- **ciaoc**: Standalone compiler.
- **ciaosi**: Script interpreter.
- **lpdoc**: Documentation Generator (info, ps, pdf, html, ...).
- **ciaopp**: Preprocessor.

Many libraries:

- Records (argument names).
- Persistent predicates.
- Transparent interface to databases.
- Interfaces to C, Java, tcl-tk, etc.
- Distributed execution.
- Internet (PiLLoW: HTML, VRML, forms, http protocol, etc.), ...
CiaoPP: The Ciao System Preprocessor

- A standalone preprocessor to the standard clause-level compiler [6].
- Performs source-to-source transformations:
  - Output: error/warning messages + transformed logic program, with
    - Results of analysis, as assertions (types, modes, sharing, non-failure, determinacy, term sizes, cost, ...).
    - Results of static checking of assertions [8, 14] (abstract verification).
    - Assertion run-time checking code.
    - Optimizations (specialization, parallelization, etc.).
- By design, a generic tool – can be applied to other systems (e.g., CHIP → CHIPRE).
- Underlying technology:
  - Modular polyvariant abstract interpretation [2, 10].
  - Modular abstract multiple specialization [17].
We demonstrate Ciaopp in use:

- Inference of complex properties of programs.
- Program debugging.
- Program validation.
- Program optimization (e.g., specialization, parallelization).
- Program documentation.

We discuss some practical issues:

- The assertion language.
- Dealing with built-ins and complex language features.
- Modular analysis (including libraries).
- Efficiency and incremental analysis (only reanalyze what is needed).

We start by describing the Ciao assertion language, used throughout the demo.
Properties and Assertions – I

- Assertion language \[13\] suitable for *multiple purposes* (see later).
- Assertions are typically *optional*.
- Properties (include *types* as a special case):
  - Arbitrary predicates, (generally) *written in the source language*.
  - Some predefined in system, some of them “native” to an analyzer.
  - Others user-defined.
  - Should be “runnable” (but property may be an approximation itself).

```prolog
:- regtype list/1. | :- typedef list ::= [];[_|list].
list([]). |   
list([_|Y]) :- list(Y). | _________________________________
______________________________|  :- regtype int/1 + impl_defined.
:- prop sorted/1. | _________________________________
sorted([]). |  :- regtype peano_int/1.
sorted([_]). | peano_int(0).
sorted([X,Y|Z]) :- X>Y, sorted([Y|Z]). | peano_int(s(X)) :- peano_int(X).
```

5
Properties and Assertions – II

• Basic assertions:

<table>
<thead>
<tr>
<th>PreC</th>
<th>PostC</th>
</tr>
</thead>
<tbody>
<tr>
<td>:- success PredictorDesc</td>
<td>PreC</td>
</tr>
<tr>
<td>:- calls PredictorDesc</td>
<td>PreC</td>
</tr>
<tr>
<td>:- comp PredictorDesc</td>
<td>PreC</td>
</tr>
</tbody>
</table>

Examples:
:- success qsort(A,B) : list(A) => ground(B).
:- calls qsort(A,B) : (list(A),var(B)).
:- comp qsort(A,B) : (list(A,int),var(B)) + (det,succeeds).

• Compound assertion (syntactic sugar):

<table>
<thead>
<tr>
<th>PreC</th>
<th>PostC</th>
</tr>
</thead>
<tbody>
<tr>
<td>:- pred PredictorDesc</td>
<td>PreC</td>
</tr>
</tbody>
</table>

Examples:
:- pred qsort(A,B) : (list(A,int),var(B)) => sorted(B) + (det,succeeds).
:- pred qsort(A,B) : (var(A),list(B,int)) => ground(A) + succeeds.
Properties and Assertions – III

• Assertion status:
  ◦ check (default) – intended semantics, to be checked.
  ◦ true, false – actual semantics, output from compiler.
  ◦ trust – actual semantics, input from user (guiding compiler).
  ◦ checked – validation: a check that has been proved (same as a true).

  :- trust pred is(X,Y) => (num(X),numexpr(Y)).

• Program point assertions:
  main :- read(X), trust(int(X)), ...

• entry: equiv. to “trust calls” (but only describes calls external to a module).

• + much more syntactic sugar, mode macros, “compatibility” properties, fields for automatic documentation [7], ...

  :- pred p/2 : list(int) * var => list(int) * int.
  :- modedef +X : nonvar(X).
  :- pred sortints(+L,-SL) :: list(int) * list(int) + sorted(SL)
      # "@var{SL} has same elements as @var{L}".
PART I: Analysis

- **ciaopp** includes two basic analyzers:
  - ◦ The PLAI generic, top-down analysis framework.
    * Several domains: modes (ground, free), independence, patterns, etc.
    * Incremental analysis, analysis of programs with delay, ...
  - ◦ Gallagher’s bottom-up type analysis.
    * Adapted to infer *parametric types* \((\text{list}(\text{int}))\) and at the *literal level*.
  - ◦ Advanced analyzers (GraCos/CASLOG) for complex properties:
    non-failure, coverage, determinism, sizes, cost, ...

- **Issues:**
  - ◦ Reporting the results \(\rightarrow\) “true” assertions.
  - ◦ Helping the analyzer \(\rightarrow\) “entry/trust” assertions.
  - ◦ Dealing with builtins \(\rightarrow\) “trust” assertions.
  - ◦ Incomplete programs \(\rightarrow\) “trust” assertions.
  - ◦ Modular programs \(\rightarrow\) “trust” assertions, interface (.itf, .asr) files.
  - ◦ Multivariance, incrementality, ...
Inference of Complex Properties: Non-failure (Intuition)

- Based on the intuitively simple notion of a set of tests “covering” the type of the input variables.
- Clause: set of primitive tests followed by various unifications and body goals.
- The tests at the beginning determine whether the clause should be executed or not (may involve pattern matching, arithmetic tests, type tests, etc.)
- Consider the predicate:
  \[
  \text{abs}(X, Y) \leftarrow X \geq 0, \ Y \text{ is } X.
  \]
  \[
  \text{abs}(X, Y) \leftarrow X < 0, \ Y \text{ is } -X.
  \]

- and a call to \text{abs}/2 with \(X\) bound to an \text{integer} and \(Y\) free.
- The test of \text{abs}/2, \(X \geq 0 \lor X < 0\), will succeed for this call.
- “The test of the predicate \text{abs}/2 covers the type of \(X\).”
- Since the rest of the body literals of \text{abs}/2 are guaranteed not to fail, at least one of the clauses will not fail, and thus the call will also not fail.
Inference of Complex Properties: Lower-Bounds on Cost (Intuition)

\[- \text{true pred append}(A, B, C): \text{list} * \text{list} * \text{var.} \]
\[\text{append([], L, L).} \]
\[\text{append([H|L], L1, [H|R]) :- append(L, L1, R).} \]

- Assuming:
  - Cost metric: number of resolution steps.
  - Argument size metric: list length.
  - Types, modes, covering, and non-failure info available.

- Let $\text{Cost}_{\text{append}}(n, m)$: cost of a call to $\text{append}/3$ with input lists of lengths $n$ and $m$.

- A difference equation can be set up for $\text{append}/3$:

  $$
  \begin{align*}
  \text{Cost}_{\text{append}}(0, m) &= 1 \quad \text{(boundary condition from first clause)}, \\
  \text{Cost}_{\text{append}}(n, m) &= 1 + \text{Cost}_{\text{append}}(n - 1, m).
  \end{align*}
  $$

- Solution obtained: $\text{Cost}_{\text{append}}(n, m) = n + 1$.

- Based on also inferring argument size relationships (relative sizes).
“Resource awareness” example (Upper-Bounds Cost Analysis)

• Given:

  \[\text{:- entry inc\_all : ground } \times \text{ var.}\]

  \text{inc\_all([],[]).}
  \text{inc\_all([H|T],[NH|NT]) :- NH is H+1, inc\_all(T,NT).}\n
• After running through ciaopp (cost analysis) we get:

  \[\text{:- entry inc\_all : ground } \times \text{ var.}\]

  \[\text{:- true pred inc\_all(A,B) : (list(A,\text{int}), var(B))}\]
  \[\text{ \ \ \ \ => (list(A,\text{int}), list(B,\text{int}))}\]
  \[\text{ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ + \text{ upper\_cost(2*length(A)+1).}\]\n
  \text{inc\_all([],[]).}
  \text{inc\_all([H|T],[NH|NT]) :- NH is H+1, inc\_all(T,NT).}\n
  which is a program with a certificate of needed resources!
• We compare actual semantics $[P]$ vs. intended semantics $\mathcal{I}$ for $P$:
  ◦ $P$ is partially correct w.r.t. $\mathcal{I}$ iff $[P] \subseteq \mathcal{I}$.
  ◦ $P$ is complete w.r.t. $\mathcal{I}$ iff $\mathcal{I} \subseteq [P]$.
  ◦ $P$ is incorrect w.r.t. $\mathcal{I}$ iff $[P] \nsubseteq \mathcal{I}$.
  ◦ $P$ is incomplete w.r.t. $\mathcal{I}$ iff $\mathcal{I} \nsubseteq [P]$.

• $\mathcal{I}$ described via (check) assertions.

• Incorrectness and incompleteness indicate that diagnosis should be performed.

• Problems: difficulty in computing $[P]$ (+ $\mathcal{I}$ incomplete, i.e., approximate).

• Approach:
  ◦ Use the abstract interpreter to infer properties of $P$.
  ◦ Compare them to the assertions.
  ◦ Generate run-time tests if anything remains to be tested.
Validation Using Abstract Interpretation

- Specification given as a semantic value $\mathcal{I}_\alpha \in D_\alpha$ and compared with $\llbracket P \rrbracket_\alpha$.

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
<th>Sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>P is partially correct w.r.t. $\mathcal{I}_\alpha$</td>
<td>$\alpha(\llbracket P \rrbracket) \subseteq \mathcal{I}_\alpha$</td>
<td>$\llbracket P \rrbracket_{\alpha^+} \subseteq \mathcal{I}_\alpha$</td>
</tr>
<tr>
<td>P is complete w.r.t. $\mathcal{I}_\alpha$</td>
<td>$\mathcal{I}_\alpha \subseteq \alpha(\llbracket P \rrbracket)$</td>
<td>$\mathcal{I}<em>\alpha \subseteq \llbracket P \rrbracket</em>{\alpha^-}$</td>
</tr>
<tr>
<td>P is incorrect w.r.t. $\mathcal{I}_\alpha$</td>
<td>$\alpha(\llbracket P \rrbracket) \not\subseteq \mathcal{I}_\alpha$</td>
<td>$\llbracket P \rrbracket_{\alpha^-} \not\subseteq \mathcal{I}<em>\alpha$, or $\llbracket P \rrbracket</em>{\alpha^+} \cap \mathcal{I}<em>\alpha = \emptyset \land \llbracket P \rrbracket</em>{\alpha} \neq \emptyset$</td>
</tr>
<tr>
<td>P is incomplete w.r.t. $\mathcal{I}_\alpha$</td>
<td>$\mathcal{I}_\alpha \not\subseteq \alpha(\llbracket P \rrbracket)$</td>
<td>$\mathcal{I}<em>\alpha \not\subseteq \llbracket P \rrbracket</em>{\alpha^+}$</td>
</tr>
</tbody>
</table>

($\llbracket P \rrbracket_{\alpha^+}$ represents that $\llbracket P \rrbracket_{\alpha} \supseteq \alpha(\llbracket P \rrbracket)$ and $\llbracket P \rrbracket_{\alpha^-}$ indicates that $\llbracket P \rrbracket_{\alpha} \subseteq \alpha(\llbracket P \rrbracket)$)

- Conclusions w.r.t. direct Galois insertions (i.e., over-approximation):
  - Suited for proving partial correctness and incompleteness w.r.t. $\mathcal{I}$.
  - It is also possible to prove incorrectness.
  - Completeness can only be proved if the abstraction is “precise.”

- Conclusion w.r.t. reversed Galois insertions (i.e., under-approximation):
  - Suited for proving completeness and incorrectness.
  - Partial correctness and incompleteness only if the abstraction is “precise.”
Integrated Validation/Diagnosis in the Ciao Preprocessor

- Syntax error/warning
- Semantic comp-time error/warning
- Interactive Diagnosis
- System run-time error
- User run-time error
- Program + RT tests
- CIAO, CHIP, ...
- Output
- Inspection
- CIAOPP
- Syntax checker
- Static Analysis
- Analysis Info
- Comparator
- RT tests Annotator
- Program
- :- entry
- :- check
- Builtins/Libs
- :- false
- :- check
- :- checked
- System run-time error

Program
:- entry
:- check
Builtins/Libs
A Program validation example

• Given:

\[
\text{:- check comp : list(int) * var + succeeds.}
\]
\[
\text{inc_all([],[]).}
\]
\[
\text{inc_all([H|T],[NH|NT]) :- NH is H+1, inc_all(T,NT).}
\]

• After running through ciaopp (non-failure analysis) we get:

\[
\text{:- true comp : list(int) * var + succeeds.}
\]
\[
\text{inc_all([],[]).}
\]
\[
\text{inc_all([H|T],[NH|NT]) :- NH is H+1, inc_all(T,NT).}
\]

which is a validated (certified) program.
Debugging with Global Analysis

- Simple bugs:
  - Undefined predicates, discontiguous, multiple arity, ...
  - Cannot be done without global analysis & a robust module system.

- Checking programs against library interfaces:
  - System predicates (builtin and library predicates):
    - Intended behavior known in advance / usually assumed to be correct.
  - If interfaces of these predicates are available as *assertions*, we can:
    - automatically compare analysis results against these specs,
    - (+ avoid analyzing the libraries over and over again).
  - Detects many bugs with no user burden (no need to use assert. language).
  - Can also be done with user-defined libraries!

- We may be interested also in checking properties of our program.
  - Price: adding *assertions* describing what we want checked (can be partial).
  - Advantage: more errors detected and automatic documentation!
Finding Bugs with Global Analysis

- Checking the calls to built-ins and libraries:
  
  \[
  \text{main}(X,Y) :- q(X,N), Y \text{ is } X+N.
  \]

  \[
  q(1,V).
  \]

  with, e.g., mode analysis an error is flagged: \(N\) is not ground.

- Checking program assertions:
  
  \[
  :- \text{pred } p(X,Y) : \text{list(num)} * \text{var} => \text{list(num)} * \text{list(num)} + \text{no}\_\text{fail}.
  \]

  \[
  p([],[]).
  \]

  \[
  p([H|T],[NH|NT]) :- q(H,NH), p(T,NT).
  \]

  \[
  q(H,NH) :- H > 0, NH = H+1.
  \]

  \[
  q(H,NH) :- H < 0, NH = H-1.
  \]

  with, e.g., type analysis an error is flagged: \(Y\) is not a list of numbers (\text{is}/2 should be used instead of \text{=/2});

  with, e.g., non-failure analysis an error is flagged: \text{=/2} should be used.
Discussion: Comparison with “Classical” Types

- Global analysis w/approximations: important role also in program development.
- Allows going beyond straight-jacket of classical type systems (Gödel, Mercury,...):

<table>
<thead>
<tr>
<th>“Traditional” Types</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compulsory (do not allow “any”)</td>
<td>Optional (allow “any”)</td>
</tr>
<tr>
<td>Expressed in a Special Language</td>
<td>Expressed in the Source Language</td>
</tr>
<tr>
<td>Limited Property Language</td>
<td>Much More General Property Language</td>
</tr>
<tr>
<td>Limit Programming Language</td>
<td>Do not Limit Programming Language</td>
</tr>
<tr>
<td>Untypable Programs Rejected</td>
<td>Run-time Checks Introduced</td>
</tr>
<tr>
<td>(Almost) Decidable</td>
<td>Approximated</td>
</tr>
<tr>
<td>“check”</td>
<td>“check” or “trust”</td>
</tr>
</tbody>
</table>

...without giving up much (types are included as just another kind of property).

- Key issues:

<table>
<thead>
<tr>
<th>Approximation</th>
<th>Suitable assertion language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Interpretation</td>
<td>Relating approximations of actual and intended semantics</td>
</tr>
</tbody>
</table>
PART III: Using Analysis Results in Program Optimization

- Eliminating run-time work at compile-time.
  - Low-level optimization.
  - Abstract specialization/partial evaluation.
    Evaluating parts of the program based on abstract information.
  - Abstract multiple specialization.
    Ditto on (possibly) multiple versions of each predicate.
- Automatic program parallelization:
  strict and non-strict Independent And-Parallelism.
- Automatic task granularity control.
- Optimization of other control rules / languages (e.g., Andorra).
- Just for fun: generating documentation!
(Multiple) Specialization

- Given the analysis output:

```prolog
main :-

..., 
true(int(X)),
( ground(X) -> write(a) ; write(b) ),
...
```

the ground(X) can be *abstractly executed* to true and the whole conditional to write(A).

- Specializer is customizable, controlled by a table of “abstract executability”.

- Can subsume traditional “partial evaluation”:
  Given `true(X=list(a))`, then, e.g., `X=[a|Y] → X=[_|Y]` (no need to test that first element is an a).

- Multiple specialization: creating multiple versions of predicates for different uses.
Automatic Program Parallelization

- Parallelization process \([2]\) starts with dependency graph:
  - edges exist if there can be a dependency,
  - conditions label edges if the dependency can be removed.

- Global analysis: reduce number of checks in conditions (also to true and false).

- Annotation: encoding of parallelism in the target parallel language:
  \[ g_1(\ldots), g_2(\ldots), g_3(\ldots) \]

```plaintext
\begin{align*}
&g_1, (g_2 \& g_3) \\
\text{Alternative:} &\quad g_1, (g_2 \& g_3) \\
\end{align*}
```

Local/Global analysis and simplification

"Annotation"
Example:

```
qs([X|L],R) :- part(L,X,L1,L2),
    qs(L2,R2), qs(L1,R1),
    app(R1,[X|R2],R).
```

Might be annotated in &-Prolog (or Ciao Prolog), using local analysis, as:

```
qs([X|L],R) :-
    part(L,X,L1,L2),
    ( indep(L1,L2) ->
        qs(L2,R2) & qs(L1,R1)
    ;
        qs(L2,R2) , qs(L1,R1) ),
    app(R1,[X|R2],R).
```

Global analysis would eliminate the `indep(L1,L2)` check.
&-Prolog/Ciao parallelizer overview

USER

Ciao:
(C)LP, FP, (Java) ...

Abstract Interpretation
(Sharing, Sharing+Freeness,
Aeqs, Def, Lsign, ...)

Dependency Info

MEL/CDG/UDG/URLP/...

Annotators (local
dependency analysis)

side−effect analysis

Parallelized Code (&)

ciao/−−Prolog
Parallel RT system

granularity analysis

PARALLELIZING COMPILER (CiaoPP)
Granularity Control

- Do not schedule tasks for parallel execution if they are too small.
- Cannot be done well completely at compile-time: work done by a call often depends on the size of its input:
  
  \[
  q([],[]). \\
  q([X|RX],[X1|RX1]) :- X1 is X +1, \ q(RX,RX1).
  \]

- **Approach** [12]:
  - generate at compile-time *functions* (to be evaluated at run-time) that efficiently approximate task size (upper and lower bounds),
  - transform programs to carry out run-time granularity control.
  - Note: size computations can be done on-the-fly [11].

- Example (with \(q\) above):
  - \(\ldots, q(X,Y) & r(X), \ldots\)
  
  \[\text{Cost} = 2 \times \text{length}(X) + 1\text{ (cost function }2 \times n + 1\text{). Assuming }\text{threshold}\text{ is 4 units:}\]
  
  \(\ldots, \text{length}(X,LX), \text{Cost is } LX \times 2 + 1, \text{ (Cost > 4 } \rightarrow \text{ } q(X,Y) & r(Z) \) \\
  \( \text{; } q(X,y), \text{ r}(X) \), \ldots\)
Granularity Control System Output

g_qsort([], []).
g_qsort([First|L1], L2) :-
    partition3o4o(First, L1, Ls, Lg, Size_Ls, Size_Lg),
    Size_Ls > 20 ->
        (Size_Lg > 20 -> g_qsort(Ls, Ls2) & g_qsort(Lg, Lg2);
         g_qsort(Ls, Ls2), s_qsort(Lg, Lg2)));
    (Size_Lg > 20 -> s_qsort(Ls, Ls2), g_qsort(Lg, Lg2);
     s_qsort(Ls, Ls2), s_qsort(Lg, Lg2)),
    append(Ls2, [First|Lg2], L2).

partition3o4o(F, [], [], [], 0, 0).
partition3o4o(F, [X|Y], [X|Y1], Y2, SL, SG) :-
    X <= F, partition3o4o(F, Y, Y1, Y2, SL1, SG), SL is SL1 + 1.
partition3o4o(F, [X|Y], Y1, [X|Y2], SL, SG) :-
    X > F, partition3o4o(F, Y, Y1, Y2, SL, SG1), xSG is SG1 + 1.

• Note: when term sizes are compared directly with a threshold: not necessary to traverse all the terms involved, only to the point at which threshold is reached.
• ciaopp is generic, i.e., it can be customized:
  ◦ For a new language: giving assertions for its built-ins and libraries (+ syntax).
  ◦ For new properties: adding a new domain to the analyzer.

• Example: chipre, preprocessor for CHIP.
Acknowledgements/Downloading the systems

• Ciao/ciaopp is a collaborative effort:
  UPM, Melbourne/Monash (incremental analysis, ...), Arizona (cost analyses, ...),
  SICS (engine)
  + Bristol, Linköping, NMSU, Leuven, Beer-Sheva, ...

• Downloading ciao, ciaopp, ciaodoc/pl2texi, and other CLIP software:
  ◦ Standard distributions:
    http://www.clip.dia.fi.upm.es/Software
  ◦ Betas (in testing or completing documentation – ask webmaster for info):
    http://www.clip.dia.fi.upm.es/Software/Beta
  ◦ User’s mailing list:
    ciao-users@clip.dia.fi.upm.es
    Subscribe by sending a message with only subscribe in the body to
    ciao-users-request@clip.dia.fi.upm.es
Recent Bibliography on the ciaopp System Components


