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
Introduction: The Ciao Program Development System

- 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).
Introduction: The Ciao Program Development System (Contd.)

- 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].
Overview

- 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).

\[
\begin{align*}
\text{- regtype list/1.} & \quad \text{- typedef list ::= [];[_|list].} \\
\text{list([]).} & \quad \text{list([_|Y]) :- list(Y).} \\
\text{- prop sorted/1.} & \quad \text{sorted([]).} \\
\text{sorted([_]).} & \quad \text{sorted([X,Y|Z]) :- X>Y, sorted([Y|Z]).} \\
\end{align*}
\]

```prolog
:- regtype list/1. | :- typedef list ::= [];[_|list].
list([]). | :- regtype int/1 + impl_defined.
list([_|Y]) :- list(Y). | :- regtype peano_int/1.
sorted([]). | peano_int(0).
sorted([_]). | peano_int(s(X)) :- peano_int(X).
sorted([X,Y|Z]) :- X>Y, sorted([Y|Z]).
```
Properties and Assertions – II

• Basic assertions:

\[
\begin{align*}
\text{:- success} & \quad \text{PredDesc} [ : \text{PreC} ] \Rightarrow \text{PostC}. \\
\text{:- calls} & \quad \text{PredDesc} : \text{PreC}. \\
\text{:- comp} & \quad \text{PredDesc} [ : \text{PreC} ] + \text{CompProps}.
\end{align*}
\]

Examples:

\[
\begin{align*}
\text{:- success qsort(A,B)} : \text{list}(A) & \Rightarrow \text{ground}(B). \\
\text{:- calls qsort(A,B)} : (\text{list}(A), \text{var}(B)). \\
\text{:- comp qsort(A,B)} : (\text{list}(A, \text{int}), \text{var}(B)) + (\text{det}, \text{succeeds}).
\end{align*}
\]

• Compound assertion (syntactic sugar):

\[
\begin{align*}
\text{:- pred} & \quad \text{PredDesc} [ : \text{PreC} ] [\Rightarrow \text{PostC}] [+ \text{Comp}].
\end{align*}
\]

Examples:

\[
\begin{align*}
\text{:- pred qsort(A,B)} : (\text{list}(A, \text{int}), \text{var}(B)) & \Rightarrow \text{sorted}(B) + (\text{det}, \text{succeeds}). \\
\text{:- pred qsort(A,B)} : (\text{var}(A), \text{list}(B, \text{int})) & \Rightarrow \text{ground}(A) + \text{succeeds}.
\end{align*}
\]
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).

\[
\text{:- trust pred is(X,Y) } \Rightarrow (\text{num}(X),\text{numexpr}(Y)).
\]

- **Program point assertions:**
  
  ```prolog
  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], ...

\[
\text{:- pred p/2 : list(int) } \star \text{ var } \Rightarrow \text{ list(int) } \star \text{ int.}
\]

\[
\text{:- modedef +X : nonvar(X).}
\]

\[
\text{:- pred sortints(+L,-SL) : list(int) } \star \text{ list(int) } + \text{ 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 (list(int)) and at the literal level.
  - Advanced analyzers (GraCos/CASLOG) for complex properties: non-failure, coverage, determinism, sizes, cost, ...

- Issues:
  - Reporting the results → “true” assertions.
  - Helping the analyzer → “entry/trust” assertions.
  - Dealing with builtins → “trust” assertions.
  - Incomplete programs → “trust” assertions.
  - Modular programs → “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 \texttt{abs/2} with \( X \) bound to an integer and \( Y \) free.

- The test of \texttt{abs/2}, \( X \geq 0 \lor X < 0 \), will succeed for this call.

- “The test of the predicate \texttt{abs/2} covers the type of \( X \).”

- Since the rest of the body literals of \texttt{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)

:- true pred append(A,B,C): list * list * var.
append([], L, L).
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:

  ```prolog
  :- entry inc_all : ground * var.

  inc_all([],[]).
  inc_all([H|T],[NH|NT]) :- NH is H+1, inc_all(T,NT).
  ```

- After running through ciaopp (cost analysis) we get:

  ```prolog
  :- entry inc_all : ground * var.

  :- true pred inc_all(A,B) : (list(A,int), var(B))
      => (list(A,int), list(B,int))
      + upper_cost(2*length(A)+1).

  inc_all([],[]).
  inc_all([H|T],[NH|NT]) :- NH is H+1, inc_all(T,NT).
  ```

  which is a program with a certificate of needed resources!
PART II: Program Validation and Diagnosis (Debugging)

- 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 $I\alpha \in D\alpha$ and compared with $[P]_\alpha$.

<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. $I\alpha$</td>
<td>$\alpha([P]) \subseteq I\alpha$</td>
<td>$[P]_{\alpha^+} \subseteq I\alpha$</td>
</tr>
<tr>
<td>P is complete w.r.t. $I\alpha$</td>
<td>$I\alpha \subseteq \alpha([P])$</td>
<td>$I\alpha \subseteq [P]_{\alpha^-}$</td>
</tr>
<tr>
<td>P is incorrect w.r.t. $I\alpha$</td>
<td>$\alpha([P]) \not\subseteq I\alpha$</td>
<td>$[P]<em>{\alpha^-} \not\subseteq I\alpha$, or $[P]</em>{\alpha^+} \cap I\alpha = \emptyset \land [P]_{\alpha} \neq \emptyset$</td>
</tr>
<tr>
<td>P is incomplete w.r.t. $I\alpha$</td>
<td>$I\alpha \not\subseteq \alpha([P])$</td>
<td>$I\alpha \not\subseteq [P]_{\alpha^+}$</td>
</tr>
</tbody>
</table>

($[P]_{\alpha^+}$ represents that $[P]_{\alpha} \supseteq \alpha([P])$ and $[P]_{\alpha^-}$ indicates that $[P]_{\alpha} \subseteq \alpha([P])$)

- Conclusions w.r.t. direct Galois insertions (i.e., over-approximation):
  ◦ Suited for proving partial correctness and incompleteness w.r.t. $I$.
  ◦ 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.”
A Program validation example

• Given:

    :- check comp : list(int) * var + succeeds.

    inc_all([],[]).
    inc_all([H|T],[NH|NT]) :- NH is H+1, inc_all(T,NT).

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

    :- true comp : list(int) * var + succeeds.

    inc_all([],[]).
    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:
  ```prolog
  main(X,Y) :- q(X,N), Y is X+N.
  q(1,V).
  with, e.g., mode analysis an error is flagged: N is not ground.
  ```

- Checking program assertions:
  ```prolog
  :- pred p(X,Y) : list(num) * var => list(num) * list(num) + no_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
  (is/2 should be used instead of =/2);
  with, e.g., non-failure analysis an error is flagged: =</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] \rightarrow 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:
  \[ g_1(\ldots), \ g_2(\ldots), \ g_3(\ldots) \]

```
( test(1−3) -> ( g_1, g_2 ) & g_3 ; \ g_1, ( g_2 & g_3 ) )
```

Alternative:
```
\ g_1, ( g_2 & g_3 )
```

Local/Global analysis and simplification

"Annotation"
• **Example:**

\[
qs([X|L],R) :- \text{part}(L,X,L1,L2), \\
qs(L2,R2), qs(L1,R1), \\
app(R1,[X|R2],R).
\]

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

\[
qs([X|L],R) :- \\
\text{part}(L,X,L1,L2), \\
( \text{indep}(L1,L2) \rightarrow \\
qs(L2,R2) \& qs(L1,R1) \\
; \quad qs(L2,R2) \text{, } qs(L1,R1) ), \\
app(R1,[X|R2],R).
\]

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

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

Annotators (local dependency analysis)
MEL/CDG/UDG/URLP/...

Parallelized Code (&)

PARALLELIZING COMPILER (CiaoPP)

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

Dependency Info

side–effect analysis

granularity analysis

Ciao/&–Prolog
Parallel RT system
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:
  \[ q([],[]) \].
  \[ q([X|RX],[X1|RX1]) :\text{-- } X1 \text{ is } X +1, \quad q(RX,RX1). \]

- Approach\(^\text{[12]}\):
  - generate at compile-time \textit{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\(^\text{[11]}\).

- Example (with \(q\) above):
  ... , \(q(X,Y) \& r(X), ... \)

\[
\text{Cost} = 2 \times \text{length}(X) + 1 \text{ (cost function } 2 \times n + 1). \text{ Assuming } \textit{threshold} \text{ is } 4 \text{ units:}
\]

... , \text{length}(X,LX), \text{ Cost is } LX\times2+1, \; (\text{ Cost } > 4 \rightarrow q(X,Y) \& r(Z) \quad ; \quad q(X,y), \quad r(X) ), ...
Granularity Control System Output

\[
g_{\text{qsort}}([], []). \]
\[
g_{\text{qsort}}([\text{First}|\text{L1}], \text{L2}) :-
\]
\[
\text{partition3o4o}(\text{First}, \text{L1}, \text{Ls}, \text{Lg}, \text{Size}_\text{Ls}, \text{Size}_\text{Lg}),
\]
\[
\text{Size}_\text{Ls} > 20 \rightarrow
\]
\[
(\text{Size}_\text{Lg} > 20 \rightarrow \text{g}_{\text{qsort}}(\text{Ls}, \text{Ls2}) \& \text{g}_{\text{qsort}}(\text{Lg}, \text{Lg2});
\]
\[
\text{g}_{\text{qsort}}(\text{Ls}, \text{Ls2}), \text{s}_{\text{qsort}}(\text{Lg}, \text{Lg2});
\]
\[
(\text{Size}_\text{Lg} > 20 \rightarrow \text{s}_{\text{qsort}}(\text{Ls}, \text{Ls2}), \text{g}_{\text{qsort}}(\text{Lg}, \text{Lg2});
\]
\[
\text{s}_{\text{qsort}}(\text{Ls}, \text{Ls2}), \text{s}_{\text{qsort}}(\text{Lg}, \text{Lg2})),
\]
\[
\text{append}(\text{Ls2}, [\text{First}|\text{Lg2}], \text{L2}).
\]

\[
\text{partition3o4o}(\text{F}, [], [], [], 0, 0).
\]
\[
\text{partition3o4o}(\text{F}, [\text{X}|\text{Y}], [\text{X}|\text{Y1}], \text{Y2}, \text{SL}, \text{SG}) :-
\]
\[
\text{X} \leq \text{F}, \text{partition3o4o}(\text{F}, \text{Y}, \text{Y1}, \text{Y2}, \text{SL1}, \text{SG}), \text{SL is SL1} + 1.
\]
\[
\text{partition3o4o}(\text{F}, [\text{X}|\text{Y}], \text{Y1}, [\text{X}|\text{Y2}], \text{SL}, \text{SG}) :-
\]
\[
\text{X} > \text{F}, \text{partition3o4o}(\text{F}, \text{Y}, \text{Y1}, \text{Y2}, \text{SL}, \text{SG1}), \text{xSG is SG1} + 1.
\]

• 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


