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

- 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).

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

:- prop sorted/1.
sorted([]).
sorted([_]).
sorted([X,Y|Z]) :- X>Y, sorted([Y|Z]).
```

```
:- typedef list ::= [];[_|list].
list([]). | list(Y). 

:- regtype int/1 + impl_defined.
peano_int(0).
peano_int(s(X)) :- peano_int(X).
```
Properties and Assertions – II

• Basic assertions:

```prolog
:- calls PredDesc : PreC .
```

Examples:

```prolog
:- 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):

```prolog
```

Examples:

```prolog
:- 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).

```prolog
:- trust pred is(X,Y) => (num(X),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], ...

```prolog
:- 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* (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:
  \[ abs(X, Y) \leftarrow X \geq 0, \ Y \text{ is } X. \]
  \[ abs(X, Y) \leftarrow X < 0, \ Y \text{ is } -X. \]
- and a call to \( abs/2 \) with \( X \) bound to an *integer* and \( Y \) free.
- The test of \( abs/2, X \geq 0 \lor X < 0 \), will succeed for this call.
- “The test of the predicate \( abs/2 \) covers the type of \( X \).”
- Since the rest of the body literals of \( abs/2 \) 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 $\text{append}/3$:
  \[
  \text{Cost}_{\text{append}}(0, m) = 1 \quad \text{(boundary condition from first clause)},
  \]
  \[
  \text{Cost}_{\text{append}}(n, m) = 1 + \text{Cost}_{\text{append}}(n - 1, m).
  \]

- 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:
  
  :- 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:
  
  :- 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 $I$ for $P$:
  - $P$ is partially correct w.r.t. $I$ iff $[P] \subseteq I$.
  - $P$ is complete w.r.t. $I$ iff $I \subseteq [P]$.
  - $P$ is incorrect w.r.t. $I$ iff $[P] \not\subseteq I$.
  - $P$ is incomplete w.r.t. $I$ iff $I \not\subseteq [P]$.

- $I$ described via (check) assertions.
- Incorrectness and incompleteness indicate that diagnosis should be performed.
- **Problems**: difficulty in computing $[P]$ (+ $I$ 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>$[P]<em>{\alpha^+} \subseteq I</em>\alpha$</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>$[P]<em>{\alpha^-} \not\subseteq I</em>\alpha$, or $[P]<em>{\alpha^+} \cap I</em>\alpha = \emptyset \land [P]_{\alpha^-} \neq \emptyset$</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:
  main(X,Y) :- q(X,N), Y is X+N.
  q(1,Y).

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

- Checking program assertions:
  :- 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:
  
  ```
  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(...), g_2(...), g_3(...) 
  ```

  ![Parallelization Diagram]

  *Local/Global analysis and simplification*

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

  *Alternative:*
  
  ```
  g_1, ( g_2 & g_3 )
  ```
Automatic Program Parallelization (Contd.)

- Example:

\[
\text{qs}([X|L], R) :- \text{part}(L, X, L_1, L_2), \\
\text{qs}(L_2, R_2), \text{qs}(L_1, R_1), \\
\text{app}(R_1, [X|R_2], R).
\]

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

\[
\text{qs}([X|L], R) :- \\
\text{part}(L, X, L_1, L_2), \\
(\text{indep}(L_1, L_2) -> \\
\text{qs}(L_2, R_2) \& \text{qs}(L_1, R_1) ; \text{qs}(L_2, R_2), \text{qs}(L_1, R_1) ), \\
\text{app}(R_1, [X|R_2], R).
\]

Global analysis would eliminate the \text{indep}(L_1, L_2) check.

&-Prolog/Ciao parallelizer overview
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:
  \( q([],[]) \).

\[
q([X\mid RX],[X1\mid RX1]) :- X1 is X + 1, \ q(RX,RX1).
\]

- 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):
  
  \[
  \ldots, q(X,Y) \ & r(X), \ldots
  \]

  Cost = \( 2 * length(X) + 1 \) (cost function \( 2 * n + 1 \)). Assuming threshold is 4 units:

  \[
  \ldots, length(X,LX), \ Cost \ is \ LX*2+1, \ (\ Cost > 4 \rightarrow q(X,Y) \ & r(Z) \ ; \ q(X,y), \ r(X) \ ), \ldots
  \]

Granularity Control System Output

\[
g_{qsort}([], []).
g_{qsort}([First\mid L1], L2) :-
\]

\[
partition3o4o(First, L1, Ls, Lg, Size_Ls, Size_Lg),
\]

\[
Size_Ls > 20 \rightarrow
\]

\[
(\ Size_Lg > 20 \rightarrow g_{qsort}(Ls, Ls2) \ & g_{qsort}(Lg, Lg2);
\]

\[
g_{qsort}(Ls, Ls2), \ s_{qsort}(Lg, Lg2));
\]

\[
(\ Size_Lg > 20 \rightarrow s_{qsort}(Ls, Ls2), g_{qsort}(Lg, Lg2);
\]

\[
s_{qsort}(Ls, Ls2), s_{qsort}(Lg, Lg2))
\]

\[
append(Ls2, [First\mid Lg2], L2).
\]

\[
partition3o4o(F, [], [], [], 0, 0).
partition3o4o(F, [X\mid Y], [X\mid Y1], Y2, SL, SG) :-
\]

\[
X <= F, \ partition3o4o(F, Y, Y1, Y2, SL1, SG), \ SL \ is \ SL1 + 1.
\]

\[
partition3o4o(F, [X\mid Y], Y1, [X\mid Y2], SL, SG) :-
\]

\[
X > F, \ partition3o4o(F, Y, Y1, Y2, SL, SG1), \ xSG \ is \ SG1 + 1.
\]

- 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.
Genericity in the Ciao Preprocessor

- 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


