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:
  - Input: logic program (optionally w/assertions [15] & syntactic extensions).
  - 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 $\rightarrow$ 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).

```
:- 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].
```

```
| :- regtype int/1 + impl_defined.
```

```
| :- regtype peano_int/1.
| peano_int(0).
| peano_int(s(X)) :- peano_int(X).
```

Properties and Assertions – II

- Basic assertions:

```
:- success  PredDesc [ : PreC ]  => PostC .
:- calls    PredDesc   : PreC .
:- comp     PredDesc [ : PreC ]  + CompProps .
```

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):

```
:- pred  PredDesc [ : PreC ] [ => PostC ] [ + Comp ] .
```

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* (`list(int)`) and at the *literal level*.
  - Advanced analyzers (GraCos/CASLOG) for complex properties: non-failure, coverage, determinism, sizes, cost, ...
- Issues:
  - Reporting the results $\longrightarrow$ "true" assertions.
  - Helping the analyzer $\longrightarrow$ "entry/trust" assertions.
  - Dealing with builtins $\longrightarrow$ "trust" assertions.
  - Incomplete programs $\longrightarrow$ "trust" assertions.
  - Modular programs $\longrightarrow$ "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:

$$abs(X, Y) \leftarrow X \geq 0,\ Y\ is\ X.$$
$$abs(X, Y) \leftarrow X < 0,\ Y\ is\ -X.$$

- and a call to $abs/2$ with X bound to an $integer$ and Y free.
- The test of $abs/2$, $X \geq 0 \vee X < 0$, 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 $\texttt{Cost}_{\texttt{append}}(n, m)$: cost of a call to `append`/3 with input lists of lengths n and m.
- A difference equation can be set up for `append`/3:

  $\texttt{Cost}_{\texttt{append}}(0, m) = 1$ (boundary condition from first clause),
  $\texttt{Cost}_{\texttt{append}}(n, m) = 1 + \texttt{Cost}_{\texttt{append}}(n - 1, m)$.
- Solution obtained: $\texttt{Cost}_{\texttt{append}}(n, m) = n + 1$.
- 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 $\mathcal{I}$ 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 $[\![P]\!] \not\subseteq \mathcal{I}$.
  - P is *incomplete* w.r.t. $\mathcal{I}$ iff $\mathcal{I} \not\subseteq [\![P]\!]$.
- $\mathcal{I}$ 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

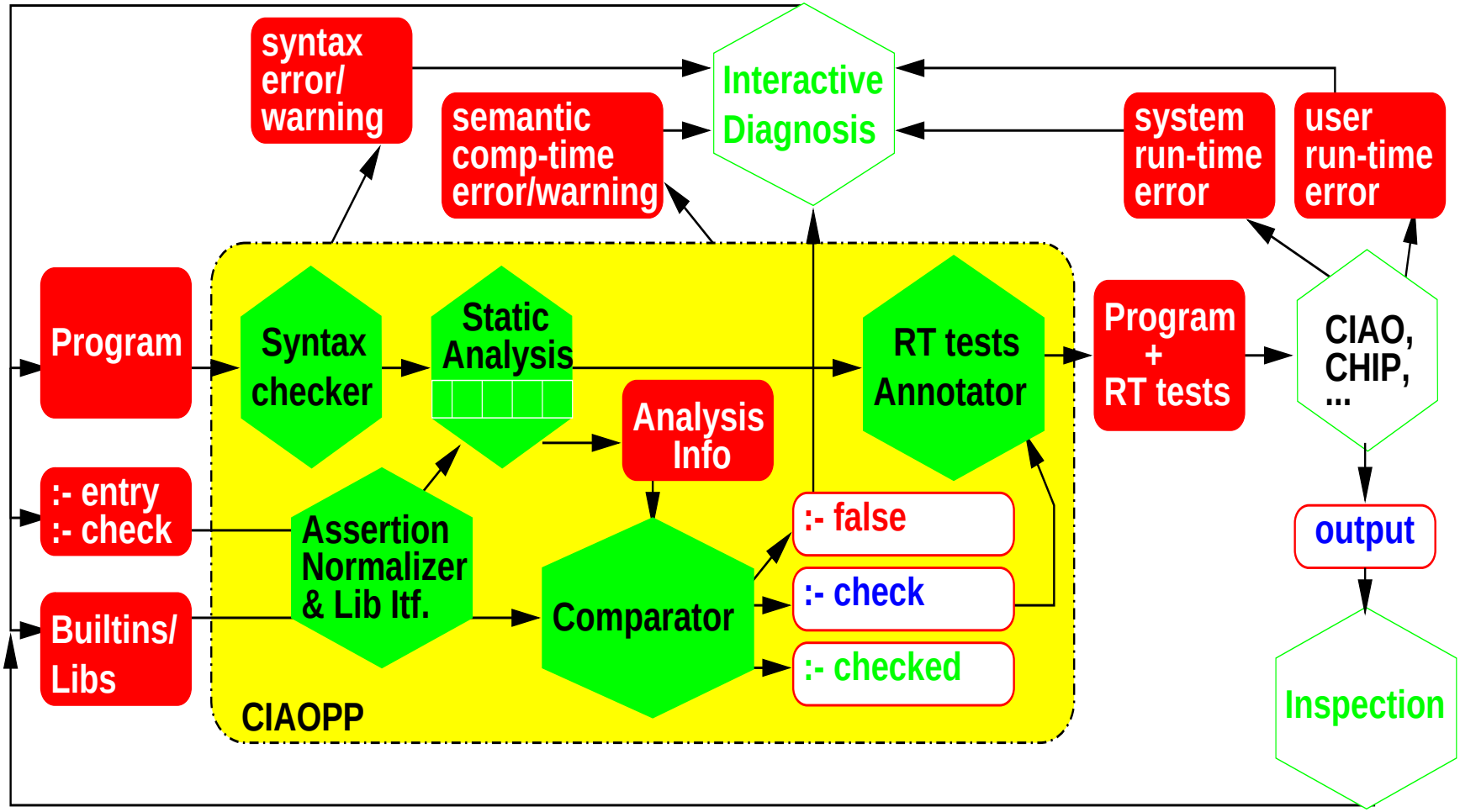
- Specification given as a semantic value $\mathcal{I}_\alpha \in D_\alpha$ and compared with $[\![P]\!]_\alpha$.

Property	**Definition**	**Sufficient condition**
P is partially correct w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^+} \subseteq \mathcal{I}_\alpha$
P is complete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \subseteq [\![P]\!]_{\alpha^-}$
P is incorrect w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \not\subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^-} \not\subseteq \mathcal{I}_\alpha$, or $[\![P]\!]_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset \wedge [\![P]\!]_\alpha \neq \emptyset$
P is incomplete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \not\subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \not\subseteq [\![P]\!]_{\alpha^+}$

($[\![P]\!]_{\alpha^+}$ 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.tn. $\mathcal{I}$.
  - 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



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,V).
```

  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,...):

"Traditional" Types	**Properties**
Compulsory (do not allow "any")	Optional (allow "any")
Expressed in a Special Language	Expressed in the Source Language
Limited Property Language	Much More General Property Language
Limit Programming Language	Do not Limit Programming Language
Untypable Programs Rejected	Run-time Checks Introduced
(Almost) Decidable	Approximated
"check"	"check" or "trust"

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

- Key issues:

Approximation	Suitable assertion language
Abstract Interpretation	Relating aproximations of actual and intended semantics

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) ),
     ...
```

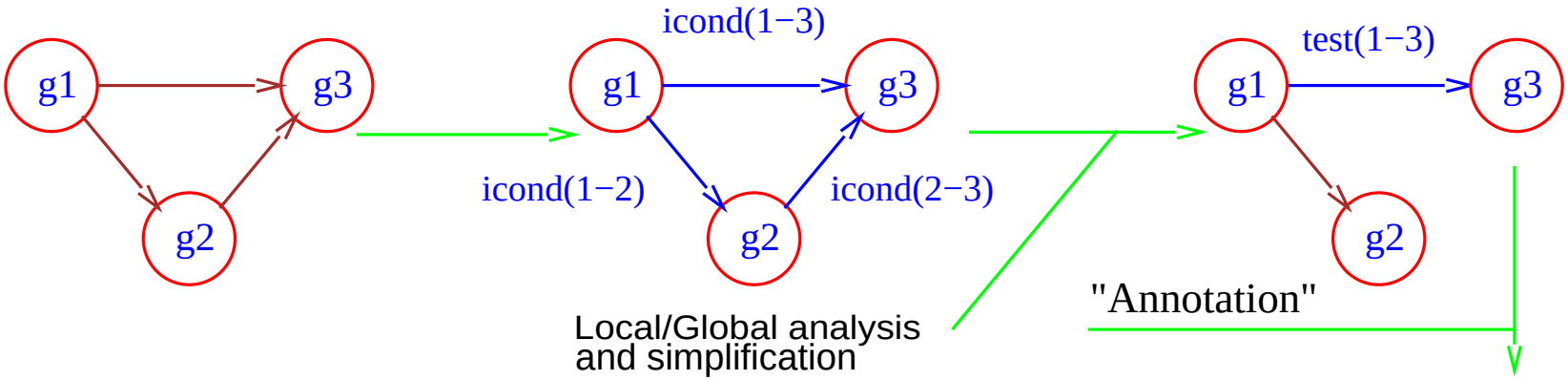
  the `ground(X)` can be *abstractly executed* to `true`
  and the whole conditional to `write(A)`.

- 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]` $\longrightarrow$ `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:

$g_1(...)$, $g_2(...)$, $g_3(...)$



(test(1−3) −> (g1, g2) & g3 ; g1, (g2 & g3))

Alternative: g1, (g2 & g3)

Automatic Program Parallelization (Contd.)

- *Example:*

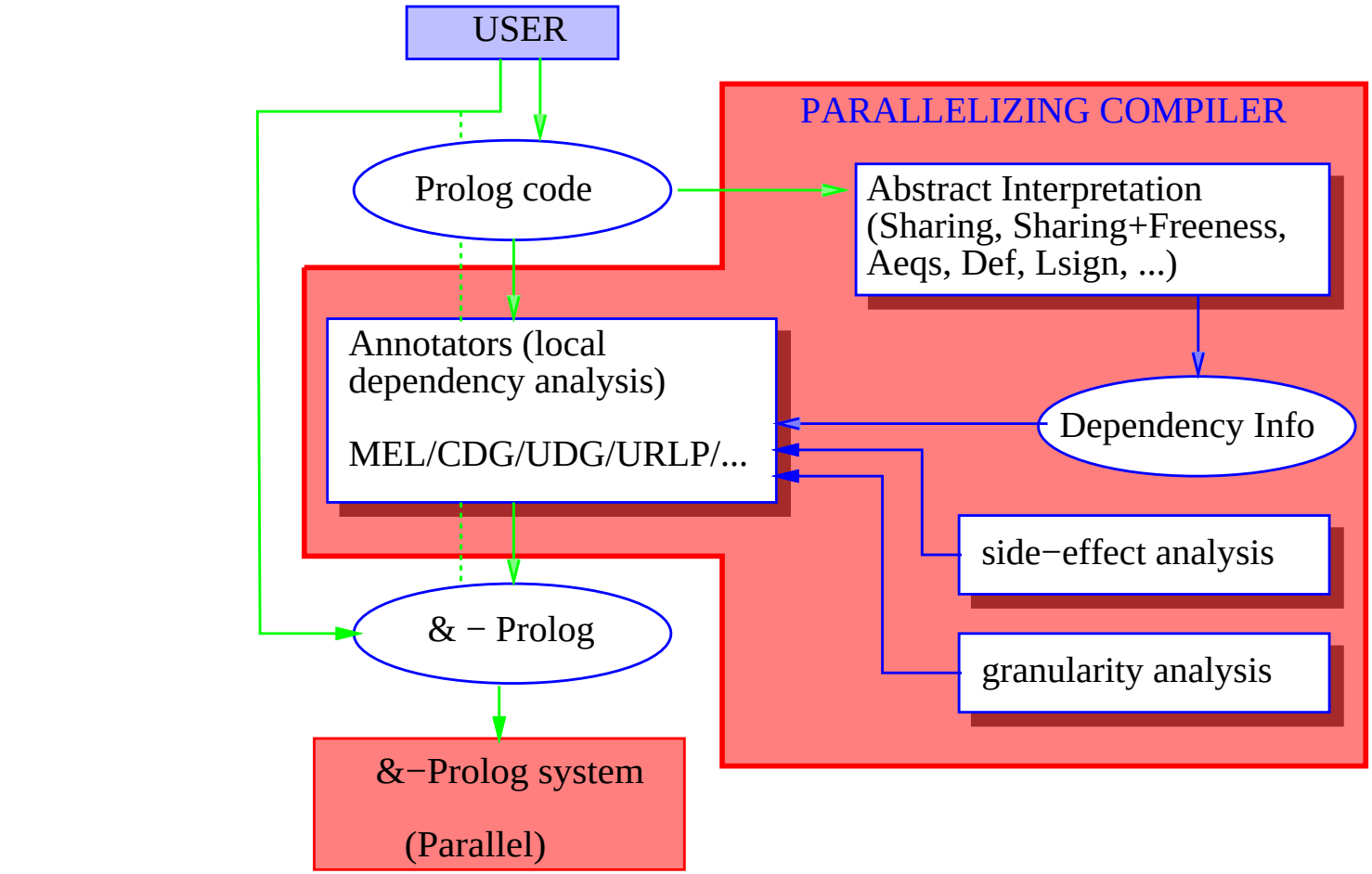
```
qs([X|L],R) :- part(L,X,L1,L2),
               qs(L2,R2), qs(L1,R1),
               app(R1,[X|R2],R).
```

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



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):

```
..., q(X,Y) & r(X), ...
```

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

```
..., length(X,LX), Cost is LX*2+1, ( Cost > 4 -> q(X,Y) & r(Z)
                                                ;  q(X,y),  r(X) ), ...
```

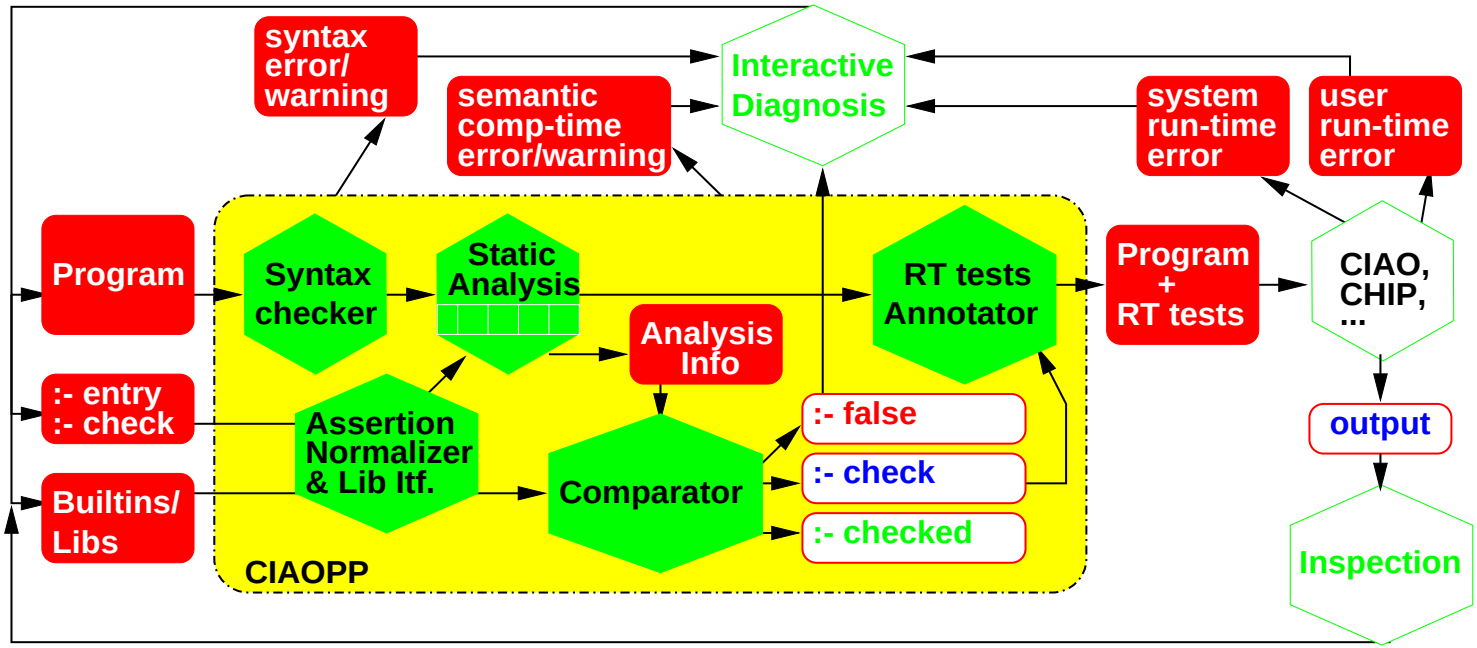
Granularity Control System Output

```
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.

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`
  - US Mirror: `http://www.cs.nmsu.edu/~clip/`... (in construction).
  - 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

[1] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[2] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.

[3] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[4] S.K. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

[5] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[6] M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 Int'l. Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[7] M. Hermenegildo and The CLIP Group. An Automatic Documentation Generator for (C)LP – Reference Manual. The Ciao System Documentation Series–TR CLIP5/97.2, Facultad de Informática, UPM, August 1997.

[8] M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifi cations, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[9] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 797–811. MIT Press, June 1995.

[10] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

[11] P. López-García and M. Hermenegildo. Effi cient Term Size Computation for Granularity Control. In *International Conference on Logic Programming*, pages 647–661, Cambridge, MA, June 1995. MIT Press, Cambridge, MA.

[12] P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.

[13] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997. Available from `ftp://clip.dia.fi.upm.es/pub/papers/assert_lang_tr_discipldeliv.ps.gz` as technical report CLIP2/97.1.

[14] G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.

[15] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[16] G. Puebla and M. Hermenegildo. Abstract Specialization and its Application to Program Parallelization. In J. Gallagher, editor, *Logic Program Synthesis and Transformation*, number 1207 in LNCS, pages 169–186. Springer-Verlag, 1997.

[17] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.