An Overview of Ciao

M. Hermenegildo[1,2]

F. Bueno[2] M. Carro[1,2] P. López-García[4,1] E. Mera[3]
J. Morales[1] G. Puebla[2] R. Haemmerlé[2]

[1]IMDEA Software Institute
[2]Technical University of Madrid (UPM)
[3]Complutense University of Madrid – UCM, Spain
[4]Spanish Research Council (CSIC), Spain

RuleML – July 19, 2011

Introduction

Objective:

- Design best possible programming language and environment, for developing challenging (semantic `:-)`) applications rapidly.

Motivating context:

- "Heroic" programming: changes, adaptation, "STOP," ...

Approach:

- Start from a small, but very *extensible* (LP-based) kernel –a language building language.
- Build gradually extensions on top of it.
- Support Prolog (as a library) but *go well beyond it*.
- Incorporate the *most useful features* from other prog. paradigms.
- Offer the *best of the dynamic and static* language approaches.
  - Provide the flexibility of dynamic languages, but with
  - *Guaranteed safety, reliability, and efficiency.*
  - Attaining *high performance* through optimization.
- Support the programmer with a *great environment*.

A Modular Language Building Language

Ciao makes it very easy to build *syntactic and semantic extensions* in a flexible and scalable way.

- Addresses shortcomings of traditional Prolog `expand_term`, etc.:
  - Expansions defined for *semantic* points: goals, terms, heads, bodies, ... (not just a global `expand_term`) → *much easier coding.*
  - All operators, expansions, flags, etc. are *module-local*.
  - Dynamic and static code clearly separated, e.g.:
    - Syntax expansion code does not necessarily end up in executables.
    - Program syntax does not necessarily affect what is read.
  - Mechanisms for defining compositions of extensions.
  - New types of operators
  - Higher-order syntax (e.g., `X(a)`), ...

→ Any extensions can be *activated* or *deactivated on per-module basis*.

→ The concept of *packages*.

A Modular Language Building Language (Contd.)

Fundamental enabler –Ciao's module/class system.

Allows also:

- Modular program devel., separate/incremental compilation.
- Modular (scalable) global analysis for detecting errors and optimizing.
- Also, building small, fast executables and embeddability (non-needed parts of the language and libraries are not included).

- All these mechanisms are easily accessible to the programmer for building extensions, restrictions (language subsets), DSLs, etc.

- Ciao is itself built in layers over a small (LP-based) *kernel*.
  - Built-ins are *in libraries* (and can be redefined or not loaded).
  - Same with all language features (loops, conditionals, functions, ',' ...).

Logic Programming

Is it still a Prolog system?

- Yes, indistinguishable to the naked eye!
  (Even won this year's Prolog programming competition! :-))
- As ISO-Prolog compliant as other popular Prologs.
- Quite compatible with de-facto standards (e.g., SICStus).
- Standard predicates, libraries, etc.

However, inside:

- No "builtins:" Prolog support is in libraries, which *can be unloaded*.
- All Prolog libraries loaded automatically for Prolog programs.

- This allows having, e.g., *pure LP* modules (no cut, no assert, ...).
- Also, other computation rules: breadth-first, iterative-deepening, Andorra, *tabling*, *fuzzy* rules, ASP, etc.

All through packages, loadable on a per-module basis.

Supporting the Best Features of Other Paradigms

Multiparadigm:

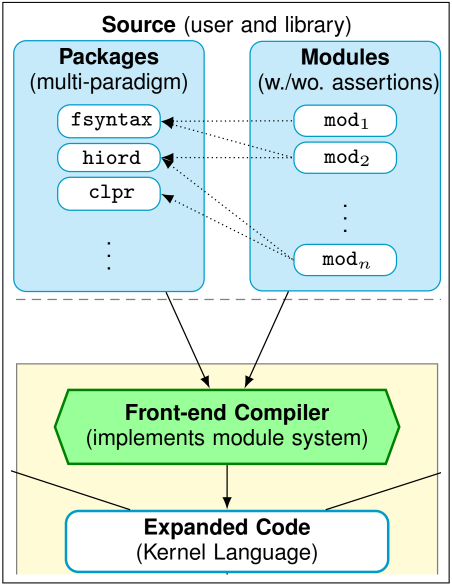
- *Constraint programming:* clpr, clpq, Leuven CHR, fd, ...
- *Functional programming:*
  - Function definitions, function calls, functional syntax for predicates.
  - *Higher-order* and *lazyness* for functions and predicates.
- *Objects:* a naturally embedded notion of classes and objects.
- *Concurrency, parallelism, distributed execution*.
- *Imperative features*: mutables, assignment, loops, cases, arrays, etc.

+ many other packages:

- Records, named argument positions.
- Logical interface to databases. Persistence.

...

Ciao Overview: Language Extensions



Dynamic vs. Static — An almost religious argument!

Dynamic languages (Prolog, Scheme, Python, Javascript, ...)

- Dynamic checking of types (and many other properties):
  - `..., A is B+C, ...`
    `B` and `C` checked to be `numexpr` by `is/2` at run time.
  - `..., arg(N,T,A), ...`
    `N` checked to be `nat` & $\leq$ `arity(T)` by `arg/3` (array bounds).
- Need to use tags (*boxing* of data) to identify type, var/nonvar, etc.
- Flexibility, compactness, rapid prototyping, scripting, ...

Static languages (ML, Haskell, Mercury, Java, ...)

- Compiler checks statically *types*.
- No dynamic checks needed for types.
- Safety guarantees (types), scalability, performance, large systems, ...

- Some languages (e.g., C) are neither:
  no checking of, e.g., array bounds at compile time or run time...

Solving the Dynamic vs. Static Dilemma

The Ciao Approach:

- Provide the flexibility of dynamic languages, but with
- *Guaranteed safety, reliability, and efficiency.*

- Use of *voluntary assertions* to express desired properties (incl. types).
  - Can be added up front, gradually, or not at all.
- Use of *advanced program analysis* (abstract interpretation) for:
  - Guaranteeing the properties as much as possible at compile-time.
  - Achieving high performance:
    - Eliminating run time checks at compile time.
    - Unboxing.
    - Specialization, slicing, ...
    - Automatic parallelization.

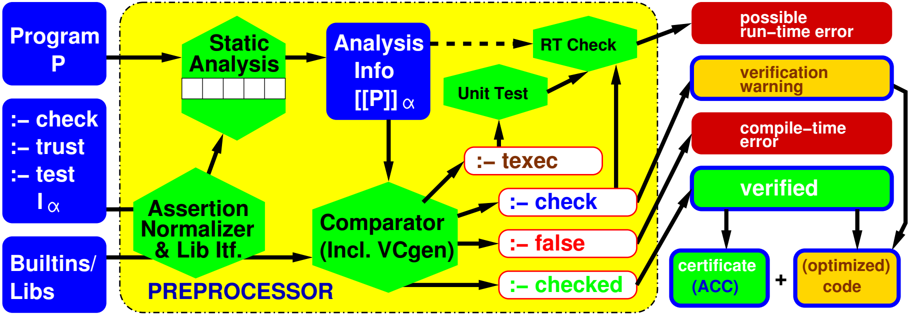
- Integrated Approach to Specification, Debugging, Verification, Testing, and Optimization.

Solving the Dynamic vs. Static Dilemma (Contd.)

Other aspects:

- Code can be interpreted or compiled. Scripting supported.
  But also separate compilation, global analysis.
- Code can be added or modified dynamically
  (but has to be marked as 'dynamic').
- Full reflection and meta-programming (but need to be declared).
- Interactive top level, embeddable source debugger.
  But compiler also creates small executables for small programs.
- Executables can be static, dynamic, or lazy load.

Integrated Approach to Specification, Debugging, Verification, Testing, and Optimization



The Assertion Language

Assertions:

:- pred Pred [:Precond] [=> Postcond] [+ Comp-formula] .

Example:

```
:- pred quicksort(X,Y) : list(int) * var => sorted(Y) + (is_det,not_fails).
:- pred quicksort(X,Y) : var * list(int) => ground(X) + non_det.
```

- Optional, can be added at any time. Provide partial specification.
- Describe calls, success, and computational behavior/invariants.
- Each `pred` typically describes a "mode" of use; the set *covers all valid calls*.
- System makes it worthwhile for the programmer to use them: e.g., autodoc.

Inst vs. Compat:

- The **:** and => fileds describe *instantiation states* by default.
- Specifying "compatibility:"

```
:- pred quicksort/2 :: list(int) * list(int).
```

The Assertion Language (Contd.)

Properties:

```
:- regtype color := green | blue | red.
:- regtype list := [] | [_|list].
:- regtype list(X) := [] | [ X|list].        ≡   list(_,[]). list(X,[H|T]) :- X(H), list(X,T).
:- prop sorted := [] | [ _ ] | [X,Y|Z] :- X > Y, sorted([Y|Z]).
```

- Arbitrary predicates (but conditions on them: termination, steadfastness, ...)
- Many predefined in libs, some of them "native" to an analyzer.
  Can also be user-defined.
- Should be visible/imported and "runnable:" used also as run-time tests!
- *Types/shapes* are a special case of property (e.g., `regtypes`).
- But also, e.g., data sizes, instantiation states, aliasing, termination, determinacy, non-failure, time, memory, ...

The Assertion Language (Contd.)

Modes (essentially "property macros"):

```
:- pred qs(+,-).             ≡  :- pred qs(X,Y) :  (nonvar(X), var(Y)).
:- pred qs(?list,?list).     ≡  :- pred qs(X,Y) :: (list(X), list(Y)).
:- pred qs(+list,-list).     ≡  :- pred qs(X,Y) :  (list(X), var(Y)) => list(Y).
```

In fact, they are defined as macros:

```
:- modedef +(A) : nonvar(A).        :- modedef +(A,X) : X(A).
:- modedef -(A) : var(A).           :- modedef -(A,X) : var(A) => X(A).
```

Can include comments:

```
:- pred qs(+list,-list)   #   "Sorts."
:- pred qs(-list,+list)   #   "Generates permutations."
```

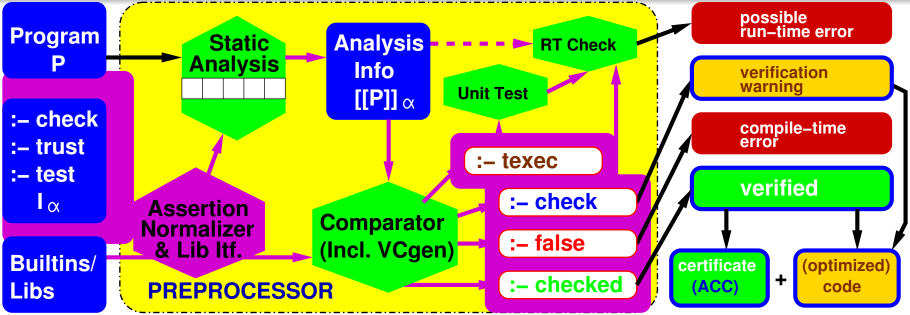
Program-point Assertions:

- Inlined with code: `..., check(( int(X), X>0 )), ....`

Assertion Status (so far "to be checked" – **check** status – default):

- Other: **trust** (guide analyzer), **true**/**false** (analysis output), **test**, etc.

The Assertion Language (Contd.)

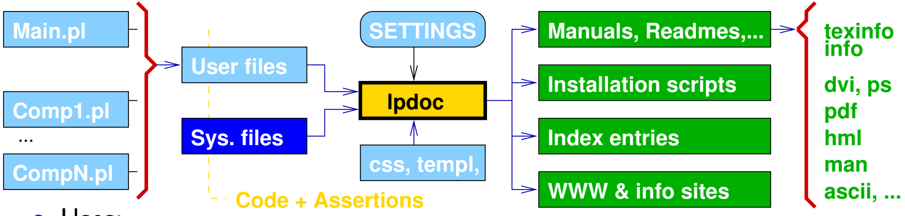


- Used everywhere, for many purposes!

- Simplest applications:
  - Generation of run-time tests.
  - Auto-documentation.

- Simple to extend also to testing.

Autodocumenter: LPdoc



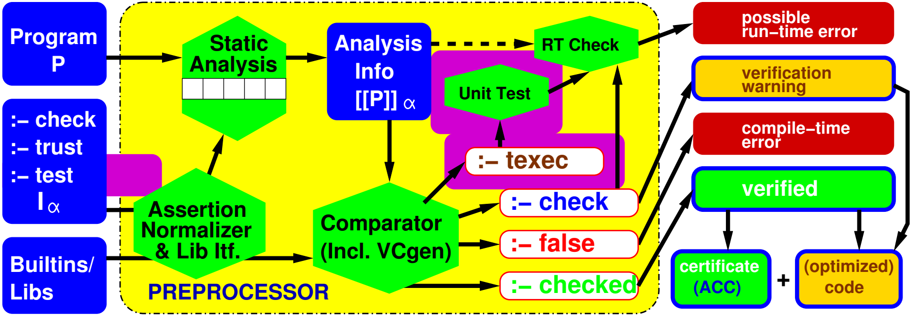
- Uses:
  - All the information that the compiler has.
  - Assertions.
  - Doc declarations (or active commens):

```
:- doc(title,"Complex numbers library").
:- doc(summary,"Provides an ADT for complex numbers.").

%! title:   Complex numbers library
%! summary: Provides an ADT for complex numbers
```

- Markup language, close to LaTeX/texinfo.
  With indices, references, figures, ...

Assertion-based Testing



Assertion-based Testing

Assertion schema used:

```
:- test Pred[:Precond][=>Postcond][+CompExecProps].
```

Such test assertions translate into:

What needs to be checked (normal assertions):

```
:- check pred Pred [:Precond] [=>Postcond] [+CompProps].
```

What test case needs to be run (test driver):

```
:-       texec   Pred [:Precond] [+Exec-Formula].
```

Many interactions within the integrated framework:

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

Verification and Error Detection using Safe Approximations

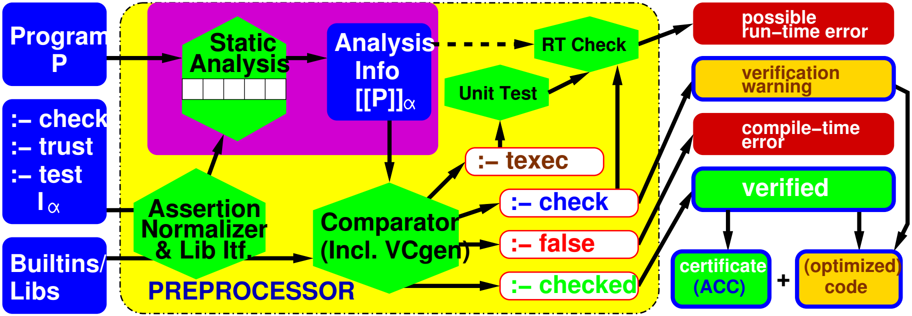
- Need to compare actual semantics $[\![P]\!]$ with intended semantics $\mathcal{I}$:

P is *partially correct* w.r.t. $\mathcal{I}$ iff	$[\![P]\!] \leq \mathcal{I}$
P is *complete* w.r.t. $\mathcal{I}$ iff	$\mathcal{I} \leq [\![P]\!]$
P is *incorrect* w.r.t. $\mathcal{I}$ iff	$[\![P]\!] \not\leq \mathcal{I}$
P is *incomplete* w.r.t. $\mathcal{I}$ iff	$\mathcal{I} \not\leq [\![P]\!]$

  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 $[\![P]\!]$ → i.e., $[\![P]\!]_{\alpha^+}$ or $[\![P]\!]_{\alpha^-}$
- Specially attractive if compiler computes (most of) $[\![P]\!]_{\alpha^+}$ anyway.

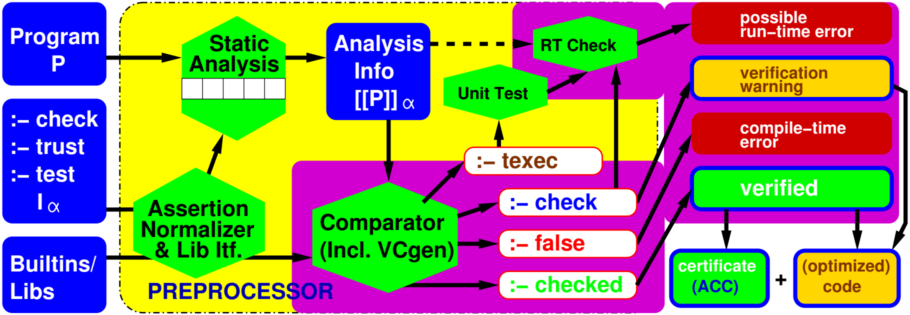
	Definition	Sufficient condition
P is prt. correct w.r.t. $\mathcal{I}_\alpha$ if	$\alpha([\![P]\!]) \leq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^+} \leq \mathcal{I}_\alpha$
P is complete w.r.t. $\mathcal{I}_\alpha$ if	$\mathcal{I}_\alpha \leq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \leq [\![P]\!]_{\alpha^=}$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

The Analyses



- 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.

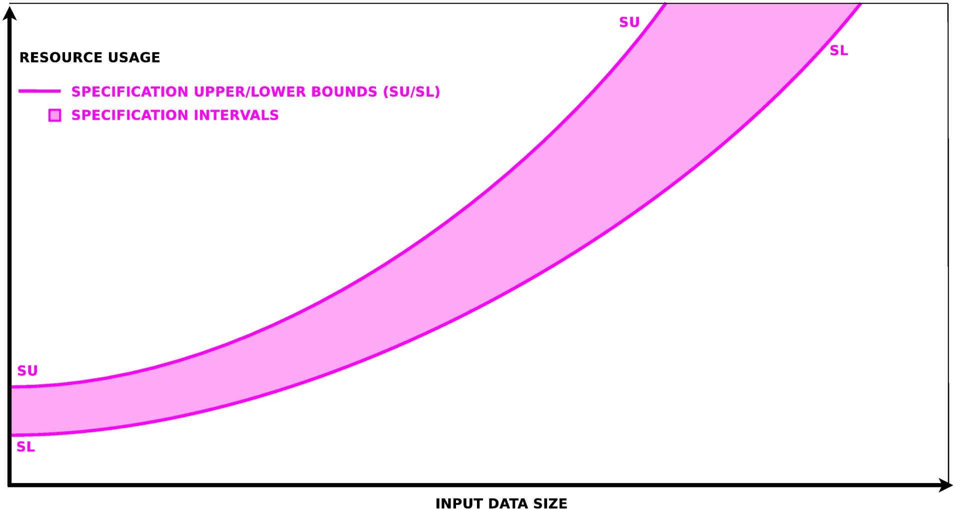
Integrated Static/Dynamic Debugging and Verification



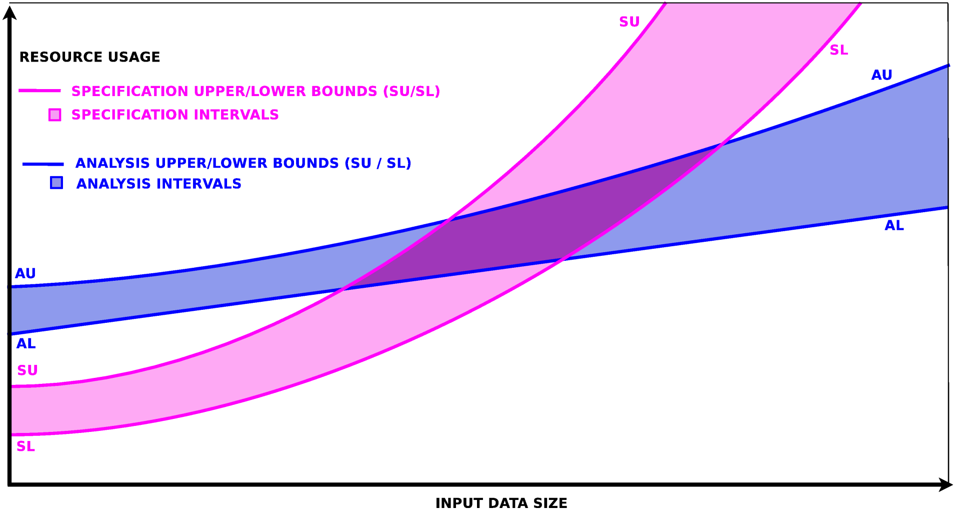
	Definition	Sufficient condition
P is prt. correct w.r.t. $\mathcal{I}_\alpha$ if	$\alpha([\![P]\!]) \leq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^+} \leq \mathcal{I}_\alpha$
P is complete w.r.t. $\mathcal{I}_\alpha$ if	$\mathcal{I}_\alpha \leq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \leq [\![P]\!]_{\alpha^=}$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

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

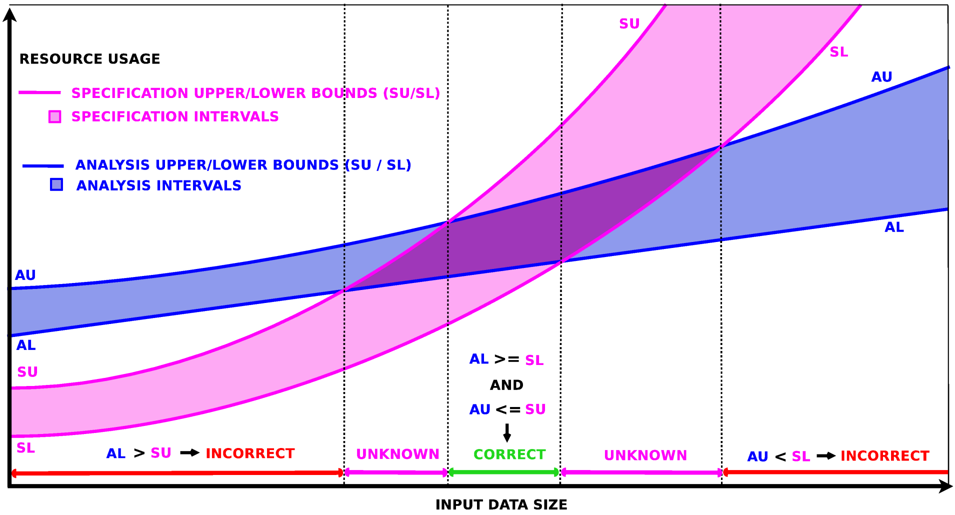
Resource Usage Verification (based on intervals)



Resource Usage Verification



Resource Usage Verification

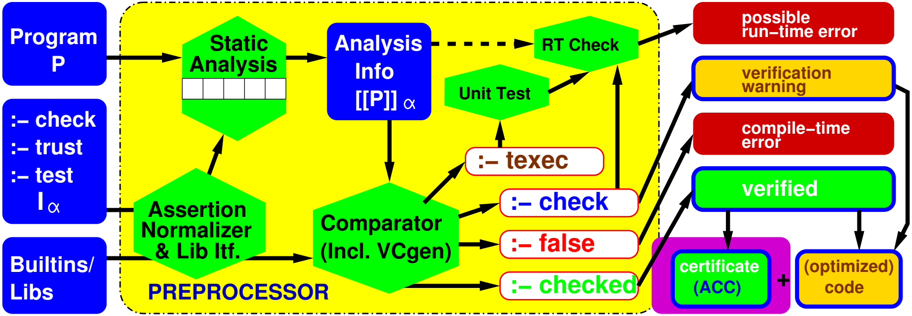


Discussion: Comparison with *Classical* Types

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

- Some key issues:
  Safe / Sound approximation — *Suitable assertion language*
  Abstract Interpretation — *Powerful abstract domains*
- Works best when properties and assertions can be expressed in the source language (i.e., source lang. supports *predicates*, *constraints*).

Abstraction-based Certification, Abstraction-Carrying Code



PRODUCER

$[\![P]\!]_\alpha$ = Analysis = **lfp**(*analysis_step*)

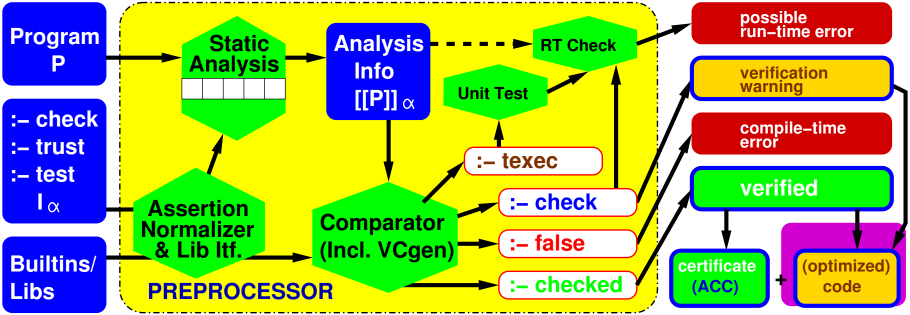
Certificate $\subset [\![P]\!]_\alpha$
Certificate $\rightarrow$
Safety Policy

CONSUMER

Checker = *analysis_step*

- Interesting extensions: reduced certificates, incrementality, ...

Optimization

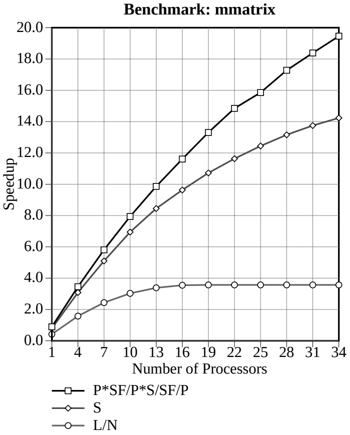


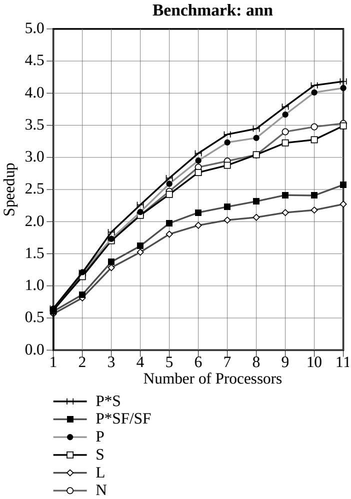
- Source-level optimizations:
  - Partial evaluation, (multiple) (abstract) specialization, ...
- Low-level optimizations:
  - Dynamic check elimination, unboxing.
  - Use of specialized instructions.
  - Optimized native code generation.

→ obtaining close-to-C performance for dynamic languages.

- Parallelization. Granularity control.

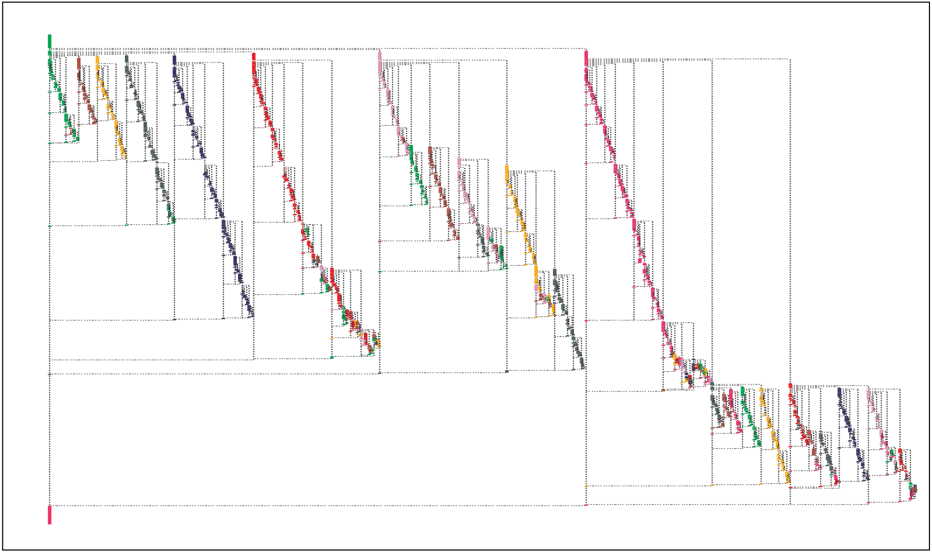
Some Speedups (Using Different Abstract Domains)



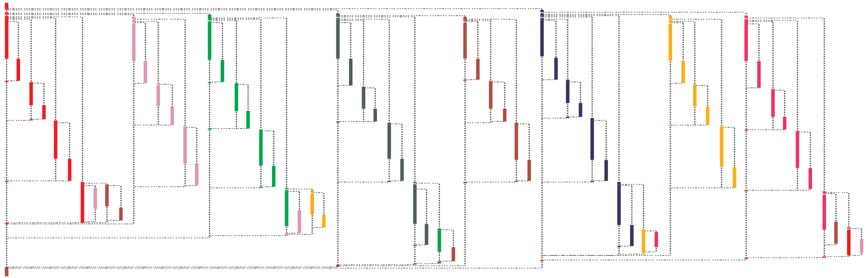


(ann: parallelizer parallelizing itself; 1-10 proc.: actual speedups on Sequent Symmetry; 10+ simulator projections from execution traces)

8 processors



8 processors, with granularity control (same scale)

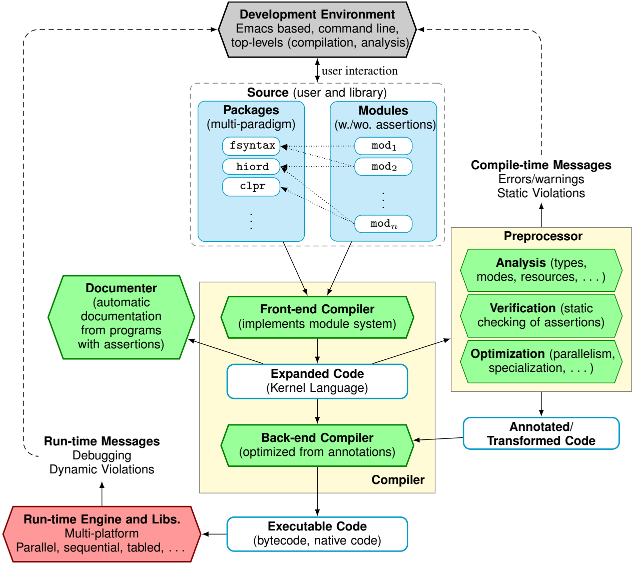


Other Relevant Ciao Features

- Extensive support for the Web:
  - PiLLoW, http(s), ODBC, XML, ZeroMQ, XPath, RDF, ...
- Extensive support for concurrency, reactivity:
  - Agents, condition-action rules, ...
- Recent applications to web services:
  - Sharing & resources for orchestration.
  - Interfaces, libraries, ...
- Compilation to javascript.

Very interested in collaborating with RuleML groups towards providing support for advanced RuleML needs!

Ciao Overview



Discussion

- 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, ...
- 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 style:
  - 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.

Some Members of The Ciao Forge

- Ciao is quite a distributed/collaborative effort:
  - Directly within the CLIP Group (UPM and IMDEA Software):

    M. Hermenegildo, K. Muthukumar, M. García de la Banda, F. Bueno, G. Puebla, M. Carro, D. Cabeza, P. López-G., R. Haemmerlé, J. Morales, E. Mera, J. Navas, M. Méndez, A. Casas, J. Correas, D. Trallero, C. Ochoa, P. Chico, M.T. Trigo, P. Pietrzak, C. Vaucheret, E. Albert, P. Arenas, S. Genaim, ...
  - Plus lots of contributors worldwide:

    G. Gupta (UT Dallas), E. Pontelli (NM State University), P. Stuckey and M. García de la Banda (Melbourne U.), K. Marriott (Monash U.), M. Bruynooghe, A. Mulkers, G. Janssens, and V. Dumortier (K.U. Leuven), S. Debray (U. of Arizona), J. Maluzynski and W. Drabent, (Linkoping U.), P. Deransart (INRIA), J. Gallagher (Roskilde University), C. Holzbauer (Austrian Research Institute for AI), M. Codish (Beer-Sheva), SICS, ...

Downloading, etc.

http://www.ciaohome.org

Provides access to:

- Latest Ciao, CiaoPP, LPdoc, etc.
- Development versions.
- Documentation.
- Mailing lists.
- etc.

Please contact us for SVN access.

Around 1,000,000 lines of (mostly Prolog) code.

Mostly **LGPL** (some packages have some variations).

All papers available on line at: http://clip.dia.fi.upm.es/clippubsbyyear and http://clip.dia.fi.upm.es/clippubsbytopic

System manual

[1] F. Bueno, M. Carro, R. Haemmerlé, M. Hermenegildo, P. López-García, E. Mera, J.F. Morales, and G. Puebla (Eds.).
The Ciao System. Ref. Manual (v1.14).
Technical report, School od C.S. (UPM), 2011.
Available at http://www.ciaohome.org.

Overall design and philosophy

[1] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, and J.F. Morales, and G. Puebla.
An Overview of The Ciao and its Design Philosophy.
Theory and Practice of Logic Programming. 2011, Cambridge University Press. http://arxiv.org/abs/1102.5497

[2] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla.
Some Techniques for Automated, Resource-Aware Distributed and Mobile Computing in a Multi-Paradigm Programming System.
In *Proc. of EURO–PAR 2004*, number 3149 in LNCS, pages 21–37. Springer-Verlag, August 2004.

[3] D. Cabeza.
An Extensible, Global Analysis Friendly Logic Programming System.
PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 2004.

[4] 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.

[5] M. Hermenegildo and The CLIP Group.
Some Methodological Issues in the Design of Ciao - A Generic, Parallel, Concurrent Constraint System.

In *Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 123–133. Springer-Verlag, May 1994.

Functions, higher order, lazyness

[1] A. Casas, D. Cabeza, and M. Hermenegildo.
A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems.
In *Eighth International Symposium on Functional and Logic Programming (FLOPS'06)*, Fuji Susono (Japan), April 2006.

[2] D. Cabeza, M. Hermenegildo, and J. Lipton.
Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction.
In *Ninth Asian Computing Science Conference (ASIAN'04)*, number 3321 in LNCS, pages 93–108. Springer-Verlag, December 2004.

Tabling

[1] P. Chico de Guzmán, M. Carro, M. Hermenegildo, Claudio Silva, Ricardo Rocha.
An Improved Continuation Call-Based Implementation of Tabling.
10th International Symposium on Practical Aspects of Declarative Languages (PADL'08), LNCS, Vol. 4902, pages 198-213, Springer-Verlag, January 2008.

[2] Pablo Chico de Guzmán Huerta, Manuel Carro, Manuel Hermenegildo.
Towards a Complete Scheme for Tabled Execution Based on Program Transformation.
11th International Symposium on Practical Aspects of Declarative Languages (PADL'09), LNCS, Num. 5418, pages 224-238, Springer-Verlag, January 2009.

Objects

[1] A. Pineda and F. Bueno.
The O'Ciao Approach to Object Oriented Logic Programming.
In *Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop)*, Copenhagen, July 2002.

[2] M. Carro and M. Hermenegildo.
A simple approach to distributed objects in prolog.

In *Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop)*, Copenhagen, July 2002.

Auto-documenter

[1] M. Hermenegildo.
A Documentation Generator for (C)LP Systems.
In *Int'l. Conf. on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345–1361. Springer-Verlag, July 2000.

Asbtract machine and low-level optimization

[1] A. Casas, M. Carro, M. Hermenegildo.
A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism.
24th International Conference on Logic Programming (ICLP'08), LNCS, Vol. 5366, pages 651-666, Springer-Verlag, December 2008.

[2] M. Carro, J. Morales, H.L. Muller, G. Puebla, and M. Hermenegildo.
High-Level Languages for Small Devices: A Case Study.
In Krisztian Flautner and Taewhan Kim, editors, *Compilers, Architecture, and Synthesis for Embedded Systems*, pages 271–281. ACM Press / Sheridan, October 2006.

[3] J. Morales, M. Carro, G. Puebla, and M. Hermenegildo.
A generator of efficient abstract machine implementations and its application to emulator minimization.
In P. Meseguer and J. Larrosa, editors, *International Conference on Logic Programming*, number 3668 in LNCS, pages 21–36. Springer Verlag, October 2005.

[4] J.F. Morales, M. Carro, and M. Hermenegildo.
Towards Description and Optimization of Abstract Machines in an Extension of Prolog.
In *Logic-Based Program Synthesis and Transformation (LOPSTR'06)*, number 4407 in LNCS, pages 77–93, July 2007.

[5] J. Morales, M. Carro, M. Hermenegildo.
Comparing Tag Scheme Variations Using an Abstract Machine Generator.
10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08), pages 32-43, ACM Press, July 2008.

[6] J. Morales, M. Carro, and M. Hermenegildo.
Improving the Compilation of Prolog to C Using Moded Types and Determinism Information.
In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, number 3057 in LNCS, pages 86–103, Heidelberg, Germany, June 2004. Springer-Verlag.

Automatic parallelization

[1] D. Cabeza, M. Hermenegildo.
Non-Strict Independence-Based Program Parallelization Using Sharing and Freeness Information.
Theoretical Computer Science, Vol. 46, Num. 410, pages 4704-4723, Elsevier Science, October 2009.

[2] M. Hermenegildo, F. Bueno, A. Casas, J. Navas, E. Mera, M. Carro, and P. López-García.
Automatic Granularity-Aware Parallelization of Programs with Predicates, Functions, and Constraints.
In *DAMP'07, ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, January 2007.

[3] A. Casas, M. Carro, and M. Hermenegildo.
Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs.
In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, The Technical University of Denmark, August 2007. Springer-Verlag.

[4] M. Hermenegildo.
Automatic Parallelization of Irregular and Pointer-Based Computations: Perspectives from Logic and Constraint Programming.
In *Proceedings of EUROPAR'97*, volume 1300 of *LNCS*, pages 31–46. Springer-Verlag, August 1997. Invited.

[5] 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.

[6] K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo.
Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism.
Journal of Logic Programming, 38(2):165–218, February 1999.

[7] D. Cabeza and M. Hermenegildo.
Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information.
In *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994. Springer-Verlag.

[8] M. Hermenegildo and K. Greene.
The &-Prolog System: Exploiting Independent And-Parallelism.
New Generation Computing, 9(3,4):233–257, 1991.

Cost analysis and granularity control in parallelism

[1] S. K. Debray, N.-W. Lin, and M. Hermenegildo.
Task Granularity Analysis in Logic Programs.
In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[2] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin.
Estimating the Computational Cost of Logic Programs.
In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[3] 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, 21(4–6):715–734, 1996.

[4] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo.
Combining Static Analysis and Profiling for Estimating Execution Times.
In *Ninth International Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 140–154. Springer-Verlag, January 2007.

[5] J. Navas, E. Mera, P. López-García, and M. Hermenegildo.
User-Definable Resource Bounds Analysis for Logic Programs.
In *23rd International Conference on Logic Programming (ICLP 2007)*, LNCS. Springer-Verlag, September 2007.

[6] E. Mera, P. López-García, M. Carro, M. Hermenegildo.
Towards Execution Time Estimation in Abstract Machine-Based Languages.
10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08), pages 174-184, ACM Press, July 2008.

The overall program development framework (CiaoPP)

[1] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla.
On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs.
In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

[2] M. Hermenegildo, G. Puebla, and F. Bueno.
Using Global Analysis, Partial Specifications, 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.

[3] G. Puebla, F. Bueno, and M. Hermenegildo.
Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs.
In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.

[4] 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.

[5] 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.

[6] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García.
Abstract Verification and Debugging of Constraint Logic Programs.
In *Recent Advances in Constraints*, number 2627 in LNCS, pages 1–14. Springer-Verlag, January 2003.

[7] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García.
Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). Invited talk.
In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

[8] E. Mera, P. López-García, M. Hermenegildo.
Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework.
em International Conference on Logic Programming (ICLP), LNCS, Num. 5649, pages 281-295, Springer-Verlag, July 2009.

Abstraction carrying code

[1] E. Albert, G. Puebla, and M. Hermenegildo.
Abstraction-Carrying Code.
In *Proc. of LPAR'04*, number 3452 in LNAI, pages 380–397. Springer-Verlag, 2005.

[2] E. Albert, G. Puebla, and M. Hermenegildo.
An Abstract Interpretation-based Approach to Mobile Code Safety.
In *Proc. of Compiler Optimization meets Compiler Verification (COCV'04)*, Electronic Notes in Theoretical Computer Science 132(1), pages 113–129. Elsevier - North Holland, April 2004.

[3] E. Albert, P. Arenas, G. Puebla, and M. Hermenegildo.
Reduced Certificates for Abstraction-Carrying Code.
In *22nd International Conference on Logic Programming (ICLP 2006)*, number 4079 in LNCS, pages 163–178. Springer-Verlag, August 2006.

[4] E. Albert, P. Arenas, and G. Puebla.
An Incremental Approach to Abstraction-Carrying Code.
In *13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'06)*, number 4246 in LNCS, pages 377–391. Springer-Verlag, November 2006.

Partial evaluation

[1] G. Puebla and M. Hermenegildo.
Abstract Specialization and its Applications.
In *ACM Partial Evaluation and Semantics based Program Manipulation (PEPM'03)*, pages 29–43. ACM Press, June 2003. Invited.

[2] E. Albert, G. Puebla, and J. Gallagher.
Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates.

In *15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, number 3901 in LNCS, pages 115–132. Springer-Verlag, April 2006.

[3] G. Puebla and C. Ochoa.
Poly-Controlled Partial Evaluation.
In *Proc. of 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, pages 261–271. ACM Press, July 2006.

[4] G. Puebla, E. Albert, and M. Hermenegildo.
Abstract Interpretation with Specialized Definitions.
In *The 13th International Static Analysis Symposium (SAS'06)*, number 4134 in LNCS, pages 107–126. Springer, August 2006.

[5] 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.

Scalability, modularity of analysis, debugging, and verification

[1] P. Pietrzak, J. Correas, G. Puebla, and M. Hermenegildo.
Context-Sensitive Multivariant Assertion Checking in Modular Programs.
In *13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'06)*, number 4246 in LNCS, pages 392–406. Springer-Verlag, November 2006.

[2] F. Bueno, M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey.
A Model for Inter-module Analysis and Optimizing Compilation.
In *Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, March 2001.

[3] G. Puebla, J. Correas, M. Hermenegildo, F. Bueno, M. García de la Banda, K. Marriott, and P. J. Stuckey.
A Generic Framework for Context-Sensitive Analysis of Modular Programs.
In M. Bruynooghe and K. Lau, editors, *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, number 3049 in LNCS, pages 234–261. Springer-Verlag, Heidelberg, Germany, August 2004.

[4] P. Pietrzak, J. Correas, G. Puebla, M. Hermenegildo.

A Practical Type Analysis for Verification of Modular Prolog Programs.
ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation (PEPM'08), pages 61-70, ACM Press, January 2008.

Some applications of the CiaoPP framework to Java bytecode

[1] M. Méndez-Lojo, M. Hermenegildo.
Precise Set Sharing Analysis for Java-style Programs.
9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08), LNCS, Num. 4905, pages 172-187, Springer-Verlag, January 2008.

[2] M. Méndez-Lojo, J. Navas, and M. Hermenegildo.
A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs.
In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, August 2007.

[3] J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
An Efficient, Context and Path Sensitive Analysis Framework for Java Programs.
In *9th Workshop on Formal Techniques for Java-like Programs FTfJP 2007*, July 2007.

[4] E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla.
Verification of Java Bytecode using Analysis and Transformation of Logic Programs.
In *Ninth International Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 124–139. Springer-Verlag, January 2007.