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.
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`) \(\rightarrow\) *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)`, ...)

\(\rightarrow\) Any extensions can be *activated* or *deactivated on per-module basis.*

\(\rightarrow\) The concept of *packages.*
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, ‘,’ ...).
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 “builtin” 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

Source (user and library)

Packages (multi-paradigm)
- fsyntax
- hiord
- clpr
- ...

Modules (w./wo. assertions)
- \(mod_1\)
- \(mod_2\)
- ...
- \(mod_n\)

Front-end Compiler
(implements module system)

Expanded Code
(Kernel Language)

Compile-time Messages
- Errors/warnings
- Static Violations

Run-time Messages
- Debugging
- Dynamic Violations
Solving The Dynamic vs. Static Dilemma

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 & ≤ 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...
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.

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:

\[\text{\texttt{:- pred Pred [:Precond] [\Rightarrow Postcond] [+ Comp-formula ]}.}\]

Example:

\[\text{\texttt{:- pred quicksort}(X,Y) : list(int) * var \Rightarrow sorted(Y) + (is\_det, not\_fails).}\]
\[\text{\texttt{:- pred quicksort}(X,Y) : var * list(int) \Rightarrow ground(X) + non\_det.}\]

- 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 \(\Rightarrow\) fields describe instantiation states by default.
- Specifying “compatibility:”

\[\text{\texttt{:- pred quicksort/2 :: list(int) * list(int)}.}\]
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”):

\[
\begin{align*}
\text{:- pred } & \text{qs}(+, -). \quad \equiv \quad \text{:- pred } \text{qs}(X, Y) : (\text{nonvar}(X), \text{var}(Y)). \\
\text{:- pred } & \text{qs}(?\text{list}, ?\text{list}). \quad \equiv \quad \text{:- pred } \text{qs}(X, Y) : (\text{list}(X), \text{list}(Y)). \\
\text{:- pred } & \text{qs}(+\text{list}, -\text{list}). \quad \equiv \quad \text{:- pred } \text{qs}(X, Y) : (\text{list}(X), \text{var}(Y)) \Rightarrow \text{list}(Y). 
\end{align*}
\]

In fact, they are defined as macros:

\[
\begin{align*}
\text{:- modedef } & +(A) : \text{nonvar}(A). \quad \text{:- modedef } +(A, X) : X(A). \\
\text{:- modedef } & -(A) : \text{var}(A). \quad \text{:- modedef } -(A, X) : \text{var}(A) \Rightarrow X(A). 
\end{align*}
\]

Can include comments:

\[
\begin{align*}
\text{:- pred } & \text{qs}(+\text{list}, -\text{list}) \quad \# \quad "\text{Sorts.}" \\
\text{:- pred } & \text{qs}(-\text{list}, +\text{list}) \quad \# \quad "\text{Generates permutations."}
\end{align*}
\]

Program-point Assertions:

- Inlined with code:
  \[
  ..., \text{check}((\text{int}(X), X>0)), ... 
  \]

Assertion Status (so far “to be checked” – check status – default):

- Other: \text{trust} (guide analyzer), \text{true}/\text{false} (analysis output), \text{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

- **Main.pl**
- **Comp1.pl**
- **CompN.pl**

**User files**
- **Sys. files**
- **SETTNGS**

- **Ipdcc**

- **Manuals, Readmes,** ...
- **Installation scripts**
- **Index entries**
- **WWW & info sites**

- **Code + Assertions**

- **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 \texttt{LaTeX}/texinfo.
  - With indices, references, figures, ...

Herme,Bue,Car,Lope,Mera,Mora,Pueb,Haem

An overview of Ciao

RuleML – July 19, 2011 16 / 35
Assertion-based Testing

Program P

:- check

:- trust

:- test

I α

Builtins/ Libs

Static Analysis

Analysis Info [[P]] α

Comparator

(INcl. VCgen)

Assertion Normalizer & Lib Itf.

RT Check

Unit Test

:- texec

:- check

:- false

:- checked

PREPROCESSOR

possible run-time error

verification warning

compile-time error

verified

certificate (ACC)

+(optimized) code
Assertion-based Testing

Assertion schema used:

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

Such test assertions translate into:

- What needs to be checked (normal assertions):
  ```prolog
  :- check pred Pred [:Precond] [=><Postcond] [+CompProps].
  ```

- What test case needs to be run (test driver):
  ```prolog
  :- 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. $\mathcal{I}$** if $[P] \leq \mathcal{I}$
- **$P$ is complete w.r.t. $\mathcal{I}$** if $\mathcal{I} \leq [P]$
- **$P$ is incorrect w.r.t. $\mathcal{I}$** if $[P] \not\leq \mathcal{I}$
- **$P$ is incomplete w.r.t. $\mathcal{I}$** if $\mathcal{I} \not\leq [P]$

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 $[P] \rightarrow$ i.e., $[P]_{\alpha^+}$ or $[P]_{\alpha^-}$

Specially attractive if compiler computes (most of) $[P]_{\alpha^+}$ anyway.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ is prt. correct w.r.t. $\mathcal{I}<em>\alpha$ if $\alpha([P]) \leq \mathcal{I}</em>\alpha$</td>
<td>$[P]<em>{\alpha^+} \leq \mathcal{I}</em>\alpha$</td>
</tr>
<tr>
<td>$P$ is complete w.r.t. $\mathcal{I}<em>\alpha$ if $\mathcal{I}</em>\alpha \leq \alpha([P])$</td>
<td>$\mathcal{I}<em>\alpha \leq [P]</em>{\alpha^=} $</td>
</tr>
<tr>
<td>$P$ is incorrect w.r.t. $\mathcal{I}<em>\alpha$ if $\alpha([P]) \not\leq \mathcal{I}</em>\alpha$</td>
<td>$[P]<em>{\alpha^=} \not\leq \mathcal{I}</em>\alpha$, or $[P]<em>{\alpha^+} \cap \mathcal{I}</em>\alpha = \emptyset \land [P]_{\alpha} \neq \emptyset$</td>
</tr>
<tr>
<td>$P$ is incomplete w.r.t. $\mathcal{I}<em>\alpha$ if $\mathcal{I}</em>\alpha \not\leq \alpha([P])$</td>
<td>$\mathcal{I}<em>\alpha \not\leq [P]</em>{\alpha^+}$</td>
</tr>
</tbody>
</table>
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.

Herme,Bue,Car,Lope,Mera,Mora,Pueb,Haem
**Integrated Static/Dynamic Debugging and Verification**

### Program P

- `:- check`
- `:- trust`
- `:- test`
- `I_α`

### Builtins/Libs

### PREPROCESSOR

- Comparator (Incl. VCgen)
- Normalizer & Lib Itf.
- Assertion Normalizer & Lib Itf.
- Static Analysis
- Analysis Info `[[P]]_α`
- RT Check
- Unit Test
- `:- check`
- `:- false`
- `:- checked`

### Possible

- run-time error
- verification warning
- compile-time error

### Verified

- (optimized) code
- certificate (ACC)

---

**Definition**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Sufficient Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>P</code> is prt. correct w.r.t. <code>I_α</code> if <code>α([[P]]) ≤ I_α</code></td>
<td><code>[[P]]_α+ ≤ I_α</code></td>
</tr>
<tr>
<td><code>P</code> is complete w.r.t. <code>I_α</code> if <code>I_α ≤ α([[P]])</code></td>
<td><code>I_α ≤ [[P]]_α=</code></td>
</tr>
<tr>
<td><code>P</code> is incorrect w.r.t. <code>I_α</code> if <code>α([[P]]) ≤ I_α</code></td>
<td><code>[[P]]_α= ≤ I_α</code>, or <code>[[P]]_α+ ∩ I_α = ∅ ∧ [[P]]_α ≠ ∅</code></td>
</tr>
<tr>
<td><code>P</code> is incomplete w.r.t. <code>I_α</code> if <code>I_α ≤ α([[P]])</code></td>
<td><code>I_α ≤ [[P]]_α+</code></td>
</tr>
</tbody>
</table>

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

Herme,Bue,Car,Lope,Mera,Mora,Pueb,Haem (IMDEA, UPM)
Resource Usage Verification (based on intervals)

RESOURCE USAGE

- SPECIFICATION UPPER/LOWER BOUNDS (SU/SL)
- SPECIFICATION INTERVALS

INPUT DATA SIZE
Resource Usage Verification

![Graph showing resource usage vs. input data size](graph.png)

- **Resource Usage**
  - Specification Upper/Lower Bounds (SU/SL)
  - Analysis Upper/Lower Bounds (SU / SL)

**Legend**
- AU: Analysis Upper Bound
- AL: Analysis Lower Bound
- SU: Specification Upper Bound
- SL: Specification Lower Bound

**X-axis:** Input Data Size
**Y-axis:** Resource Usage

Herme, Bue, Car, Lope, Mera, Mora, Pueb, Haem (IMDEA, UPM)

An overview of Ciao

Resource Usage Verification

RESOURCE USAGE

- SPECIFICATION UPPER/LOWER BOUNDS (SU/SL)
- SPECIFICATION INTERVALS

- ANALYSIS UPPER/LOWER BOUNDS (SU / SL)
- ANALYSIS INTERVALS

INPUT DATA SIZE

AL > SU → INCORRECT
AL >= SL AND AU <= SU → CORRECT
AU < SL → INCORRECT
## Discussion: Comparison with Classical Types

<table>
<thead>
<tr>
<th>“Traditional” Types</th>
<th>Ciao Assertion-based Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Properties&quot; limited by decidability</td>
<td>Much more general property language</td>
</tr>
<tr>
<td>May need to limit prog. lang.</td>
<td>No need to limit prog. lang.</td>
</tr>
<tr>
<td>“Untypable” programs rejected</td>
<td>Run-time checks introduced</td>
</tr>
<tr>
<td>(Almost) Decidable</td>
<td>Decidable + Undecidable(approximated)</td>
</tr>
<tr>
<td>Expressed in a different language</td>
<td>Expressed in the source language</td>
</tr>
<tr>
<td>Types must be defined</td>
<td>Types can be defined or inferred</td>
</tr>
<tr>
<td>Assertions are only of type “check”</td>
<td>“check”, “trust”, ...</td>
</tr>
<tr>
<td>Type signatures &amp; assertions different</td>
<td>Type signatures are assertions</td>
</tr>
</tbody>
</table>

- Some key issues:
  - *Safe / Sound approximation*
  - *Abstract Interpretation*
  - *Suitable assertion language*
  - *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

Program P

::= check
::= trust
::= test

Builtins/Libs

PREPROCESSOR

Static Analysis

Analysis Info \([P]_\alpha\)

Comparator (Incl. VCgen)

RT Check

Unit Test

Assertion Normalizer & Lib Itf.

Program P

Possible run-time error

Verification warning

Compile-time error

Verified

Certificate (ACC)

(optimized) code

Certificate \(\subset \[P]_\alpha\)

Safety Policy

\[P]_\alpha = \text{Analysis} = \text{lfp}(\text{analysis\_step})

Interesting extensions: reduced certificates, incrementality, ...

PRODUCER

CONSUMER

Certificate \(\subset \[P]_\alpha\) → Checker = analysis\_step
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)
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

Development Environment
- Emacs based, command line, top-levels (compilation, analysis)

Source (user and library)
- Packages (multi-paradigm)
  - fsyntax
  - hiord
  - clpr
  - ...
- Modules (w./wo. assertions)
  - mod_1
  - mod_2
  - ...
  - mod_n

user interaction

Compiler
- Front-end Compiler (implements module system)
- Expanded Code (Kernel Language)
- Back-end Compiler (optimized from annotations)
- Executable Code (bytecode, native code)

Preprocessor
- Analysis (types, modes, resources, ...)
- Verification (static checking of assertions)
- Optimization (parallelism, specialization, ...)
- Annotated/Transformed Code

Documenter
- (automatic documentation from programs with assertions)

Compile-time Messages
- Errors/warnings
- Static Violations

Run-time Engine and Libs.
- Multi-platform
- Parallel, sequential, tabled, ...

Run-time Messages
- Debugging
- Dynamic Violations
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):

- Plus lots of contributors worldwide:
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


Overall design and philosophy


Functions, higher order, lazyness

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.

Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction.

Tabling

An Improved Continuation Call-Based Implementation of Tabling.

Towards a Complete Scheme for Tabled Execution Based on Program Transformation.

Objects

The O’Ciao Approach to Object Oriented Logic Programming.

A simple approach to distributed objects in prolog.
Auto-documenter

A Documentation Generator for (C)LP Systems.

Abstract machine and low-level optimization

A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism.

High-Level Languages for Small Devices: A Case Study.

A generator of efficient abstract machine implementations and its application to emulator minimization.

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.

Comparing Tag Scheme Variations Using an Abstract Machine Generator.

Improving the Compilation of Prolog to C Using Modeled Types and Determinism Information.
Automatic parallelization


Cost analysis and granularity control in parallelism

Task Granularity Analysis in Logic Programs.

Estimating the Computational Cost of Logic Programs.

A Methodology for Granularity Based Control of Parallelism in Logic Programs.


User-Definable Resource Bounds Analysis for Logic Programs.

Towards Execution Time Estimation in Abstract Machine-Based Languages.
The overall program development framework (CiaoPP)


Abstraction carrying code

Abstraction-Carrying Code.


Reduced Certificates for Abstraction-Carrying Code.

An Incremental Approach to Abstraction-Carrying Code.

Partial evaluation

Abstract Specialization and its Applications.

Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates.

Poly-Controlled Partial Evaluation.

Abstract Interpretation with Specialized Definitions.

Abstract Multiple Specialization and its Application to Program Parallelization.

Scalability, modularity of analysis, debugging, and verification

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.

A Model for Inter-module Analysis and Optimizing Compilation.

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.

A Practical Type Analysis for Verification of Modular Prolog Programs.

Some applications of the CiaoPP framework to Java bytecode

Precise Set Sharing Analysis for Java-style Programs.

A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs.

An Efficient, Context and Path Sensitive Analysis Framework for Java Programs.

Verification of Java Bytecode using Analysis and Transformation of Logic Programs.