

An Overview of Ciao and its Design Philosophy

M. V. HERMENEGILDO^{1,2} F. BUENO¹ M. CARRO¹

P. LÓPEZ-GARCÍA^{2,4} E. MERA³ J. F. MORALES² G. PUEBLA¹

¹*Universidad Politécnica de Madrid (UPM)*

(e-mail: bueno@fi.upm.es, mcarro@fi.upm.es, german@fi.upm.es)

²*Madrid Institute of Advanced Studies in Software Development Technology (IMDEA Software)*

(e-mail: manuel.hermenegildo@imdea.org, pedro.lopez@imdea.org, josef.morales@imdea.org)

³*Universidad Complutense de Madrid (UCM)*

(e-mail: edison@fdi.ucm.es)

⁴*Spanish Research Council (CSIC)*

Abstract

We provide an overall description of the Ciao multiparadigm programming system emphasizing some of the novel aspects and motivations behind its design and implementation. An important aspect of Ciao is that, in addition to supporting logic programming (and, in particular, Prolog), it provides the programmer with a large number of useful features from different programming paradigms and styles, and that the use of each of these features (including those of Prolog) can be turned on and off at will for each program module. Thus, a given module may be using, e.g., higher order functions and constraints, while another module may be using assignment, predicates, Prolog meta-programming, and concurrency. Furthermore, the language is designed to be extensible in a simple and modular way. Another important aspect of Ciao is its programming environment, which provides a powerful preprocessor (with an associated assertion language) capable of statically finding non-trivial bugs, verifying that programs comply with specifications, and performing many types of optimizations (including automatic parallelization). Such optimizations produce code that is highly competitive with other dynamic languages or, with the (experimental) optimizing compiler, even that of static languages, all while retaining the flexibility and interactive development of a dynamic language. This compilation architecture supports modularity and separate compilation throughout. The environment also includes a powerful auto-documenter and a unit testing framework, both closely integrated with the assertion system. The paper provides an informal overview of the language and program development environment. It aims at illustrating the design philosophy rather than at being exhaustive, which would be impossible in a single journal paper, pointing instead to previous Ciao literature.

KEYWORDS: Prolog, Logic Programming System, Assertions, Verification, Extensible Languages.

1 Origins and Initial Motivations

Ciao (Hermenegildo et al 1994; Hermenegildo et al 1999; Bueno et al. 2009; Hermenegildo and The Ciao Development Team 2006) is a modern, multiparadigm programming language with an advanced programming environment. The ultimate motivation behind the system is to develop a combination of programming language and development tools that together help programmers produce in less time and with less effort code that has fewer or no bugs. Ciao aims at combining the flexibility of dynamic/scripting languages with the

guarantees and performance of static languages. It is designed to run very efficiently on platforms ranging from small embedded processors to powerful multicore architectures. Figure 1 shows an overview of the Ciao system architecture and the relationships among its components, which will be explained throughout the paper.

Ciao has its main roots in the &-Prolog language and system (Hermenegildo and Greene 1991). &-Prolog's design was aimed at achieving higher performance than state of the art sequential logic programming systems by exploiting parallelism, in particular, and-parallelism (Hermenegildo and Rossi 1995). This required the development of a specialized abstract machine, derived from early versions of SICStus Prolog (Swedish Institute for Computer Science 2009), capable of running a large number of (possibly non-deterministic) goals in parallel (Hermenegildo 1986; Hermenegildo and Greene 1991). The source language was also extended in order to allow expressing parallelism and concurrency in programs, and later to support constraint programming, including the concurrent and parallel execution of such programs (García de la Banda et al. 1996).

Parallelization was done either by hand or by means of the &-Prolog compiler, which was capable of automatically annotating programs for parallel execution (Muthukumar and Hermenegildo 1990; Muthukumar et al. 1999). This required developing advanced program analysis technology based on abstract interpretation (Cousot and Cousot 1977), which led to the development of the PLAI analyzer (Warren et al. 1988; Hermenegildo et al. 1992; Muthukumar and Hermenegildo 1992), based on Bruynooghe's approach (Bruynooghe 1991) but using a highly-efficient fixpoint including memo tables, convergence acceleration, dependency tracking, etc. This analyzer inferred program properties such as independence among program variables (Muthukumar and Hermenegildo 1992; Muthukumar and Hermenegildo 1991), absence of side effects, non-failure (Bueno et al. 2004), determinacy (López-García et al. 2010), data structure shape and instantiation state ("moded types") (Saglam and Gallagher 1995; Vaucheret and Bueno 2002), or upper and lower bounds on the sizes of data structures and the cost of procedures (Debray et al. 1990; Debray and Lin 1993; Debray et al. 1997). This was instrumental for performing automatic granularity control (Debray et al. 1990; López-García et al. 1996). In addition to automatic parallelization the &-Prolog compiler performed other optimizations such as multiple (abstract) specialization (Puebla and Hermenegildo 1995). Additional work was also performed to extend the system to support other computation rules, such as the Andorra principle (Warren 1993; Olmedilla et al. 1993) and other sublanguages and control rules.

In the process of gradually extending the capabilities of the &-Prolog system in the late 80's/early 90's two things became clear. Firstly, the wealth of information inferred by the analyzers would also be very useful as an aid in the program development process. This led to the idea of the Ciao assertion language and preprocessor, two fundamental components of the Ciao system (even if neither of them are strictly required for developing or compiling programs). The Ciao assertion language (Puebla et al. 2000b) provides a homogeneous framework which allows, among other things, static and dynamic verification to work cooperatively in a unified way. The Ciao Preprocessor (CiaoPP (Hermenegildo et al. 1999; Puebla et al. 2000a; Hermenegildo et al. 2005)) is a powerful tool capable of statically finding non-trivial bugs, verifying that the program complies with specifications (written in the assertion language), and performing many types of program optimizations.

A second realization was that many desirable language extensions could be supported

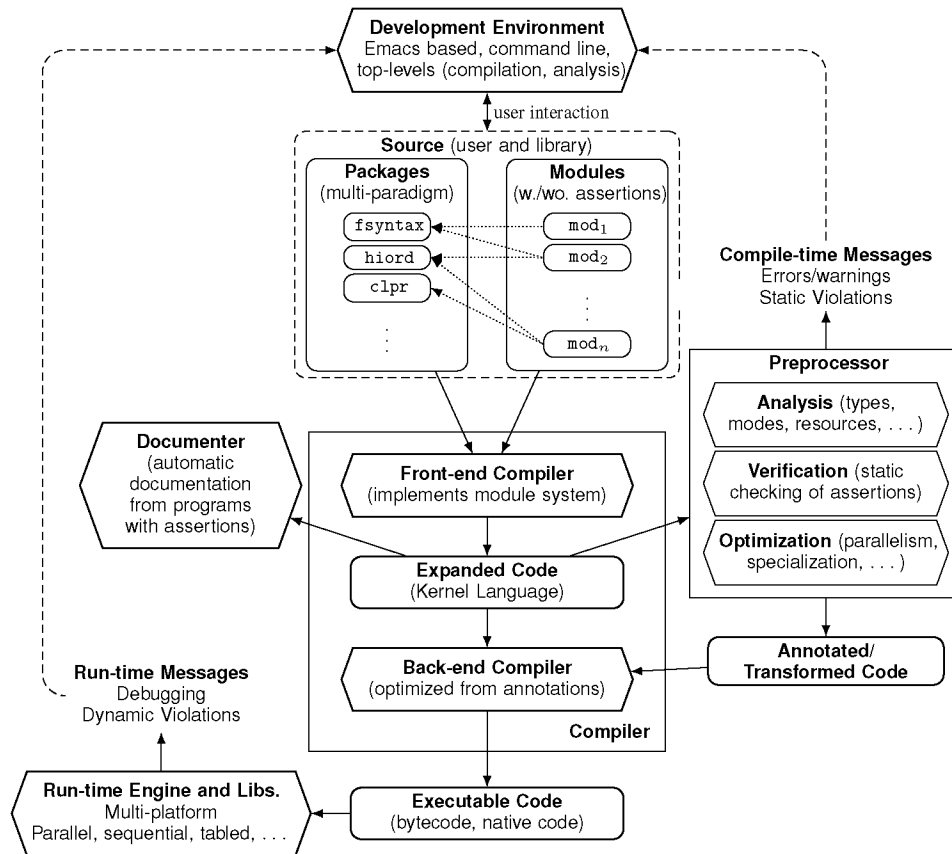


Fig. 1. A high-level view of the Ciao system.

efficiently within the same system if the underlying machinery implemented a relatively limited set of basic constructs (a *kernel language*) (Hermenegildo et al 1994; Hermenegildo et al 1999) coupled with an easily programmable and modular way of defining new syntax and giving semantics to it in terms of that kernel language. This idea is not exclusive to Ciao, but in Ciao the facilities that enable building up from a simple kernel are explicitly available from the system programmer level to the application programmer level. The need to be able to define extensions based on some basic blocks led to the development of a novel module system (Cabeza and Hermenegildo 2000a) which allows writing language extensions (*packages*) by grouping together syntactic definitions, compilation options, and plugins to the compiler. The mechanisms provided for adding new syntax to the language and giving semantics to such syntax can be activated or deactivated on a per-compilation unit basis without interfering with other units. As a result all Ciao operators, “builtins,” and most other syntactic and semantic language constructs are user-modifiable and live in *libraries*.¹ The Ciao module system also addresses the needs for modularity deriving from global analysis. We will start precisely with the introduction of the user view of packages.

¹ In fact, some Ciao packages are portable with little modification to other logic and constraint logic programming systems. Others require support from the kernel language (e.g. concurrency), to provide the desired semantics or efficiency. In any case, packages offer a *modularized* view of language extensions to the user.

```

1 :- module(_, _, [functional, lazy]).
2
3 nrev([])      := [].
4 nrev([H|T])  := ~conc(nrev(T), [H]).
5
6 conc([], L)  := L.
7 conc([H|T], K) := [H | conc(T, K)].
8
9 fact(N) := N=0 ? 1
10         | N>0 ? N * fact(--N).
11
12 :- lazy fun_eval nums_from/1.
13 nums_from(X) := [X | nums_from(X+1)].
14
15 :- use_module(library('lazy/lazy_lib'), [take/3]).
16 nums(N) := ~take(N, nums_from(0)).

```

Fig. 2. Some examples in Ciao functional notation.

2 Supporting Multiple Paradigms and Useful Features

Packages allow Ciao to support multiple programming paradigms and styles in a single program. The different source-level sub-languages are supported by a compilation process stated by the corresponding package, typically via a set of rules defining *source-to-source* transformations into the kernel language. This kernel is essentially pure Prolog plus a number of basic, instrumental additional functionalities (such as the cut, non-logical predicates such as `var/1` or `assert/1`, threads, attributed variables, etc.), all of which are in principle not visible to the user but can be used if needed at the kernel level to support higher-level functionality. However, the actual nature of the kernel language is actually less important than the extensibility mechanisms which allow these extensions to be, from the point of view of the compiler, analyzers, autodocumenter, and language users, on a par with the native builtins. We will now show some examples of how the extensibility provided by the module system allows Ciao to incorporate the fundamental constructs from a number of programming paradigms.

We will use the examples in Fig. 2 to illustrate general concepts regarding the module system and its extensibility. In Ciao the first and second arguments of a `module` declaration (line 1) hold the module name and list of exports in the standard way. “_” in the first argument means that the name of the module is the name of the file, without suffix, and in the second one that all definitions are exported. The third argument states a list of packages to be loaded (`functional` and `lazy` in this case, which provide functional notation and lazy evaluation). Packages are Ciao files which contain syntax and compilation rules and which are loaded by the compiler as plugins and unloaded when compilation finishes. Packages only modify the syntax and semantics of the module from where they are loaded, and therefore other modules can use packages introducing incompatible syntax / semantics without clashing. Packages can also be loaded using `use_package` declarations throughout the module.

Functional Programming: functional notation (Casas et al. 2006) is provided by a set of packages which, besides a convenient syntax to define predicates using a function-like

layout, gives support for semantic extensions which include higher-order facilities (e.g., predicate abstractions and applications thereof) and, if so required, lazy evaluation. Semantically, the extension is related to logic-functional languages like Curry (Hanus et al) but relies on flattening and resolution, using `freeze/2` for lazy evaluation, instead of narrowing. For illustration, Fig. 2 lists a number of examples using the Ciao functional notation. Thanks to the packages loaded by the `module` declaration, `nrev` and `conc` can be written in functional style by using multiple `:=/2` definitions. The `~` prefix operator in the second rule for `nrev` states that its argument (`conc`) is an interpreted function (a call to a predicate), as opposed to a data structure to unify with and return as a result of function invocation. This *eval* mark can be omitted when the predicate is marked for functional syntax. The recursive call to `nrev` does not need such a clarification because it is called within its own definition. The list constructor in `conc` is not marked for evaluation, and therefore it stands for a data structure instead of a predicate call.

`fact` is written using a disjunction (marked by “|”) of guards (delimited by “?”) which together commit the system to the first matching choice. Arithmetic operators are assumed to be evaluable by default, but this can be turned off with a special declaration. `nums_from` is declared lazy, which makes it possible to write a recursion which is executed only up to the extent it is necessary. In this case, it is called by `take` (imported from a library of lazy functions/predicates) which in turns allows `nums` to (lazily) return a list of *N* numbers starting at 0.

The following queries produce the expected answer:

```
?- use_package(functional).
?- X = ~nrev([1,2,3]).
   X = [3,2,1]
?- [3,2,1] = ~nrev(X).
   X = [1,2,3]
```

Loading the `functional` package in the top level allows using functional notation in it—the top level behaves in this sense essentially in the same way as a module. Since in general, functional *notation* is just syntax and thus no directionality is implied, the second query to `nrev/2` just instantiates its argument.

However, as mentioned before, other constructs such as conditionals do commit the system to the first matching case. The assertion language includes `func` assertions aimed at enforcing strictly “functional” behavior (e.g., being single moded, in the sense that a fixed set of inputs must always be ground and for them a single output is produced, etc.), and generating *assertions* (see later) which ensure that the code is used in a functional way.

Fig. 3 lists more examples using `functional` and other packages, and the result after applying just the transformations brought in by the `functional` package. Note the use of higher order in `list_of`: a predicate is called using a syntax which has a variable in the place of a predicate name. This is possible thanks to the `hiord` package (more on it later) which adds the necessary syntax and a compile-time translation into `call/N`.

Classic and ISO-Prolog: Ciao provides, through convenient defaults, an excellent Prolog system with support for ISO-Prolog. Other classical “builtins” expected by users, and which are provided by modern Prolog systems (YAP, SWI-Prolog, Quintus Prolog, SICStus Prolog, XSB, GNU Prolog, B-Prolog, BinProlog, etc.), are also conveniently available. In line with its design philosophy, in Ciao all of these features are optional and brought in

```

1 :- module(someprops, _, [functional, hiord]).
2
3 color := red | blue | green.
4
5 list := [] | [_ | list].
6
7 list_of(T) := [] | [~T | list_of(T)].
8
9 sorted := [] | [_].
10 sorted([X,Y|Z]) :- X @< Y, sorted([Y|Z]).

```

```

1 :- module(someprops, _, []).
2
3 color(red).    color(blue).    color(green).
4
5 list([]).
6 list(_|T) :- list(T).
7
8 :- use_module(engine(hiord_rt)).
9
10 list_of(_, []).
11 list_of(T, [X|Xs]) :- call(T, X), list_of(T, Xs).
12
13 sorted([]).    sorted([_]).
14 sorted([X,Y|Z]) :- X @< Y, sorted([Y|Z]).

```

Fig. 3. Examples in Ciao functional notation and state of translation after applying the `functional` and `hiord` packages.

```

:- module(h, [main/1]).

main :- write("Hello world!").

```

```

:- module(h, [main/1], [classic]).

main :- write("Hello world!").

```

Fig. 4. Two equivalent Prolog modules.

from libraries rather than being part of the language. This is done in such a way that classical Prolog code runs without modifications: the Prolog libraries are automatically loaded when module declarations have only the first two arguments, which is the type of module declaration used by most Prolog systems (see Fig. 4, left). This is equivalent to loading only the “`classic`” package (Fig. 4, right).

The set of ISO builtins and other ISO compliance-related features (e.g., the exceptions they throw) are triggered by loading the `iso` package (included in `classic`). Facilities for testing ISO compliance (Section 5.4) are also available.

The `classic` Prolog package is also loaded by default in user files (i.e., those without a module declaration) that do not load any packages explicitly via a `use_package` declaration. Also, the system top level comes up by default in Prolog mode. This can be tailored by creating a `~/.ciaorc` initialization file which, among other purposes, can be used to state packages to be loaded into the top level. As a result of these defaults, Ciao users who come to the system looking for a Prolog implementation do get what they expect. If they

do not poke further into the menus and manuals, they may never realize that Ciao is in fact quite a different beast under the hood.

Other Logic Programming Flavors: alternatively to the above, by not loading the classic Prolog package(s) the user can restrict a given module to use only pure logic programming, without any of Prolog’s impure features.² That means that if a call to `assert` were to appear within the module, it would be signaled by the compiler as a call to an undefined predicate. Features such as, for example, declarative I/O, can be added to such pure modules by loading additional libraries. This also allows adding individual features of Prolog to the pure kernel on a needed basis.

Higher-order logic programming with predicate abstractions (similar to *closures*) is supported through the `hiord` package. This is also illustrated in Fig. 3, where the `list_of/2` predicate receives a unary predicate which is applied to all the arguments of a list. As a further example of the capabilities of the `hiord` package, consider the queries:

```
?- use_package(hiord), use_module(library(hiordlib)).
?- P = ( _(X,Y) :- Y = f(X) ),      map([1, 3, 2], P, R).
```

where, after loading the higher-order package `hiord` and instantiating `P` to the anonymous predicate `_(X,Y) :- Y = f(X)`, the call `map([1, 3, 2], P, R)` applies `P` to each element of the list `[1, 3, 2]` producing `R = [f(1), f(3), f(2)]`. The (reversed) query works as expected, too:

```
?- P = ( _(X,Y) :- Y = f(X) ),      map(M, P, [f(1), f(3), f(2)]).
   M = [1, 3, 2]
```

If there is a free variable, say `V`, in the predicate abstraction and a variable with the same name `V` in the clause within which the anonymous predicate is defined, the variable in the predicate abstraction is bound to the value of the variable in the clause. Otherwise it is a free variable, in the logical sense (as any other existential variable in a clause). This is independent from the environment where the predicate abstraction is applied, and therefore closures have syntactic scoping.

Additional Computation Rules: in addition to the usual depth-first, left-to-right execution of Prolog, other computation rules such as breadth-first, iterative deepening, tabling (see later), and the Andorra model are available, again by loading suitable packages. This has proved particularly useful when teaching, since it allows postponing the introduction of the (often useful in practice) quirks of Prolog (see the slides of a course starting with pure logic programming and breadth-first search in <http://www.cliplab.org/logalg>).

Constraint Programming: several constraint solvers and classes of constraints using these solvers are supported including `CLP(Q)`, `CLP(R)` (a derivative of (Holzbaur 1994)), and a basic but usable `CLP(FD)` solver.³ The constraint languages and solvers, which are built on more basic blocks such as attributed variables (Holzbaur 1992) and/or the higher-level Constraint Handling Rules (CHR) (Frühwirth 2009), also available in Ciao, are extensible at the user level.

² The current implementation –as of version 1.13– does still leave a few builtins visible, some of them useful for debugging. To avoid the loading of any impure builtins in 1.13 the `pure` pseudo-package should be used.

³ `CLP(X)` stands for a Constraint Logic Programming System parametrized by the constraint domain \mathcal{X} .

```

1 :- module(_,_,[fsyntax,clpqf]).
2
3 fact(.=. 0) := .=. 1.
4 fact(N)      := .=. N*fact(.=. N-1) :- N .>. 0.
5
6 sorted := [] | [_].
7 sorted([X,Y|Z]) :- X .<. Y, sorted([Y|Z]).

```

Fig. 5. Ciao constraints (combined with functional notation).

Fig. 5 provides two examples using Ciao CLP(\mathcal{Q}) constraints, combined with functional notation. For example, line 3 can be read as: if the input argument of `fact` is constrained to 0 then the “output” argument is constrained to 1. In the next line, if the argument of `fact` is constrained to be greater than 0 then the “output” is constrained to be equal to `N*fact(.=. N-1)`. The two definitions (`fact` and `sorted`) can be called with their arguments in any state of instantiation. For example, the query

`?- sorted(X).`

returns (blanks in the answers have been edited to save space):

`X = [] ? ;`

`X = [_] ? ;`

`X = [_A, _B], _A .<. _B ? ;`

`X = [_A, _B, _C], _B .<. _C, _A .<. _B ?`

etc. As many other CLP systems Ciao is not, at the moment, a highly specialized constraint system, and it does not intend to compete with very high performance systems like, e.g., Gecode (Schulte and Stuckey 2008) or Comet (Van Hentenryck and Michael 2005). The purpose of the constraint solving support present in Ciao is to offer some reasonable functionality for medium-sized problems and to be able to explore new possibilities in the combination of paradigms.

Object-Oriented Programming: object oriented-style programming has been classically provided in Ciao through the `O'Ciao class` and `object` packages (Pineda and Bueno 2002). These packages provide capabilities for class definition, object instantiation, encapsulation and replication of state, inheritance, interfaces, etc. These features are designed to be natural extensions of the underlying module system. There is current work performed within the “`optimcomp`” branch (see later) revisiting these issues in the context of abstract mechanisms for passing, maintaining, and updating different notions of state. These extensions have also introduced imperative control structures and nested syntactic scopes.

Concurrency, Parallelism, and Distributed Execution: other packages bring in different capabilities for expressing concurrency (including a concurrent, shared version of the internal fact database which can be used for synchronization (Carro and Hermenegildo 1999)), distribution, and parallel execution (Cabeza and Hermenegildo 1995; Casas et al. 2008). A notion of “active objects” also allows compiling objects so that they are ultimately mapped to a standalone process, which can then be transparently accessed by the rest of an application. This provides simple ways to implement servers and services in general.

In addition to the programming paradigm-specific characteristics above, many additional features are available through libraries (that can also be activated or deactivated on a *per-module / class* basis), including:


```

1 :- module(someprops, _, [functional, hiord, assertions]).
2 :- prop color/1.      color := red | blue | green.
3 :- prop list/1.       list := [] | [_ | list].
4 :- prop list_of/2.    list_of(T) := [] | [~T | list_of(T)].
5 :- prop sorted/1.     sorted := [] | [_].
6                       sorted([X,Y|Z]) :- X @< Y, sorted([Y|Z]).

```

Fig. 6. Examples of state property definitions.

Structures with named arguments (feature terms), a trimmed-down version of ψ -terms (Aït-Kaci 1993) which translates structure unifications to Prolog unifications, adding no overhead to the execution when argument names can be statically resolved, and a small overhead when they are resolved at run time.

Partial support for advanced higher-order logic programming features, like higher-order unification, based on the algorithms used in λ Prolog (Wolfram 1992) (experimental).

Persistence, which allows Ciao to transparently save and restore the state of selected facts of the dynamic database of a program on exit and startup. This is the basis of a high-level interface with databases (Correas et al. 2004).

Tabled evaluation (Chen and Warren 1996), pioneered by XSB (experimental).

Answer Set Programming (ASP) (El-Khatib et al. 2005), which makes it possible to execute logic programs under the *stable model semantics* (experimental).

WWW programming, which establishes a direct mapping of HTML / XML and other formats to Herbrand terms, allowing the manipulation of WWW-related data easily through unification, writing CGIs, etc. (Cabeza and Hermenegildo 2001).

3 Ciao Assertions

An important feature of Ciao is the availability of a rich, multi-purpose assertion language. We now introduce (a subset of) this assertion language. Note that a great deal of the capabilities of Ciao for supporting and processing assertions draws on its extensibility features which are used to define and give semantics to the assertion language without having to change the low-level compiler.

Ciao Assertion Language Syntax and Meaning: Assertions are linguistic constructs which allow expressing properties of programs. Syntactically they appear as an extended set of declarations, and semantically they allow talking about preconditions, (conditional-) postconditions, whole executions, program points, etc. For clarity of exposition, we will focus on the most commonly-used subset of the Ciao assertion language: *pred* assertions and program point assertions. A detailed description of the full language can be found in (Puebla et al. 2000b; Bueno et al. 2009).

The first subset, *pred assertions*, is used to describe a particular predicate. They can be used to state preconditions and postconditions on the (values of) variables in the computation of predicates, as well as global properties of such computations (such as, e.g., the number of execution steps, determinacy, or the usage of some other resource). Fig. 7 includes a number of *pred* assertions whose syntax is made available through the *assertions* package. For example, the assertion (line 5): `:- pred nrev(A,B) : list(A) => list(B).`

```

1 :- module(_, [nrev/2], [assertions, nativeprops, functional]).
2 :- entry nrev/2 : {list, ground} * var.
3 :- use_module(someprops).
4
5 :- pred nrev(A, B) : list(A) => list(B).
6 :- pred nrev(A, B) : list_of(color, A) => list_of(color, B).
7 :- pred nrev(A, B) : list(A) + (not_fails, is_det, terminates).
8 :- pred nrev(A, _) : list(A) + steps_o(length(A)).
9
10 nrev([])      := [].
11 nrev([H|L])   := ~conc(nrev(L), [H]).
12
13 :- pred conc(A,B,C) : list(A) => size_ub(C, length(A)+length(B))
14                               + steps_o(length(A)).
15
16 conc([], L)   := L.
17 conc([H|L], K) := [ H | conc(L,K) ].

```

Fig. 7. Naive reverse with some –partially erroneous– assertions.

expresses that calls to predicate `nrev/2` with the first argument bound to a list are admissible, and that if such calls succeed then the second argument should also be bound to a list. `list/1` is an example of a *state property* –a prop, for short: a predicate which expresses properties of the (values of) variables. Other examples are defined in Fig. 6 (`sorted/1`, `color/1`, `list_of/2`), or arithmetic predicates such as `>/2`, etc. Note that `A` in `list(A)` above refers to the first argument of `nrev/2`. We could have used the parametric type `list_of/2` (also defined in Fig. 6), whose first argument is a type parameter, and written `list_of(term, A)` instead of `list(A)`, where the type `term/1` denotes any term. As an additional example using the parametric type `list_of/2`, the assertion in line 6 of Fig. 7 expresses that for any call to predicate `nrev/2` with the first argument bound to a list of colors, if the call succeeds, then the second argument is also bound to a list of colors.

State properties defined by the user and exported/imported as usual. In Fig. 7 some properties (`list/1`, `list_of/2`, `color/1`) are imported from the user module `someprops` (Fig. 6) and others (e.g., `size_ub/2`) from the system’s `nativeprops`. In any case props need to be marked explicitly as such (see Fig. 6) and this flags that they need to meet some restrictions (Puebla et al. 2000b; Bueno et al. 2009). E.g., their execution should terminate for any possible call since, as discussed later, props will not only be checked at compile time, but may also be involved in run-time checks. Types are just a particular case (further restriction) of state properties. Different type systems, such as regular types (`regtypes`), Hindley-Milner (`hmtypes`), etc., are provided as libraries. Since, e.g., `list_of/2` in Fig. 6 is a property that is in addition a regular type, this can be flagged as `:- prop list_of/2 + regtype.` or, more compactly, `:- regtype list_of/2.` Most properties (including types) are “runnable” (useful for run-time checking), and can be interacted with, i.e., the answers to a query `?- use_package(someprops), X = ~list.` are: `X = []`, `X = [_]`, `X = [_,_]`, `X = [_,_,_]`, etc. Note also that assertions such as the one in line 5 provide information not only on (a generalization of) types but also on modes.

In general `pred` assertions follow the schema:

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

Pred is a *predicate descriptor*, i.e., a predicate symbol applied to distinct free variables, such as, e.g., `nrev(A,B)`. *Precond* and *Postcond* are logic formulas about execution states, that we call *StateFormulas*. An execution state is defined by the bindings of values to variables in a given execution step (in logic programming terminology, a substitution, plus any global state). An atomic *StateFormula* (such as, e.g., `list(X)`, `X > 3`, or `sorted(X)`) is a literal whose predicate symbol corresponds to a state property. A *StateFormula* can also be a conjunction or disjunction of *StateFormulas*. Standard (C)LP syntax is used, with comma representing conjunction (e.g., “`(list(X), list(Y))`”) and semicolon disjunction (e.g., “`(list(X) ; int(X))`”). *Precond* is the precondition under which the `pred` assertion is applicable. *Postcond* states a conditional postcondition, i.e., it expresses that in any call to *Pred*, if *Precond* holds in the calling state and the computation of the call succeeds, then *Postcond* should also succeed in the success state. If *Precond* is omitted, the assertion is equivalent to: `:- pred Pred : true => Postcond.` and it is interpreted as “for any call to *Pred* which succeeds, *Postcond* should succeed in the success state.” As Fig. 7 shows, there can be several `pred` assertions for the same predicate. The set of preconditions (*Precond*) in those assertions is considered *closed* in the sense that they must cover all valid calls to the predicate.

Finally, `pred` assertions can include a *CompProps* field, used to describe properties of the whole computation of the calls to predicate *Pred* that meet precondition *Precond*. For example, the assertion in line 8 of Fig. 7, states that for any call to predicate `nrev/2` with the first argument bound to a list, the number of resolution steps, given as a function on the length of list *A*, is in $O(\text{length}(A))$ (i.e., such function is linear in $\text{length}(A)$).⁴ The assertion in line 7 of Fig. 7 is an example where *CompProps* is a conjunction: it expresses that the previous calls do not fail without first producing at least one solution, are deterministic (i.e., they produce at most one solution at most once), and terminate. Thus, in this case, *CompProps* describes a terminating functional computation. The rest of the assertions in Fig. 7 will be explained later, in the appropriate sections.

In order to facilitate writing assertions, Ciao also provides additional syntactic sugar such as *modes* and cartesian product notation. For example, consider the following set of `pred` assertions providing information on a reversible sorting predicate:

```
:- pred sort/2 : list(num) * var => list(num) * list(num) + is_det.
:- pred sort/2 : var * list(num) => list(num) * list(num) + non_det.
```

(in addition, curly brackets can be used to group properties –see Fig. 9). Using Ciao’s *isomodes* library, which provides syntax and meaning for the ISO instantiation operators, this can also be expressed as:

```
:- pred sort(+list(num), -list(num)) + is_det.
:- pred sort(-list(num), +list(num)) + non_det.
```

The `pred` assertion schema is in fact syntactic sugar for combinations of atomic assertions of the following three types:

```
:- calls   Pred [: Precond].
:- success Pred [: Precond] [=> Postcond].
:- comp   Pred [: Precond] [+ CompProps].
```

⁴ This is of course false, but we will let the compiler tell us –see later.

which describe all the admissible call states, the success states, and computational properties for each set of admissible call states (in this order).

Program-point assertions are of the form `check(StateFormula)` and they can be placed at the locations in programs in which a new literal may be added. They should be interpreted as “whenever computation reaches a state corresponding to the program point in which the assertion is, *StateFormula* should hold.” For example,

```
check((list_of(color, A), var(B)))
```

is a program-point assertion, where A and B are variables of the clause where the assertion appears.

Assertion status: Independently of the schema, each assertion can be in a *verification status*, marked by prefixing the assertion itself with the keywords, `check`, `trust`, `true`, `checked`, and `false`. This specifies respectively whether the assertion is provided by the programmer and is to be checked or to be trusted, or is the output of static analysis and thus correct (safely approximated) information, or the result of processing an input assertion and proving it correct or false, as will be discussed in the next section. The `check` status is assumed by default when no explicit status keyword is present (as in the examples so far).

Uses of assertions: as we will see, assertions find many uses in Ciao, ranging from testing to verification and documentation (for the latter, see `1pdoc` (Hermenegildo 2000)). In addition to describing the properties of the module in which they appear, assertions also allow programmers to describe properties of modules / classes which are not yet written or are written in other languages.⁵ This makes it possible to run checkers / verifiers / documenters against partially developed code.

4 The Ciao Unified Assertion Framework

We now describe the Ciao unified assertion framework (Bueno et al. 1997; Hermenegildo et al. 1999; Puebla et al. 2000b), implemented in the Ciao preprocessor, CiaoPP. Fig. 8 depicts the overall architecture. Hexagons represent tools and arrows indicate the communication paths among them. It is a design objective of the framework that most of this communication be performed also in terms of assertions. This has the advantage that at any point in the process the information is easily readable by the user. The input to the process is the user program, *optionally* including a set of assertions; this set always includes any assertion present for predicates exported by any libraries used (left part of Fig. 8).

Run-time checking of assertions: after (assertion) normalization (which, e.g., takes away syntactic sugar) the *RT-check* module transforms the program by adding run-time checks to it that encode the meaning of the assertions (we assume for now that the *Comparator* simply passes the assertions through). Note that the fact that properties are written in the source language and *runnable* is very useful in this process. Failure of these checks raises run-time errors referring to the corresponding assertion. *Correctness* of the transformation requires that the transformed program only produce an error if the assertion is in fact violated.

⁵ This is also done in other languages but, in contrast with Ciao, different kinds of assertions for each purpose are often used.

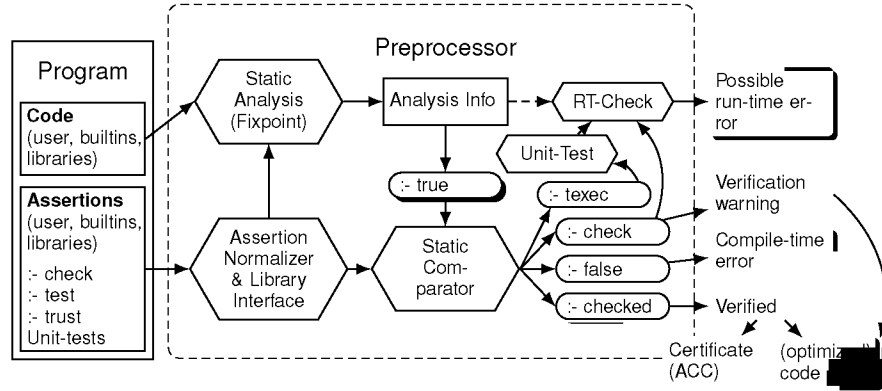


Fig. 8. The Ciao assertion framework (CiaoPP’s verification/testing architecture).

Compile-time checking of assertions: even though run-time checking can detect violations of specifications, it cannot guarantee that an assertion holds. Also, it introduces run-time overhead. The framework performs compile-time checking of assertions by *comparing* the results of *Static Analysis* (Fig. 8) with the assertions (Bueno et al. 1997; Hermenegildo et al. 1999). This analysis is typically performed by abstract interpretation (Cousot and Cousot 1977) or any other mechanism that provides *safe* upper or lower approximations of relevant properties, so that comparison with assertions is meaningful despite precision losses in the analysis. The type of analysis may be selected by the user or determined automatically based on the properties appearing in the assertions. Analysis results are given using also the assertion language, to ensure interoperability and make them understandable by the programmer. As a possible result of the comparison, assertions may be proved to hold, in which case they get *checked* status –Fig. 8. If all assertions are *checked* then the program is *verified*. In that case a certificate can be generated that can be shipped with programs and checked easily at the receiving end (using the *abstraction carrying code* approach (Albert et al. 2008)). As another possible result, assertions can be proved not to hold, in which case they get *false* status and a *compile-time error* is reported. Even if a program contains no assertions, it can be checked against the assertions contained in the libraries used by the program, potentially catching bugs at compile time. Finally, and most importantly, if it is not possible to prove nor to disprove (part of) an assertion, then such assertion (or part) is left as a *check* assertion, for which optionally run-time checks can be generated as described above. This can optionally produce a *verification warning*.

The fact that the system deals throughout with safe approximations of the meaning of the program, and that remaining in *check* status is an acceptable outcome of the comparison process, allows dealing with complex properties in a correct way. For example, in CiaoPP the programmer has the possibility of stating assertions about the efficiency of the program (lower and/or upper bounds on the computational cost of procedures (López-García et al. 2010)) which the system will try to verify or falsify, thus performing automatic debugging and verification of the *performance* of programs (see Section 5.2). Other interesting properties are handled such as data structure shape (including pointer sharing), bounds on data structure sizes, and other operational properties, as well as procedure-level properties such as determinacy (López-García et al. 2010), non-failure (Bueno et al. 2004), termi-

```

1 :- module(qsort, [qsort/2], [assertions, functional]).
2 :- use_module(compare, [geq/2, lt/2]).
3 :- entry qsort/2 : {list(num), ground} * var.
4
5 qsort([])      := [].
6 qsort([X|L])  := ~conc(qsort(L1), [X|qsort(L2)])
7               :- partition(L, X, L1, L2).
8
9 partition([], _B, [], []).
10 partition([E|R], C, [E|Left1], Right) :-
11     lt(E, C), partition(R, C, Left1, Right).
12 partition([E|R], C, Left, [E|Right1]) :-
13     geq(E, C), partition(R, C, Left, Right1).

```

Fig. 9. A modular qsort program.

nation, and bounds on the execution time (Mera et al. 2008), and the consumption of a large class of user-defined resources (Navas et al. 2007). Assertion checking in CiaoPP is also module-aware (Pietrzak et al. 2006; Pietrzak et al. 2008). Finally, the information from analysis can be used to optimize the program in later compilation stages, as we will discuss later.

5 Static Verification, Debugging, Run-Time Checking, and Unit Testing in Practice

We now present some examples which illustrate the use of the Ciao assertion framework discussed in the previous section, as implemented in CiaoPP. We also introduce some more examples of the assertion language as we proceed.

5.1 Automatic Inference of (Non-Trivial) Code Properties

We first illustrate with examples the automatic inference of code properties (box “Static Analysis” in Fig. 8). Modes and types are inferred, as mentioned before, using different methods including (Muthukumar and Hermenegildo 1992; Muthukumar and Hermenegildo 1991) for modes and (Saglam and Gallagher 1995; Vaucheret and Bueno 2002) for types. As also mentioned before, CiaoPP includes a non-failure analysis (Bueno et al. 2004), which can detect procedures and goals that can be guaranteed not to fail, i.e., to produce at least one solution or not to terminate. It also can detect predicates that are “covered”, i.e., such that for any input (included in the calling type of the predicate), there is at least one clause whose “test” (head unification and body builtins) succeeds. CiaoPP also includes a determinacy analysis (López-García et al. 2010), which can detect predicates which produce at most one solution at most once, or predicates whose clause tests are mutually exclusive, even if they are not deterministic because they call other predicates that can produce more than one solution (it means that the predicate does not perform backtracking at the level of its clauses).

Consider again the naive reverse program in Fig. 7. The assertion in line 2 is an example

of an `entry` assertion: a `pred` assertion addressing calls from outside the module.⁶ It informs the CiaoPP analyzers that in all external calls to `nrev/2`, the first argument will be a ground list and the second one a free variable. Using only the information specified in the `entry` assertion, the aforementioned analyses infer different sorts of information which include, among others, that expressed by the following assertion:

```
:- true pred nrev(A,B): ( list(A), var(B) ) => ( list(A), list(B) )
    + ( not_fails, covered, is_det, mut_exclusive ).
```

As mentioned before, CiaoPP can also infer lower and upper bounds on the sizes of terms and the computational cost of predicates (Debray et al. 1997; Debray et al. 1990; Debray and Lin 1993), including user-defined resources (Navas et al. 2007). The cost bounds are expressed as functions on the sizes of the input arguments and yield the number of resolution steps. Note that obtaining a finite upper bound on cost also implies proving *termination* of the predicate.

As an example, the following assertion is part of the output of the lower-bounds analysis (that also includes a non-failure analysis, without which a trivial lower bound of 0 would be derived):

```
:- true pred conc(A,B,C) : ( list(A), list(B), var(C) )
    => ( list(A), list(B), list(C),
        size_lb(A,length(A)), size_lb(B,length(B)),
        size_lb(C,length(B)+length(A)) )
    + ( not_fails, covered, steps_lb(length(A)+1) ).
```

Note that in this example the size measure used is list length. The property `size_lb(C,length(B)+length(A))` means that a (lower) bound on the size of the third argument of `conc/3` is the sum of the sizes of the first and second arguments. The inferred lower bound on computational steps is the length of the first argument of `conc/3` plus one. The `length/1` property used in the previous assertion is just the `length/2` predicate called using functional syntax, that carries the last argument. CiaoPP currently uses some predefined metrics for measuring the “size” of an input, such as list length, term size, term depth, or integer value. These are automatically assigned to the predicate arguments involved in the size and cost analysis according to the previously inferred type information. A new, experimental version of the size analyzers is in development that can deal with user-defined size metrics (i.e., predicates) and is also able to synthesize automatically size metrics.

5.2 Static (Performance) Verification and Debugging

We now illustrate static verification and debugging, i.e., statically proving or disproving program assertions (i.e., specifications). This corresponds to the “Static Comparator” box in Fig. 8. We focus on verification of the resource usage of programs, such as lower and/or upper bounds on execution steps or user defined resources, but the process also applies to more traditional properties such as types and modes. Consider the assertion in line 8 of Fig. 7, which states that `nrev` should be linear in the length of the (input) argument A. With compile-time error checking turned on, CiaoPP automatically selects mode, type,

⁶ Note that in CiaoPP the `pred` assertions of exported predicates can be used optionally instead of `entry`.

```

5 :- pred qsort(A,B) => (ground(B),sorted_num_list(B)).
6
7 :- prop sorted_num_list/1.
8
9 sorted_num_list([]).
10 sorted_num_list([X]):- num(X).
11 sorted_num_list([X,Y|Z]):- num(X),num(Y),geq(Y,X),
12                             sorted_num_list([Y|Z]).
13 qsort([],[]).
14 qsort([X|L],R) :- partition(L,X,L1,L2),
15                  qsort(L2,R2), qsort(L1,R1),
16                  conc(R2,[X|R1],R).

```

Fig. 10. An example for run-time checking.

non-failure, and lower/upper-bound cost analyses and issues the following error message (corresponding to the “compile-time error” exit in Fig. 8):

```

ERROR: False assertion:
      :- pred nrev(A, _) : list(A) + steps_o(length(A))
      because on comp nrev:nrev(A,_):
      [generic_comp] : steps_lb(0.5*exp(length(A),2)+1.5*length(A)+1)

```

This message states that `nrev` will take at least $\frac{\text{length}(A)^2 + 3 \text{length}(A)}{2} + 1$ resolution steps (a safe lower bound inferred by the cost analyzer), while the assertion requires the cost to be in $O(\text{length}(A))$ resolution steps. As a result, the worst case asymptotic complexity stated in the user-provided assertion is proved wrong by the lower bound cost assertion inferred by the analysis. Note that upper-bound cost assertions can be proved to hold by means of upper-bound cost analysis if the bound computed by analysis is lower or equal than the upper bound stated by the user in the assertion. The converse holds for lower-bound cost assertions (Bueno et al. 1997; López-García et al. 2010). Thanks to this functionality, CiaoPP can also certify programs with resource consumption assurances as well as efficiently checking such certificates (Hermenegildo et al. 2004).

5.3 Run-Time Checking

As mentioned before, (parts of) assertions which cannot be verified at compile time (see again Fig. 8) are translated into run-time checks via a program transformation. As an example, consider the assertion, property definitions, and (wrong) definition of `qsort/2` in Fig. 10 (where `partition/4` and `conc/3` are defined as in Figures 9 and 2 respectively). The assertion states that `qsort/2` always returns a ground, *sorted* list of numbers. The program contains a bug to be discovered. With run-time checking turned on, the following query produces the listed results:

```

?- qsort([1,2],X).
{In /tmp/qsort.pl
ERROR: (lns 5-5) Run-time check failure in assertion for: qsort:qsort/2.
In *success*, unsatisfied property: sorted_num_list.
ERROR: (lns 13-16) Failed in qsort:qsort/2.}

```

Two errors are reported for a single run-time check failure: the first error shows the actual assertion being violated and the second marks the first clause of the predicate which

violates the assertion. However, not enough information is provided to determine which literal made the erroneous call. It is also possible to increase the verbosity level of the messages and to produce a call stack dump up to the exact program point where the violation occurs, showing for each predicate the body literal that led to the violation:

```
?- set_ciao_flag(rtchecks_callloc,literal),
   set_ciao_flag(rtchecks_namefmt,long), use_module('/tmp/qsort.pl').
yes
?- qsort([3,1,2],X).
{In /tmp/qsort.pl
ERROR: (lns 5-5) Run-time check failure in assertion for: qsort:qsort(A,B).
In *success*, unsatisfied property: sorted_num_list(B).
Because: ['B'=[2,1]].
ERROR: (lns 13-16) Failed in qsort:qsort(A,B).
ERROR: (lns 13-16) Failed when invocation of qsort:qsort([X|L],R)
           called qsort:qsort(L1,R1) in its body.}
{In /tmp/qsort.pl
ERROR: (lns 5-5) Run-time check failure in assertion for: qsort:qsort(A,B).
In *success*, unsatisfied property: sorted_num_list(B).
Because: ['B'=[3,2,1]].
ERROR: (lns 13-16) Failed in qsort:qsort(A,B).}
```

The output makes it easier to locate the error since the call stack dump provides the list of calling predicates. Note that the first part of the assertion is not violated, since B is ground. However, on success the output of `qsort/2` is a sorted list but in reverse order, which gives us a hint: the variables R1 and R2 in the call to `conc/3` are swapped by mistake.

5.4 Unit Testing

Unit tests need to express on one hand *what to execute* and on the other hand *what to check* (at run time). A key characteristic of the Ciao approach to unit testing (see (Mera et al. 2009) for a full description) is that it (re)uses the assertion language for expressing what to check. This avoids redundancies and allows reusing the same assertions and properties used for static and/or run-time checking. However, the assertion language does include a minimal number of additional elements for expressing *what to execute*. In particular, it includes the following assertion schema: `:- texec Pred [: Precond] [+ ExecProps]`.

which states that we want to execute (as a test) a call to *Pred* with its variables instantiated to values that satisfy *Precond*. *ExecProps* is a conjunction of properties describing how to drive this execution. As an example, the assertion:

```
:- texec conc(A, B, C) : (A=[1,2],B=[3],var(C)).
```

expresses that the testing harness should execute a call to `conc/3` with the first and second arguments bound to `[1,2]` and `[3]` respectively and the third one unbound.

In our approach many of the properties that can be used in *Precond* (e.g., types) can also be used as value generators for those variables, so that input data can be automatically generated for the unit tests (see e.g., the technique described in (Gómez-Zamalloa et al. 2008)). However, there are also some properties that are specific for this purpose, such as, e.g., random value generators.

We can define a complete unit test using the `texec` assertion together with other assertions expressing *what to check at run time* such as, for example:

```

:- check success conc(A,B,C): (A=[1,2],B=[3],var(C)) => C=[1,2,3].
:- check comp    conc(A,B,C): (A=[1,2],B=[3],var(C)) + not_fails.

```

The success assertion states that if a call to `conc/3` with the first and second arguments bound to `[1,2]` and `[3]` respectively and the third one unbound terminates with success, then the third argument should be bound to `[1,2,3]`. The comp assertion says that such a call should not fail.

One additional advantage of Ciao's unified framework is that the execution expressed by a *Precond* in a `texec` assertion for unit testing can also trigger the checking of parts of other assertions that could not be checked at compile time and thus remain as run-time checks. This way, a single set of run-time checking machinery can deal with both run-time checks and unit tests. Conversely, static checking of assertions can safely avoid (possibly parts of) unit test execution (see Fig. 8 again), so that sometimes unit tests can be checked without ever running them.

Finally, the system provides as syntactic sugar another predicate assertion schema, the `test` schema: `:- test Pred [: Precond] [=> Postcond] [+ CompExecProps]`.

This assertion is interpreted as the combination of the following three assertions:

```

:-      texec   Pred [: Precond] [+ ExecProps].
:- check success Pred [: Precond] [=> Postcond].
:- check comp   Pred [: Precond] [+ CompProps].

```

For example, the assertion:

```

:- test conc(A,B,C): (A=[1,2],B=[3],var(C))=> C=[1,2,3] + not_fails.

```

is conceptually equivalent to the three (`texec`, `success`, `comp`) shown previously as examples (*CompExecProps* being the conjunction of *ExecProps* and *CompProps*).

The assertion language not only allows checking single solutions (as it is done in the previous `test` assertion for `conc/3`), but also multiple solutions to calls. In addition, it includes a set of predefined properties that can be used in *ExecProps* that are specially useful in the context of unit tests, including: an upper bound *N* on the number of solutions to be checked (`try_sols(N)`); expressing that the execution of the unit test should be repeated *N* times (`times(N)`); that a test execution should throw a particular exception (`exception(Excep)`); or that a predicate should write a given string into the current output stream (`user_output(String)`) or the current error stream (`user_error(String)`). Similarly, properties are provided that are useful in *Precond*, for example, to generate random input data with a given probability distribution (e.g., for floating point numbers, including special cases like *infinite*, *not-a-number*, or *zero* with sign).

The testing mechanism has proved very useful in practice. For example, with it we have developed a battery of tests that are used for checking ISO-Prolog compliance in Ciao. The set contains 976 unit tests, based on the *Stdprolog* application (Szabó and Szeredi 2006).

6 High Performance with Less Effort

A potential benefit of strongly typed languages is performance: the compiler can generate more efficient code with the additional type and mode information that the user provides. Performance is a good thing, of course. However, it is also attractive to avoid putting the burden of efficient compilation on the user by requiring the presence of many program

declarations: the compiler should certainly take advantage of any information given by the user, but if the information is not available, it should do the work of inferring such program properties whenever possible. This is the approach taken in Ciao: as we have seen before, when assertions are not present in the program, Ciao's analyzers try to *infer* them. Most of these analyses are performed at the kernel language level, so that the same analyzers are used for several of the supported programming models.

High-level optimization: the information inferred by the global analyzers is used to perform high-level optimizations, including multiple abstract specialization (Puebla and Hermenegildo 1995), partial evaluation (Puebla et al. 2006), dead code removal, goal reordering, reduction of concurrency / dynamic scheduling (Puebla et al. 1997), etc.

Optimizing compilation: the objective is again to achieve the best of both worlds: with no assertions or analysis information, the low-level Ciao compiler (`ciaooc` (Cabeza and Hermenegildo 2000b)) generates code which is competitive in speed and size with the best dynamically typed systems. And then, when useful information is present, either coming from the user or inferred by the system analyzers, the experimental optimizing compiler, `optimcomp` (see, e.g., (Morales et al. 2004) for an early description) can produce code that is competitive with that of strongly-typed systems. Ciao's highly optimized compilation has been successfully tested, for example, in applications with tight memory and real-time constraints (Carro et al. 2006), obtaining a 7-fold speed-up w.r.t. the default bytecode compilation. The performance of the latter is already similar to that of state-of-the-art abstract machine-based systems. The application involved the real-time spatial placement of sound sources for a virtual reality suit, and ran in a small ("Gumstix") processor embedded within a headset. Interestingly, this performance level is only around 20-40% slower than a comparable (but more involved) implementation in C of the same application.

ImProlog: driven by the need of producing efficient final code in extreme cases, we have also introduced in the more experimental parts of the system the design and compilation of a variant of Prolog (which we termed *ImProlog*) which, besides assertions for types and modes, introduces imperative features such as *low-level pointers* and *destructive assignment*. This restricted subset of the merge of the imperative and logic paradigms is present (in beta) in the `optimcomp` branch and has been used to write a complete WAM emulator including its instructions (Morales et al. 2009), and part of its lower-level data structures (Morales et al. 2008). This source code is subject to several analysis and optimization stages to generate highly efficient C code. This approach is backed by some early performance numbers, which show this automatically generated machine to be on average just 8% slower than that of a highly optimized emulator, such as YAP 5.1.2 (Costa et al. 2002) (and actually faster in some benchmarks), and 44% faster than the stock Ciao emulator. In this case, some of the annotations ImProlog takes advantage of cannot be inferred by the analyzers because, for example, they address issues (such as word size) which depend on the targeted architecture, which must be entered by hand.

Automatic parallelization: a particularly interesting optimization performed by CiaoPP, in the same vein of obtaining high performance with less effort from the programmer, and which is inherited from the &-Prolog system, is *automatic parallelization* (Hermenegildo 1997; Gupta et al. 2001). This is specially relevant nowadays given that the wide availability of multicore processors has made parallel computers mainstream. We illustrate this by

```

qsort([X|L],R) :-
    partition(L,X,L1,L2),
    ( indep(L1, L2) ->
        qsort(L2,R2) & qsort(L1,R1)
    ;
        qsort(L2,R2), qsort(L1,R1) ),
    conc(R1,[X|R2],R).

```

Fig. 11. Parallel QuickSort w/run-time checks.

```

qsort([X|L],R) :-
    partition(L,X,L1,L2),
    qsort(L2,R2) &
    qsort(L1,R1),
    conc(R1,[X|R2],R).

```

Fig. 12. Parallel QuickSort.

means of a simple example using goal-level program parallelization (Bueno et al. 1999; Casas et al. 2007). This optimization is performed as a source-to-source transformation, in which the input program is *annotated* with parallel expressions as a result. The parallelization algorithms, or annotators (Muthukumar et al. 1999), exploit parallelism under certain *independence* conditions, which allow guaranteeing interesting correctness and no-slowdown properties for the parallelized programs (Hermenegildo and Rossi 1995; García de la Banda et al. 2000). This process is made more complex by the presence of variables shared among goals and pointers among data structures at run time.

Consider the program in Fig. 9 (with `conc/3` defined as in Fig. 2). A possible parallelization (obtained in this case with the “MEL” annotator (Muthukumar et al. 1999)) is shown in Fig. 11, which means that, provided that `L1` and `L2` do not have variables in common at run time, then the recursive calls to `qsort` can be run in parallel. Assuming that `lt/2` and `geq/2` in Fig. 9 need their arguments to be ground (note that this may be either inferred by analyzing the implementation of `lt/2` and `geq/2` or stated by the user using suitable assertions), the information collected by the abstract interpreter using, e.g., mode and sharing/freeness analysis, can determine that `L1` and `L2` are ground after `partition`, and therefore they do not have variables to share. As a result, the independence check and the corresponding conditional is simplified via abstract executability and the annotator yields instead the code in Fig. 12, which is much more efficient since it has no run-time check. This check simplification process is described in detail in (Bueno et al. 1999) where the impact of abstract interpretation in the effectiveness of the resulting parallel expressions is also studied.

The checks in the above example aim at *strict* independent and-parallelism (Hermenegildo and Rossi 1995). However, the annotators are parametrized on the notion of independence. Different checks can be used for different independence notions: non-strict independence (Cabeza and Hermenegildo 1994), constraint-based independence (García de la Banda et al. 2000), etc. Moreover, all forms of and-parallelism in logic programs can be seen as independent and-parallelism, provided the definition of independence is applied at the appropriate granularity level.⁷

Ciao currently includes low-level, native support for the creation of (POSIX-based) threads at the O.S. level which are used as support for independent and-parallel execution (Casas et al. 2008). Task stealing is used to achieve independence between the number of O.S. threads and the number of parallel goals (Hermenegildo 1986; Hermenegildo and Greene 1991).

⁷ For example, stream and-parallelism can be seen as independent and-parallelism if the independence of “bindings” rather than goals is considered.

Granularity control: the information produced by the CiaoPP cost analyzers is also used to perform combined compile–time/run–time resource control. An example of this is *task granularity control* (López-García et al. 1996) of parallelized code. Such parallel code can be the output of the process mentioned above or code parallelized manually. In general, this run-time granularity control process includes computing sizes of terms involved in granularity control, evaluating cost functions, and comparing the result with a threshold to decide between parallel and sequential execution. However, there are optimizations to this general process, such as cost function simplification and improved term size computation.

Visualization of parallel executions: a tool (VisAndOr (Carro et al. 1993)) for depicting parallel executions was developed and used to help programmers and system developers understand the program behavior and task scheduling performed. This is very useful for tuning the abstract machine and the automatic parallelizers.

7 Incremental Compilation and Other Support for Programming in the Small and in the Large

In addition to all the functionality provided by the preprocessor and assertions, programming in the large is further supported again by the module system (Cabeza and Hermenegildo 2000a). This design is the real enabler of Ciao’s modular program development tools, effective global program analysis, modular static debugging, and module-based automatic incremental compilation and optimization. The analyzers and compiler take advantage of the module system and module dependencies to reanalyze / recompile only the required parts of the application modules after one or more of them is changed, automatically and implicitly, without any need to define “makefiles” or similar dependency-related additional files, or to call explicitly any “make”-style command.

Application deployment is enhanced beyond the traditional Prolog top level, since the system offers a full-featured interpreter but also supports the use of Ciao as a scripting language and a compiled language. Several types of executables can be easily built, from multiarchitecture *bytecode* executables to single-architecture, standalone executables. Multiple platforms are supported, including the very common Linux, Windows, Mac OS X, and other Un*x-based OSs, such as Solaris. Due to the explicit effort in keeping the requirements of the virtual machine to a minimum, the effort of porting to new operating systems has so far been reduced. Ciao is known to run on several architectures, including Intel, Power PC, SPARC, and XScale / ARM processors.

Modular distribution of user and system code in Ciao, coupled with modular analysis, allows the generation of stripped executables containing only those builtins and libraries used by the program. Those reduced-size executables allow programming in the small when strict space constraints are present.

Flexible development of applications and libraries that use components written in several languages is also facilitated, by means of compiler and abstract machine support for multiple bidirectional foreign interfaces to C/C++, Java, Tcl/Tk, SQL databases (through a notion of predicate persistence), etc. The interfaces are described by using assertions, as previously stated, and any necessary glue code is automatically generated from them.

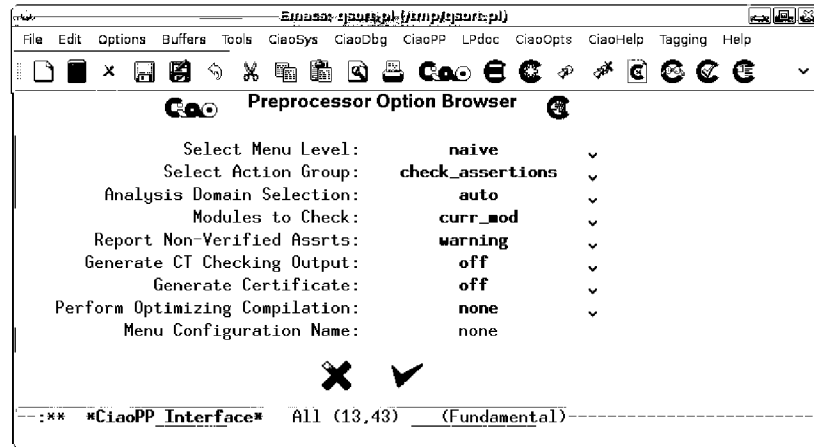


Fig. 13. Menu for compile-time / run-time checking of assertions.

8 An Advanced Integrated Development Environment

Another design objective of Ciao has been to provide a truly productive program development environment that integrates all of the tools mentioned before in order to allow the development of correct and efficient programs in as little time and with as little effort as possible. This includes a *rich graphical development interface*, based on the latest graphical versions of Emacs and offering menu and widget-based interfaces with direct access to the top-level/debugger, preprocessor, and autodocumenter, as well as an embeddable source-level debugger with breakpoints, and several profiling and execution visualization tools. In addition, a plugin with very similar functionality is also available for the Eclipse programming environment.⁸

The programming environment makes it possible to start the top level, the debugger, or the preprocessor, and to load the current module within them by pressing a button or via a pair of keystrokes. Tracing the execution in the debugger makes the current statement in the program be highlighted in an additional buffer containing the debugged file.

The environment also provides automated access to the documentation, extensive syntax highlighting, auto-completion, auto-location of errors in the source, etc., and is highly customizable (to set, for example, alternative installation directories or the location of some binaries). The direct access to the preprocessor allows interactive control of all the static debugging, verification, and program transformation facilities. For example, Fig. 13 shows the menu that allows choosing the different options for compile-time and run-time checking of assertions (this menu is the “naive” one, that offers reduced and convenient defaults for all others; selecting “expert” mode allows changing all options).

As another example, Fig. 14 shows CiaoPP indicating a semantic error in the source. In particular, it is the cost-related error discussed in Section 5.2 where the compiler detects (statically!) that the definition of `nrev` does not comply with the assertion requiring it to be of linear complexity.

The direct access to the auto-documentation facilities (Hermenegildo 2000) allows the

⁸ See <http://eclipse.ime.usp.br/projetos/grad/plugin-prolog/index.html>.

```

File Edit Options Buffers Tools CiaoSys CiaoDbg CiaoPP LPdoc CiaoOpts CiaoHelp Help

:- module(_, [nrev/2], [assertions, nativeprops, functional]).
:- entry nrev/2 : (list, ground) * var.
:- use_module(someprops).

:- pred nrev(A, B) : list(A) => list(B).
:- pred nrev(A, B) : list_of(color, A) => list_of(color, B).
:- pred nrev(A, B) : list(A) + (not_fails, is_det, terminates).
:- pred nrev(A, _) : list(A) + steps_o(length(A)).

nrev([], _) :- [].
nrev([H|L], _) :- ~conc(nrev(L), [H]).

:- pred conc(A,B,C) : list(A) => size_ub(C,length(A)+length(B))
                        + steps_o(length(A)).
conc([], L) = L.
conc([H|L], K) : [ H | conc(L,K) ].

-- nrev.pl All (8,1) (Ciao)-----
{ERROR(ctchecks_pred_messages): (lins 8-8) False assertion:
:- check comp nrev(A,B)
      : list(A)
      + steps_o(length(A)).

because
on comp nrev:nrev(A,_) :

[[generic_comp] : steps_ub(0.5*exp(length(A),2)+1.5*length(A)+1),steps_lb(0.5*
*exp(length(A),2)+1.5*length(A)+1),not_fails,covered,mult_exclusive,is_det

-1:** *Ciao-Preprocessor* 97% (271,0) (Ciao/CiaoPP/LPdoc Listener: run)---

```

Fig. 14. Error location in the source –a cost error.

easy generation of human-readable program documentation from the current file in a variety of formats from the assertions, directives, and machine-readable comments present in the program being developed or in the system’s libraries, as well as all other program information available to the compiler. This direct access to the documenter and on a per-module basis is very useful in practice for incrementally building documentation and making sure that, for example, cross references between files are well resolved and that the documentation itself is well structured and formatted.

9 Some Final Thoughts: Dynamic vs. Static Languages, Parallelism

We now provide as conclusions some final thoughts regarding how the now fairly classical Ciao approach fares in the light of recent trends. We argue that in fact many of the motivations and ideas behind the development of Ciao and CiaoPP over the years have acquired presently even more importance.

The environment in which much software needs to be developed nowadays (decoupled software development, use of components and services, increased interoperability constraints, need for dynamic update or self-reconfiguration, mash-ups) is posing requirements which align with the classical arguments for dynamic languages but which in fact go beyond them. Examples of often required dynamic features include making it possible to (partially) test and verify applications which are partially developed, and which will never be “complete” or “final,” or which evolve over time in an asynchronous, decentralized fashion (e.g., software service-based systems). These requirements, coupled with the intrinsic agility in development of dynamic programming languages such as Python, Ruby, Lua, JavaScript, Perl, PHP, etc. (with Scheme or Prolog also in this class) have made such languages a very attractive option for a number of purposes that go well beyond simple

scripting. Parts written in these languages often become essential components (or even the whole implementation) of full, mainstream applications.

At the same time, detecting errors at compile time and inferring properties required to optimize programs are still important issues in real-world applications. Thus, strong arguments are also made for static languages. For example, modern logic and functional languages (e.g., Mercury (Somogyi et al. 1996) or Haskell (Hudak et al. 1992)) impose strong type-related requirements such as that all types (and, when relevant, modes) have to be defined explicitly or that all procedures have to be “well-typed” and “well-moded.” One argument supporting this approach is that it clarifies interfaces and meanings and facilitates “programming in the large” by making large programs more maintainable and better documented. Also, the compiler can use the static information to generate more specific code, which can be better in several ways (e.g., performance-wise).

In the design of Ciao we certainly had the latter arguments in mind, but we also wanted Ciao to be useful (as the scripting languages) for highly dynamic scenarios such as those listed above, for “programming in the small,” for prototyping, for developing simple scripts, or simply for experimenting with the solution to a problem. We felt that compulsory type and mode declarations, and other related restrictions, can sometimes get in the way in these contexts.

The solution we came up with involves the rich Ciao assertion language and the Ciao methodology for dealing with such assertions (Bueno et al. 1997; Hermenegildo et al. 1999; Puebla et al. 2000b), which implies making a best effort to infer and check these properties statically, using powerful and rigorous static analysis tools based on safe approximations, while accepting that complete verification or validation may not always be possible and run-time checks may be needed. This approach opens up the possibility of dealing in a uniform way with a wide variety of properties besides types (e.g., rich modes, determinacy, non-failure, sharing/aliasing, term linearity, cost,...), while at the same time making assertions *optional*. We argue that this solution has made Ciao very useful for programming both in the small and in the large, combining effectively the advantages of the strongly typed and untyped language approaches. In contrast, systems which focus exclusively on automatic compile-time checking are often rather strict about the properties which the user can write. This is understandable because otherwise the underlying static analyses are of little use for proving the assertions.

In this sense, the Ciao model is related to the *soft typing* approach (Cartwright and Fagan 1991), but without being restricted to types. It is also related to the NU-Prolog debugger (Naish et al. 1989), which performed compile-time checking of decidable (regular) types and also allowed calling Prolog predicates at run time as a form of dynamic type checks. However, as mentioned before, compile-time inference and checking in the Ciao model is not restricted to types (nor requires properties to be decidable), and it draws many new synergies from its novel combination of assertion language, properties, certification, run-time checking, testing, etc. The practical relevance of the combination of static and dynamic features is in fact illustrated by the many other languages and frameworks which have been proposed lately aiming at bringing together ideas of both worlds. This includes recent work in gradual typing for Scheme (Tobin-Hochstadt and Felleisen 2008) (and the related PLT-Scheme/Racket language) or Prolog (Schrijvers et al. 2008), the recent uses of “contracts” in verification (Logozzo et al.), and the pragmatic view-

point of (Lamport and Paulson 1999), but applied to programming languages rather than specification languages. The fifth edition of ECMAScript, on which the JavaScript and ActionScript languages are based, includes optional (soft-)type declarations to allow the compiler to generate more efficient code and detect more errors. The Tamarin project (Mozilla 2008) intends to use this additional information to generate faster code. The RPython (Ancona et al. 2007) language imposes constraints on the programs to ensure that they can be statically typed. RPython is moving forward as a general purpose language. This line has also brought the development of safe versions of traditional languages, such as, e.g., CCured (Necula et al. 2005) or Cyclone (Jim et al. 2002) for C, as well as of systems that offer capabilities similar to those of the Ciao assertion preprocessor, such as Deputy (<http://deputy.cs.berkeley.edu/>) or Spec# (Leavens et al. 2007).

We believe that Ciao has pushed and is continuing to push the state of the art in solving this currently very relevant and challenging conundrum between statically and dynamically checked languages. It pioneered what we believe is the most promising approach in order to be able to obtain the best of both worlds: the combination of a flexible, multi-purpose assertion language with strong program analysis technology. This allows support for dynamic language features while at the same time having the capability of achieving the performance and efficiency of static systems. We believe that a good part of the power of the Ciao approach also comes from the synergy that arises from using the same framework and assertion language for different tasks (static verification, run-time checking, unit testing, documentation, ...) and its interaction with the design of Ciao itself (its module system, its extensibility, or the support for predicates and constraints). The fact that properties are written in the source language is instrumental in allowing assertions which cannot be statically verified to be translated easily into run-time checks, and this is instrumental in turn in allowing users to get some benefits even if a certain property cannot be verified at compile time. The assertion language design also allows a smooth integration with unit testing. Moreover, as (parts of) the unit tests that can be verified at compile time are eliminated, sometimes unit tests can be checked without ever running them.

Another interesting current trend where Ciao's early design choices have become quite relevant is parallelism. Multi-core processors are already the norm, and the number of cores is expected to grow in the foreseeable future. This has renewed the interest in language-related designs and tools which can simplify the intrinsically difficult (Karp and Babb 1988) but currently necessary task of parallelizing programs. In the Ciao approach programmers can choose between expressing manually the parallelism with high-level constructs, letting the compiler discover the parallelism, or a combination of both. The parallelizer also checks manual parallelizations for correctness and, conversely, programmers can easily inspect and improve the (source level) parallelizations produced by the compiler. These capabilities rely (again) on the use of CiaoPP's powerful, modular, and incremental abstract interpretation-based static program analyzers. This approach was pioneered by &-Prolog and Ciao (arguably one of the first direct uses of abstract interpretation in a real compiler), and seems the most promising nowadays, being adopted by many systems (see, e.g., (Hermenegildo 1997) for further discussion).

Probing Further. The reader is encouraged to explore the system, its documentation, and the tutorial papers that have been published on it. At the time of writing, work is progressing on the new 1.14 system version which includes significant enhancements with respect to the previous major release (1.10). In addition to the autodocumenter, new versions also include within the default distribution the CiaoPP preprocessor (initially beta versions), which was previously distributed on demand and installed separately. The latest version of Ciao, 1.13, which is essentially a series of release candidates for 1.14 has now been available for some time from the Ciao web site (snapshots) and subversion repository.

Contact / download info / license: the latest versions of Ciao can be downloaded from <http://www.ciaohome.org> or <http://www.cliplab.org>. Ciao is free software protected to remain so by the GNU LGPL license, and can be used freely to develop both free and commercial applications.

References

- AÏT-KACI, H. 1993. An Introduction to LIFE – Programming with Logic, Inheritance, Functions and Equations. In *Proceedings of the 1993 International Symposium on Logic Programming*, D. Miller, Ed. MIT Press, 52–68.
- ALBERT, E., PUEBLA, G., AND HERMENEGILDO, M. 2008. Abstraction-Carrying Code: A Model for Mobile Code Safety. *New Generation Computing* 26, 2.
- ANCONA, D., ANCONA, M., CUNI, A., AND MATSAKIS, N. D. 2007. RPython: a Step towards Reconciling Dynamically and Statically Typed OO Languages. In *DLS'07*. ACM, 53–64.
- BRUYNOGHE, M. 1991. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming* 10, 91–124.
- BUENO, F., CABEZA, D., CARRO, M., HERMENEGILDO, M., LÓPEZ-GARCÍA, P., AND PUEBLA- (EDS.), G. 2009. The Ciao System. Ref. Manual (v1.13). Tech. rep., School of Computer Science, T.U. of Madrid (UPM). Available at <http://www.ciaohome.org>.
- BUENO, F., DERANSART, P., DRABENT, W., FERRAND, G., HERMENEGILDO, M., MALUSZYSKI, J., AND PUEBLA, G. 1997. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l WS on Automated Debugging—AADEBUG*. U. Linköping Press, 155–170.
- BUENO, F., GARCÍA DE LA BANDA, M., AND HERMENEGILDO, M. 1999. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM TOPLAS* 21, 2 (March), 189–238.

- BUENO, F., LÓPEZ-GARCÍA, P., AND HERMENEGILDO, M. 2004. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *FLOPS'04*. Number 2998 in LNCS. Springer-Verlag, 100–116.
- CABEZA, D. AND HERMENEGILDO, M. 1994. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *SAS'94*. Number 864 in LNCS. Springer-Verlag, 297–313.
- CABEZA, D. AND HERMENEGILDO, M. 1995. Distributed Concurrent Constraint Execution in the CIAO System. In *COMPULOG-NET'95*. U. Utrecht / T.U. Madrid, Utrecht, NL.
- CABEZA, D. AND HERMENEGILDO, M. 2000a. A New Module System for Prolog. In *International Conference CL 2000*. LNAI, vol. 1861. Springer-Verlag, 131–148.
- CABEZA, D. AND HERMENEGILDO, M. 2000b. The Ciao Modular, Standalone Compiler and Its Generic Program Processing Library. In *Special Issue on Parallelism and Implementation of (C)LP Systems*. ENTCS, vol. 30(3). Elsevier - North Holland.
- CABEZA, D. AND HERMENEGILDO, M. 2001. Distributed WWW Programming using (Ciao) Prolog and the PiLLoW Library. *TPLP* 1, 3 (May), 251–282.
- CARRO, M., GÓMEZ, L., AND HERMENEGILDO, M. 1993. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *ICLP'93*. MIT Press, 184–201.
- CARRO, M. AND HERMENEGILDO, M. 1999. Concurrency in Prolog Using Threads and a Shared Database. In *1999 International Conference on Logic Programming*. MIT Press, Cambridge, MA, USA, 320–334.
- CARRO, M., MORALES, J., MULLER, H., PUEBLA, G., AND HERMENEGILDO, M. 2006. High-Level Languages for Small Devices: A Case Study. In *Compilers, Architecture, and Synthesis for Embedded Systems*, K. Flautner and T. Kim, Eds. ACM Press / Sheridan, 271–281.
- CARTWRIGHT, R. AND FAGAN, M. 1991. Soft Typing. In *PLDI'91*. SIGPLAN, ACM, 278–292.
- CASAS, A., CABEZA, D., AND HERMENEGILDO, M. 2006. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *FLOPS'06*. 142–162.
- CASAS, A., CARRO, M., AND HERMENEGILDO, M. 2007. Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs. In *LOPSTR'07*. Number 4915 in LNCS. Springer-Verlag, 138–153.
- CASAS, A., CARRO, M., AND HERMENEGILDO, M. 2008. A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism. In *24th International Conference on Logic Programming (ICLP'08)*, M. García de la Banda and E. Pontelli, Eds. LNCS, vol. 5366. Springer-Verlag, 651–666.
- CHEN, W. AND WARREN, D. S. 1996. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM* 43, 1 (January), 20–74.
- CORREAS, J., GOMEZ, J. M., CARRO, M., CABEZA, D., AND HERMENEGILDO, M. 2004. A Generic Persistence Model for CLP Systems (And Two Useful Implementations). In *PADL'04*. Number 3057 in LNCS. Springer-Verlag, 104–119.
- COSTA, V. S., DAMAS, L., REIS, R., AND AZEVEDO, R. 2002. *YAP User's Manual*. <http://www.dcc.fc.up.pt/~vsc/Yap>.
- COUSOT, P. AND COUSOT, R. 1977. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77*. ACM, 238–252.
- DEBRAY, S. K. AND LIN, N. W. 1993. Cost analysis of logic programs. *ACM TOPLAS* 15, 5 (November), 826–875.
- DEBRAY, S. K., LIN, N.-W., AND HERMENEGILDO, M. 1990. Task Granularity Analysis in Logic Programs. In *Proc. PLDI'90*. ACM, 174–188.
- DEBRAY, S. K., LÓPEZ-GARCÍA, P., HERMENEGILDO, M., AND LIN, N.-W. 1997. Lower Bound Cost Estimation for Logic Programs. In *ILPS'97*. MIT Press.

- EL-KHATIB, O., PONTELLI, E., AND SON, T. C. 2005. Integrating an Answer Set Solver into Prolog: ASP-PROLOG. In *LPNMR*. 399–404.
- FRÜHWIRTH, T. 2009. *Constraint Handling Rules*. Cambridge University Press.
- GARCÍA DE LA BANDA, M., BUENO, F., AND HERMENEGILDO, M. 1996. Towards Independent And-Parallelism in CLP. In *PLILP'96*. Number 1140 in LNCS. Springer-Verlag, 77–91.
- GARCÍA DE LA BANDA, M., HERMENEGILDO, M., AND MARRIOTT, K. 2000. Independence in CLP Languages. *ACM TOPLAS* 22, 2 (March), 269–339.
- GÓMEZ-ZAMALLOA, M., ALBERT, E., AND PUEBLA, G. 2008. On the Generation of Test Data for Prolog by Partial Evaluation. In *Workshop on Logic-based methods in Programming Environments (WLPE'08)*. 26–43. Report number: WLPE/2008/06.
- GUPTA, G., PONTELLI, E., ALI, K., CARLSSON, M., AND HERMENEGILDO, M. 2001. Parallel Execution of Prolog Programs: a Survey. *ACM TOPLAS* 23, 4 (July), 472–602.
- HANUS ET AL, M. Curry: An Integrated Functional Logic Language. <http://www.informatik.uni-kiel.de/~mh/curry/report.html>.
- HERMENEGILDO, M. 1986. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *ICLP'86*. LNCS, vol. 225. Springer-Verlag, 25–40.
- HERMENEGILDO, M. 1997. Automatic Parallelization of Irregular and Pointer-Based Computations: Perspectives from Logic and Constraint Programming. In *EURO-PAR'97*. Number 1300 in LNCS. Springer-Verlag, 31–46.
- HERMENEGILDO, M. 2000. A Documentation Generator for (C)LP Systems. In *International Conference CL 2000*. LNAI, vol. 1861. Springer-Verlag, 1345–1361.
- HERMENEGILDO, M., ALBERT, E., LÓPEZ-GARCÍA, P., AND PUEBLA, G. 2004. Some Techniques for Automated, Resource-Aware Distributed and Mobile Computing in a Multi-Paradigm Programming System. In *EURO-PAR'04*. Number 3149 in LNCS. Springer-Verlag, 21–37.
- HERMENEGILDO, M. AND GREENE, K. 1991. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing* 9, 3,4, 233–257.
- HERMENEGILDO, M., PUEBLA, G., AND BUENO, F. 1999. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In *The Logic Programming Paradigm: a 25-Year Perspective*. Springer-Verlag, 161–192.
- HERMENEGILDO, M., PUEBLA, G., BUENO, F., AND GARCÍA, P. L. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming* 58, 1–2.
- HERMENEGILDO, M. AND ROSSI, F. 1995. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming* 22, 1, 1–45.
- HERMENEGILDO, M. AND THE CIAO DEVELOPMENT TEAM. 2006. Why Ciao? –An Overview of the Ciao System's Design Philosophy. Tech. Rep. CLIP7/2006.0, UPM. Available from: <http://cliplab.org/papers/ciao-philosophy-note-tr.pdf>.
- HERMENEGILDO, M., WARREN, R., AND DEBRAY, S. K. 1992. Global Flow Analysis as a Practical Compilation Tool. *JLP* 13, 4 (August), 349–367.
- HERMENEGILDO ET AL, M. 1994. 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. Springer-Verlag, 123–133.
- HERMENEGILDO ET AL, M. 1999. 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*. Nova Science, 65–85.
- HOLZBAUR, C. 1992. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *PLILP'92*. LNCS631, Springer Verlag, 260–268.
- HOLZBAUR, C. 1994. *SICSStus 2.1/DMCAI Clp 2.1.1 User's Manual*. University of Vienna.

- HUDAK, P., PEYTON-JONES, S., WADLER, P., BOUTEL, B., FAIRBAIRN, J., FASEL, J., GUZMAN, M. M., HAMMOND, K., HUGHES, J., JOHNSON, T., KIEBURTZ, D., NIKHIL, R., PARTAIN, W., AND PETERSON, J. 1992. Report on the Programming Language Haskell. *Haskell Special Issue, ACM Sigplan Notices* 27, 5.
- JIM, T., MORRISETT, J. G., GROSSMAN, D., HICKS, M. W., CHENEY, J., AND WANG, Y. 2002. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, C. S. Ellis, Ed. USENIX, 275–288.
- KARP, A. AND BABB, R. 1988. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software*.
- LAMPORT, L. AND PAULSON, L. C. 1999. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems* 21, 3 (May), 14.
- LEAVENS, G. T., LEINO, K. R. M., AND MÜLLER, P. 2007. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.* 19, 2, 159–189.
- LOGOZZO ET AL., F. Clousot. <http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx>.
- LÓPEZ-GARCÍA, P., BUENO, F., AND HERMENEGILDO, M. 2010. Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information. *New Generation Computing* 28, 2, 117–206.
- LÓPEZ-GARCÍA, P., DARMAWAN, L., AND BUENO, F. 2010. A Framework for Verification and Debugging of Resource Usage Properties. In *Technical Communications of ICLP*. LIPIcs, vol. 7. Schloss Dagstuhl, 104–113.
- LÓPEZ-GARCÍA, P., HERMENEGILDO, M., AND DEBRAY, S. K. 1996. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation* 21, 715–734.
- MERA, E., LÓPEZ-GARCÍA, P., CARRO, M., AND HERMENEGILDO, M. 2008. Towards Execution Time Estimation in Abstract Machine-Based Languages. In *PPDP'08*. ACM Press, 174–184.
- MERA, E., LÓPEZ-GARCÍA, P., AND HERMENEGILDO, M. 2009. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *ICLP'09*. Number 5649 in LNCS. Springer-Verlag, 281–295.
- MORALES, J., CARRO, M., AND HERMENEGILDO, M. 2004. Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In *PADL'04*. Number 3057 in LNCS. Springer-Verlag, 86–103.
- MORALES, J., CARRO, M., AND HERMENEGILDO, M. 2008. Comparing Tag Scheme Variations Using an Abstract Machine Generator. In *PPDP'08*. ACM Press, 32–43.
- MORALES, J., CARRO, M., AND HERMENEGILDO, M. 2009. Description and Optimization of Abstract Machines in a Dialect of Prolog. Technical Report CLIP4/2009.0, Technical University of Madrid (UPM), School of Computer Science, UPM. October.
- MOZILLA. 2008. Tamarin Project. Available at <http://www.mozilla.org/projects/tamarin/>.
- MUTHUKUMAR, K., BUENO, F., DE LA BANDA, M. G., AND HERMENEGILDO, M. 1999. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *JLP* 38, 2 (February), 165–218.
- MUTHUKUMAR, K. AND HERMENEGILDO, M. 1990. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *ICLP'90*. MIT Press, 221–237.
- MUTHUKUMAR, K. AND HERMENEGILDO, M. 1991. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *ICLP'91*. MIT Press, 49–63.
- MUTHUKUMAR, K. AND HERMENEGILDO, M. 1992. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *JLP* 13, 2/3 (July), 315–347.
- NAISH, L., DART, P. W., AND ZOBEL, J. 1989. The NU-Prolog Debugging Environment. In *Proceedings of ICLP'89*, A. Porto, Ed. MIT Press, 521–536.

- NAVAS, J., MERA, E., LÓPEZ-GARCÍA, P., AND HERMENEGILDO, M. 2007. User-Definable Resource Bounds Analysis for Logic Programs. In *ICLP'07*. Number 4670 in LNCS. 348–363.
- NECULA, G. C., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. 2005. Ccured: type-safe retrofitting of legacy software. *ACM TOPLAS* 27, 3, 477–526.
- OLMEDILLA, M., BUENO, F., AND HERMENEGILDO, M. 1993. Automatic Exploitation of Non-Determinate Independent And-Parallelism in the Basic Andorra Model. In *LOPSTR'93*. Workshops in Computing. Springer-Verlag, 177–195.
- PIETRZAK, P., CORREAS, J., PUEBLA, G., AND HERMENEGILDO, M. 2006. Context-Sensitive Multivariant Assertion Checking in Modular Programs. In *LPAR'06*. Number 4246 in LNCS. Springer-Verlag, 392–406.
- PIETRZAK, P., CORREAS, J., PUEBLA, G., AND HERMENEGILDO, M. 2008. A Practical Type Analysis for Verification of Modular Prolog Programs. In *PEPM'08*. ACM Press, 61–70.
- PINEDA, A. AND BUENO, F. 2002. The O'Ciao Approach to Object Oriented Logic Programming. In *CICLOPS'02*.
- PUEBLA, G., ALBERT, E., AND HERMENEGILDO, M. 2006. Abstract Interpretation with Specialized Definitions. In *SAS'06*. Number 4134 in LNCS. Springer-Verlag, 107–126.
- PUEBLA, G., BUENO, F., AND HERMENEGILDO, M. 2000a. A Generic Preprocessor for Program Validation and Debugging. In *Analysis and Visualization Tools for Constraint Programming*. Number 1870 in LNCS. Springer-Verlag, 63–107.
- PUEBLA, G., BUENO, F., AND HERMENEGILDO, M. 2000b. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*. Number 1870 in LNCS. Springer-Verlag, 23–61.
- PUEBLA, G., GARCÍA DE LA BANDA, M., MARRIOTT, K., AND STUCKEY, P. 1997. Optimization of Logic Programs with Dynamic Scheduling. In *ICLP'97*. MIT Press, 93–107.
- PUEBLA, G. AND HERMENEGILDO, M. 1995. Implementation of Multiple Specialization in Logic Programs. In *PEPM'95*. ACM Press, 77–87.
- SAGLAM, H. AND GALLAGHER, J. 1995. Approximating Constraint Logic Programs Using Polymorphic Types and Regular Descriptions. Technical Report CSTR-95-17, Dep. of Computer Science, U. of Bristol, Bristol BS8 1TR.
- SCHRIJVERS, T., COSTA, V. S., WIELEMAKER, J., AND DEMOEN, B. 2008. Towards Typed Prolog. In *ICLP'08*. Number 5366 in LNCS. Springer, 693–697.
- SCHULTE, C. AND STUCKEY, P. J. 2008. Efficient Constraint Propagation Engines. *TOPLAS* 31, 1 (December), 2:1–2:43.
- SOMOGYI, Z., HENDERSON, F., AND CONWAY, T. 1996. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *JLP* 29, 1–3 (October), 17–64.
- Swedish Institute for Computer Science 2009. *SICStus Prolog User's Manual*, 4.1.1 ed. Swedish Institute for Computer Science, PO Box 1263, S-164 28 Kista, Sweden. Available from <http://www.sics.se/sicstus/>.
- SZABÓ, P. AND SZEREDI, P. 2006. Improving the ISO Prolog Standard by Analyzing Compliance Test Results. In *ICLP'06*. 257–269.
- TOBIN-HOCHSTADT, S. AND FELLEISEN, M. 2008. The Design and Implementation of Typed Scheme. In *POPL*. ACM, 395–406.
- VAN HENTENRYCK, P. AND MICHAEL, L. 2005. *Constraint-Based Local Search*. MIT Press.
- VAUCHERET, C. AND BUENO, F. 2002. More Precise yet Efficient Type Inference for Logic Programs. In *SAS'02*. Number 2477. Springer-Verlag, 102–116.
- WARREN, D. 1993. Logic Programming Languages, Parallel Implementations, and the Andorra Model. *ICLP'93*.
- WARREN, R., HERMENEGILDO, M., AND DEBRAY, S. K. 1988. On the Practicality of Global Flow Analysis of Logic Programs. In *ICLP'88*. MIT Press, 684–699.
- WOLFRAM, D. 1992. A Semantics for λ Prolog. Technical report prg-tr-8-92, University of Oxford.