An Overview of the Ciao Multiparadigm Language and Program Development Environment and its Design Philosophy

M. V. Hermenegildo F. Bueno M. Carro
P. López J.F. Morales G. Puebla

IMDEA Institute for Software Development Technology
Universidad Politécnica de Madrid
University of New Mexico
Universidad Complutense de Madrid

Abstract. We describe some of the novel aspects and motivations behind the design and implementation of the Ciao multiparadigm programming system. An important aspect of Ciao is that 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 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 objects, predicates, 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 program optimizations. Such optimizations produce code that is highly competitive with other dynamic languages or, when the highest levels of optimization are used, even that of static languages, all while retaining the interactive development environment of a dynamic language. The environment also includes a powerful auto-documenter. 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 the format of a paper, pointing instead to the existing literature on the system.

1 Origins and Initial Motivations

Ciao [50,48,5,25,49] 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 which has fewer or no bugs and which also performs very efficiently on platforms from small embedded processors to powerful multicore architectures. Ciao has its main roots in the &-Prolog language and system [51]. &-Prolog's

design was aimed at achieving higher performance than state of the art sequential logic programming systems by exploiting (and-)parallelism. This required the development of a specialized abstract machine capable of running in parallel and very efficiently a large number of (possibly non-deterministic) goals [44,51] (the abstract machine was derived from early versions of SICStus Prolog). It also required extending the source language to allow expressing parallelism and concurrency in programs. This made it possible for the user to parallelize programs manually in a relatively simple way. The system was later extended to support constraint programming, including the concurrent and parallel execution of such programs [30]. Significant work was done with Ugo Montanari and Francesca Rossi in this context through the development of a true concurrency semantics that implied the possibility of exploiting truly maximum parallelism in the execution of constraint programs [68,10,11]. Additional work was also performed to extend the system to support other computation rules, such as the Andorra principle [90,7,79,9], other sublanguages, etc.[1]

The experience of this process of gradual extension of the capabilities of the &-Prolog system inspired some of the fundamental concepts underlying the Ciao system. In particular, while each of the functionalities mentioned above (Andorra execution, constraint programming, concurrent programming, etc.) was typically implemented up to that time by a separate (and complex) system comprising compiler, abstract machine, etc. we observed that all such extensions could in fact be supported efficiently within the same system provided the underlying machinery implemented a relatively limited set of basic constructs (a *kernel language*) [50,48], coupled with an easily programmable and modular way of defining new syntax and giving semantics to it in terms of that kernel language. This approach is, of course, not exclusive to Ciao, but in Ciao the facilities that enable building from a simple kernel are very explicitly available (and their use encouraged) from the system programmer level to the application programmer level.

[1] It is interesting to note that a great deal of this initial work on the design and implementation of Ciao occurred within the ACCLAIM EU project, in which the Andorra Kernel Language (AKL) and its successor, the Oz language [42], were also developed. The great group of people involved in the project, including Ugo Montanari, Seif Haridi, Gert Smolka, Peter VanRoy, David Warren, and many others resulted in a very fruitful collaboration that effectively gave birth to modern multi-paradigm languages. Within this collaborative context, Ciao took different paths to AKL and Oz in many aspects, including for example the use of assertions and global analysis support, the fact that in Ciao non-determinism (backtracking) is implicit, or the use of a (Prolog-derived) syntax aimed at easily supporting meta-programming. As another example, in Ciao the language has always been sequential by default, i.e., concurrency has to be added explicitly by the user (or the parallelizer), whereas in the original Oz and AKL designs the language was concurrent by default (although this has been changed in later Oz designs). Interesting aspects of Oz include for example the extensive development and use of computational spaces as first-level constructs.

This is one of the fundamental capabilities of the Ciao system, which effectively allows Ciao to support multiple programming paradigms and styles. In Ciao all operators, "builtins," and most other syntactic and semantic language constructs such as conditionals or loops are not part of a predefined "language." Instead they are user-modifiable constructs living in *libraries* which can be loaded or unloaded at will thanks to the notion of *packages* [15]. This is the mechanism which allows adding new syntax to the language and giving semantics to this syntax. Most importantly, such packages, and thus the restrictions and extensions to the language that they provide, can be activated or deactivated separately on a per-module/class basis without interfering with each other.[2] The different source-level constructs (and sub-languages / DSLs) are supported by a compilation process defined within the corresponding package, typically via a set of rules defining *source-to-source* transformation into the kernel language, with the (rather infrequent) help of modules or classes written in an external language using one of the several interfaces provided. The approach of compiling to a common kernel implies that the programming styles that Ciao implements share much at both the semantic and implementation levels, and they naturally reuse significant portions of the compiler, documenter, abstract machine, etc.

Another fundamental characteristic of the Ciao system is that it provides a powerful preprocessor, called CiaoPP [8,52,53], which is capable of statically finding non-trivial bugs, verifying that the program complies with specifications, and performing many types of program optimizations. A key ingredient for the above task is the Ciao assertion language [82]. While not strictly required for developing or compiling programs, the preprocessor and assertion language are important and distinctive components of the Ciao design and they also have their origin in earlier work stemming from &-Prolog. In particular, the &-Prolog compiler included a parallelizer which was capable of automatically annotating programs for parallel execution [51,74,71]. This required developing advanced program analysis technology based on abstract interpretation [27] (leading to the development of the PLAI analyzer [91,73,55,76]) which allowed inferring program properties such as independence among program variables [73,75], absence of side effects [72], non-failure [36], data structure shape and instantiation state ("moded types") [87], or even being able to infer upper and lower bounds on the sizes of data structures and the cost of procedures [33,32,37,34], which was instrumental for performing automatic granularity control [33,66,65]. Also, in addition to automatic parallelization the &-Prolog compiler performed other optimizations such as multiple (abstract) specialization [84]. While the &-Prolog inference technology was aimed at performing program optimizations to maximize execution speed and minimize resource consumption, interacting with the system it soon became clear that the wealth of information inferred by the analyzers would also be very useful as an aid in the program development process, and this led to the idea of the Ciao assertion language and preprocessor, as we will discuss later.

[2] In fact, some Ciao packages are intended to be portable so that they can be used with little modification in other logic and constraint logic programming systems.

```
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. 1. Some examples in Ciao functional notation.

2 Supporting Multiple Paradigms

We will now show some examples of how the extensibility of the kernel language mentioned before allows Ciao to incorporate the fundamental constructs from a number of programming paradigms. In particular, the system currently offers, as a combination of syntactic and semantic extensions, the following programming models:

– *Functional Programming:* functional notation is provided by a set of packages which, besides a convenient syntax to define functions (or predicates using a function-like layout), gives support for semantic extensions which include higher-order facilities (e.g., function abstractions and applications thereof) and, if so required, lazy evaluation. For illustration, Figure 1 lists a number of examples using the Ciao functional notation. `nrev` and `conc` are written by using multiple `:=/2` definitions. `fact` is written using a disjunction of guards (which actually commits the system to the first matching choice). The `~` prefix (eval, which can often be omitted) is the opposite of quote and states that its argument is a call (as opposed to a data structure to unify with). All of this syntax is defined in the `functional` package, which is loaded into the module (line 1). `nums_from` is declared lazy, which is possible thanks to the `lazy` package, also loaded into the module. An important point is that these packages only modify the syntax and semantics of this module, and other modules can use any other packages. Finally, `nums` uses `take` from the library of lazy functions/predicates.
In general, functional notation is just syntax and thus the following query (loading the functional package in the top level allows using functional notation –the top level behaves in this sense exactly as a module):

```
:- module(_, _, [functional, hiord, bfall]).

color := red | blue | green.

list := [] | [_ | list].

list_of(T) := [] | [~T | list_of(T)].
```

```
:- module(_, _, [hiord, bfall]).

color(red). color(blue). color(green).

list([]).
list([_|T]) :- list(T).

list_of(_, []).
list_of(T, [X|Xs]) :- T(X), list_of(T, Xs).
```

Fig. 2. Examples in Ciao functional notation and state of translation after applying the `functional` package.

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

produces the answer:

```
X = [1,2,3]
```

As mentioned before other constructs such as conditionals do commit the system to the first matching case. More 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.) can be enforced using *assertions*, to be discussed later. Figure 2 lists more examples using `functional` and other packages, and the result after applying just the transformations implied by the `functional` package. Note the use of higher order in `list_of`. More details on Ciao's functional notation can be found in [21].

– *Logic Programming Flavors:* a set of packages (which are loaded by default when a Prolog module is read in) provide support for full ISO-Prolog and a number of other classical "builtins" expected by users to be provided by Prolog systems —except that of course in Ciao rather than builtins all of them are optional features brought in from the libraries.[3] This is signaled by

[3] The support of Prolog is done in such a way that Prolog code runs without modification, and the system top level comes up by default in Prolog mode. As a result, many Ciao users which come to the system looking for a good Prolog implementation do get what they expect and, if they do not poke further into the menus and manuals, may never realize that Ciao is in fact quite a different beast.

simply not using the third argument (the one devoted to listing packages) in module declarations.
Alternatively, by avoiding the loading of the Prolog packages the user can restrict the module to use only pure logic programming, without any of Prolog's impure features. For example, the second listing in Figure 2 is a pure logic programming module. If a call to a Prolog builtin such as `assert` were to appear within the module it would be signaled by the compiler as calling 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 on a needed basis.
Higher-order logic programming with predicate abstractions is supported through the `hiord` package. This is also illustrated in the second listing in Figure 2. 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 predicate abstraction `_(X,Y) :- Y = f(X)`, `map([1, 3, 2], P, R)` is applied to `P` producing:

```
R = [f(1), f(3), f(2)]
```

The (reversed) query:

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

produces:

```
M = [1, 3, 2]
```

– *Additional Computation Rules:* In addition to the usual depth-first, left-to-right execution regime of Prolog, and again by loading suitable packages, other computation rules such as breadth-first, iterative deepening, Andorra model, etc. are available. As an example in the second listing in Figure 2 any calls to `color`, `list`, and `list_of` will be executed breadth-first.[4]

[4] The possibility of using different control rules has shown useful not only in applications but also (and very specially) when teaching logic programming. In our experience, it is cumbersome to make the first introductory lectures to logic programming using Prolog since the particular (albeit often practically useful) quirks and the subsequent non-termination of Prolog get in the way of teaching the fundamental concepts of logic programming. We have found that it makes perfect sense to start with a purer logic language, with better termination and fairness characteristics. The Ciao breadth-first mode has proved quite useful for this (see `http://www.cliplab.org/logalg` for the slides of our course based on this approach). Once the beauty of pure logic programming is experienced the student is then introduced to the practical and powerful choices made in the design of Prolog, and later to topics and functionality beyond Prolog, such as those outlined in this document, all within the same system.

```
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. 3. Ciao constraints (combined with functional notation).

Tabling [24] is currently being added using an approach which relies mostly on a source-to-source program transformation (which is performed using a package) and an external C library which is accessed using one of the available foreign interfaces [29]. The underlying abstract machine did not have to be changed, and therefore sequential execution is left essentially untouched.

– *Constraint Programming:* several constraint solvers and classes of constraints using these solvers are supported including CLP($\mathcal{Q}$), CLP($\mathcal{R}$) (a derivation of [57]), and CLP($\mathcal{FD}$). The constraint languages and solvers, which are built on more basic blocks such as attributed variables [56] and/or the higher-level Constraint Handling Rules (CHR) [39,88], are also extensible at the user level. As an example, Figure 3 provides two examples using Ciao CLP(Q) constraints, combined with functional notation. For example, line 3 can be read as: if the 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:

```
X = [] ? ;

X = [_] ? ;

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

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

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

```
_B .<. _C,
_A .<. _B ?
```

etc.

- *Object-Oriented Programming:* object-oriented programming is provided by the O'Ciao `class` and `object` packages [80]. 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.
- *Concurrency, Parallelism, and Distributed Execution:* other packages bring in extensive capabilities for expressing concurrency (including a concurrent, shared version of the internal fact database which can be used for synchronization), distribution, and parallel execution [14,11,23]. 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 characteristics that are specific to certain programming paradigms, many other additional features are available through libraries such as, e.g.:

- Structures with named arguments (feature terms), a trimmed-down version of ψ-terms [2] which makes it possible to compile statically all structure unifications to Prolog unifications, which ensures that using them adds no overhead to the execution.
- Persistence, which makes it possible to transparently save and restore the state of selected facts of the dynamic database of a program on exit and startup. This is the base of a high-level interface with databases [26].
- Answer set programming [38], an alternative logic programming model based on the *stable model semantics* [40].
- WWW programming, where Ciao provides libraries to easily establish a mapping between HTML / XML and Herbrand terms, to easily handle (generate / transform / inspect / ...) them in order to for example, write CGIs (or complete web sites) quite easily Ciao [17].

Again, all of these can be activated or deactivated on a *per-module / class* basis.

3 The Ciao Approach to Assertions

Many languages (e.g. Mercury [89,43] or Haskell [59,58], to cite some modern, well-known examples from the logic and functional programming communities) impose certain type-related requirements, e.g., all types (and, when relevant, modes) used have to be defined explicitly or all procedures have to be "well-typed" and "well-moded".

One argument in favor of such declarations and restrictions is that they can be useful to clarify interfaces and meanings, and in general to make large programs more maintainable and well documented, facilitating "programming in

the large." Besides, the compiler may use them to generate more specific code, which can be better in several ways (e.g., performance-wise).

We certainly agree with this! But at the same time we also wanted Ciao to be useful (as, say, Prolog, Scheme, or, more recently, Python) for "programming in the small," prototyping, developing simple scripts, or simply experimenting while trying to find a solution to a problem, ... and for this we feel type and mode declarations and other related restrictions can sometimes get in the way.

Fortunately, we came up with a unique solution to this apparent conundrum: Ciao includes a very rich assertion language (and a methodology for dealing with such assertions) [52,82] which allows expressing not only classical types, but also a much wider variety of properties (modes, determinacy, non-failure, cost, ...), but in Ciao *these assertions are optional*. This solution makes Ciao very useful both for programming in the small and in the large. We believe that Ciao's solution to the issue of assertions combines effectively the advantages of the strongly typed and untyped language approaches, bringing the best of both worlds to the programmer, but within a broader scope which, as we will see, makes it possible to use a uniform language to express more program properties (and, therefore, to interact with a tool able to check or infer / reconstruct them). This is, in some sense, related to the *soft typing* approach, pioneered in [20], but it differs from it in that it is not restricted to types. Instead, the framework is open regarding the kind of properties that can be expressed in the corresponding assertions. As an example, systems which aim at performing automatic compile-time checking are often rather strict about the properties which the user can write in assertions. This is understandable because otherwise, the underlying static analyses are of little use for proving the assertions. In our case, we use the same assertion language for different purposes, including run-time checking. Therefore, the user may use properties which go beyond those which the static analyses in the system can prove. Of course, even though such assertions may sometimes be useful at compile-time for certain purposes, the user cannot expect CiaoPP to automatically always be able to verify such assertions statically.

As an example, Figure 4 includes the definitions of `nrev` and `conc` (similarly to Figure 1) but also states some program properties expressed using the Ciao assertion language (whose syntax and semantics are made available to the module by means of the `assertions` package). For example, the assertion in line 4 expresses that when `nrev` is called (`:`) the first argument should be a list, and the second one should be a list on success (`=>`). The `+` field in `comp` assertions can contain a conjunction of global properties of the *computation* of the predicate (as the one in line 7). The predicates which are used in such assertions can be in libraries (such as the `nativeprops` library used in the figure) or defined by the user. For example, the definitions in Figure 2 can be used as types in assertions (e.g., line 4 in Figure 4).

```
:- module(_, [nrev/2], [assertions, fsyntax, nativeprops]).
:- entry nrev/2 : {list, ground} * var.

    :- pred nrev(A, B)  : list(A) => list(B).
    :- success nrev(A, B) => size_o(B, length(A)).
    :- comp nrev(_, _)  + (not_fails, is_det).
    :- comp nrev(A, _)  + steps_o(length(A)).


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

    :- comp conc(_, _, _) + (terminates, non_det).
    :- comp conc(A, _, _) + steps_o(length(A)).

conc([],    L) := L.
conc([H|L], K) := [ H | ~conc(L,K) ].
```

Fig. 4. Naive reverse with some –partially erroneous– assertions.

4 Program Documentation, Static Debugging, and Verification

One of the most useful characteristics of the assertions used in Ciao is that they are designed to serve many purposes. First, any assertions present in programs can be processed by an autodocumenter (`lpdoc` [46]) in order to generate useful documentation. Also, assertions are analyzed interactively during program development by the system preprocessor (CiaoPP) which can *find non-trivial bugs* statically, *verify* that the program complies with the assertions, or even generate automatically proofs of correctness that can be shipped with programs and checked easily at the receiving end (using the *proof/abstraction carrying code* approach [3]). Even if a program contains no user-provided assertions, Ciao can check the program against the assertions contained in the libraries used by the program, thus potentially catching additional bugs at compile time. If the system cannot prove nor disprove some property at compile time, the system can (optionally again) introduce a run-time check for such property in the executable. For homogeneity, and to ease information exchange among the autodocumenter and the different checkers and analyzers, analysis results are reported using also the assertion language —which, since it is readable by humans, can be inspected by a programmer, for example to make sure that the results of the analyses agree with the intended meaning of the program.

Interestingly, the same underlying technology (global analysis based on abstract interpretation) that allows the system to obtain useful results even when assertions are not present for all predicates, also allows dealing with complex properties, beyond classical types, in a safe way. As a result, for example,

```
:- module(qsort, [qsort/2], [assertions, fsyntax]).
:- use_module(compare, [geq/2, lt/2]).
:- entry qsort/2 : {list(num), ground} * var.

qsort([])     := [].
qsort([X|L]) := ~append(~qsort(L1), [X|~qsort(L2)])
              :- partition(L, X, L1, L2).

append([],     X) := X.
append([H|X], Y) := [H | ~append(X,Y)].

partition([],_B,[],[]).
partition([E|R],C,[E|Left1],Right) :-
        lt(E,C),  partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]) :-
        geq(E,C), partition(R,C,Left,Right1).
```

Fig. 5. A modular `qsort` program.

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 [37,35]) which the system will try to verify or falsify, thus performing automatic debugging and validation of the *performance* of programs. Many other interesting properties of the predicates and literals of the program can be handled, such as data structure shape (including pointer sharing), bounds on data structure sizes, and other operational variable instantiation properties, as well as procedure-level properties such as determinacy [63], non-failure [12,36], termination, and bounds on the execution time [67], as well as on the consumption a large class user-defined resources [77].

Assertions also allow programmers to describe the relevant properties of modules or classes which are not yet written or are written in other languages. This is also done in other languages but often using different types of assertions for each purpose. In contrast in Ciao the same assertion language is used again for this task. This, interestingly, makes it possible to run checkers / verifiers / documenters against code which is only partially developed: the traditional "stubs", which have to be changed later on for a working version, can be replaced by an assertion declaring how the predicate should behave, with the advantage that this declared behavior can effectively be checked against its uses.

We will now present some examples which illustrate how these capabilities are used in practice, and which also help introduce some aspects of the assertion language. The first example will illustrate automatic inference of non-trivial code properties while the second will focus on the use of assertions in verification and debugging, and more specifically to detect problems in the expected performance of a program.

As mentioned before, CiaoPP includes a non-failure analysis, based on [36] and [12], 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 based on [63], which can detect predicates which produce at most one solution, 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). We will use the code in Figure 5. The aforementioned analyses infer different types of information which include, among others, that expressed by the following assertion:

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

which expresses that, if `qsort(A, B)` is called with a list of numbers in `A` and a variable in `B`, then `B` will on exit be a list of numbers and the predicate will not fail, will give at most one solution, and will not perform backtracking at the level of its clauses (the `+` field in `pred` assertions can contain a conjunction of global properties of the *computation* of the predicate.)

CiaoPP can also infer lower and upper bounds on the sizes of terms and the computational cost of predicates [37,35]. The cost bounds are expressed as functions on the sizes of the input arguments and yield the number of resolution steps. Various measures can be used for the "size" of an input, such as list length, term size, term depth, integer value, etc. 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 upper bounds analysis:

```
:- true pred append(A,B,C)
         : ( list(A,num), list1(B,num), var(C) )
        => ( list(A,num), list1(B,num), list1(C,num),
             size_ub(A,length(A)), size_ub(B,length(B)),
             size_ub(C,length(B)+length(A)) )
         + steps_ub(length(A)+1).
```

Note that in this example the size measure used is list length. The property `size_ub(C,length(B)+length(A)` means that an (upper) bound on the size of the third argument of `append/3` is the sum of the sizes of the first and second arguments. The inferred upper bound on computational steps is the length of the first argument of `append/3`.

The following is the output of the lower-bounds analysis:

```
:- true pred append(A,B,C)
         : ( list(A,num), list1(B,num), var(C) )
        => ( list(A,num), list1(B,num), list1(C,num),
             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) ).
```

The lower-bounds analysis uses information from the non-failure analysis, without which a trivial lower bound of 0 would be derived.

As a second example, we illustrate how in CiaoPP it is possible to state assertions about the efficiency of the program which the system will try to verify or falsify, thus implementing a form of performance debugging and validation. This is done by specifying lower and/or upper bounds on the computational cost of predicates (given in number of execution steps). Consider for example again the naive reverse program in Figure 4. The assertion in line 7 states that `nrev` should be linear in the length of the (input) argument `A`. CiaoPP can be used to verify (or disprove) this assumption by running the analyzer (as before) to infer bounds on costs and then comparing them with the assertion. In fact, `nrev` is of course quadratic. With compile-time error checking turned on, and mode, type, non-failure and lower-bound cost analysis selected, CiaoPP issues the following error message:

```
ERROR: false comp assertion:
          :- comp nrev(A,B) : true => steps_o(length(A))
       because in the computation the following holds:
          steps_lb(0.5*exp(length(A),2)+1.5*length(A)+1)
```

This message states that `nrev` will take at least $\frac{length(A)^2+3\ length(A)}{2} + 1$ resolution steps (which is the cost analysis output), while the assertion requires the cost to be in $O(length(A))$ 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. This allows detecting the inconsistency and proving that the program does not satisfy the efficiency requirements imposed. 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 [8]. Thanks to this functionality, CiaoPP can also certify programs with resource consumption assurances as well as efficiently checking such certificates [47].

5 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, we do not want to put 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. 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

Fig. 6. Headset with a Gumstix processor (left) and 3-D compass (right).

performed at the kernel language level, so that the same analyzers are used for several of the supported programming models. The information inferred by the global analyzers is used to perform optimizations, including multiple abstract specialization [85], partial evaluation [81], dead code removal, goal reordering, reduction of concurrency / dynamic scheduling [83], low-level optimization (including optimized compilation to native code via C), and others [53].

The objective is again to achieve the best of both worlds: with no assertions or analysis information the low-level Ciao compiler (`ciaoc` [16]) generates code which is highly 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 optimizing compiler (see, e.g., [69] for an early description) can produce code that is competitive with that coming from strongly-typed systems. Ciao's highly optimized compilation has been successfully tested for example in applications with tight resource usage constraints (including real-time) [19], obtaining a 7-fold speed-up w.r.t. the default bytecode compilation (the performance of which is similar to that of state of the art abstract machine-based systems). The application in hand was real-time spacial placement of sound sources for a virtual reality suit and ran in a small ("Gumstix") processor embedded within a headset (Figure 6). It is interesting to note that this level of performance is only around 20-40% slower than a comparable implementation in C of the same application.

A particularly interesting optimization performed by CiaoPP, and which is inherited from the &-Prolog system, is *automatic parallelization* [45,41]. This is specially relevant nowadays given that the wide availability of multicore processors has made parallel computers mainstream. We illustrate this by means of a simple example using goal-level program parallelization [6,22]. This optimization is performed as a source-to-source transformation, in which the input program is *annotated* with parallel expressions. The parallelization algorithms, or annotators [71], exploit parallelism under certain *independence* conditions, which allow

guaranteeing interesting correctness and no-slowdown properties for the parallelized programs [54,31]. This process is made more complex by the presence of variables shared among goals and pointers among data structures at run-time. Let us consider again the program in Figure 5. A possible parallelization (obtained in this case with the "MEL" annotator [71]) is:

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

which indicates 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 Figure 5 need their arguments to be ground (note that this may be either inferred by analyzing the implementation of `lt/2` and `geq/2` or by stated by the user using suitable assertions), the information inferred 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 test and the corresponding conditional is simplified via abstract executability and the annotator yields instead the following code:

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

which is much more efficient since it has no run-time test. This test simplification process is described in detail in [6] where the impact of abstract interpretation in the effectiveness of the resulting parallel expressions is also studied.

The tests in the above example aim at *strict* independent and-parallelism. However, the annotators are parametrized on the notion of independence. Different tests can be used for different independence notions: non-strict independence [13], constraint-based independence [31], 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.[5]

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* [65] of parallelized code. Such parallel code can be the output of the process mentioned above or code parallelized manually. In general,

[5] For example, stream and-parallelism can be seen as independent and-parallelism if the independence of "bindings" rather than goals is considered.

this run-time granularity control process involves 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. Optimizations to this general process include cost function simplification and improved term size computation [64].

6 Incremental Compilation: 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/class system [15,80]. This design is the real enabler of Ciao's modular program development support 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 part of the application modules after one of them is changed, without any need to define "makefiles" or similar dependency-related additional files.

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 Windows, Linux, Mac Os X, and many other Un*x-based OSs.

Modular distribution of user and system code in Ciao, coupled with modular analysis, allows the generation of stripped executables, with only those builtins and libraries used by the program. Those reduced-size executables permit 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 allowed, by means of compiler and abstract machine support for multiple bidirectional foreign interfaces to C/C++, Java, Tcl/Tk, SQL databases (with a notion of predicate persistence), etc. The interfaces are expressed in (and any necessary glue code automatically generated from) descriptions written in the assertion language, as previously stated.

7 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 fulfill the objective of allowing 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

Fig. 7. A program file and the top-level interpreter.

source-level debugger with breakpoints, and several execution visualization tools. In addition, a plugin with very similar functionality is also available for the Eclipse environment.

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. Figure 7 shows a source file (with syntax highlighting, top level, menus, buttons, etc.). Tracing the execution in the debugger makes the current statement in the program be highlighted in an additional buffer containing the debugged file (Figure 8).

The environment provides also automated access to the documentation, extensive syntax highlighting, auto-completion, or auto-location of errors in the source, and is highly customizable (to set, for example, alternative installation directories or the location of some binaries). Figure 9 shows the location in the source of a simple syntactic error. The direct access to the preprocessor allows interactive control of all the static debugging, verification, and program transformation facilities. As an example, Figure 10 shows CiaoPP signaling in the source a semantic error (in the same way as the previous simple syntactic er-

Fig. 8. The source debugger in action.

ror). In particular, it is the cost-related error discussed previously in which 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 [46] allows using a pair of keystrokes to generate 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 in order to build documentation incrementally and to make sure that, for example, cross references between files are are well resolved and that the documentation itself is well structured and formatted. As a further example of the different components and capabilities of the environment, Figure 11 shows a VisAndOr [18] depiction of an and-parallel execution.

Fig. 9. Error location in the source –a simple syntactic error.

8 Some Final Thoughts on Parallelism, Dynamic Languages, and Mainstream Programming

Interestingly many of the motivations behind the development of Ciao over the years have acquired presently even more crucial importance. Parallelism capabilities are becoming ubiquitous thanks to the widespread use of multi-core processors. Indeed, most laptops on the market contain two cores (typically capable of running up to four threads simultaneously) and single-chip, 8-core servers are now in widespread use. Furthermore, the trend is that the number of on-chip cores will double with each processor generation. In this context, being able to exploit such parallel execution capabilities in programs as easily as possible becomes more and more a necessity. However, it is well-known [61] that parallelizing programs is a hard challenge. This has renewed interest in language-related designs and tools which can simplify the task of producing parallel programs.

At the same time, 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-

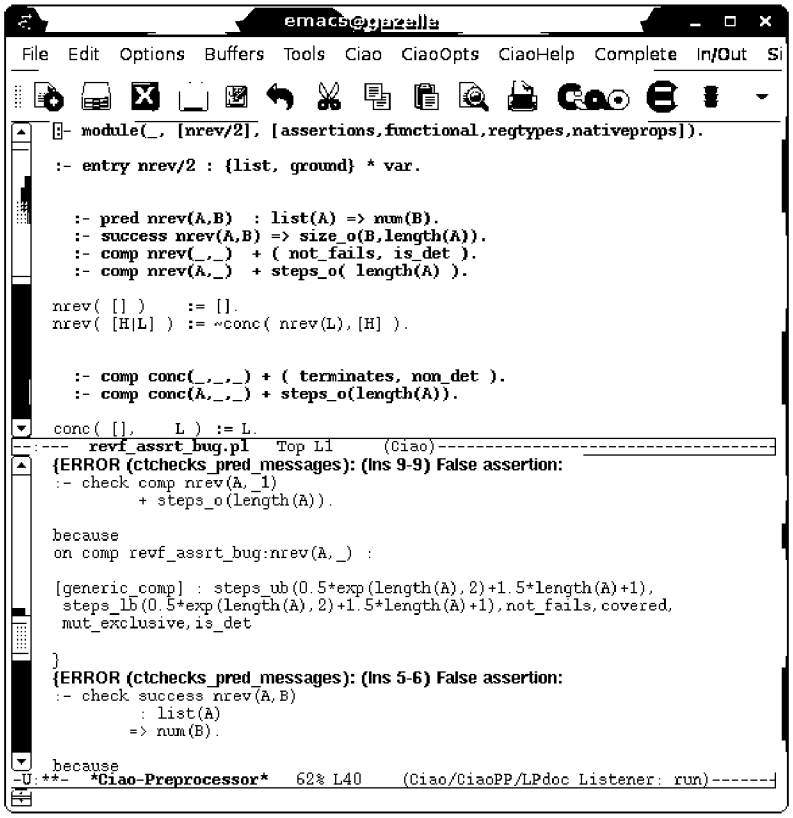

Fig. 10. Error location in the source –a cost error.

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 need to have flexibility in their APIs because they need to have a variable number of arguments or their "entry points" evolve over time in an asynchronous, decentralized fashion (e.g., services, including web services). These requirements, coupled with their intrinsic agility in development, have made dynamic programming languages (such as Python, Ruby, Lua, JavaScript, Perl, PHP, etc.) a very attractive option in recent years for a number of purposes that go well beyond simple scripting. Parts written in these languages often become essential components (if not the central implementation vehicle) of mainstream applications. The practical relevance of dynamic features is also illustrated by the many successful languages and frameworks which aim at bringing together ideas of both worlds. For example, Objective-C [28], which mixes C, object orientation, and the possibility of having dynamically typed

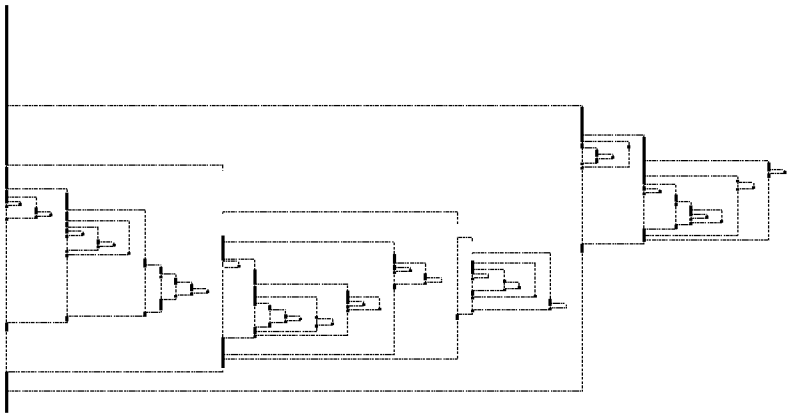

Fig. 11. VisAndOr depiction of an and-parallel execution of QuickSort.

variables and messages, is currently used as the base of Mac OS X, and it was used before in NextStep. Other frameworks, such as Java and .NET, are intensely working on ensuring and improving the interoperability among dynamic and static languages by including support for dynamicity in their virtual machines. Another example is the future fourth revision of ECMAScript [1] on which the JavaScript and ActionScript languages are based, that will include optional (soft-)type declarations to allow the compiler to generate more efficient code and detect more errors. The Tamarin project [70] intends to use this additional information to generate faster code. For Python, the PyPy project [86] designed a language, RPython [4] that imposes constraints on the programs to ensure that they can be statically typed. RPython is moving forward as a general purpose language.

At the same, detecting errors at compile-time and inferring properties required to optimize programs, are still important issues in real-world applications. This has also brought the development of safe versions of traditional languages, such as, e.g., CCured [78] or Cyclone [60] for C, as well as of systems that offer capabilities similar to those of the Ciao assertion preprocessor, such as Necula et al.'s Deputy [6] or Leino et al.'s Spec# [62].

We believe that Ciao has pioneered and is continuing to push the state of the art in these currently very relevant and challenging areas, and offers a unique combination of features which directly address many of these challenges. The Ciao approach to exploiting parallelism provides powerful parallelizers and at the same time allows programmer and parallelizer to cooperate. Programmers can choose between expressing manually the parallelism with high-level constructs, letting the compiler discover the parallelism, or a combination of both. Parts of a program can be parallelized by hand and other parts automatically. Furthermore, the parallelizer also checks manual parallelizations for correctness.

[6] `http://deputy.cs.berkeley.edu/`

Finally, the output of the parallelizer is expressed in the same high level language, which means that programmers can easily inspect (and improve) the parallelizations produced by the compiler. At the heart of these capabilities are CiaoPP's powerful, modular, and incremental abstract interpretation-based program analyzers. The use of this technology was pioneered by &-Prolog and Ciao (it was arguably the first use of abstract interpretation in a real compiler) and we continue to believe it is the most promising nowadays, and they are being adopted or will be adopted by many systems (see, e.g., [45] for further discussion of this topic).

Regarding the conundrum between statically and dynamically checked languages, Ciao has also pioneered and continues to push the state of the art of 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 sophisticated assertion processing based on 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. It also allows being able to work in a seamless way with a large class of properties, some of them even user-defined, and which go well beyond traditional types. Again, at the heart of these capabilities are CiaoPP's abstract interpretation-based analyzers.

Finally, we also believe that Ciao's language design offers unique possibilities due to its simple and powerful extensibility features, which not only allow to selectively bring in the constructs of multiple programming paradigms, but also make it possible for the programmer to easily extend (and restrict) the language as needed, syntactically and semantically, and to quickly design domain-specific languages.

Probing Further

The reader is encouraged to explore the system, its documentation, and the tutorial papers that have been published on it. We are currently working on the new 1.14 system version which includes significant enhancements with respect to the previous version (1.10). In addition to the autodocumenter, we plan to include a beta version of the preprocessor in the default Ciao distribution (up to now, CiaoPP was only distributed on demand and installed separately). Ciao 1.14 is available already on demand from the Ciao subversion repository.

But, Why is it Called Ciao?

After reading the previous paragraphs the reader may have already seen the logic behind the "Ciao Prolog" phrase. Ciao is an interesting word which is used both to say *hello* and *goodbye*. Ciao intends to be a truly excellent, high-performance, and freely available ISO-Prolog system which can be used as a classical Prolog, in both academic and industrial environments (and, in particular, to introduce users to Prolog and to constraint and logic programming –the *hello* Prolog part). But Ciao is also a new-generation, multiparadigm programming language and

sophisticated program development environment for large, complex applications which goes well beyond Prolog and other classical logic programming languages –the *goodbye* Prolog part. And it has the advantage (when compared to other modern systems that support different forms of logic programming) that it does so while keeping full Prolog compatibility when desired.

Contact / download info

`http://www.ciaohome.org`	The Ciao Development Team
`http://www.cliplab.org`	Technical U. of Madrid, Spain
`ciao@clip.dia.fi.upm.es`	U. of New Mexico, USA
	U. Complutense de Madrid, Spain
	IMDEA-Institute for Software Development Technology

References

1. ECMA 2008. ECMAscript Edition 4 Specification Wiki, 2008. Available at `http://wiki.ecmascript.org`.
2. Hassan Aït-Kaci. An Introduction to LIFE – Programming with Logic, Inheritance, Functions and Equations. In *Proceedings of the 1993 International Symposium on Logic Programming*, 1993.
3. E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.

4. Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a Step towards Reconciling Dynamically and Statically Typed OO Languages. In *DLS '07: Proceedings of the 2007 Symposium on Dynamic Languages*, pages 53–64, New York, NY, USA, 2007. ACM.
5. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.
6. F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM TOPLAS*, 21(2):189–238, March 1999.
7. F. Bueno, S. K. Debray, M. García de la Banda, and M. Hermenegildo. Transformation-based Implementation and Optimization of Programs Exploiting the Basic Andorra Model. Technical Report CLIP11/95.0, Facultad de Informática, UPM, May 1995.
8. F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.
9. F. Bueno and M. Hermenegildo. An Automatic Translation Scheme from Prolog to the Andorra Kernel Language. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems*, volume 2, pages 759–769. Institute for New Generation Computer Technology (ICOT), June 1992.
10. F. Bueno, M. Hermenegildo, U. Montanari, and F. Rossi. From Eventual to Atomic and Locally Atomic CC Programs: A Concurrent Semantics. In *Fourth International Conference on Algebraic and Logic Programming*, number 850 in LNCS, pages 114–132. Springer-Verlag, September 1994.
11. F. Bueno, M. Hermenegildo, U. Montanari, and F. Rossi. Partial Order and Contextual Net Semantics for Atomic and Locally Atomic CC Programs. *Science of Computer Programming*, 30:51–82, January 1998. Special CCP95 Workshop issue.
12. F. Bueno, P. López-García, and M. Hermenegildo. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *FLOPS'04)*, pages 100–116. Springer LNCS 2998, 2004.
13. D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994. Springer-Verlag.
14. D. Cabeza and M. Hermenegildo. Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the 1995 COMPULOG-NET Workshop on Parallelism and Implementation Technologies*, Utrecht, NL, September 1995. U. Utrecht / T.U. Madrid. Available from `http://www.cliplab.org/`.
15. D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.
16. D. Cabeza and M. Hermenegildo. The Ciao Modular, Standalone Compiler and Its Generic Program Processing Library. In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.
17. D. Cabeza and M. Hermenegildo. Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library. *Theory and Practice of Logic Programming*, 1(3):251–282, May 2001.

18. M. Carro, L. Gómez, and M. Hermenegildo. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *1993 International Conference on Logic Programming*, pages 184–201. MIT Press, June 1993.
19. M. Carro, J. Morales, H.L. Muller, G. Puebla, and M. Hermenegildo. High-Level Languages for Small Devices: A Case Study. In Krisztian Flautner and Taewhan Kim, editors, *Compilers, Architecture, and Synthesis for Embedded Systems*, pages 271–281. ACM Press / Sheridan, October 2006.
20. Robert Cartwright and Mike Fagan. Soft Typing. In *Programming Language Design and Implementation (PLDI 1991)*, pages 278–292. SIGPLAN, ACM, 1991.
21. A. Casas, D. Cabeza, and M. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *FLOPS'06*, Fuji Susono (Japan), April 2006.
22. A. Casas, M. Carro, and M. Hermenegildo. Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, number 4915 in LNCS, pages 138–153, The Technical University of Denmark, August 2007. Springer-Verlag.
23. A. Casas, M. Carro, and M. Hermenegildo. Towards a High-Level Implementation of Execution Primitives for Non-restricted, Independent And-parallelism. In D.S. Warren and P. Hudak, editors, *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, volume 4902 of *LNCS*, pages 230–247. Springer-Verlag, January 2008.
24. Weidong Chen and David S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.
25. The Ciao Development Team. The Ciao Multiparadigm Language and Program Development Environment, November 2006. The ALP Newsletter 19(3). The Association for Logic Programming. Available from `http://www.logicprogramming.org/newsletter/nov06/index.html`.
26. J. Correas, J. M. Gomez, M. Carro, D. Cabeza, and M. Hermenegildo. A Generic Persistence Model for CLP Systems (And Two Useful Implementations). In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, number 3057 in LNCS, pages 104–119, Heidelberg, Germany, June 2004. Springer-Verlag.
27. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.
28. Brad J. Cox. *Object Oriented Programming: An Evolutionary Approach*. Addison Wesley, 1991. Additional information available at `http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf`.
29. P. Chico de Guzmán, M. Carro, M. Hermenegildo, Claudio Silva, and Ricardo Rocha. An Improved Continuation Call-Based Implementation of Tabling. In D.S. Warren and P. Hudak, editors, *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, volume 4902 of *LNCS*, pages 198–213. Springer-Verlag, January 2008.
30. M. García de la Banda, F. Bueno, and M. Hermenegildo. Towards Independent And-Parallelism in CLP. In *Programming Languages: Implementation, Logics, and Programs*, number 1140 in LNCS, pages 77–91, Aachen, Germany, September 1996. Springer-Verlag.
31. M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in CLP Languages. *ACM Transactions on Programming Languages and Systems*, 22(2):269–339, March 2000.

32. S. K. Debray and N. W. Lin. Cost analysis of logic programs. *TOPLAS*, 15(5), 1993.
33. S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. PLDI'90*, pages 174–188. ACM, June 1990.
34. S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.
35. S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *ILPS'97*. MIT Press, 1997.
36. S.K. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *ICLP'97*, pages 48–62. MIT Press, 1997.
37. S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.
38. Omar El-Khatib, Enrico Pontelli, and Tran Cao Son. Integrating an Answer Set Solver into Prolog: ASP-PROLOG. In *LPNMR*, pages 399–404, 2005.
39. Thom Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3), October 1998.
40. Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *International Conference on Logic Programming 1988*, pages 1070–1080, 1988.
41. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, July 2001.
42. S. Haridi and N. Franzén. *The Oz Tutorial*. DFKI, February 2000. Available from http://www.mozart-oz.org.
43. F. Henderson, Z. Somogyi, and T. Conway. Determinism Analysis in the Mercury Compiler. In *Proc. Australian Computer Science Conference*, pages 337–346, Melbourne, Australia, January 1996.
44. M. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.
45. M. Hermenegildo. Automatic Parallelization of Irregular and Pointer-Based Computations: Perspectives from Logic and Constraint Programming. In *Proceedings of EUROPAR'97*, volume 1300 of *LNCS*, pages 31–46. Springer-Verlag, August 1997.
46. M. Hermenegildo. A Documentation Generator for (C)LP Systems. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345–1361. Springer-Verlag, July 2000.
47. M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Some Techniques for Automated, Resource-Aware Distributed and Mobile Computing in a Multi-Paradigm Programming System. In *Proc. of EURO–PAR 2004*, number 3149 in LNCS, pages 21–37. Springer-Verlag, August 2004.
48. M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

49. M. Hermenegildo and The Ciao Development Team. Why Ciao? –An Overview of the Ciao System's Design Philosophy. Technical Report CLIP7/2006.0, Technical University of Madrid (UPM), School of Computer Science, UPM, December 2006. Available from: `http://cliplab.org/papers/ciao-philosophy-note-tr.pdf`.
50. M. Hermenegildo and The CLIP Group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel, Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 123–133. Springer-Verlag, May 1994.
51. M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.
52. M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, 1999.
53. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
54. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.
55. M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.
56. C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *1992 International Symposium on Programming Language Implementation and Logic Programming*, pages 260–268. LNCS631, Springer Verlag, August 1992.
57. C. Holzbaur. *SICStus 2.1/DMCAI Clp 2.1.1 User's Manual*. University of Vienna, 1994.
58. P. Hudak, S. Peyton-Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell. *Haskell Special Issue, ACM Sigplan Notices*, 27(5), 1992.
59. Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: Being Lazy with Class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM.
60. Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In Carla Schlatter Ellis, editor, *USENIX Annual Technical Conference, General Track*, pages 275–288. USENIX, 2002.
61. A.H. Karp and R.C. Babb. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software*, September 1988.
62. Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, 19(2):159–189, 2007.
63. P. López-García, F. Bueno, and M. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *LOPSTR'04*, pages 19–35, 2005.
64. P. López-García and M. Hermenegildo. Efficient Term Size Computation for Granularity Control. In *Proc. of ICLP'95*, 1995.

65. P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21:715–734, 1996.
66. P. López-García, M. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In *Proc. of First International Symposium on Parallel Symbolic Computation, PASCO'94*, 1994.
67. E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining Static Analysis and Profiling for Estimating Execution Times. In *PADL'07*, number 4354 in LNCS. Springer-Verlag, 2007.
68. U. Montanari, F. Rossi, F. Bueno, M. García de la Banda, and M. Hermenegildo. Towards a Concurrent Semantics-based Analysis of CC and CLP. In *Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 151–161. Springer-Verlag, May 1994.
69. J. Morales, M. Carro, and M. Hermenegildo. Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, number 3057 in LNCS, pages 86–103, Heidelberg, Germany, June 2004. Springer-Verlag.
70. Mozilla. Tamarin Project, 2008. Available at `http://www.mozilla.org/projects/tamarin/`.
71. K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*, 38(2):165–218, February 1999.
72. K. Muthukumar and M. Hermenegildo. Complete and Efficient Methods for Supporting Side Effects in Independent/Restricted And-parallelism. In *1989 International Conference on Logic Programming*, pages 80–101. MIT Press, June 1989.
73. K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.
74. K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Int'l. Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.
75. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *ICLP*, 1991.
76. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *JLP*, 13(2/3):315–347, 1992.
77. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-definable resource bounds analysis for logic programs. In *ICLP*, LNCS, 2007.
78. George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.
79. M. Olmedilla, F. Bueno, and M. Hermenegildo. Automatic Exploitation of Non-Determinate Independent And-Parallelism in the Basic Andorra Model. In *Logic Program Synthesis and Transformation, 1993*, Workshops in Computing, pages 177–195. Springer-Verlag, July 1993.
80. A. Pineda and F. Bueno. The O'Ciao Approach to Object Oriented Logic Programming. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop)*, Copenhagen, July 2002.

81. G. Puebla, E. Albert, and M. Hermenegildo. Abstract Interpretation with Specialized Definitions. In *Proc. of SAS'06*, number 4134 in LNCS. Springer, 2006.
82. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–61. Springer LNCS 1870, 2000.
83. G. Puebla, M. García de la Banda, K. Marriott, and P. Stuckey. Optimization of Logic Programs with Dynamic Scheduling. In *1997 International Conference on Logic Programming*, pages 93–107, Cambridge, MA, June 1997. MIT Press.
84. G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. of PEPM'95*, pages 77–87. ACM Press, June 1995.
85. G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *JLP*, 41(2&3):279–316, November 1999.
86. A. Rigo and S. Pedroni. PyPy's Approach to Virtual Machine Construction. In *Dynamic Languages Symposium 2006*. ACM Press, October 2006.
87. H. Saglam and J. Gallagher. Approximating constraint logic programs using polymorphic types and regular descriptions. Technical Report CSTR-95-17, Department of Computer Science, University of Bristol, Bristol BS8 1TR, 1995.
88. Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Belgium, June 2005.
89. Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *JLP*, 29(1–3), October 1996.
90. D.H.D. Warren. Logic Programming Languages, Parallel Implementations, and the Andorra Model. Invited talk, slides presented at ICLP'93, 1993.
91. R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.