FACULTAD DE INFORMÁTICA

UNIVERSIDAD POLITÉCNICA DE MADRID

MASTER THESIS

MASTER IN ARTIFICIAL INTELLIGENCE RESEARCH

DYNAMIC CHECKING OF ASSERTIONS FOR HIGHER-ORDER PREDICATES

AUTHOR: NATALIIA STULOVA
SUPERVISOR: MANUEL HERMENEGILDO
CO-SUPERVISOR: JOSÉ F. MORALES

JULY, 2013

Abstract

In order to improve the quality of software products various techniques can be used within the software development life cycle. A topic that has received significant interest in recent years has been the technique of program validation via static and/or dynamic checking of user–provided assertions. Such assertions can be considered a (partial) program specification in the form of annotations in the source code. Together with analysis tools, they allow detecting incorrect program behaviors, studying resource consumption, and reasoning about various other properties of the program.

This approach has proven quite useful in the domain of Logic Programming, where one of the richest designs and implementations of a combination of assertion language and analysis tools can be found in the Ciao system. One of the features of this system is the support of higher–order. Higher–order logic programming extends the expressiveness of conventional first–order logic programming, both syntactical and semantically. While quite useful in practice, higher-order also poses challenges in analysis and verification.

This thesis contributes to solving the problem of analysis, checking, and specification of higher–order logic programs. It proposes a higher-order extension of the traditional Ciao assertion language and of the CiaoPP run-time verification mechanisms. We explore different alternatives for checking, highlighting the benefits and drawbacks of each of them. We also present a prototype implementation, based on the current language extension mechanism of Ciao, and some early experimental results. We expect this first approach to the problem to serve as a basis for the design and development of more sophisticated checking and analysis tools of higher-order programs.

Resumen

Varias técnicas pueden usarse para mejorar la calidad de los productos *software* durante su ciclo de desarrollo. Una técnica que ha suscitado un interés significativo durante los últimos años es la validación de programas mediante análisis estático y/o dinámico de aserciones proporcionadas por el usuario. Dichas aserciones pueden considerarse como una especificación (parcial) del programa, en forma de anotaciones en el código. Junto con herramientas de análisis, permiten inferir comportamientos incorrectos del programa, estudiar el consumo de recursos y razonar acerca de varias otras propiedades del programa.

Estos métodos han resultado muy útiles en el contexto de la Programación Lógica, donde uno de los diseños e implementaciones de análisis y lenguaje de aserciones más ricos puede encontrarse en el sistema Ciao. Una de las características de este sistema es el soporte a la programación de orden superior. La programación lógica de orden superior extiende la expresividad de la programación lógica convencional de primer orden tanto sintáctica como semánticamente. Aunque muy útil en la práctica, el orden superior plantea retos en el análisis y la verificación.

Esta tesis contribuye a la solución de problema del análisis, comprobación y especificación de programas lógicos de orden superior. Propone una extensión del lenguaje de aserciones de Ciao y del sistema de comprobación en tiempo de ejecución por parte de CiaoPP. Exploramos diferentes alternativas para la comprobación en tiempo de ejecución, destacando los beneficios e inconvenientes de cada una de ellas. También presentamos un prototipo de implementación, basado en el mecanismo de extensiones de lenguaje de Ciao, así como algunos resultados experimentales preliminares. Esperamos que la dirección adoptada en este trabajo para atacar estos problemas sirva como base para el diseño y desarrollo de herramientas más sofisticadas de análisis y comprobación de programas de orden superior.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rising complexity of software has brought about an increased need for advanced development and debugging environments. One of the topics of greatest interest is the particular issue of program validation and debugging via direct static and/or dynamic checking of user–provided *assertions* [1, 2, 3, 4, 5, 6, 7], which can be considered as the starting point for correctness validation and debugging. For the environments that support such reasoning mechanism, the following features are desirable:

- *optional* assertion annotation: specifications may be given only for some parts of the program and even for those parts the information given may be incomplete;
- supporting assertions which are much more general than traditional type declarations, and such that it may be statically *undecidable* whether they hold or not for a given program;
- automatically *generating* assertions, especially for parts of the program for which there are no user–specified assertions.

A consequence of these assumptions is that the overall framework typically needs to deal throughout with approximations [8, 9, 10] rather than directly with the concrete semantics of a program. Another desirable feature of the framework and the assertion language is that the design must support dynamic checking of assertions (run–time tests) in addition to static checking. This approach was embodied within the Ciao system, strongly motivated by the availability of powerful and mature static analyzers for (constraint) logic programs, generally based on abstract interpretation [9].

Despite the advancements of those tools and techniques, that work has focused mainly on relatively *flat* programs and languages. An interesting and heavily used abstraction mechanism consists in the introduction of mechanisms for writing *higher-order* programs. Higher-order programming was initially borrowed from computation models such as λ-calculus and introduced in the functional programming paradigm. In higher-order, functions can be passed as arguments or returned as arguments of other functions. With higher-order, the program structure of different problems can be shared (e.g., sorting a list of elements) and reused for different instances (e.g., different order relations). Many extensions and variants of Prolog extend the idea for logic programming, so that variables can be unified with predicates [11, 12].

In this work we explore different solutions for enhancing the capabilities of the Ciao assertion language and dynamic checking framework to support higher-order programming.

1.1 Possibilities of Static and Dynamic Analyses

The starting point for this work is the Ciao [13] language together with the CiaoPP preprocessor [14, 15, 16, 13], which is able to perform static global program analysis (see Fig 1.1 from [13] for the details of preprocessor and assertion framework architectures). The system includes several abstract analysis domains developed by several groups in the LP and CLP communities and can infer both variable– and procedure–level properties such as data structure shape and instantiation state ("moded types") together with pointer sharing [17, 18, 19, 20, 18], definiteness, freeness, independence among program variables [21, 22], absence of side effects [23], determinacy [24, 25], 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, termination, non–failure [26, 27]), lower and upper bounds on the sizes of terms and the computational cost of predicates [28, 29, 30, 31, 32, 8, 32], bounds on the execution time [33] and the consumption of a large class of user-defined resources [34]. Thanks to this functionality, CiaoPP can also certify programs with resource consumption assurances as well as efficiently checking such certificates [35].

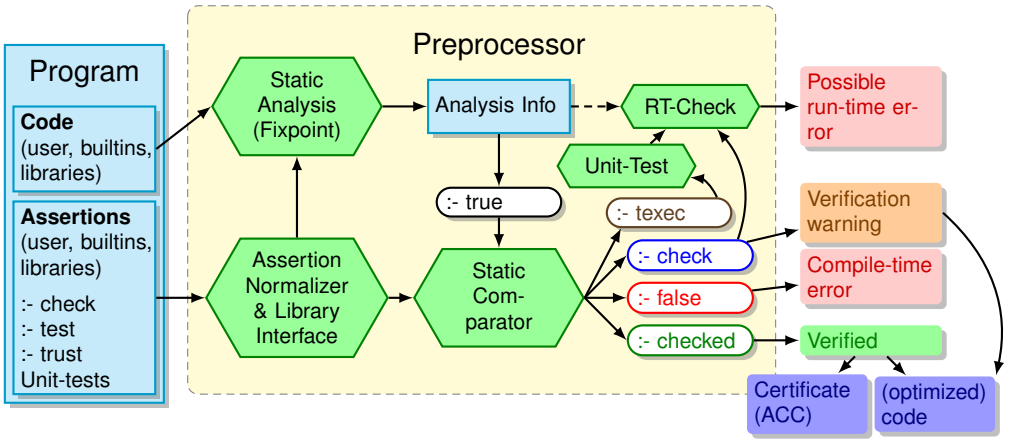

Figure 1.1: The Ciao assertion framework (CiaoPP's verification/testing architecture).

After performing static analysis the framework compares the obtained results with assertions, which allows to prove or disprove some of them, and, thus, reason whether the corresponding program properties hold or not. However, for some cases it is impossible to perform static checks, and the only suitable solution consists of introducing and performing run–time checks.

1.2 Dynamic Checking of Higher–order Predicates

When higher–order programming is introduced in the language it becomes necessary to describe properties about the predicates that are *bound* to variables. However, the current Ciao assertion language and verification tools only include very limited facilities for describing the predicate arguments of higher–order predicates and the corresponding calls: essentially the basic typing required by the module system (meta-predicate declarations). The capability of the static analyzers for dealing with higher order is also quite limited. Still, the framework itself serves as the perfect foundation for introducing improved analysis facilities, allowing both the extension of the assertion language with the necessary properties and the addition of the corresponding reasoning and checking mechanisms.

Since the first objective of this work is to enhance the specification of logic programs by extending it with better ways of describing higher–order, we intentionally restrict our approach to dynamic checking. This consists in defining the evaluation mechanisms able to detect violations of specifications that include properties about higher–order predicates.

1.3 Structure of the Document

This document is organized as follows:

- Chapter 2 describes the State of the Art in assertion–based analysis for logic programs, concentrating on the approach adopted in Ciao. It provides a description of its rich assertion language and gives an overview of the higher–order extension to Ciao.
- Chapter 3 shows the inadequacies of the State of the Art for the specification and verification of higher–order predicates, describes a new extension of the Ciao assertion language that allows asserting statements about higher–order calls, and details several alternatives for run–time checking.
- Chapter 4 presents a working prototype implementation of the extensions presented in Chapter 3.
- Chapter 5 describes the evaluation process and presents some experimental results on performance.
- Chapter 6 draws some conclusions and presents ideas for future work.

Chapter 2

State of the Art

Several research areas need to be considered as background for the discussion of possible solutions to the problem of enhancing the specification of logic programs. The recent achievements in the area of program annotation with *assertions* and corresponding analyses can be seen as a first foundation stone of such discussion. Next, the discussion should take into account the practical issues of higher–order programming extensions, available in the domain of logic programming. Finally, the related areas, such as meta–programming, should not be omitted in order to have a precise feeling of the domain of the problem.

This chapter presents an overview of techniques corresponding to the aforementioned research areas, to provide background on the field that this thesis covers. Section 2.1 provides a brief introduction to the theoretical foundation of program analysis based on assertions, described further in Section 2.2. Then, the description of Ciao assertion language is outlined in Section 2.3. The peculiarities of introducing higher–order to logic programming are described in Section 2.4, and Section 2.5 illustrates the relation of the area of meta–programming to the problem.

2.1 The Foundation for Advanced Static and Dynamic Program Analysis

As emphasized in [14], the technique of Abstract Interpretation [9] has allowed the development of sophisticated program analyses which are at the same time provably correct and practical. The semantic approximations produced by such analyses have been traditionally applied to high– and low–level *optimizations* during program compilation, including *program transformation*. More recently, novel and promising applications of semantic approximations have been proposed in the more general context of program development, such as *verification* and *debugging*. Some basic concepts from abstract interpretation are recalled below.

In the setting of *fixpoint semantics*, a (monotonic) semantic operator S_P is associated with each program P. This S_P function operates on a semantic domain D which is generally assumed to be a complete lattice or, more generally, a chain complete partial order.

The meaning of the program $[\![P]\!]$ is defined as the least fixpoint of the S_P operator, i.e., $[\![P]\!] = \mathrm{lfp}(S_P)$.

In the abstract interpretation technique, the program P is interpreted over a non–standard domain called the *abstract* domain D_α which is simpler than the *concrete* domain D. The abstract domain D_α is usually constructed with the objective of computing safe approximations of the semantics of programs, and the semantics w.r.t. this abstract domain, i.e., the *abstract semantics* of the program, is computed (or approximated) by replacing the operators in the program by their abstract counterparts. The abstract domain D_α also has a lattice structure. The concrete and abstract domains are related via a pair of monotonic mappings: *abstraction* $\alpha : D \mapsto D_\alpha$, and *concretization* $\gamma : D_\alpha \mapsto D$, which relate the two domains by a Galois insertion (or a Galois connection) [9].

One of the fundamental results of abstract interpretation is that an abstract semantic operator S_P^α for a program P can be defined which is correct w.r.t. S_P in the sense that $\gamma(\mathrm{lfp}(S_P^\alpha))$ is an approximation of $[\![P]\!]$, and, if certain conditions hold (e.g., ascending chains are finite in the D_α lattice), then the computation of $\mathrm{lfp}(S_P^\alpha)$ terminates in a finite number of steps. The result of abstract interpretation for a program P, $\mathrm{lfp}(S_P^\alpha)$, is further denoted as $[\![P]\!]_\alpha$.

Typically, abstract interpretation guarantees that $[\![P]\!]_\alpha$ is an *over*–approximation of the abstract semantics of the program itself, $\alpha([\![P]\!])$. Thus, we have that $[\![P]\!]_\alpha \supseteq \alpha([\![P]\!])$, which is further denoted as $[\![P]\!]_{\alpha^+}$. Alternatively, the analysis can be designed to safely *under*–approximate the actual semantics, and then $[\![P]\!]_\alpha \subseteq \alpha([\![P]\!])$ holds, which is further denoted as $[\![P]\!]_{\alpha^-}$.

Both program verification and debugging compare the *actual semantics* of the program, i.e., $[\![P]\!]$, with an *intended semantics* for the same program, further denoted by $\mathcal{I}$. The abstract interpretation technique allows computing *safe* approximations of the program semantics.

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

Table 2.1: Set theoretic formulation of verification problems.

In the approach adopted in Ciao [8], the abstract approximation $[\![P]\!]_\alpha$ is actually computed over the concrete semantics of the program $[\![P]\!]$ and is compared directly to the (also approximate) intention (which is given in terms of *assertions* [36]), following almost directly the scheme of Table 2.1.

This approach assumes that the program specification is given as a semantic value $\mathcal{I}_\alpha \in D_\alpha$. The implications of comparing $\mathcal{I}_\alpha$ and $[\![P]\!]_\alpha$, which is an approximation of $\alpha([\![P]\!])$, are relevant on our context. In Table 2.2 [8] (sufficient) conditions for correctness and completeness w.r.t. $\mathcal{I}_\alpha$ are proposed, which can be used when $[\![P]\!]$ is approximated.

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

Table 2.2: Validation problems using approximations.

Analyses which over–approximate the actual semantics (i.e., those denoted as $[\![P]\!]_{\alpha^+}$), are specially suited for proving partial correctness and incompleteness with respect to the abstract specification $\mathcal{I}_\alpha$. It will also be sometimes possible to prove incorrectness in the extreme case in which the semantics inferred for the program is incompatible with the abstract specification, i.e., when $[\![P]\!]_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset$. It will only be possible to prove total correctness if the abstraction is *precise*, i.e., $[\![P]\!]_\alpha = \alpha([\![P]\!])$.

On the other hand, $[\![P]\!]_{\alpha^-}$ denotes the (less frequent) case in which analysis under–approximates the actual semantics. In such case, it will be possible to prove completeness and incorrectness. In this case, partial correctness and incompleteness can only be proved if the analysis is precise.

Performing these validation tasks can result in the validation of P with respect to $\mathcal{I}$, i.e., proving that P is partially correct and/or complete with respect to $\mathcal{I}$, or in the detection of incorrectness and/or incompleteness *symptoms*, which would flag the existence of errors in P, and in which case a process of diagnosis should be started to locate such errors.

2.2 Assertion–based Analysis

Assertions are linguistic constructions which allow expressing properties of programs [36]. Syntactically they appear as an extended set of declarations, and semantically they allow talking about preconditions, (conditional–) postconditions, whole executions, program points, etc [13]. Assertions have been used in the past in different contexts and for different purposes related to program development:

Run–time checking: In this context, assertions express properties about the run–time behavior of the program which *should hold* if the program is correct [37] (see [38] for an application in Constraint Logic Programming (CLP) [39]).

Replacing the oracle: In declarative debugging [40], the existence of an *oracle* (normally the user) which is capable of answering questions about the intended behavior of the program is assumed. Here also assertions are used to express properties which *should hold* for the program to be correct [1, 2, 4].

Compile–time checking: In this context, assertions express properties about the program which are intended to be checked at compile–time. The result of such check-

ing may indicate either that the assertions actually hold and the program is *validated* w.r.t. the assertions or that the assertions do not hold, and then the program is *incorrect* w.r.t. the assertions [37]. Again, these are properties which *should hold*, i.e., otherwise a bug exists in the program. An example of this kind of assertions are type declarations (e.g., [41, 42], functional languages, etc.), which have been shown to be useful in debugging.

Providing information to the optimizer: Assertions have also been proposed as a means of providing information to an optimizer in order to perform additional optimizations during code generation. In this context, assertions do not express properties which should hold for the program, but rather properties which *do hold* for the program at hand. If the program is not correct, the properties which hold may not coincide with the properties which should hold.

General communication with the compiler: In a setting where there is both a static inference system, such as an abstract interpreter [9, 43], and an optimizer, assertions have also been proposed as a means of allowing the user to provide additional information to the analyzer [3], which it can use both to increase the precision of the information it infers and/or to perform additional optimizations during code generation [44, 45, 46, 47]. Also, assertions can be used to represent analysis output in a user-friendly way and to communicate different modules of the compiler which deal with analysis information [3]. In this context, assertions express both properties which should hold and properties which *do hold* for the program in hand.

Program documentation: Assertions have also been used to document programs and to automatically generate manuals (as inspired by the "literate programming" style [48, 49]). In this application, assertions may express both properties which *do hold* or which *should hold* for the program in hand.

In addition to the classification given above, made according to the context in which assertions are used, assertions can be classified according to many other criteria. For example, as mentioned above, in some cases the assertions express properties which should hold (intended properties) while in others the assertions express properties which actually hold (actual properties) for the program.

One of the design choices behind the Ciao assertion language [8, 15, 36, 50] was to allow expressing any property which is of interest for any of the debugging (and validation) tools in the environment. The other one was to allow the assertion language to be independent of the particular CLP platform in which it is applied and the constraint domains supported. Thus, it was chosen not to restrict too much beforehand the kind of properties which can be expressed with assertions. A fundamental motivation behind this choice was the frequent availability in Ciao of tools which can handle quite rich properties, through techniques such as approximations and abstract interpretation [9].

However, not all tools within the development environment are capable of dealing with *all* properties expressible in Ciao assertion language. This was the main reason for proposing the use of the same assertion language for all of them. This facilitates communication among the different tools and enables easy reuse of information, i.e., once

a property has been stated there is no need to repeat it for the different tools. Each tool could then only make use of the part of the information given as assertions which the tool *understands* and could deal *safely* with the part of the information it does not understand.

In the Ciao assertion language, assertions are always instances of some *assertion schema* together with a reference to which part of the program (predicate or program point) the assertion refers to and, depending on the schema used, one or two *logic formulae*. Whereas the assertion language has a fixed set of assertion schemas, the user has a high degree of freedom for defining the logic formulae for the properties considered of interest. Thus, the whole assertion language is determined by a set of assertion schemas and the way in which "logic formulae" can be built.

2.2.1 The Role of Properties

Whereas each kind of assertion indicates *when*, i.e., in which states or sequences of states, to check the given properties, the properties themselves define *what* to check. This section revises the use of properties for run–time checks according to [15]. In order to make it possible to check a property at run–time, some *code* must exist somewhere in the system that performs this check. If the set of properties were fixed, the code to be used when performing the run–time tests could be contained in a predefined library. However, if one of the language design objectives is to allow the user to define new, quite general properties it is necessary to provide an option of *writing the definitions of properties in the source language*.

A property may be a built–in predicate or constraint (such as `integer(X)` or `X>5`, and including extra–logical properties such as `var(X)`), an expression built using conjunctions (and/or disjunctions) of properties, or, in principle, any predicate defined by the user, using the full underlying CLP language. Some limitations are useful in practice, for instance, to avoid the behavior of the program to change in a fundamental way depending on whether the run–time tests are being performed or not.

In Ciao the user is required to ensure that the execution of properties terminate for any possible initial state. Also, checking a property should not change the answers computed by the program or produce unexpected side–effects. Regarding computed answers, in principle properties are not allowed to further instantiate their arguments or add new constraints. Regarding side–effects, it is required that the code defining the property does not perform input/output, add/delete clauses, etc. which may interfere with the program behavior.

There are two fundamental classes of properties:

- the properties that refer to a particular execution state, namely, *properties of execution states*;
- the properties that refer to a sequence of states, called *properties of computations*.

Yet another point of view allows differentiating *instantiation* and *compatibility* properties. The following (combined) example from [36] gives the intuition for these two notions:

Consider the following definition of the property predicate `list/1`:

```
list([]).
list([_|Xs]):- list(Xs).
```

In this definition of property `list` it is not obvious which one of the following two possibilities we mean exactly: "the argument is *instantiated* to a list" (let us indicate this property with the property predicate `inst_to_list`), or "if any part of the argument is instantiated, this instantiation must be compatible with the argument being a list" (we will associate this property with the property predicate `compat_with_list`). For example, `inst_to_list` should be true for the terms `[]`, `[1,2]`, and `[X,Y]`, but should not for `X` and `[a|X]`. In turn, `compat_with_list` should be true for `[]`, `X`, `[1,2]`, and `[a|X]`, but should not be for `[a|1]` and `a`.

As a general rule, the properties that can be used directly for checking for compatibility should be *downwards closed*, i.e., once they hold they will keep on holding in every state accessible in forwards execution. Also, properties about execution states are interpreted by default as instantiation properties.

2.2.2 Compile–time Analysis

In Ciao the actual checking of assertions at compile–time [15] is performed as follows (precise details on how to reduce assertions at compile–time can be found in [51]). The properties which appear in the user-provided `check` assertions are compared one by one with the properties inferred by the analysis. An assertion is validated if all its properties are *implied* by the analysis results (preconditions require special consideration in this process). On the other hand, errors are detected if any property specified is *incompatible* with the analysis results. If it is not possible to prove nor to disprove an assertion, then such assertion is left as a `check` assertion, for which run–time checks might be generated. However, if some properties are implied but others cannot be proved nor disproved, the assertion as a whole can be *simplified*, in the sense of reducing the number of properties which have to be checked at run–time.

As a result, compile–time checking must be able to deal (at least safely) with properties that have perhaps been written with run–time checking in mind or for which no specific analysis is available. Conversely, the run–time checking machinery must also be able to deal correctly with properties that are primarily meant for compile–time checking.

In this context properties can be divided into classes *from the point of view of a given analysis*. First, *native* properties are those which are directly "understood" (abstracted) by this analysis. This is the case for example of properties like `ground` or `var` for a mode analysis, `does_not_fail` for a non–failure analysis, `terminates` for a termination analysis, or a predicate defining a (regular) type for a regular type analysis, etc.

If a property appearing in an assertion is native of an analysis then it is often possible to either prove it or disprove it, provided that the analysis is accurate enough and the "direction" of approximation performed by the analysis is the appropriate one [51, 8]

(this is the case for the properties `var` and `does_not_fail` in the example above). It can be specified that the properties are *abstractly reducible* (to either true or false), or abstractly executable [52]. If the analysis is *precise* (in the sense defined before, i.e., that the abstract operations do not lose information beyond the abstraction implied by the abstraction function used [9]) and, obviously, terminates, then the native properties will be decidable in all cases. However, since there may in general be cases in which some such properties remain for run–time checking (and because in the Ciao framework the definitions of properties can be called from user programs) it is required that there be an executable definition of all properties available in the system.

There are properties which can be proved (or disproved) at compile–time by a given analyzer but for which no *accurate* definition can be written in the underlying language. An extreme example of this is the property `terminates`, for which it is obviously not possible to define a run–time test which will give a warning if it does not hold. For these properties, an *approximate* definition may be given, and this approximation should be correct in the usual sense that all errors flagged should be errors, but there may be errors that go unchecked.

In summary, it is not necessary that the executable definition of all properties be an exact implementation of a given property, but the user must provide, or import, some code for each property and understand and take into account the impact of approximation being performed in the property definition when using these properties in assertions.

Conversely, and again for a given analysis, there may be properties which are defined precisely and are perfectly executable at run–time, but which may not be *native* for that analysis. For them, the analysis may not be capable of obtaining an exact representation (abstraction). However, a useful approximation (usually an over–approximation) of such property can be obtained by *directly analyzing the code which defines the property*.

In general, typical analyzers obtain over–approximations of properties, i.e., they succeed for a superset of the cases in which the exact property would succeed. However, for the case of properties in preconditions of `success` or `comp` assertions, under–approximations (i.e., the approximation succeeds for a subset of the cases in which the exact property would succeed) rather than over–approximations should be considered.

2.3 The Ciao Assertion Language

As pointed out in [36], when reasoning about whether a certain program behaves as indicated by a set of assertions, it is often useful to restrict the discussion to a set of *valid* initial queries. Informally, a program is correct when it behaves according to the user's intention for any input data satisfying certain preconditions. Such input data can be seen as *valid* input data, and the corresponding queries as *valid queries*. In what follows it is assumed that program debugging and validation is always performed w.r.t. a given set of (descriptions of) valid queries.

Very often, the properties of a program which are interesting to express by means of assertions are related to the run–time behavior of the program. For this, the *operational* semantics of the program needs to be considered. The operational semantics of a program is in terms of its *derivations* which are sequences of reductions between *execution states*.

An execution state $\langle G \mid \theta \rangle$ consists of the current goal G and the current constraint store (or *store* for short) θ which contains information on the values of variables. The way in which a state is transformed into another one is determined by the operational semantics and the program code. One of the advantages of CLP is that in addition to the operational semantics, programs also have a *declarative* meaning or semantics which is independent of the particular details on how the program is executed.

Every assertion A is conceptually composed of two logic formulae which are referred to as app_A and sat_A. The formula app_A determines the *applicability set* of the assertion: a context s is in the applicability set of A iff app_A takes the value *true* in s. Also, an assertion A is *applicable* in context s iff app_A holds in s. The formula sat_A determines the *satisfiability set* of the assertion: a context s is in the satisfiability set of A iff sat_A takes the value *true* in s. If it can be proved that there is a context which is in the applicability set of an assertion A but is not in its satisfiability set then the program is definitely incorrect w.r.t. A. Conversely, if it can be proved that every context in which A is applicable is in the satisfiability set of A then the program is validated w.r.t. A. The following sections describe a repertoire of *assertion schemas*, used in the Ciao assertion language. Such schemas can be seen as templates which when properly instantiated define in a simple and clear way the required formulae app_A and sat_A.

2.3.1 Assertion Schemas for Execution States

When considering the operational behavior of a program, it is natural to associate (sets of) execution states with certain syntactic elements of the program. Typically, a program can be seen as composed of a set of *predicates* (also known as *procedures*). Alternatively, a program can be seen, at a finer-grained level, as composed of a set of *program points*. Thus, we first introduce several assertion schemas whose applicability contexts are related to a given predicate. Then we introduce an assertion schema whose applicability context is related to a particular program point. We refer to the former kind of assertions as *predicate* assertions, and to the second ones as *program-point* assertions. Though a simple program transformation technique can be used to express program-point assertions in terms of predicate assertions, we maintain program-point assertions in our language for pragmatic reasons.

As a general rule, we restrict the properties expressible by means of assertions about execution states to those which refer to the values of certain variables in the store of the corresponding execution state. This has the advantage that in order to check whether the app_A and sat_A logic formulae hold or not it suffices to inspect the store at the corresponding execution state.

An Assertion Schema for Success States

This assertion schema is used in order to express properties which should hold on termination of any successful computation of a given predicate. They can be expressed in our assertion language using the assertion schema:

`:- success` *Pred* `=>` *Postcond* `.`

This assertion schema has to be instantiated with suitable values for *Pred* and *Postcond*. *Pred* is a *predicate descriptor*, i.e., it has a predicate symbol as main functor and all arguments are distinct free variables, and *Postcond* is a logic formula about execution states, and which plays the role of the sat_A formula.

The resulting assertion should be interpreted as "the assertion is applicable in those execution states which correspond to success states of a computation of *Pred*, and its satisfiability set has those states in which *Postcond* holds."

An Assertion Schema for Success States with Preconditions

The `success` schema can be used when the applicability set of an assertion is the set of success states for a given predicate. However, it is often useful to consider more restricted applicability sets. The success schema with precondition takes the form:

`:- success` *Pred* `:` *Precond* `=>` *Postcond* `.`

and it should be interpreted as "the assertion is applicable to those execution states which correspond to success states of a computation of *Pred* which was originated by a calling state in which *Precond* holds, and its satisfiability set has those states in which *Postcond* holds."

An Assertion Schema for Call States

This assertion schema has an aim to express properties which should hold in any call to a given predicate. Assertions built using this schema can be used to check whether any of the calls for the predicate is not in the expected set of calls (i.e., the call is "inadmissible" [53]). This schema has the form:

`:- calls` *Pred* `:` *Precond* `.`

This assertion schema has to be instantiated with a predicate descriptor *Pred* and a logic formula about execution states *Precond*. The resulting assertion should be interpreted as "the assertion is applicable in those execution states which correspond to calling states to *Pred*, and its satisfiability set has those states in which *Precond* holds."

As mentioned in [15], from the point of view of their use in debugging, `calls` assertions are conceptually somewhat different from `success` and `comp` assertions (see later). Introducing `calls` assertions is a good idea even for correct predicates because the fact that a predicate is correct does not guarantee that it is called in the proper way in other parts of the program.

An Assertion Schema for Query States

It is often the case that one wants to describe the exported uses of a given predicate, i.e., its valid queries. This is for example the case also in traditional preconditions of a program. Thus, in addition to describing calling and success states, assertions are used to describe *query states*, i.e., valid input data. In terms of the operational semantics, in which program executions are sequences of states, query states are the initial states in such sequences.

These can be described in Ciao assertion language using the `entry` schema, which has the form:

`:- entry` *Pred* : *Precond*.

where, *Pred* is a predicate descriptor and *Precond* is a logic formula about execution states. It should be interpreted as "the assertion is applicable in those execution states which correspond to initial queries to *Pred*, and the satisfiability set has those states in which *Precond* holds."

It can be noted that `entry` and `calls` schemas are syntactically (and semantically) similar. However, their applicability set is different. The `entry` assertion only applies to the initial calls to *Pred*, whereas the `calls` assertion applies to any call to *Pred*, including all recursive (internal) calls. Thus, `entry` assertions allow providing more precise descriptions of initial calls, as the properties expressed do not need to hold for the internal calls.

Program–point Assertions

As already mentioned, usually, when considering operational semantics of a program, in addition to predicates we also have the notion of *program points*. The program points are considered as the places in a program in which a new literal may be added, i.e., before the first literal (if any) of a clause, between two literals, and after the last literal (if any) of a clause. For simplicity, program–point assertions can be introduced to a program by adding a new literal at the corresponding program point. This literal is of the form:

`check`(*Cond*).

and it should be interpreted as "the assertion is applicable in those execution states originated at the program point in which the assertion appears and its satisfiability set has those states in which *Cond* holds."

Logic Formulae about Execution States

This section describes how the aforementioned formulae are defined in Ciao assertion language. Both conjunctions and disjunctions are allowed in the formulae, are written in the usual CLP syntax. Thus, logic formulae about execution states can be:

- An atom of the form $p(t_1, \ldots, t_n)$ with $n \geq 0$, where p/n is a *property predicate*;
- An expression of the form ($F1$, $F2$) where $F1$ and $F2$ are logic formulae about execution states and, as usual in CLP, the comma should be interpreted as conjunction;
- An expression of the form ($F1$; $F2$) where $F1$ and $F2$ are logic formulae about execution states and, as usual in CLP, the semicolon should be interpreted as disjunction.

2.3.2 An Asertion Schema for Declarative Semantics

As already mentioned, one of the main features of CLP is the existence of a *declarative semantics* which allows concentrating on *what* the program computes and not on *how* it should be computed. Consider the case of CLP(D), where D is the domain of values. For example, in classical logic programming D is the Herbrand Universe.

The assertion schema which allows stating properties which should hold in the least D-model $[\![P]\!]$ of a program (otherwise the program is incorrect) is described below:

`:- inmodel` *Pred* `=>` *Cond*.

where *Pred* is a predicate descriptor and *Cond* is a logical formula about D-atoms. It should be interpreted as "the applicability set of the assertion has those D-atoms in $[\![P]\!]$ whose predicate symbol is that of *Pred* and the satisfiability set is the set of D-atoms whose predicate symbol is that of *Pred* which satisfy the property *Cond*."

In fact, a program which is correct w.r.t. an assertion '`:- inmodel` *Pred* `=>` *Cond*' is also correct w.r.t. the assertion '`:- success` *Pred* `=>` *Cond*' due to correctness of the operational semantics (but not vice versa due to possible incompleteness of the operational semantics). A further difference between `inmodel` and `success` assertions is that in `inmodel` it is not possible to add preconditions since the declarative semantics does not capture calls to predicates.

2.3.3 Assertion Schemas for Program Completeness

As seen above, there is a similarity between `success` and `inmodel` assertions in that they both express properties about the *answers* of predicates. More precisely, `success` assertions express properties of the *computed answers* of predicates, i.e., those generated by the operational semantics, whereas `inmodel` assertions refer to *correct answers*, i.e., those which are in the declarative semantics of the program (its least D-model). When considering answers to predicates, one particular aspect to reason about is *correctness* of the program, which corresponds to answering the question: Are all the actual answers of the program in the set of intended answers? Conversely, another aspect we can reason about is the well known concept of *completeness* of the program, which corresponds to answering the question: Are all the intended answers of the program in the set of actual answers? In other words, a program is complete when it does not fail to produce any expected answer. Clearly, we would like our program to be both correct and complete w.r.t. our intention. This corresponds to the classical notion of *total correctness*, as opposed to the previous notion of correctness, which is also known as *partial correctness*.

Below is described another kind of assertions which is a variation of the `inmodel` and `success` assertions to reason about completeness of programs.Such assertions can be distinguished from the previous ones because the arrow (`=>`) now points in the reverse direction, i.e., `<=`. For example, an assertion of the form:

`:- inmodel` *Pred* `<=` *Cond*.

should be interpreted as "any D-atom of the form $p(d_1, \ldots, d_n)$ whose predicate symbol is the same as that in *Pred* and on which *Cond* holds should be in $[\![P]\!]$."

Completeness assertions for operational semantics can be written using the following schema (optional "fields" appear in square brackets):

`:- success` *Pred* [`:` *Precond*] `<=` *Postcond* .

which should be interpreted as "any call to predicate *Pred* which on the calling state satisfies *Precond* must have as success states at least all those states which satisfy *Postcond*."

2.3.4 Status of Assertions

The assertion language should be able to express both intended and actual properties of programs. However, all the assertions presented in the examples in previous sections relate to intended properties. The Ciao assertion language allows adding in front of an assertion a flag which clearly identifies the *status* of the assertion. The status indicates whether the assertion refers to intended or actual properties, and possibly some additional information.

Five different status are considered. They are listed below, grouped according to who is usually the generator of such assertions:

- For assertions written by the user:

  `check` The assertion expresses an intended property. Note that the assertion may hold or not in the current version of the program.

  `trust` The assertion expresses an actual property. The difference with status `true` introduced below is that this information is given by the user and it may not be possible to infer it automatically.

- For assertions which are results of static analyses:

  `true` The assertion expresses an actual property of the current version of the program. Such property has been automatically inferred.

- For assertions which are the result of static checking:

  `checked` A `check` assertion which expresses an intended property is rewritten with the status `checked` during compile–time checking when such property is proved to actually hold in the current version of the program for any valid initial query.

  `false` Similarly, a `check` assertion is rewritten with the status `false` during compile–time checking when such property is proved not to hold in the current version of the program for some valid initial query.

2.3.5 An Assertion Schema for Computations

The assertion schema named `comp` relates to *computations*, where by computation it is meant the (ordered) execution tree of all derivations of a goal from a calling state. The need for it arises when considering properties which refer to the computation of the predicate (rather than the input-output behavior), which are not expressible with the previous schemas. Examples of properties of the computation which we may be interested in are: non–failure, termination, determinacy, non–suspension, non–floundering, etc. In the Ciao language, this sort of properties are expressed using the schema:

`:- comp` *Pred* [`:` *Precond*] + *Comp-prop* .

where *Pred* is a predicate descriptor, *Precond* is a logic formula on execution states, and *Comp-prop* is a logic formula on computations. It can be interpreted as "the applicability set of the assertion is the set of computations of $Pred$ in which the logic formula on states *Precond* holds at the calling state, and its satisfiability set has all computations in which the logic formula on computations *Comp-prop* holds."

2.3.6 Syntax of the Assertion Language

The summarized syntax of the assertions is presented with the following two formal grammars. The first one defines the syntax of program assertions, from the non–terminal *program–assert*:

program-assert ::= *predicate-assert*
| *prog-point-assert*
predicate-assert ::= `:-` *stat-flag pred-assert* .
| `:-` *entry* .
pred-assert ::= `calls` *pred-cond*
| `success` *pred-cond direction state-log-formula*
| `inmodel` *pred-desc direction state-log-formula*
| `comp` *pred-cond* + *comp-log-formula*
entry ::= `entry` *pred-cond*
pred-cond ::= *pred-desc*
| *pred-desc* : *state-log-formula*
pred-desc ::= *Pred-name*
| *Pred-name*(*args*)
args ::= *Var*
| *Var*,*args*
state-log-formula ::= (*state-log-formula* , *state-log-formula*)
| (*state-log-formula* ; *state-log-formula*)
| `compat`(*State-prop*)
| *State-prop*
comp-log-formula ::= *comp-log-formula* , *comp-log-formula*
| *comp-log-formula* ; *comp-log-formula*
| *Comp-prop*

stat-flag ::= *status*
| ϵ
status ::= `check`
| `true`
| `checked`
| `trust`
| `false`
direction ::= `=>`
| `<=`
prog-point-assert ::= *status*`(`*state-log-formula*`)`

There are some non–terminals in the grammar which are not defined. This is because they are constraint domain- and/or platform-dependent. They can be easily distinguished in the previous grammar because their name starts with a capital letter:

Pred-name As we are interested in having an assertion language which looks homogeneous with the CLP language used, we admit as *Pred-name* any valid name for a predicate in the underlying CLP language. Usually, non-empty strings of characters which start with a lower–case letter.

Var It corresponds to the syntax for variables in the CLP language. Usually, non–empty strings of characters which start with a capital letter. As mentioned before, it is assumed that all variables in the same predicate description are distinct.

State-prop An atom of a `prop` property predicate.

Comp-prop An atom of a `cprop` property predicate.

The second one defines the syntax of assertions for declaring property predicates, from the non–terminal *prop–assert*:

prop-assert ::= *prop-exp*
| *approx-exp*
prop-exp ::= `:-` *is-flag prop pred-spec* `.`
prop ::= `prop`
| `cprop`
is-flag ::= `[` *is-idlist* `]`
| *Is-id*
| ϵ
is-idlist ::= *Is-id* `,` *is-idlist*
| *Is-id*
pred-spec ::= *Pred-name/Number*
approx-assert ::= `:-` *approx approx-exp* `.`
approx ::= `proves`
| `disproves`
approx-exp ::= *State-prop* `:` *state-log-formula*
| *Comp-prop* `:` *comp-log-formula*

The new non-terminals in the grammar are as follows:

Is-id A constant of the language which uniquely identifies an inference system in the debugging system being used.

Number A number (which is meant to be the arity of a predicate).

Assertion Grouping Possibilities

The motivation for introducing compound assertions is twofold:

- when more than one `success` (resp. `comp`) assertion is given by the user for the same predicate, in the user's mind this set is usually meant to cover all the different uses of the predicate. In such cases, the disjunction of the preconditions in all the `success` (resp. `comp` assertions) is often a description of the possible calls to the predicate. The user would have to explicitly write down a `calls` assertion to express this. It would be desirable to have the `calls` assertion be automatically generated in such cases for the set of assertions, rather than having to add it manually. Compound assertions allow this.

- a disadvantage of the assertion `success`, `calls`, `entry` and `comp` schemas is that it is often the case that in order to express a series of properties of a predicate, several of them need to be written.

Each compound assertion is translated into one, two, or even three basic predicate assertions, depending on how many of the fields in the compound assertion are given. Compound assertions are built using the `pred` schema, which has the form:

`:-` `pred` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-prop*]`.`

Both basic and compound assertions may be given for a program. Compound assertions are only a more compact way to write what otherwise would have to be written as a set of basic assertions. The syntax grammar presented previously does not include this extension. Table 2.3 presents how a compound assertion is translated into basic `success` and `comp` assertions. Generation of `calls` assertions from compound assertions is more involved, as the set of all compound assertions for one predicate must cover all possible calls to that predicate.

Field	Translation if given	Otherwise
`=>` *Postcond*	`success` *Pred* `:` *Precond* `=>` *Postcond*	$\emptyset$
`+` *Comp-prop*	`comp` *Pred* `:` *Precond* `+` *Comp-prop*	$\emptyset$

Table 2.3: Compound assertions transformation into basic ones.

Yet another way of grouping properties, stated in assertions, is to specify the properties of some arguments both at the call and success states of the predicate. To avoid repeating the properties, the syntax of the following example can be used:

```
:- pred qsort(A,B) :: ( list(A), list(B) ).
```

which is equivalent to:

```
:- pred qsort(A,B) : ( compat(list(A)), compat(list(B)) ) =>
                     ( compat(list(A)), compat(list(B)) ).
```

This kind of writing can also be "in-lined" into the predicate arguments. For example, the following assertion is equivalent to the two ones above:

```
:- pred qsort(list,list).
```

2.3.7 Run–time Checking

This section describes a possible scheme for translation of a program with assertions into code which will perform run–time checking, from [15]. The checking of the properties can be an *instantiation check* or a *compatibility check*.

Success Assertions. A possible translation scheme for `success` assertions (ones used to express properties which should hold on termination of any successful computation of a given predicate) into run-time tests is the following. Let A(`p/n`) represent the set of current assertions for predicate `p` of arity `n`. Let S be the set $\{Postcond$ s.t. '`:- success` $p(X1,...,Xn)$ `=>` $Postcond$' $\in A(p/n)\}$. Then the translation is:

```
p(X1,...,Xn):- new_p(X1,...,Xn), check(S).
```

where `new_p` is a renaming of predicate `p`.

Let RS be the set $\{(Precond, Postcond)$ s.t. '`:- success` $p(X1,...,Xn)$ `:` $Precond$ `=>` $Postcond$' $\in A(p/n)\}$. A possible translation scheme for `success` assertions with a precondition is as follows:

```
p(X1,...,Xn):-
        collect_valid_postc(RS,S),
        new_p(X1,...,Xn), check(S).
```

The predicate `collect_valid_postc/2` collects the postconditions of all pairs in `RS` such that the precondition holds.

Calls Assertions. A possible translation scheme for `calls` assertions (ones used to express properties which should hold in any call to a given predicate) into run-time tests follows. Let C be the set $\{Cond$ s.t. '`:- calls` $p(X1,...,Xn)$ `:` $Cond \in A(p/n)\}$. Then the translation is:

```
p(X1,...,Xn):-  check(C), new_p(X1,...,Xn).
```

Comp Assertions. Let RC be the set $\{(Prec, Comp\_prop)$ s.t. '`:- comp` $p(X1, ..., Xn)$: $Prec$ + $Comp\_prop$' $\in A(p/n)\}$. Then, a possible translation scheme of `comp` assertions (ones used to express properties about computations of a predicate) into run–time tests is as follows:

```
p(X1,...,Xn):-
        collect_valid_postc(RC,C),
        add_arg(C,new_p(X1,...,Xn),C1),
        ( C1 == [] ->
          call(new_p(X1,...,Xn)) %% then
        ; call_list(C1) ).       %% else
```

where the predicate `add_arg` adds the goal `new_p(X1,...,Xn)` as the first argument to any property of the computation, and `call_list` calls each goal in the argument list.

2.4 Higher–order Logic Programming

This section does not deeply discuss the recent advances in higher–order programming in general, as they are out of scope of the present work, but the interested reader is referred to relevant bibliography [54, 55, 56, 11]. The intent of it is to provide the description of the Ciao implementation of higher–order programming, introduced in [57, 12, 58].

2.4.1 Higher–order Extensions in Ciao

As it was highlighted in [54], the term "higher–order logic" is ambiguous with several widespread readings used:

- from the point of view of philosophy and mathematics, logic can be divided into first–order and second-order. The latter is a formal basis for all of mathematics and, as a consequence of Gödel's first incompleteness theorem, cannot be recursively axiomatized.
- from the point of view of a proof theorist, all logics correspond to formal systems that are recursively presented and a higher–order logic is no different. The main distinction between a higher–order and a first–order logic is the presence in the former of predicate variables and comprehension, i.e., the ability to form abstractions over formula expressions.
- to many working in automated deduction, higher–order logic refers to any computational logic that contains typed terms and/or variables of some higher–order type, although not necessarily of predicate type.

According to [55], there are two main ways of adding meaningful higher–order extensions to the first–order base of Prolog:

- add a mechanism to handle predicates as data objects;

- add set expression implementations.

The former task is inspired by the facilities, provided by functional languages, in which it is possible to have a function call that is determined at run–time and, thus, may be constructed from parameters passed to the caller function. In turn there are two main ways for passing such higher–order parameters:

- pass a term that can be bound to an existing predicate at run–time;
- pass a so–called anonymous predicate, or predicate abstraction, through the means of a lambda expression.

However, this solution does not really introduce the power of higher–order expressions, as it can be easily translated into first–order, so at the end it is rather "syntactic sugar" for conventional syntax. The latter one, though, has an aim to combine the solutions generated by backtracking into a single data structure and in such a manner that introduces what is called *set expressions*. The main difficulty in implementing such functionality is remembering already discovered correct solutions when backtracking to get new ones, and, in turn, allow these expressions to be backtrackable themselves, so they can be used freely in any context.

The approach implemented in Ciao covers both of these approaches, supporting a form of higher–order untyped logic programming with predicate abstractions [57, 50, 12]. Predicate abstractions are Ciao's translation to logic programming of the lambda expressions of functional programming: they define unnamed predicates which will be ultimately executed by a higher–order call, unifying its arguments appropriately.

2.4.2 Implementation Issues

One of the foundations of Ciao, that enables, among other things, the underlying support for various analysis techniques and an easy way for performing language extensions, is a (predicate–based) module system [59]. Modules allow dividing programs into separate parts, which have their own independent name spaces and interfaces for communicating with the rest of the parts of the program. This provides the necessary support for reusing program patterns by means of having library predicates that can be customized in different ways when used in different modules (and, thus, in different *contexts*). While traditional design patterns for reusing code in Prolog are mainly represented with "skeletons and techniques" [60, 61, 56], cliches [62], program schemata [63] and logic description schemata[64], higher–order code can be seen as a more powerful alternative [56].

In the rest of the section we adopt several definitions from [65, 66] to address the practical issues of higher–order programming in modular systems, concentrating on the Ciao approach.

Module is a set of Prolog clauses associated to a unique identifier, the *module name*.

Goal is a logical formula over a definite set of predicate calls to be satisfied.

Closure is a callable term used to construct a goal by appending one or more additional arguments.

Qualified goal `M:G` is a classical Prolog goal `G` prefixed by a module name `M` in which it must be interpreted (a *context*).

Definition context for a predicate is a module that contains the definition of it.

Calling context for a predicate is a module from which a predicate is called. This can be a module where the predicate is defined in the case of a local call or another module assuming that the predicate is within scope.

Visibility. A predicate is *visible* from some context if it can be called from this particular context without any qualification.

Accessibility. A predicate is *accessible* from some context if it can be called from this particular context with or without qualification.

Meta–predicate is a predicate that handles arguments to be interpreted as goals. Those arguments are called meta–arguments.

The latter definition requires more attention than others, which we will devote to it in Section 2.4.3, but for the rest of this section the definition provided above is enough. Basically, here we consider meta–predicates as a way of supporting higher–order functionality in the presence of modularity by means of correct handling of contexts.

One of the key issues regarding modular systems are visibility rules. In Ciao the set of predicates visible in a module is comprised of the predicates, defined in this module plus the predicates imported from other modules. The *default module* for a given predicate name is the one which contains the definition of the predicate which will be called using the predicate name without module qualification. Such policy requires special handling for meta–programming usages, since in those cases functors can become predicate names and vice–versa.

Although module qualifications make it possible to refer to a predicate from a module which is not the default for it, thus allowing some call customization, still this will not work for predicates not visible from a module. In Ciao module qualifications are used only for disambiguating predicate names and not for changing the naming context. Thus, if in a module a local, unexported (and thus not visible out of the module) predicate is passed to a predicate argument of higher–order predicate, that in turn was imported from another module, the corresponding higher–order call in a module will fail because of the definition context of the higher–order predicate which is different from its calling context.

At the same time if the higher–order predicate is declared also as a meta–predicate, these context mismatching difficulties can be overcome. In Ciao the manipulation of such meta–data through the module system is possible through an advanced version of the `meta_predicate/1` directive. Before calling the meta–predicates, the system dynamically translates the meta–arguments into an internal representation containing the goal and the context in which the goal must be called. Since this translation is done before

calling the meta–predicate, the system correctly selects the context in which the meta–data must be called. As far as the system does not document any predicate able to create or manipulate the internal data, the protection of the code is preserved. However, this approach is a well–known source of misunderstandings, as it often leads to mixing the concepts of higher–order and meta–programming, which is discussed in the following section.

2.4.3 The Difference Between Meta– and Higher–order Predicates

In the present work we follow the point of view of [57], where the concepts of meta– and higher–order programming are separated, unlike the position, expressed in [66]. Additional information on the meta–programming facilities in logic programming can be found in [67, 68, 69].

In the Ciao approach the notion of higher–order programming implies having data, different from ordinary data, which represent processing units (in logic programming the natural choice will be predicates), and the ability of calling them instantiating their arguments. This good behavior is ensured by the following two principles:

- Higher–order data is syntactically differentiated from ordinary data. In this way any name of the program can be easily told apart as a predicate name or a data functor name. Additionally, a predicate name occurs always with its proper arity.

- Higher–order data is like an "abstract data type": it is regarded as a "black box" which cannot be inspected or manipulated, except by specific operations defined on it. This other property makes higher–order more amenable to effective global analysis.

At the same time, meta–programming can be seen as the manipulation of data representing code with the purpose of later executing it by its lifting to an executable status. Regarding the interaction of higher–order and modularity, and in order to respect module separation, names inside a predicate abstraction appearing in the code of a module are interpreted in the context of that module, and not in the context of the module where the apply builtin is ultimately invoked.

To sum up, higher–order predicates can be seen as mechanisms that execute their predicate arguments as "black boxes" without inspecting their structure, while the predicate arguments of meta–predicates are term expressions, enriched with the appropriate execution context for them. Detailed overviews of meta–predicate implementations in other LP systems can be found in [65, 66].

2.5 The Meta–programming Approach

One of the first approaches to the problem of reasoning about higher–order types in the context of logic programs is described in [70]. The main difference here from our intended approach is that the authors perform the analysis of *meta–types*, a related, but different concept. A meta–type can be seen as a pattern of a program part: a clause pattern like

`Head :- Body`, directive patterns, structure patterns, etc. PA–props can be derived from meta–types, but not vice versa. Still, the approach proposed by them covers some higher–order cases as well.

The authors highlight that in the general case a description of a type for a meta–structure is more complex than a description of a first–order type and they differentiate three kinds (*dimensions*) of such information:

- type–arguments and arity;
- argument application (currying);
- *nature* of the meta–type.

The first dimension can provide some information about the type–arguments of the meta–type and its arity. The type–arguments and arity of the meta–type must fit the type–arguments given in a definition of a predicate or in a predicate declaration, respectively.

The second dimension can provide a clear distinction between a function or a callable predicate with its arguments and a functor, as it is possible for both to appear in the place of a meta–type.

The third dimension can provide a description of the *nature* of a meta–type. For instance, for the previously described `map/3` predicate a meta–type declaration with a specified nature could look as follows:

```
:- pred map( meta(trmName(T1,T2) of mpred), list(T1), list(T2)).
```

where the first argument should now be a declared predicate (have a *nature mpred*). The concept of *nature* is hierarchical, as shown in Fig 2.1. The nature `mterm` gives no restrictions for meta–structures, its first refinement allows to separate the type definition `mtype` and the predicate declaration `mpred`, and several more cases are considered.

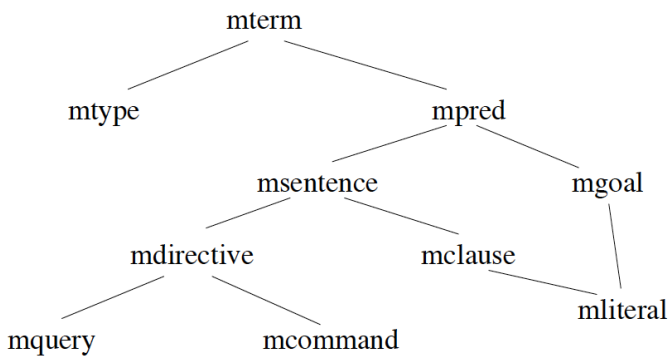

Figure 2.1: The hierarchy of natures for meta–types.

Finally, the overall combination of the three dimensions of meta–types and also an introduced “capsulation” of a standard `call/1` predicate, `fcall/N` (in a way similar to the Ciao suite of `call/N` predicates), which is capable of inspecting the structure of its argument(s), allows even more specified meta–types usage, as, for instance, in the case of `map/3`:

```
:- pred map( meta(trmName(list(T),list(T)) of mpred), list(@T),
list(@T) ).
```

where now it is possible to specify the types not only for the arguments of `map/3` itself, but also the types that are accepted by its first predicate argument.

Although this approach looks promising in the line of reasoning about higher–order calls, there are many limitations that make it undesirable for our purposes. We believe that using assertions for the analysis of predicate arguments of higher–order calls provides more flexibility and more possibilities, including:

- more flexible handling of types (including user–defined);
- a possibility of exploiting the directionality of types;
- a possibility of describing several I/O patterns for a predicate.

Chapter 3

Approach

Although the Ciao assertion language is quite expressive when introducing and checking various statements about different kinds of properties of first–order predicates, the ways in which the CiaoPP preprocessor can be provided with information about predicate arguments that contain higher–order terms are comparatively limited, as well as the amount of static or dynamic checks that can be performed for such arguments. One of the reasons behind this is that the predicate arguments of higher–order calls are instantiated in a manner that cannot always be checked statically and even adding run–time checks for such predicates is not a trivial task. However, higher–order predicates are extremely useful in many cases and it is desirable to fill this gap in the assertion language and add reasoning support for higher–order .

We start with the general idea of performing run–time checks of assertions for higher–order predicates as a way of reasoning about instantiation states of predicate arguments of higher–order calls. This can be achieved as a result of forming the necessary foundation basis, which consists of several key elements:

- allow writing assertions about predicates whose arguments may be instantiated with predicate abstractions;
- reuse the existing assertion language for this purpose minimizing the syntactic and semantic complexity introduced;
- separate the implicit data transformations problem (introduced by meta–predicates) from the task of dealing with predicates as data (higher–order programming approach);
- ensure that the language extension introduced is compatible with analyzers.

In the following we illustrate some practical uses of higher–order programming in logic programming. Those examples will allow us to motivate the need for an extended assertion language and verification tools. Then, we will provide the overview of the problem together with possibilities for solving it.

3.1 Higher-order: Uses and Challenges for Specification

3.1.1 Passing Data as Predicates

In declarative programming, program data is always explicitly passed as program arguments. However, efficient data encoding often needs the use of complex data structures. In some cases, Prolog facts provide a convinient and efficient way of storing data. However this has a main inconvenient in first-order programs: the predicate that stores the data has a fixed name in the algorithm. This problem can be solved with higher–order programming, by passing the predicate that stores the input data as an argument of the algorithm.

Example (Graph–related Problems: Finding Paths) When solving graph–related problems, such as graph traversal, one of the key issues is a way to represent the graph structure. There are many possibilities to do it in Prolog. Consider the following small graph and its representations, from [71] (Fig. 3.1).

```
% Arc-clause form
  arc(m,q,7).
  arc(p,q,9).
  arc(p,m,5).
% Graph-term form
  digraph([k,m,p,q],
      [a(m,q,7),a(p,m,5),a(p,q,9)])
% Adjacency-list form
  [n(k,[]), n(m,[q/7]), n(p,[m/5,q/9]),n(q,[])]
% Human-friendly form
  [p>q/9, m>q/7, k, p>m/5]
```

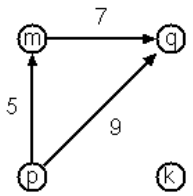

Figure 3.1: Simple graph and its possible representations in Prolog.

For finding a path between two nodes the author proposes the code in Fig 3.2.

However, this solution could be greatly generalized using higher–order calls and a simple representation of a graph by a predicate `arc/3` (Fig. 3.3). This approach has its benefits like the simplicity of graph representation, that there is no need to keep a list of edges and pass it as a parameter, and that edges and vertices can be easily added or removed. The main drawback here is that we lose the guarantee that the data that is being processed is a valid graph representation. Thus it would be desirable to check whether the predicate arguments of `path/4` and `path1/4` are instantiated to a suitable predicate. In this case we may want to specify that `Pred` should be a predicate with *atomic* (atom or integer) arguments.

3.1.2 Data Structure Manipulation (Classic higher–order Predicates)

Manipulation of algebraic data structures (like lists or trees) is often implemented as programs that perform structural recursion on data. Most of those programs show an specific structure where only some specific parts change (the constructors for the data structure,

```
path(G,A,B,P) :- path1(G,A,[B],P).

path1(_,A,[A|P1],[A|P1]).
path1(G,A,[Y|P1],P) :-
        adjacent(X,Y,G),
        \+ memberchk(X,[Y|P1]),
        path1(G,A,[X,Y|P1],P).

% A useful predicate: adjacent/3
adjacent(X,Y,graph(_,Es))   :- member(e(X,Y),  Es).
adjacent(X,Y,graph(_,Es))   :- member(e(Y,X),  Es).
adjacent(X,Y,graph(_,Es))   :- member(e(X,Y,_),Es).
adjacent(X,Y,graph(_,Es))   :- member(e(Y,X,_),Es).
adjacent(X,Y,digraph(_,As)) :- member(a(X,Y),  As).
adjacent(X,Y,digraph(_,As)) :- member(a(X,Y,_),As).
```

Figure 3.2: Path finding solution.

```
path(Pred,A,B,P) :- path1(Pred,A,[B],P).

path1(_   ,A,[A|P1],[A|P1]).
path1(Pred,A,[Y|P1],P     ) :-
        Pred(X,Y,_),
        \+ memberchk(X,[Y|P1]),
        path1(G,A,[X,Y|P1],P).
```

Figure 3.3: An example of finding paths in graph with higher–order calls.

mappings between elements, filtering criteria, etc.). Classical higher–order predicates borrowed from functional programming, like `foldl/4`, `map/3`, `filter/3`, `split/4`, etc. fall into this category.

Example (Minimum Element of a List) One of the possible examples that fall into the category of classical uses of higher–order is a predicate that finds the minimal element in a list according to some order releation. A possible implementation is shown in Fig. 3.4. Let us consider a case in which valid input data is represented by integers and lists of integers. In such a case it would be desirable to check that the second argument is a predicate that accepts only integers as inputs. However, the current assertion language implementation allows only to specify that we expect a callable term in the position of the second argument.

3.1.3 Support for Generic Programming

Despite *generic programming* often requires sophisticated language mechanisms and type systems, higher–order programming can be used as the basis for abstracting concrete and efficient algorithms. For example, sorting algorithms can be parameterized to work with

```
:- meta_predicate minimum(_, pred(2), _).
:- pred minimum(?List, +SmallerThan, ?Minimum)
   : list * callable * term .

minimum([X|Xs], Pred, Min) :-
        min(Xs, Pred, X, Min).

min([],     _Pred, M,       M  ).
min([X|Xs], Pred, CurrMin, Min) :-
        (  Pred(CurrMin, X)
        -> min(Xs, Pred, CurrMin, Min)
        ;  min(Xs, Pred, X, Min)
        ).
```

Figure 3.4: An example of `min/3` implementation in Ciao.

different data structures or order relations.

Example (Parameterized Sorting Algorithms) The famous quicksort algorithm implemented in Prolog is shown in Fig. 3.5. If one would like to change the standard total ordering of elements, used in `partition/4`, the only way for this implementation is only to rewrite the predicate itself. Making `partition/4` a higher–order predicate eliminates this problem, but at the same time one may want a way to perform a check whether a passed comparison predicate is compatible with the data arguments.

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

partition([],_B,[],[]).
partition([E|R],C,[E|Left1],Right):-
        E @< C,
        partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        E @>= C,
        partition(R,C,Left,Right1).

append([],X,X).
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
```

Figure 3.5: An example of quicksort implementation in Ciao.

3.2 Scope

Considering the aforementioned issues and keeping in mind the possibilities of the Ciao machinery for (assertion) language extensions and the peculiarities of higher–order implementation, we come up with the following approach to the problem:

- we extend the set of properties that can be used in assertions, with new ones to describe predicate arguments in higher–order predicates, calling them *properties of predicate abstractions (PA–props)*;
- we exploit full assertions as parts of these properties to keep and reuse the expressiveness of the assertion language in predicate specifications;
- we reuse the existing machinery of run–time checks for regular assertions to perform the checks for PA–props ;
- we introduce an additional layer of source–to–source transformations to add the support for higher–order properties within the framework of run–time checks for the first–order predicates;

A number of other issues deserve mentioning: First, the properties that we are introducing fall into the category of *instantiation* properties, and not *compatibility* ones (recall this from p. 10 in Chap. 2). In addition to the implementation issues, that we have to take into account, the meaning of *compatibility* properties is not straightforward in this context.

Next, in this work we aim at a simple first solution that does not change the machinery of analyzers and the CiaoPP preprocessor and performs at most some module–local analysis, but we do not attempt to perform any reasoning about the behavior of the whole program and/or imported libraries.

3.3 An Extension to the Assertion Language

Several questions should be discussed before we come up with a set of properties for predicate arguments of higher–order predicates that are suitable for the approach considered above. First, we need to choose the way that statements about higher–order variables are introduced. There are two main ways to do this:

- add new types of assertions;
- add new properties and use them in the current set of assertions.

The former case may sound more natural but at the same time is requires performing general changes not only in the structure of the assertion language, but also in all the tools that make use of it, which is undesirable. The latter approach does not have such demands and can be implemented as a combination of a set of new properties for predicate arguments and a corresponding translation, which manages the mentioned matching of

arguments and regular assertions. Our design choice for the language extension follows exactly this approach.

Next, we need to specify the way of introducing new properties. Two possible solutions can be considered:

- allow the user to write some ad–hoc higher–order property definitions.
- provide some built–in mechanism to allow providing full assertions within PA–props .

The second option is much more elegant and desirable, and will also facilitate program understanding and automated analysis, since it is essentially a recursive extension of the assertion language.

Keeping in mind these considerations, we present below a description of the PA–props that represent predicate arguments in assertions for higher–order predicates. Our definition of PA–props follows in its syntax the compound form of the predicate assertion (see Sec. 2.3.6, p. 19), as it is considered to represent a predicate with its behavior and not a data property.

Another major issue regarding PA–props is the issue of `meta_predicate` declarations, which are required in many cases by the low–level compiler. As we have described before (see Sec 2.4.3), we distinguish pure higher–order programming style and meta–programing facilities. However, to be able to take into account both issues, our PA–props implementation should also reflect this separation. Meta–programming support is already implemented in Ciao, so there is no need to for a re–implementation. Thus, we concentrate our attention on the higher–order part, narrowing meta–programming issues as much as possible.

We introduce PA–props as arguments of binary properties that relate both the *caller* higher–order predicate and the first–order *callee* predicate, passed as parameter. To refer to the meta–nature of the caller (typically expressed by a `meta_predicate` declaration), we provide two kinds of those properties: ones that imply `meta_predicate` declaration for the caller, and ones that don't do it.

In our language extension we chose the following schemas for PA–prop declarations:

- Explicit syntax:

  ' ' (*Arg1, . . ., ArgN*) [: *Precond*] [=> *Postcond*] [+ *Comp-prop*]

- Abridged syntax:

  ' ' / *Arity* [: *Precond*] [=> *Postcond*] [+ *Comp-prop*]

where the mandatory part of the declaration (' ' / *Arity* in abridged form) is syntactically similar to Ciao's predicate abstraction (with *Arity* denoting a number of arguments of the *callee*), and all the optional parts (enclosed in square brackets) keep the same meaning as they do in the regular assertions. Thus, the mentioned properties take the form:

- for arguments of higher–order *caller* that need to be defined as a meta–predicate:

`mprd(` *Callee* , *PA–prop* `)`

- for arguments of higher–order *caller* that need not to be defined as a meta–predicate:

  `prd(` *Callee* , *PA–prop* `)`

```
:- module(usr_lib,_,[hiord,assertions,hiochk]).

:- meta_predicate minimum(_, pred(2), _).
:- pred minimum(?List, +SmallerThan, ?Minimum)
   : list * prd(''/2 : (int * int)) * term .

minimum([X|Xs], Pred, Min) :-
        min(Xs, Pred, X, Min).
```

Figure 3.6: An example of higher–order assertion for the `minimum/3` predicate (abridged syntax).

We illustrate the use of PA–props with the following two examples, that derive from the `minimum/3` predicate (see Fig. 3.4).The first example (see Fig. 3.6) refers to the case when the `meta_predicate` declaration is explicit, the second example (see Fig. 3.7) is semantically identical to the first one. The PA–prop `''/2 : (int * int)` denotes a predicate with two arguments, both of which should be instantiated to integers at the moment of the higher–order call.

```
:- module(usr_lib,_,[hiord,assertions,hiochk]).

:- pred minimum(?List, +SmallerThan, ?Minimum)
   : list(List),
     mprd(SmallerThan,''(A,B):(int(A),int(B))).

minimum([X|Xs], Pred, Min) :-
        min(Xs, Pred, X, Min).
```

Figure 3.7: An example of higher–order assertion for the `minimum/3` predicate (with implicit `meta_predicate` declaration).

3.4 Extending the Runtime Checking Machinery

As described in Section 2.2, in Ciao run–time checks for predicates are performed by means of the assertion language and source–to–source program transformations. To address the extension of the checking machinery to support our extended assertion language several solutions are possible, keeping in mind the specificity of dealing with higher–order arguments in modular systems.

In this section we introduce several alternatives to the extension of the current run–time checks mechanism. Although we concentrate on performing checks for PA–props during runtime, we see several possibilities to associate higher–order calls with corresponding checks for PA–props . We call the process of such association *checks propagation* and such calls *affected calls*. It can be performed either statically (at compile time) or dynamically (at runtime).

For those PA–props checks, that are propagated statically, we define the *scope* of the check propagation:

- *predicate*, if the check is performed for each higher–order call, that occurs in clauses of the predicate, in assertion for which a PA–prop appears.
- *module*, if the check is performed for all *affected* higher–order calls that appear in one module.

For those PA–props checks, that are propagated dynamically, we define the *temporal direction* of the check propagation:

- the check for a PA–prop is propagated through *subsequent* higher–order calls, if it is performed for those *affected* calls, that occur after the program point in which the execution reaches the PA–prop check definition in some predicate assertion.
- the check for a PA–prop is propagated through *all* higher–order calls, if it is performed for all *affected* higher–order calls that take place during the program execution.

We illustrate these check propagation choices in Figure 3.8.

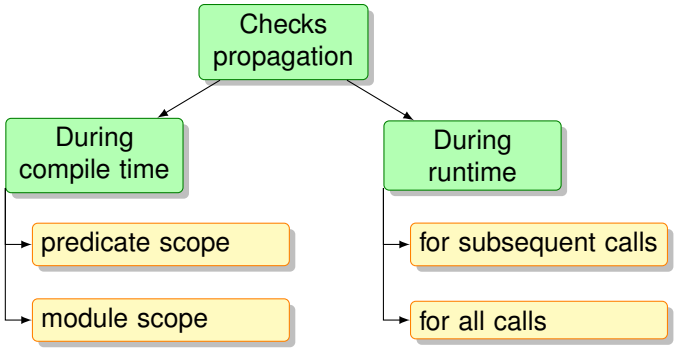

Figure 3.8: Checks for PA–props propagation possibilities.

We distinguish the possible alternatives by the checks propagation strategy they implement. Of course, it would be desirable to have all *affected* higher–order calls checked, but at the same time another major objective is to minimize the computational effort involved in the checking as much as possible, without losing checking precision.

Below we discuss the characteristics of the possible implementations of each of the alternatives. We provide a reference name for each of them in order to keep track of the features of each of them.

3.4.1 Static Checks Propagation Alternatives

ChkLocal Alternative

The first alternative to consider is *ChkLocal*, which reflects the static propagation strategy in *predicate* scope for the checks of PA–props . The type of checking done is similar to that of the regular *calls* and *success* predicate assertions: it implies a local check(s) inside the body of the corresponding higher–order predicate for all the higher–order calls.

To illustrate this and other alternatives, let us consider a small program, with a higher–order predicate `c/1`, which accepts a unary predicate as its argument, and an assertion for `c/1` with a PA–prop , which requires the argument to be compatible with unary predicate with argument, for which property `one/1` holds (please assume that `one/1` is defined as `:- regtype one/1. one(1).`).

Then, the *ChkLocal* alternative would allow detecting incorrect higher–order calls within a body of `c/1`, for instance, `P(0)` (we denote such places as `Chk`):

```
:- pred c(P) : prd(P,''/1: int).
c(P) :- Chk.
```

This alternative introduces minimal overhead in the program, which can be reduced even more if we decide to treat checks here as *entry*/*exit* pairs (thus not repeating the PA–prop check for recursive calls of the calling higher–order predicate). However, the precision of the checking in this case will be comparatively poor, which will be shown in the following examples.

ChkPropMod Alternative

The next alternative to consider is *ChkPrpMod*, which reflects static propagation strategy in *module* scope for the checks of PA–props . It is similar to the previous one, but extends it by propagating the PA–prop check to those places, where other affected higher–order calls occur (within the same module). This approach does not propagate PA–props across modules, but for the programs which have all higher–order calls within one module this alternative probably represents the best trade–off between the checking precision and computational cost.

Consider a simple extension of the small program we used to illustrate the previous alternative:

```
:- pred c(P) : prd(P,''/1: int).
c(P) :- Chk_1, d(P), Chk_3.

d(P) :- Chk_2.
```

With the *ChkPropMod* alternative the incorrect higher–order calls could be detected not only within the body of predicate `c/1` (positions $\texttt{Chk}_1$ and $\texttt{Chk}_3$), but also inside the body of `d/1` (position $\texttt{Chk}_2$).

The common feature that these two alternatives share is that the propagation of the PA–props is performed at compile time, minimizing the overhead that may be introduced by other solutions that implement PA–prop propagation at run time.

3.4.2 Dynamic Checks Propagation Alternatives

DelayChk Alternative

The alternative, that supports the check propagation for *subsequent* higher–order calls, is the one we call *DelayChk*. The main idea behind it is to store a list of existing PA–props checks and for each higher–order call try to match it with an appropriate check as soon as the execution mechanism can perform a call (thus, *delaying* the check until the very moment before or after the corresponding higher–order call).

Let us now extend our simple example by passing predicate variables between higher–order predicates that are defined in different modules `m1` and `m2`:

```
% m1.pl                                  % m2.pl
:- pred c(P) : prd(P,''/1: int).        e(P) :- Chk3
c(P) :- Chk1, d(P), Chk5.

d(P) :- Chk2, e(P), Chk4.
```

With the *DelayChk* alternative the incorrect higher–order calls now could be detected not only for the affected calls that occur in the bodies of predicates `c/1` and `d/1` (positions $\texttt{Chk}_1$, $\texttt{Chk}_2$, $\texttt{Chk}_4$ and $\texttt{Chk}_5$), but also for those that take place in the body of `e/1` (position $\texttt{Chk}_3$).

The main benefit of this approach is that no higher–order calls affected by a check are omitted. Still the drawback is that it requires major changes in the call handling mechanism.

TblChk Alternative

The alternative that supports the check propagation for *all* higher–order calls is the one we call *TblChk*. The main idea behind it is to keep track of all the calls and their successes/failures for a particular program during its execution. Thus, would be possible to check them by looking through the table of calls and informing the user for which cases the PA–prop checks have failed/succeeded.

Let us extend our sample program once more, so now there is a higher–order predicate `b/1`, that starts passing a variable, instantiated to a predicate of arity 1, to other higher–order predicates before there occurs any assertion that could specify PA–prop for this variable:

```
% m1.pl                                  % m2.pl
b(P) :- Chk1, c(P), Chk7.                e(P) :- Chk4.

:- pred c(P) : prd(P,''/1: int).
c(P) :- Chk2, d(P), Chk6.

d(P) :- Chk3, e(P), Chk5.
```

With the *TblChk* alternative the checks for PA–prop now would be performed not only for the higher–order calls that occur after the assertion for `c/1` (positions $\texttt{Chk}_2$ – $\texttt{Chk}_6$), but also for those that take place in the body of `b/1` (positions $\texttt{Chk}_1$ and $\texttt{Chk}_7$).

Still, the main limitation of this approach is the need for storing the table of higher–order calls, which can obviously be quite costly for large programs. Moreover, in this scenario the checks of PA–props can be applied only to those calls that appear in the program after them.

3.4.3 Measuring the Precision of Checking

We describe here a family of artificial test programs to measure the efficiency of each of the alternatives described before. Let us assume five predicates `a/1`, `b/1`, `c/1`, `d/1`, `e/1`. Predicates `b/1`, `c/1`, and `d/1` are all defined in the same module, while `a/1` and `e/1` are both defined in a different module. We define a family of programs $\texttt{T}_i$ as follows:

```
a(P) :- Chk1,b(P),Chk9.
b(P) :- Chk2,c(P),Chk8.   :- regtype one/1. one(1).
c(P) :- Chk3,d(P),Chk7.   :- pred c(P) : prd(''/1 : one).
d(P) :- Chk4,e(P),Chk6.
e(P) :- Chk5.
```

where $\texttt{Chk}_j$ is `P(0)` if $i = j$ or `true` otherwise. We assume that `P` is instantiated at the time of calling `a/1` with a predicate of arity one. Only `c/1` has an assertion that specifies what property should hold for `P`. In our case, `P(0)` is a wrong call. The test program $\texttt{T}_i$ will measure if the HO-checking method is able to detect a wrong call at the i-th location. We mark calls to `P(0)` as "×" when a wrong call is detected, or "–" otherwise. The detection of wrong calls for each alternative is shown in Table 3.1. Note that only the *ideal* case is able to detect incompatibility between higher–order predicates before they are actually called (the $\texttt{T}_0$ case).

	Ideal	*ChkLocal*	*ChkPropMod*	*DelayChk*	*TblChk*
$\mathtt{T}_0$	×	–	–	–	–
$\mathtt{T}_1$	×	–	–	–	×
$\mathtt{T}_2$	×	–	–	–	×
$\mathtt{T}_3$	×	×	×	×	×
$\mathtt{T}_4$	×	–	×	×	×
$\mathtt{T}_5$	×	–	–	×	×
$\mathtt{T}_6$	×	–	×	×	×
$\mathtt{T}_7$	×	×	×	×	×
$\mathtt{T}_8$	×	–	–	×	×
$\mathtt{T}_9$	×	–	–	×	×

Table 3.1: Precision of the different HO run–time checking alternatives.

Chapter 4

Implementation

In this chapter we present the implementation of some of the alternatives presented in Chapter 3, using the language extension mechanism of Ciao and the existing run–time checking facilities, which we will also briefly describe. Our approach extends the language in two directions, reusing the existing extensions for assertions and run–time checking. Syntactically, it extends the assertion language with higher–order types (PA–props), that allow the specification of predicate abstractions with some desired properties (written themselves with the assertion language). Semantically, it extends the run–time checks facilities to check such higher–order types (reusing all the run–time checking machinery for the first-order case).

4.1 Language Extensions in Ciao

In Ciao, language extensions can be implemented as different groups of syntactic definitions and translation rules, that can be stacked one on top of the other, extending a core base language [72]. Extensions are encoded by means of *packages*: libraries that define extensions to the language with a well defined structure. Such extensions are local to the module or user file defining them [59]. This not only improves modularity, it also allows the use of different extensions in different parts of the same application.

Packages consist of a main source file, which includes syntactic declarations (e.g., operator declarations), the auxiliary run–time code, and other declarations that specify the modules that implement the language translation. Packages are used in modules just by including their main file, using the special `use_package` directive. The `load_compilation_module/1` directive specifies the modules that contain the auxiliary code that is needed at compile–time to perform program translations. This directive allows separating code that will be used at compilation time (e.g., the code used for program transformations) from code which will be used at run–time. It loads the module defined by its argument into the compiler. The effects usually achieved through conventional `term_expansion/2` in Prolog can be obtained in Ciao by means of four different, more specialized directives, which, again, affect only the current module and are (by default) only active at compile–time.

Each of such directives defines a different target for the translations. The argument for all of them is a predicate indicator of arity 2 or 3. When reading a file, a general-purpose

module processing library (embedded in the Ciao compiler [73]) invokes such translation predicates at the right times, instantiating the first argument with the item to be translated. If the predicate is of arity 3, the optional third argument is instantiated with the name of the module where the translation is being done. If the call to the expansion predicate is successful, the term returned by the predicate in the second argument is used to replace the original. Otherwise, the original item is kept. The directives are:

`add_sentence_trans/1`: declares a translation of the terms read by the compiler which affects the rest of the current text (module or user file). For each subsequent term (directive, fact, clause, assertion, etc.) read by the compiler, the translation predicate is called to obtain a new term which will be used by the compiler in place of the term present in the file.

`add_term_trans/1`: declares a translation of the terms and sub–terms read by the compiler which affects the rest of the current text. This translation is performed after all translations defined by `add_sentence_trans/1` are done. For each subsequent term read by the compiler, and recursively any subterm included in such a term, the translation predicate is called to possibly obtain a new term to replace the old one.

`add_clause_trans/1`: declares a translation of the clauses of the current text. This translation is performed after `add_sentence_trans/1` and `add_term_trans/1` translations. This kind of translation is defined for more involved translations and is related to the compiling procedure of Ciao. The usefulness of this translation is that information on the interface of related modules is available when it is performed.

`add_goal_trans/1`: declares a translation of the goals present in the clauses of the current text. The translation is performed after all translations defined by directives `add_sentence_trans/1`, `add_term_trans/1` and `add_clause_trans/1` (optionally) are done. For each clause read by the compiler, the translation predicate is called with each goal present in the clause to possibly obtain another goal to replace the original one, and the translation is subsequently applied to the resulting goal. This translation continues recursively, until a fixpoint is reached and it is aware of `meta_predicate` definitions.

This interaction order among translations is shown in Fig. 4.1 (taken from [59]).

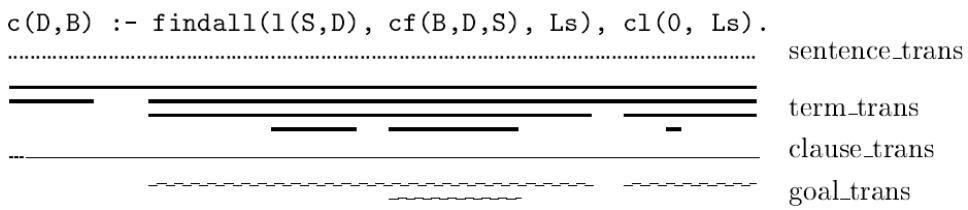

Figure 4.1: The order and subjects of source-to-source translations.

The proposed higher–order assertion syntax is implemented as a Ciao package that performs a source-to-source translation. In particular, we have used the `add_sentence_trans/1`

directive. A sentence translation is a predicate which will be called by the compiler to possibly convert each term (clause, fact, directive, input, etc.) read by the compiler to a new term, which will be used in place of the original term.

We exploit sentence translation both for translating higher–order properties in assertions and the corresponding higher–order calls in predicate bodies in order to be able to perform run–time checks. Though the conventional path is to use goal translation in the latter case, it is not suitable way of source transformation in our approach due to the aspect of dealing with higher–order meta–predicates and their meta–arguments, that are goals themselves and thus would fall into an infinite translation loop if not taken care of.

4.2 Run–time Checks

Run–time assertion checking is performed in the Ciao framework mainly by means of source-to-source translation. This is based on transforming the program into a new one, which preserves the semantics of the original program and at the same time checks during its execution if the program assertions are violated for some particular concrete data.

One approach to this translation is described in [74], implemented in `rtchecks` package. There is an alternative mechanism built into the CiaoPP processor, but in this work we have used the `rtchecks` library as the starting point, since is is a smaller, standalone package.

The Ciao run–time checking system is composed of a set of transformations, to be performed by the preprocessor, and a library containing a number of primitives that the transformed programs will call. Applying the transformation that is called `transforming procedure definitions`, the original predicate is rewritten so that it performs the run–time checks itself, each time it is called, and calls to it are left unchanged. This procedure can be seen as done in two steps (see Fig. 4.2 (adapted from [74]) for process illustration).

```
p :-
   entry-checks,
   exit-preconditions-checks,
   ext-comp-checks (p1),
   exit-postconditions-checks.


% p renamed to p1 within
% module
```

```
p1 :-
   calls-checks,
   success-preconditions-checks,
   comp-checks(
     call stack(p2, locator)),
   success-postconditions-checks.
p2 :- body0.
...
p2 :- bodyN.
```

Figure 4.2: Ciao run–time checks program transformation.

In this transformation the original predicate `p` is renamed to `p2` and a new definition of `p`, which performs the run–time checks, is added:

1. run–time checks, corresponding to, e.g., entry and exit assertions before and after a call, are added to a new predicate `p1`. The objective of this first transformation is to separate external calls from internal ones.

2. p1 is defined so that it calls predicate p2 and performs all run–time checks corresponding to each type of predicate–level assertions, i.e., calls, success, or comp in the right place.

4.3 Meta–types and Higher–order types

In most Prolog implementations, as well as in Ciao, there is no explicit syntax for dealing with higher–order predicates. Instead, a form of meta–programming is supported by meta_predicate declarations. As mentioned before, it is important to distinguish between the two notions of meta–types and higher–order types.

In Ciao meta–predicate declarations introduce an implicit *casting* operation that composes the execution context at the caller site with the actual argument, to create a meta-term. That is, those meta–predicate declarations are required when passing predicates across modules (that is, when changing execution context). In the most general sense, meta–programming involves manipulation of meta-terms (e.g., deconstructing goals, manipulating goal arguments, etc.). However, in this work we are only interested in executing those meta-terms. In practice, we can map this particular view of meta-terms to predicate abstractions.

Having no direct notion of higher–order predicates, our implementation of higher–order assertions will need to take this peculiarity into account. We take care of it by performing a transformation between meta–types (which involve an implicit casting) and PA–props (that assume that data is a predicate abstraction, a native and opaque element). Basically, this can be seen as removal of the syntactic sugar for meta–types. Such source expansion is illustrated in Fig. 4.3. [1]

```
1
2
3 :- pred p(P,...)
4    :(mprd(P,''/2:(int*int))).
```

```
1 :- meta_predicate
     p(pred(2),...).
2 :- pred p(P,...)
3    :(prd(P,''/2:(int*int))).
```

Figure 4.3: Meta–types to PA–props translation example: mprd/2 to prd/2.

Note that in general, unlike PA–props, meta–types cannot be propagated or inferred without semantically affecting the program.

Another interesting issue regarding meta_predicate declarations is that in many cases only one declaration is valid for each predicate. For example, consider the erroneous example in Fig. 4.4: the user–provided declaration describes p as a higher–order predicate with first argument a predicate of arity 2, but the meta–type states that a predicate of arity 1 is expected. This is not a problem if we describe just higher–order types (as there can be one clause of p using each different version).

[1] An alternative to this approach is the introduction of special syntax to build predicate abstractions.

```
1 :- meta_predicate p(pred(2),?).
2 :- pred p/2:(mprd(''/1:(list)) * list).
```

Figure 4.4: Meta–types translation example: incompatible `meta_predicate` declarations.

4.4 A Source-to-Source Transformation-Based Approach for Higher–order Checking

We perform run–time checking of the properties introduced in Section 3.3 by means of a source-to-source transformation that introduces checks at the program points where higher–order call are invoked. This transformation consists on a two–pass process. For our task it is desirable to extract PA–props from assertions in the first phase and to modify the corresponding higher–order calls in the second. In the first pass, PA–prop information is propagated though the program variables. During the second pass, this type information is used to transform unchecked higher–order calls into checked higher–order calls. In this way, we can effectively check PA–props at run–time during the program execution. [2]

To describe the translation, we adopt the following definitions:

Caller: a higher–order predicate with at least one argument that is instantiated to a predicate a run–time.

Callee: a predicate (usually, first–order) passed as an argument of a higher–order caller.

Check: a tuple of four parameters that relate the predicate argument of the caller with the corresponding meta–type from the corresponding predicate assertion:

- caller head;
- integer number that corresponds to a position of predicate argument for which a PA–prop is provided by user;
- the PA–prop itself;
- the name of the module that is the definition context of the caller.

The translation establishes the relation between the PA–prop and the call of the corresponding callee. It is done in two steps. During the first one a predicate assertion for the caller is expanded so that a check, composed of a meta–type definition, is asserted to a program as a fact of a special `data` predicate. During the second step, each corresponding callee invocation of the form `call(Callee,Args)` is substituted with a call to an auxiliary first–order predicate. We expand the check as a regular predicate assertion for the auxiliary predicate. This transformation is illustrated in Fig. 4.5.

[2] Obviously, this method will only detect incorrect executions in programs where PA–props are violated at HO call locations.

```
:- pred p(P,...)
    :(prd(P,''/2:(int*int))).
p(P,Arg1,...,ArgN) :-
   ...,
   P(P,Arg1,ArgN),
   ....



% P(...) == call(P,...)
```

```
:- pred p(P,...).
p(P,Arg1,...,ArgN) :-
   ...,
   p_1(P,Arg1,ArgN),
   ....

:- pred p_1(P,Q,R)
   :(int(Q)*int(R)).
p_1(X,A,B):-
   call(X,A,B).
```

Figure 4.5: PA–props translation example: `call/N` expansion.

Please note that the arity of the auxiliary predicate is larger than the arity of the anonymous predicate from the meta–type by 1, due to the necessity of passing the callee as an argument of the system `call/N` predicate.

Below we explain the policy for PA–prop propagation adopted in the translation. This kind of source-to-source transformation covers those cases in which callee is local to the module of the higher–order predicate itself, for which no assertions with PA–props are given. To make the run–time checking as precise as possible, we propagate the check from caller to callee, changing respectively the first and the second parameters of the check. This transformation is illustrated in Fig. 4.6.

```
:- pred p(P,...)
    :(prd(P,''/2:(int*int))).
p(P,Arg1,...,ArgN) :-
   ...,
   q(P,Arg1,ArgN),
   ....

q(Pr,Arg1,...,ArgM) :-
   ...,
   Pr(ArgM,Arg1),
   ....



%
```

```
:- pred p(P,...).
p(P,Arg1,...,ArgN) :-
   ...,
   q(P,Arg1,ArgN),
   ....

q(Pr,Arg1,...,ArgM) :-
   ...,
   q_1(ArgM,Arg1),
   ....

:- pred q_1(P,Q,R)
   :(int(Q)*int(R)).
q_1(X,A,B):-
   call(X,A,B).
```

Figure 4.6: PA–props translation example: check propagation.

4.4.1 General Translation Workflow

The translation mechanism proposed covers several tasks:

1. For the properties that imply a `meta_predicate` declaration for the caller, consider two cases (and later treat the higher–order part):

- if there is already such a declaration, check whether it is consistent with the meta–type definition and report an error if not;
- if there is no such declaration, add one.

2. For higher–order properties, replace all the higher–order calls in bodies of corresponding callers by calls to the auxiliary first–order predicates and for each auxiliary predicate add an assertion, corresponding to the meta–type from the caller declaration.

The translation has two principal phases:

- source-to-source transformations of higher–order predicates and assertions with PA–props;
- source-to-source transformations of meta–programming parts.

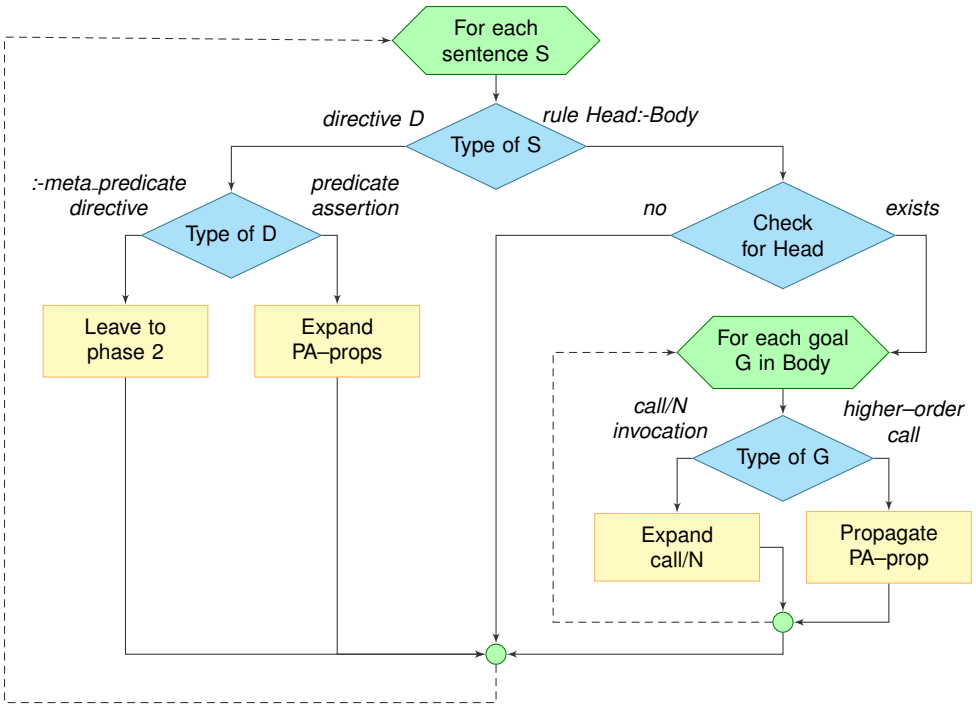

Figure 4.7: Source-to-source translation: phase 1 of 2.

During the first phase (see Fig. 4.7) source expansion for higher–order takes place. While the translation processes the module sentence–by–sentence it infers checks to be combined with corresponding higher–order calls, expands such calls in the manner described previously, but both read and inferred `meta_predicate` declarations are kept as `data` predicates facts until the beginning of the second translation phase.

The second phase of the translation (see Fig. 4.8) begins when the end of the module is read. During it both read and inferred `meta_predicate` declarations are analyzed and if no incompatibility errors occur, they are returned to the module.

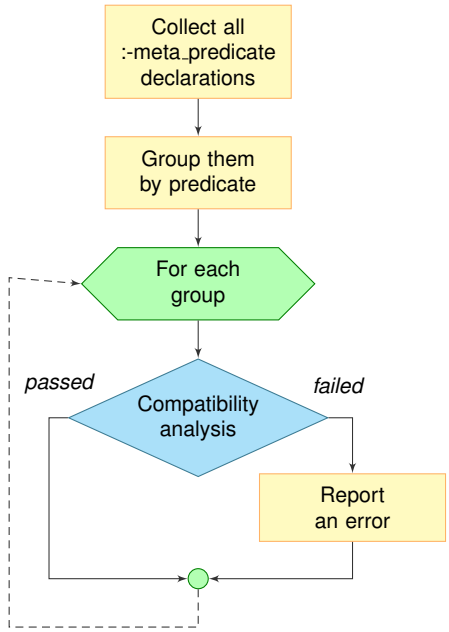

Figure 4.8: Source-to-source translation: phase 2 of 2.

After these source-to-source transformations take place, all the PA–props and corresponding calls are rewritten as conventional first–order predicates and assertions. This way the consequent translation performed by the `rtchecks` package is able to construct such run–time checks that precisely represent the PA–props for the corresponding calls.

Chapter 5

Evaluation and Experimental Results

We present below some experimental results from our implementation within the Ciao system of the proposed run–time checking of properties of PA–props of predicate arguments of higher–order predicates. To evaluate the performance of programs using such checks we measure the increase in execution time caused by the checking-related program transformations and the increase in compilation time caused by PA–prop propagation.

We distinguish two main sources of run–time overhead that slow down the performance:

- run–time checks for regular properties in (predicate) assertions;
- run–time checks of PA–props in predicate assertions.

The former source of run–time overhead is the one corresponding to the source transformation, performed by `rtchecks` Ciao package, while the latter is introduced by our implementation of PA–props checks.

We start with the performance evaluation of the recursive predicate `qsort/2-3` of Fig. 3.5. We are particularly interested in performing checks of the instantiation states of the variables `E` and `C`, which are used in the basic comparison operation within the `partition/4-5` predicate. We chose the following four benchmarks for performance tests:

1. Optimal case performance as explicit instantiation check;
2. Conventional predicate assertion check at run–time;
3. Check for a PA–prop propagated through `partition/5`;
4. Check for a PA–prop propagated through `qsort/3`;

The first benchmark is shown in Fig. 5.1. In this version the type checking is introduced by hand (`lessthan/2` clause). We consider this version, which directly calls the necessary instantiation checks at the right places, as optimal in terms of run–time overhead (before doing program optimizations statically). We take it as the baseline for comparison with other versions.

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

partition([]   ,_,[]       ,[]   ).
partition([E|R],C,[E|Left1],Right):-
        lessthan(E,C),!,
        partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        partition(R,C,Left,Right1).

lessthan(A,B) :- integer(A), integer(B), !, A @< B.
```

Figure 5.1: Benchmark 1: Hand–coded type check.

The second benchmark is shown in Fig. 5.2. Here the types of the arguments of the comparison predicate are checked with the current system implementation of run–time checks. This use case reflects the run–time overhead added by the `rtchecks` package and it can be considered as a measure of the real performance impact in practice.

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

partition([]   ,_,[]       ,[]   ).
partition([E|R],C,[E|Left1],Right):-
        lessthan1(E,C),!,
        partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        partition(R,C,Left,Right1).

:- pred lessthan1(X,Y) : (int(X),int(Y)).
lessthan1(A,B) :- A @< B.
```

Figure 5.2: Benchmark 2: First–order run–time checks.

The third benchmark is shown in Fig. 5.3. Here we use higher–order versions of both `qsort/2` and `partition/4` and to check the types of the arguments of the call `Pr(E,C)` we specify a PA–prop for the predicate argument of the caller `partition/5`. In this case no PA–prop propagation takes place.

```
:- meta_predicate qsort(?,pred(2),?).
qsort([],_,  []).
qsort([X|L],P,R) :-
        partition(L,P,X,L1,L2),
        qsort(L2,P,R2),
        qsort(L1,P,R1),
        append(R1,[X|R2],R).

:- pred partition(A,T,B,C,D) : (mprd(T,''/2:(int*int))).
partition([] ,_  ,_,[]       ,[]   ).
partition([E|R],Pr,C,[E|Left1],Right):-
        Pr(E,C),!,
        partition(R,Pr,C,Left1,Right).
partition([E|R],Pr,C,Left,[E|Right1]):-
        partition(R,Pr,C,Left,Right1).
```

Figure 5.3: Benchmark 3: PA–prop check propagation 1

The fourth benchmark is shown in Fig. 5.4. It is similar to the third benchmark, the only difference between them is that the PA–prop of the predicate variable is specified for `qsort/3`, so here the PA–prop propagation takes place, from caller `qsort/3` to callee `partition/5`.

```
:- pred qsort(L1,T,L2) : (mprd(T,''/2:(int*int))).
qsort([], _, []).
qsort([X|L],P,R) :-
        partition(L,P,X,L1,L2),
        qsort(L2,P,R2),
        qsort(L1,P,R1),
        append(R1,[X|R2],R).

partition([],_,   _,[]       ,[]   ).
partition([E|R],Pr,C,[E|Left1],Right):-
        Pr(E,C),!,
        partition(R,Pr,C,Left1,Right).
partition([E|R],Pr,C,Left,[E|Right1]):-
        partition(R,Pr,C,Left,Right1).
```

Figure 5.4: Benchmark 4: PA–prop check propagation 2

The benchmarks described above were executed on two different computers, whose specifications are described in Table 5.1.

Parameter	MacBookAir2,1 [1]	Dell Vostro 1440 [2]
Processor Name	Intel Core 2 Duo	Intel Core i3
Processor Speed	2.13 GHz	2.40 GHz
Number Of Processors	1	1
Total Number Of Cores	2	4
RAM	2 GB	4 GB
Bus Speed	1.07 GHz	1.333 GHz
OS	Mac OS X 10.6.8	Linux Mint 15 “Olivia”
Ciao version	1.15.0	1.15.0

Table 5.1: Hardware and Software Specifications.

Table 5.2 presents the execution time values for each benchmark (in milliseconds) using as input lists of integer numbers of different lengths. We estimated the average time of 5 consecutive runs on the same input and repeated the experiment for lists of 1024, 2048, 4096, and 8192 elements.

List length		1024		2048		4096		8192	
Computer		[1]	[2]	[1]	[2]	[1]	[2]	[1]	[2]
First order	Pure	3	2	5	5	11	8	23	20
	Hand–coded	*3*	*2*	*6*	*8*	*14*	*14*	*35*	*35*
	Run–time check	170	136	439	350	962	719	2398	1805
Higher order	Pure	17	13	32	36	74	67	180	186
	Check 1	197	155	444	380	1226	854	2927	1902
	Check 2	198	162	444	374	1166	854	2751	1890

Table 5.2: Benchmark execution times in milliseconds.

The results show that the run–time overhead is added not only by the run–time checks, but also by expansions performed because of `meta_predicate` declaration. However, the impact of the latter is not significant, compared to the former one.

Table 5.3 presents the rations of execution times of the benchmarks which use run–time checks compared to the execution time of hand–coded type checks (which, as we have mentioned, we consider as the optimal performance case). Table 5.4 presents the execution time ratios of benchmarks 3 and 4 (which evaluate run–time added by PA–props check propagation) compared to benchmark 2 (first–order run–time checks), whose performance we see as an average practical case.

List length		1024		2048		4096		8192	
Computer		[1]	[2]	[1]	[2]	[1]	[2]	[1]	[2]
First order	Run–time check	55.34	56.67	68.56	43.80	66.87	49.94	69.21	55.02
Higher order	Check 1	64.41	64.67	69.35	47.50	85.26	59.33	84.49	57.98
	Check 2	64.62	67.67	69.35	46.80	81.07	59.28	79.41	57.61

Table 5.3: Benchmark execution time ratios (compared to hand–written type checks).

List length		1024		2048		4096		8192	
Computer		[1]	[2]	[1]	[2]	[1]	[2]	[1]	[2]
Higher order	Check 1	1.16	1.14	1.01	1.08	1.28	1.19	1.22	1.05
	Check 2	1.17	1.19	1.01	1.07	1.21	1.19	1.15	1.05

Table 5.4: Benchmark execution time ratios (compared to first–order run–time checks).

These ratios allow us to conclude that the overhead, introduced by our approach is not significantly larger than the the one already added by system run–time checks, and at the same time it allows to discover those errors at run–time that are usually left undiscovered.

Chapter 6

Conclusions and Future Work

In the present work we have introduced an extension of the Ciao assertion language for higher–order predicates. We identified that previously to this work, there was limited support for the specification of the behavior of higher–order predicates in the domain of (C)LP programming, especially about instantiations of their predicate arguments.

We have also outlined several possible solutions that allow us to perform run–time checking of these properties. They vary in trade–offs between expressiveness and computational effort required for their implementation, which we have discussed in Chapter 3 from a theoretical point of view. Along with the discussion of benefits and drawbacks of each we have narrowed our attention to several alternatives, most suitable for the implementation of a prototype checker.

We have developed a prototype implementation of the run–time checking of higher–order properties as a part of the Ciao system and its powerful preprocessor, capable of performing various types of source code analysis. The preliminary experimental results are encouraging and prove that the problem of checking higher–order predicates can be tackled – for small but realistic cases – with assertion–based run–time checks with reasonable overhead.

Still, a lot of work remains to be done before such checking can be performed in an elegant and reliable manner in larger, real-life code. We foresee several possible improvements as future work:

- A basic goal is to improve the performance for the checks for higher–order types, which implicitly requires also optimizing further the run–time checking machinery for first-order programs, since from our experiments this seems to be the main component of the overhead.

- We also plan to study additional cases for the semantics of PA–props assertion checking. For example, there exist other possible interpretations of the scope of checking for higher–order predicates, like checking properties just for the derivations under the predicate that introduce the assertion with PA–props. Checking in those semantics may have quite different overheads than those described in this work.

- It would desirable to have support not only for instantiation properties, but also for compatibility ones.
- Finally, the property propagation pass should be reformulated as an abstract interpretation-based analysis. When plugged into the CiaoPP analysis framework, this will give much better precision and performance (as many run–time checks can be detected and removed, e.g., by abstract program specialization). It will also make the transformation-based approach more powerful for modular applications (e.g., with inter-modular analysis).

Bibliography

[1] W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*, pages 573–581, 1988.

[2] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.

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

[4] J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden, May 1997. U. of Linköping Press.

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

[6] K. Apt, editor. *From Logic Programming to Prolog*. Prentice-Hall, Hemel Hempstead, Hertfordshire, England, 1997.

[7] Claude Laï. Assertions with Constraints for CLP Debugging. In Pierre Deransart, Manuel V. Hermenegildo, and Jan Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, volume 1870 of *Lecture Notes in Computer Science*, pages 109–120. Springer, 2000.

[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] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL'77)*. ACM Press, 1977.

[10] M. Hermenegildo and The CLIP Group. Programming with Global Analysis. In *Proceedings of ILPS'97*, pages 49–52, Cambridge, MA, October 1997. MIT Press. (abstract of invited talk).

[11] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

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

[13] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012. http://arxiv.org/abs/1102.5497.

[14] 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, 2005.

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

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

[17] 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.

[18] C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.

[19] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[20] J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Fourth International Symposium on Practical Aspects of Declarative Languages*, number 2257 in LNCS, pages 243–261. Springer-Verlag, January 2002.

[21] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992. Originally published as Technical Report FIM 59.1/IA/90, Computer Science Dept, Universidad Politecnica de Madrid, Spain, August 1990.

[22] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *International Conference on Logic Programming (ICLP 1991)*, pages 49–63. MIT Press, June 1991.

[23] 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.

[24] P. López-García, F. Bueno, and M. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 19–35. Springer-Verlag, August 2005.

[25] P. López-García, F. Bueno, and M. Hermenegildo. Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information. *New Generation Computing*, 28(2):117–206, 2010.

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

[27] F. Bueno, P. López-García, and M. Hermenegildo. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany, April 2004. Springer-Verlag.

[28] 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.

[29] 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.

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

[31] S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[32] P. López-García, L. Darmawan, and F. Bueno. A Framework for Verification and Debugging of Resource Usage Properties. In M. Hermenegildo and T. Schaub, editors, *Technical Communications of the 26th Int'l. Conference on Logic Programming (ICLP'10)*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 104–113, Dagstuhl, Germany, July 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

[34] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*. Springer, 2007.

[35] 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.

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

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

[38] E. Vetillard. *Utilisation de Declarations en Programmation Logique avec Contraintes*. PhD thesis, U. of Aix-Marseilles II, 1994.

[39] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[40] E. Shapiro. *Algorithmic Program Debugging*. ACM Distiguished Dissertation. MIT Press, 1982.

[41] P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, Cambridge MA, 1994.

[42] Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1–3):17–64, October 1996.

[43] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, September 1996.

[44] 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.

[45] P. Van Roy and A.M. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

[46] K. Marriott and P. Stuckey. The 3 R's of Optimizing Constraint Logic Programs: Refinement, Removal, and Reordering. In *19th. Annual ACM Conf. on Principles of Programming Languages*. ACM, 1992.

[47] A. Kelly, A. Macdonald, K. Marriott, P. Stuckey, and R. Yap. Effectiveness of optimizing compilation for CLP(R). In *Proceedings of Joint International Conference and Symposium on Logic Programming*, pages 37–51. MIT Press, 1996.

[48] D. Knuth. Literate programming. *Computer Journal*, 27:97–111, 1984.

[49] D. Cordes and M. Brown. The Literate Programming Paradigm. *IEEE Computer Magazine*, June 1991.

[50] F. Bueno, M. Carro, M. Hermenegildo, R. Haemmerlé, P. López-García, E. Mera, , J.F. Morales, and G. Puebla-(Eds.). The Ciao System. Ref. Manual (v1.14). Technical report, July 2011. Available at `http://ciao-lang.org`.

[51] G. Puebla, F. Bueno, and M. Hermenegildo. A Framework for Assertion-based Debugging in Constraint Logic Programming. In *Proceedings of the JICSLP'98 Workshop on Types for CLP*, pages 3–15, Manchester, UK, June 1998.

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

[53] L. Naish. A three-valued declarative debugging scheme. In *8th Workshop on Logic Programming Environments*, July 1997. ICLP Post-Conference Workshop.

[54] Gopalan Nadathur and Dale Miller. Higher–Order Logic Programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5. Oxford University Press, 1998.

[55] D.H.D. Warren. Higher-order extensions to prolog: are they needed? In J.E. Hayes, Donald Michie, and Y-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd., Chicester, England, 1982.

[56] Lee Naish. Higher-order Logic Programming. Technical Report 96/2, Department of Computer Science, University of Melbourne, Melbourne, Australia, feb 1996. URL: http://www.cs.mu.oz.au/~lee/papers/ho/.

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

[58] D. Cabeza and M. Hermenegildo. Higher-order Logic Programming in Ciao. Technical Report CLIP7/99.0, Facultad de Informática, UPM, September 1999.

[59] 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.

[60] Leon Sterling and Marc Kirschenbaum. Applying Techniques to Skeletons. In Jean-Marie Jacquet, editor, *ICLP Workshop on Construction of Logic Programs*, pages 127–140. Wiley, 1991.

[61] Leon Sterling. Patterns for Prolog Programming. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2407 of *Lecture Notes in Computer Science*, pages 374–401. Springer, 2002.

[62] Dave Barker-Plummer. Cliche Programming in PROLOG. Technical report, Duke University, Durham, NC, USA, 1989.

[63] Timothy S. Gegg-Harrison. Representing Logic Program Schemata in lambda-Prolog. In Leon Sterling, editor, *ICLP*, pages 467–481. MIT Press, 1995.

[64] Yves Deville. *Logic Programming: Systematic Program Development*. International Series in Logic Programming. Addison-Wesley, 1990.

[65] Rémy Haemmerlé and François Fages. Modules for Prolog Revisited. In Sandro Etalle and Miroslaw Truszczynski, editors, *ICLP*, volume 4079 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2006.

[66] Paulo Moura. Meta-predicate Semantics. In Germán Vidal, editor, *LOPSTR*, volume 7225 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2011.

[67] Alberto Pettorossi, editor. *Meta-Programming in Logic, 3rd International Workshop, META-92,Uppsala, Sweden, June 10-12, 1992, Proceedings*, volume 649 of *Lecture Notes in Computer Science*. Springer, 1992.

[68] Iliano Cervesato and Gianfranco Rossi. Logic Meta-Programming Facilities in 'LOG. In Pettorossi [67], pages 148–161.

[69] Patricia Hill and John Gallagher. Meta-programming in logic programming. Technical Report 94.22, School of Computer Studies, University of Leeds, 1994. To be published in *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 5*. Oxford Science Publications, Oxford University Press.

[70] C. Beierle, R. Kloos, and G. Meyer. A pragmatic type concept for Prolog supporting polymorphism, subtyping, and meta-programming. In *Proc. of the ICLP'99 Workshop on Verification of Logic Programs, Las Cruces*, Electronic Notes in Theoretical Computer Science, volume 30, issue 1. Elsevier, 2000.

[71] Werner Hett. Prolog Problems: 6. Graphs. https://sites.google.com/site/prologsite/prolog-problems/6. Accessed: 2013-05-23.

[72] J. F. Morales, M. V. Hermenegildo, and R. Haemmerlé. Modular Extensions for Modular (Logic) Languages. In *Proceeding of the 21th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'11)*, volume 7225 of *LNCS*, pages 139–154, Odense, Denmark, July 2011. Springer.

[73] 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.

[74] E. Mera, P. López-García, and M. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th International Conference on Logic Programming (ICLP'09)*, number 5649 in LNCS, pages 281–295. Springer-Verlag, July 2009.