Estudio y desarrollo de nuevas extensiones sintácticas funcionales para Ciao Prolog

Study and development of new functional syntactic extensions for Ciao Prolog

Trabajo de Fin de Máster
Curso 2023–2024

Autor

Víctor Luque Santamaría

Directores

Jose Francisco Morales Caballero

Manuel de Hermenegildo Salinas

Máster en Métodos Formales en Ingeniería Informática

Facultad de Informática

Universidad Complutense de Madrid

Estudio y desarrollo de nuevas extensiones sintácticas funcionales para Ciao Prolog

Study and development of new functional syntactic extensions for Ciao Prolog

Trabajo de Fin de Máster en Métodos Formales en Ingeniería Informática

Autor
Víctor Luque Santamaría

Directores
Jose Francisco Morales Caballero
Manuel de Hermenegildo Salinas

Convocatoria: *Junio 2024*

Máster en Métodos Formales en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

17 de Junio de 2024

Agradecimientos

A mis tutores del trabajo, por mantener siempre una actitud positiva y una ayuda activa durante todo el proceso. Por su atención, su búsqueda de disponibilidad y, sobre todo, su confianza hacia mis habilidades.

A todos mis seres queridos que me han acompañado en la realización de este trabajo, a mantener un buen estado de ánimo para lograrlo. Especialmente, a mi madre por preocuparse por el estado del proyecto y de mi salud durante su realización; a mi padre por ayudarme a expresar las ideas del trabajo para mantener una conversación y por mostrar su interés aún sin saber de los detalles; y a Lucía, por estar siempre a mi lado antes, durante y después de mis esfuerzos por llevar a cabo el proyecto.

Resumen

Estudio y desarrollo de nuevas extensiones sintácticas funcionales para Ciao Prolog

En el área de lenguajes de programación, la idea de combinar paradigmas es de interés, tanto para mejorar la comodidad y productividad del programador, como para mejorar la expresividad y aumentar las funcionalidades del lenguaje. Una de las características del paradigma de la programación funcional que es atractivo trasladar a otros paradigmas es el excelente soporte del orden superior y la brevedad sintáctica. En este trabajo estudiamos estos temas en el contexto de Prolog. Existen múltiples propuestas para obtener orden superior y otras características de la programación funcional junto con la programación lógica. Muchas de ellas consisten en incorporar parte de la semántica de los lenguajes funcionales, como el cálculo lambda simplemente tipado, o añadir fragmentos de lógica de orden superior, por ejemplo definiendo la unificación de expresiones lambda. En este trabajo nos centramos sin embargo en la solución propuesta por el sistema Ciao Prolog, que incluye la posibilidad de programar en estilo funcional y con orden superior, pero sin cambiar la semántica operacional de su base en Prolog. Esta solución está basada en transformaciones sintácticas, gracias a las capacidades de extensión del lenguaje Ciao y su sistema de módulos. Presentamos nuevas extensiones que añaden funcionalidad y mejoran el soporte en Ciao de la notación funcional y del orden superior, siguiendo en la línea de no cambiar la semántica operacional de Prolog. La idea principal parte del trabajo realizado en 2004 sobre la propuesta *Hiord*, que se ha mejorado a través de un análisis de soluciones similares presentes en otros lenguajes de programación.

En este trabajo se presenta la solución actual de Ciao junto con las nuevas extensiones para notación funcional, las soluciones ofrecidas por otros lenguajes con funcionalidades similares, y una discusión de su implementación.

Palabras clave

Programación lógica, programación funcional, programación multiparadigma, orden superior, extensiones sintácticas, programación lógico-funcional, programación modular.

Abstract

Study and development of new functional syntactic extensions for Ciao Prolog

In programming languages, the idea of mixing programming paradigms is of interest, both for improving programmer productivity and convenience, and for enhancing language expressiveness and functionality. Some characteristics of the functional programming paradigm that it is attractive to carry over to other paradigms is the excellent support for higher order and the syntactic compactness. In this work we study these issues in the context of Prolog. There are multiple solutions for achieving higher order and other functional programming characteristics along with logic programming. Many of these change the semantics to include part of the semantics of functional languages, such as the simply typed lambda calculus, or to add fragments of higher-order logics, such as for example lambda term unification. In this work we concentrate instead on the Ciao Prolog system, which includes the possibility of programming in a functional style and using higher order, but without changing the underlying Prolog operational semantics. This approach is based on syntactic transformations, thanks to the language extension capabilities and module system of the Ciao language. We present new extensions that add functionality and improve the support in Ciao for functional notation and higher order, while continuing in the line of not changing the underlying Prolog semantics. The main idea is based on work done in 2004 the *Hiord* proposal, which has been revisited in this work through an analysis of similar solutions in other programming languages.

This work presents the current Ciao solution along with new extensions for functional notation, explores other languages with similar functionalities, and discusses their implementation.

Keywords

Logic programming, functional programming, multiparadigm programming, higher order, syntactic extensions, functional-logic programming, modular programming.

Contents

List of figures

List of tables

.

Chapter 1

Introduction

"Coding is today's language of creativity. All our children deserve a chance to become creators rather than consumers of computer science."
— Maria Klawe

1.1 Motivation

Adhering to what is suggested in Maria Klawe's quote, this work is dedicated to remarking the relevance of having a good programming language that is enriched with the latest modern features to ease and enhance code elegance, readability and creativity to write down problems and applications. Having a good programming language has to be something that makes the task of coding not tedious to carry out. Principally, the main purpose is to have readability, maintainability and expressive power in order to ease the process of writing down an idea to express it to a computer and to improve possible teamwork and communication between coders.

Logic programming had as one of its main motivations to ease human reasoning about problems while expressing a problem and its solution. In general, since the first programming languages were created, many others have been created to enrich the existing ones focusing on other philosophies and purposes. Ciao Prolog has the philosophy to encompass as much as possible while introducing the least noise.

If one takes a look at the most popular and widely-used general-purpose programming languages, it becomes clear that they remain relevant by doing what this project is about. Having an adequate understanding and a broad vision of modern languages, these ideas can start growing a language in the directions of improving the expressiveness and, sometimes, the functionality.

To design a new feature in a programming language, it is reasonable to have extensive programming experience and to try to understand other solutions and the latest language updates. Mainly, understanding why several languages follow the same evolution can help to know if your language needs new features or not. It is especially important to ensure that any improvements made are concise and don't introduce future problems that could hinder the later evolution of the language.

1.2 Objectives

The main goal of this work is to increase the expressivity of the Ciao Prolog logic programming language. Ciao Prolog provides a modular and extensible design to enhance its expressive power incrementally without having to change the existing implementation. Its design also provides features for the combination of functionalities. The expressive power and the features of a Turing-complete programming language are enough to write any program. Nevertheless, a language design can be enhanced by improving its syntax or semantics in order to reduce coding effort, program readability or maintainability. Thus, the objectives in this work are focused on research for human understanding and language expressiveness, while preserving the existing features and efficiency. Moreover, any extension must consider potential ambiguity and interaction with both the existing features. Taking this into account, the objectives set out for this work are:

1. Study pure functional languages to identify interesing features or syntactic constructs, as well as examples to translate into Ciao Prolog (where by features we mean constructs such as, e.g., *let rec*, *where*, auxiliary functions, closures, etc.).

2. Translate with the current syntax into Ciao in order to detect possible new (missing) features worth being included as a language extension.

3. Develop a formal description (as deduction rules) of the semantics or translation rules (scoping, meaning/design) for the new features.

4. Contribute an extended design of the *fsyntax* package, which is used currently to support functional syntax and other related features in Ciao Prolog, taking into account backwards compatibility with the existing solution.

1.3 Work Plan

This section presents the steps to accomplish the former objectives. First of all, the goals described in this work require familiarity with both Prolog and Ciao Prolog. Ciao Prolog is a modern Prolog dialect that builds up from a logic-based simple kernel and it is designed to be portable, extensible, and modular. Its package system, that enables modular implementation of syntactic and semantic language extensions, greatly facilitates the implementation of multiparadigm programming.

Besides knowledge of Ciao Prolog, this work requires studing other modern general-purpose programming languages that support functional programming.

As a next step in the work plan, we will collect representative code snippets for those languages (e.g., from benchmarks, language tutorials, etc.) and try to translate them to Ciao, as idiomatically as possible. Of course, since Prolog is already Turing complete, a working translation is always possible in Ciao, however the purpose is to preserve as much as possible to the original code style using the multiparadigm extensions. This step is necessary to spot important syntactic or semantic differences between languages.

The search for code examples has been focused on popular coding challenge projects, such as the *99 Prolog Problems*, that was initially proposed for Prolog and ported (idiomatically) to several functional programming languages, or the *Advent of Code* (Wastl, 2023).

Once the translation is complete, the next step is writing down the enhancements proposals for the functional syntax package. This design proposal embraces both an aesthetic vision and a commitment to elegance. To get this done, we have to take into account many cases not only in the languages where the feature was found, but also in the possible translation into Ciao Prolog. Note that there are many features that can be implemented but not all of them will have a positive impact on the language.

Once the features to extend the language are decided, we will focus on the study of the existing functional notation package, in order to implement the new extensions.

1.4 Structure of the work

The structure of this thesis is the following:

1. Introduction.

2. Preliminaries, which include a review of logic programming, Prolog, Ciao Prolog, and Ciao's *fsyntax* package.

3. The main body of the work, discussing the different extensions considered and the comparisons with other languages.

4. A brief overview of other related work.

5. Suggestions for future work, including a discussion of some related extensions under development.

1.5 Contributors

The author of this thesis is not responsible for the implementation of the extensions presented in this work, but rather a key contributor to their conceptual design. The main implementation of the code was carried out by one of the directors, José F. Morales. The author's contribution has focused on conducting extensive research across various languages.

This research provided a well-rounded, in-depth perspective that was crucial in shaping the syntax and expressivity of the resulting extensions. Additionally, the other director of the project, Manuel Hermenegildo, played a significant role in the design process thanks to his deep expertise in the object language Ciao Prolog and all of the topics related.

1.6 Associated code

The associated code (Google Drive) of the current implementation of the extensions presented in this work is part of a huge project (Ciao Prolog) and is still under development. But it has been extracted and uploaded into the folder of the link in the bibliography. Of course, future changes could be found when the final version is released and this repository will remain unchanged to leave it as it was at the point this work was realized.

Chapter 2

Preliminaries

To begin with the topic of this project, it is important to introduce some notions of logic programming. Logic programming has been studied for a long time and there exists several general-purpose (and Turing-complete) logic programming languages, like Prolog. To understand the foundations of Prolog, we must talk about a subset of first order logic formulas and the corresponding theorem proving method that makes computations feasible: Herbrandization and SLD-resolution (Selective Linear Definite resolution) with Horn clauses (Kowalski and Kuehner, 1971; Apt and van Emden, 1982). Of course, there exist other approaches for logic programming and many extensions such as SLDNF (SLD with Negation as Failure (Clark, 1978)), TLP (Tabled Logic Programming (Tamaki and Sato, 1986; Swift and Warren, 2012)), CLP (Constraint Logic Programming (Jaffar and Lassez, 1987; Van Hentenryck, 1991; Marriot and Stuckey, 1998)), but they are out of the scope of this project.

2.1 Introduction to logic programming

Logic serves as a cornerstone for reasoning and provides a formal and precise language for expressing goals, knowledge, and assumptions. The purpose of creating a paradigm based on it lies in the aim of providing a language that is nearer to the human thinking process rather than to the underlying machinery of a computer. Logic programming, as well as functional programming, comes from abstract models based on logic approaches to supply declarative ways to define a program.

Rather than solving a problem by relying on the operational translation of a solution, logic programming allows expressing the knowledge about the problem and – ideally – formalizing its solution without necessarily concerning oneself with the specifics of the underlying machine model. This provides a deeper insight on the concepts that define a problem and its solution, so that the programmer can think of the problem assumptions and let the computer solve them to achieve the goals given.

Name	**Syntax (infix)**	**Meaning**
Conjunction	$A \wedge B$	True iff both terms are true
Disjunction	$A \vee B$	True iff at least one of the terms are true
Negation	$\neg A$	True iff the term is false and viceversa
Implication	$A \Rightarrow B$	True iff either A is false or if both are true

Table 2.1: Connectives

The execution of the program consists of an attempt to prove a goal statement based on the formalism that defines the problem given a set of assumptions for it.

2.1.1 From logic to Prolog

As mentioned before, logic programming is based on logic, and, more specifically, Prolog is based on a computable part of it. To continue with the explanation, we must start with propositional and subsequently first-order logic.

Propositional logic. In propositional logic we define **proposition**s as statements that can be said to be either true or false. This is very important to understand many aspects when enabling multiparadigm extensions on Ciao Prolog because other paradigms have different foundations and operational semantics (e.g., functional is based on reducing its functions to their corresponding values).

A **formula** in propositional logic is defined inductively from atomic formulas or **propositions**, and **connectives** which, as the name suggests, connect formulas.

An intepretation in propositional logic assigns *true* or *false* values to each formula. Table 2.1 shows the connectives and their meaning.

At this point we can already explain relevant concepts that play a significant role in logic. Propositional logic formulas can be classified as:

- *Valid* formula, when it is true for any value assigned to the propositions. For example: $p \Rightarrow p$.
- *Unsatisfiable* formula, when it is false for any value assigned to their propositions. For example: $p \wedge \neg p$.
- *Satisfiable* formula, when there is a value assignment that makes the formula true. For example: $p \vee \neg p$.

Another important concept is logical consequence. A formula Q is a logical consequence of formula P, or in other words P logically entails Q (denoted $P \models Q$), if Q is true for all the assignments that make P true. For example:

$$p \wedge q \models p$$

Propositional formulas allow arbitrary nesting of connectives but Prolog execution is based on the resolution algorithm and it requires formulas in a precise form called Conjunctive Normal Form (CNF). Any propositional formula can be transformed into this form preserving its satisfiability. A *clause* is a formula where if there is more than one proposition, they are connected only by the *and* connective. Propositions within this conjunction can be negated or non-negated. All possible formulas can be converted into a set of clauses that are connected by the *or* connective, that is, an "or of ands", a canonical form that is called conjunctive normal form. An example could be:

$$a \vee (b \wedge c) \vee (\neg a \wedge d)$$

First-order logic. Let us now move to first-order logic (FOL), also referred to as predicate logic. The difference between propositional and first-order logic is essentially that propositions are generalized to *predicates*, which may have arguments that contain *terms*, and the introduction of logical variables as part of such terms. In FOL there are actually three kinds of terms: *constants*, *variables* and *compound terms*. Predicates define the relations between these terms that are true within a domain (or program). The meaning of these predicates can be defined using the same connectives as in propositional logic, but we extend the logic with another concept due to the existence of variables. This new concept around variables in a sentence is called quantifier because such quantifiers define the extent of the meaning of the variables within predicates. There are two types: the **universal** quantifier ($\forall$) that means that the variable can be instantiated to any possible value and it will make the predicate true, and the **existential** quantifier ($\exists$) that means that there is at least one instantiation of the variable to a possible value that makes the predicate to be true. Terms can be classified into two types: ground and non-ground. The ground ones are those that are completely specified, meaning that all the atoms are ground and all the compound terms with ground terms as arguments are ground, recursively. Variables and terms where variables occur are called non-ground terms. The Herbrand universe of a program in Prolog is the set of all ground terms that can be represented with the symbols that appear in that program (Lloyd, 1987; Apt, 1990). In predicate logic the elements of clauses (that are thus connected by $\wedge$ and may be negated or not) are referred to as *literals* or *atomic formulas* (van Emden and Kowalski, 1976) and take the form $P(t_1, \cdots, t_n)$ where P is a predicate symbol and the t_i are terms. Non-negated atomic formulas are *positive* literals and negated atomic formulas are *negative* literals.

Logic programs and Prolog. A logic program is composed of a set of clauses, defining one or more predicates. In general in logic programming, and in Prolog in particular, a restricted form of clauses is used called Horn clauses. Horn clauses are those that are formed as a disjunction of literals where *all these literal are negative*

	Disjunction form	**Implication form**	**Prolog translation**
Definite Clause	$\neg p \vee \neg q \vee ... \vee \neg t \vee u$	$u \leftarrow p \wedge q \wedge ... \wedge t$	u :- p, q, ... , t.
Fact	u	$u \leftarrow true$	u.
Goal Clause	$\neg p \vee \neg q \vee ... \vee \neg t$	$false \leftarrow p \wedge q \wedge ... \wedge t$	:- p, q, ... , t.

Table 2.2: Horn Clauses (assuming p, q, etc. are atomic formulas)

except one, called the head or objective. This translates into an implication that can also be seen as expressing that "to accomplish the truth of the objective literal, the conjunction of all the rest ones must be true." Clauses can be either facts (clauses with a single, non-negated literal) or rules (Horn clauses with more than one literal). Facts are the simplest way to state a relation. Each fact simply relates a set of objects, represented by the terms involved. I.e., it establishes a relation between certain terms. Rules also define a relation between terms but by referring to other relations. *Definite clauses* are those that have exactly one non-negated literal. Facts and rules are thus all definite clauses. A *query* or goal statement is a clause that only contains one *negated* literal and can be seen as a question looking for the truth or falsity of the existence of a non-query statement in a specific program. That is, if there exists a predicate or set of predicates that makes the question true the query succeeds and if not it fails. Table 2.2 illustrates the three types of clauses in Prolog.

Such predicates defined by facts and rules can also be seen as procedure declarations, as we will see in the next section. Then, the empty clause, $\Box$, can be seen as a halt statement, and, finally, the goal clause is what we have called a query, where all the literals have to be accomplished to make the sentence true.

Sentences in first-order logic can be turned into Horn clauses by the process described in (Nilsson, 1971): simplify the formula to contain only $\forall$, $\exists$, $\wedge$ $\vee$, $\neg$, connectives (through implication identity: $\neg P \vee Q$ is equivalent to $P \Rightarrow Q$), then move in negations to be applying at literal granularity (with deMorgan's laws and negation of quantifiers). Then comes the *Skolemnization* phase, that consists in removing the existentially quantified variables, and finally the removal of universal quantifiers, after which we have a formula that can be converted easily to CNF (Clause Normal Form) and that can be converted subsequently to a Horn clause by the implication identity. Note that the last conversion is not always possible due to the possible negation of literals, since a Horn clause can only have one literal negated (the objective or head of the clause). In those cases, it is possible to redefine such negated literals to be their contrary, use other resolution mechanisms that include negation as failure (SLDNF), etc.

When the results are Horn clauses, as in 2.2, a Prolog program can be easily obtained (Apt and van Emden, 1982).

Prolog syntax. Although Prolog programs correspond to Horn clauses, some conventions are necessary for typing those formulas easily as code. Constants in Prolog are numbers and *atoms*, understood as atomic terms.[1] Atoms in Prolog are represented as identifiers starting with a lowercase letter. Compound terms are the ones constructed by a function symbol and its arguments. The arguments are other terms and the number of arguments of the function symbol is called its arity. The function symbol is represented by a name starting with a lowercase letter and it is followed by parentheses embracing its arguments. Variables, written starting with an uppercase letter, represent unspecified terms. Indeed, they can have any value within the Herbrand universe of the program in which they appear.

Prolog execution. Prolog can be seen as the procedural interpretation of predicate logic as a programming language (van Emden and Kowalski, 1976). A rule can be seen as a *procedure declaration*, its head is the *procedure name* and its body the *procedure body*, where each literal is a *procedure call*. Computation is initiated by a goal statement (i.e., a query), which is the *initial procedure call*, and the process of subsequent *procedure invocations* does the work of finding a refutation for the goal statement. The concrete process used to obtain this refutation (i.e., to derive the empty clause $\Box$) is SLD-resolution (Kowalski and Kuehner, 1971; Apt and van Emden, 1982), a specialized version of the resolution algorithm (Robinson, 1965). The *resolvent* used by the SLD-resolution corresponds to the stack of the procedure calls. *Unification* is used as the parameter passing and pattern matching. Finally, and as a special case for logic programming, some mechanism is used to explore different possible execution paths, which allows dealing with unification failure and possibly finding several solutions for the query.

As a result of this procedural interpretation, we have a practical programming language. In fact, logic programming has been successfully used in many applications and it is an ideal language for many purposes (see, e.g., (Warren et al., 2023) and the links provided there). To this end, while keeping the deep roots in logic and automated theorem proving, some assumptions and simplifications are usually made to make Prolog a practical programming language. For example, Prolog systems usually stick to depth-first search. This means that clauses are visited in the same order in which they appear in the program, and literals are called in sequence, in the order in which they appear in the clause, as in a traditional programming language.

Example 1 *Let us see a short Prolog example which we will also use to illustrate the operational semantics of Prolog. Consider the following database of facts representing the father and mother relationships between individuals in a family, and rules for the new derived grandparent relation among these individuals:*

[1]Not to be confused with atomic *formulas*.

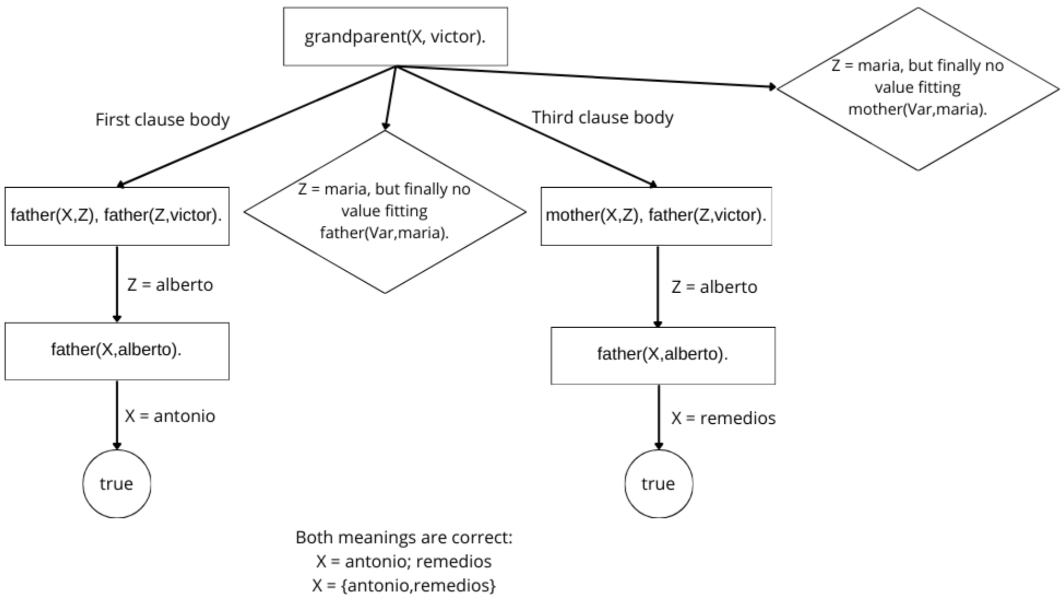
grandparent(X, victor).
First clause body
Third clause body
Z = maria, but finally no value fitting mother(Var,maria).
father(X,Z), father(Z,victor).
Z = maria, but finally no value fitting father(Var,maria).
mother(X,Z), father(Z,victor).
Z = alberto
father(X,alberto).
X = antonio
true
Z = alberto
father(X,alberto).
X = remedios
true
Both meanings are correct:
X = antonio; remedios
X = {antonio,remedios}

Figure 2.1: Search tree representation for query resolution

```
father(alberto,victor).
father(alberto,alvaro).
mother(maria,victor).
mother(maria,alvaro).
father(antonio,alberto).
mother(remedios,alberto).

grandparent(X,Y) :- father(Z,Y), father(X,Z).
grandparent(X,Y) :- mother(Z,Y), father(X,Z).
grandparent(X,Y) :- father(Z,Y), mother(X,Z).
grandparent(X,Y) :- mother(Z,Y), mother(X,Z).
```

Figure 2.1, shows the execution tree for an initial query (call) `grandparent(X, victor)`*. Each of the branches of the tree represents an execution path (a sequence of calls) starting from the query and finishing in either "true" (the empty clause □), in which case that execution path was successful, or in failure, in which case that execution path was not successful. The values of variables at the end of a successful execution are the* answers *or output to the query. As we mentioned before and we can see in the figure, a program can have several solutions. Another interesting aspect is that we could also use as initial call for example* `grandparent(antonio, Y)` *and we would get answers for* `Y`*. I.e., procedure arguments are not fixed as inputs or outputs and the flow of the computation can go in either direction. This can result in a completely tree with a different set of executions.*

The previous example illustrates the fact that, in contrast to other paradigms, variables in Prolog may be bound to some data (a fully or partially instantiated term) or be unbound. The translation into the underlying predicate (first-order) logic semantics of Prolog is that they fit all the possible interpretations that a sentence has, i.e., instantiated values for the variables (if any) that make the sentence true.

Also note that there are other strategies which can be used to explore the search space instead of the standard depth-first search. Most Prologs use this strategy and Ciao Prolog has this strategy as default, although it also provides other strategies such as breath-first, iterative deepening, determinate literals-first scheduling, etc. Also, most Prologs have "delay" predicates that allow dynamic reordering of execution.

To make Prolog a practical programming language there are system-provided predicates (referred to as "built-ins") that are integral to the logical functionality of the language (Sterling and Shapiro (1987); PROLOG). These predicates are called to perform in the underlying machinery common useful tasks such as for example evaluating arithmetic, supporting extra-logical behaviour like I/O operations, treating errors (not logical failure but rather system undesired behaviour and user exceptions), implementing meta-logical behaviors like treating variables as objects of the language, or controlling the behaviour of the unification and resolution algorithms, etc.

Also, a mechanism exists in order to avoid undesired computations for a predicate by controlling the instantiation of its arguments: *modes*. Modes describe how the arguments to predicates are intended to be used: as inputs, outputs, or either. In this sense, when using a predicate, the user can specify through modes the expected purposes for the arguments of a predicate and avoid possible errors (such as, e.g., non-termination or non-specified computations) if the predicate is used with other modes.

2.1.2 Meta-programming and higher-order

We now turn to the issue of higher-order programming, i.e., where procedures are allowed to have other procedures as arguments, to return procedures as results, or both. The extension of pure logic programming as presented so far to higher-order programming is not straightforward, since, in first order logic, logic variables can be bound only to terms, and not to literals. Fully supporting higher-order means that variables can be bound to predicates and that goals can be formed using variables instead of predicate symbols. Thus, the underlying logic of the programming language should be based on second-order logic, but this has the known problem of being undecidable. Despite this, several practical approaches have been proposed that allow logic variables to be bound to predicates, ranging from the meta-programming facilities present in most Prologs to more involved ones that extend the semantics by increasing the capabilities of the resolution algorithm. Others mix logic programming semantics with that or other languages that support higher-order.

2.1.2.1 Meta-programming in Prolog

An important Prolog feature is meta-programming, which has served traditionally as basis for a practical form of higher-order programming. Traditionally Prolog does not make any differentiation between functor symbols and predicate symbols and, thus, terms and goals are indistinguishable. Meta-programming allows calling terms as if they were goals. To this end, all Prolog systems implement the `call/1` built-in that allows calling terms as if they were literals and most also include the family of `call/n` predicates that allow applying a predicate symbol, represented by an atom or a term with some arguments, to an additional set of arguments and calling the resulting term.

The presence of a traditional (i.e., "predicate based") Prolog module system, where predicates can belong to separate name spaces but functors are global makes all this somewhat more involved because it requires keeping a mapping between functor symbols and predicate symbols for a particular module. Ciao Prolog has its own primitives to carry out this process, called meta-expansion of a term (the compiled version) that represents a predicate in a certain syntax and the meta-call (the correct call to the compiled code) of expanded terms.

Another problem to be addressed is unification, an in particular the case or unifying two terms that are meta (i.e., callable). There exist several alternatives that can be considered, with different levels of practicality, e.g., does unifying such meta-terms mean that they have to be the same *object* (in addition to unifying *shared* variables as will be explained later), that their definition must be syntactically identical, or that the two predicates are the same if they have the same semantics (e.g., give the same solutions)? Prolog systems (and in particular Ciao Prolog) take the first view and only ensure that the terms unify as in the first case. We will return to this issue in section 3.1.1. [2]

An important consideration is that the `call` primitives mentioned above work only in one direction: the only cases supported are those where the variable containing the name of the predicate to be called is instantiated. The predicate name cannot be a variable (an unknown value) when the meta-call is made. This is done in order to avoid the previously mentioned undecidability of higher-order logic, and preserve the semi-decidability of the first-order logic semantics. Note that this is essentially the same restriction as in functional programming, where the function to be called has to be known at run time when a function is called.

[2]There are other semantics and implementation such as *residuation* (delaying meta-calls until they are sufficiently instantiated), using *attributed variables*, and treating higher-order terms as constraints.

2.1.2.2 Hilog

Hilog (Chen et al., 1993) is an extension of Prolog that presents a more flexible syntax without changing the underlying predicate logic semantics. Most of the higher-order support of these Prolog extensions is also based on the *call* predicate. The purpose of the original syntax of Hilog is to eliminate the distinction of nonlogical parameters; thus, constants, functions (functors in Prolog) and predicates have the same meaning. All these parameters, in principle, are arityless, and what makes them to be one object or another is to give them a finite number of arguments. Also, Hilog presents an elegant syntax to manage the sets of solutions with parameterized predicates:

```
empl_earning_more_than_boss(Boss)(Empl) ← supervises(Boss,Empl),
    salary(Boss, B_Sal), salary(Empl,E_Sal),
    E_Sal > B_Sal.
```

In the previous code there is the set of the employees that satisfy the formula for each boss represented by `empl_earning_more_than_boss(Boss)`. Later, the possibility of indistinctly having arguments as any possible nonlogical parameters allows the user to code second-order predicates and/or add parameters like the following ones:

```
closure(R)(X,Y) ← R(X, Y).
closure(R)(X,Y) ← R(X, Z), closure(R)(Z,Y).
```

```
decomposition(X(Y,Z)) ← decomposition(Y), decomposition(Z).
```

The first declares the transitive closure of a given relation, thus creating a second-order predicate that parameterizes other relations/predicates. The second one parameterizes the variable X being a function (functor in Prolog) of arity 2, and this predicate will be independent of the function name to traverse all the domain functions of arity 2 in the program database.

In order to define the unification of higher-order terms, Hilog relies on intensionality over extensionality. Intensionality refers to the ability to treat predicates as first-class entities, which allows them to be passed as arguments, returned as results, and manipulated within the language. This contrasts with extensional approaches, where predicates are defined by their input-output behaviour and are typically not treated as data objects themselves.

2.1.2.3 λProlog

Another relevant extension of Prolog is λProlog (Nadathur and Miller, 1988). Historically, λProlog originates from theorem provers, where solving second-order logic or higher-order logic problems becomes interesting. This extension combines the underlying Horn clauses basis of Prolog with higher-order unification and the theoretical framework from lambda calculus (Barendregt, 1981; Selinger, 2008).

In contrast to Hilog, terms in λProlog include λ-terms and those must be treated during unification. Unification of λ-terms is based on a specialized built-in depth-first search algorithm. Note that higher-order unification is undecidable, and thus this algorithm is incomplete, although in practice good enough for most of λProlog's applications. There are many other interesting theoretical issues over the different forms in which λProlog extends the underlying logic of plain Prolog. In any case, perhaps because of the complex nature and undecidability issues involved in the higher-order unification approach used in λProlog, in the approaches we propose, we take the more conventional solution of unifying only identical terms and requiring meta-calls to be sufficiently instantiated.

2.1.2.4 Hiord

Hiord (Cabeza et al., 2004), which is the starting point of this work, proposed an approach for the introduction of higher-order programming with the objectives of compatibility with existing Prolog semantics, ease of translation to WAM-compilable code, and amenabilty to static analyses and optimizations of the code through program transformations. It proposes the notion of **Predicate Abstractions**, that are the structures that allow writing anonymous predicates as new terms, considered in the syntax and semantics of the system, in a concise way. Moreover, the fact that it is designed to be compatible with existing Prolog systems and its implementation as a Ciao Prolog package loadable within a modular extension of Prolog, makes it even easier to incorporate higher-order programming into logic programming systems.

The Hiord formalization explains what the set of variables are, and specifically how they are interpreted and mapped (substitution) in an abstraction. The paper has its own extended abstract syntax and semantics using partial combinatory algebras in order to present a model theory w.r.t. them. What is relevant for the current work is that, later, in the hiord-1 restriction, which is from where the implementation of our work started, also describes the syntax and semantics presented inside the object language of Ciao Prolog. The paper shows the new higher-order terms, including the closures with the predicate abstractions, including how to treat the shared variables and existentially quantified variables. Informally, given the set of variables inherited from the outer scope, the free variables are only the ones that do not appear in the former context. In fact, a similarity with lambda calculus abstraction can be observed (Weslley College). More concretely, in the syntax used, the existentially quantified variables inside the "{}" are the ones that are not explicitly shared with the parent context:

```
  { SharedVariables -> ''(AbtractionVariables) :- G}
```

where *SharedVariables* is a comma separated list of variables that are *shared* from the parent context, and *AbtractionVariables* are comma separated head variables of the abstraction working as arguments like in a normal predicate definition. The rest of the variables are existentially quantified variables, and, thus, local to the abstraction.

Inside the {} the user can write any predicate that is allowed in the module's context. The only difference is that the predicate is anonymous (in many languages anonymous predicates/methods/functions are the simile of a lambda abstraction) with ''. That means that the predicate must be compilable (mainly, all of its clauses must have same number of arguments).

2.2 Ciao Prolog modules and language extension mechanisms

As previously mentioned, this project is centered around Ciao Prolog, a specific extended version of Prolog which, among other features, supports multiparadigm programming in a syntactic way, thanks to its special language extension capabilities. In this section we review these capabilities and how they have been used to support functional syntax and other aspects of functional programming.

2.2.1 The Ciao Prolog module system

Ciao Prolog has a specific module system (Cabeza and Hermenegildo, 1999), which is designed to be compatible with the traditional de-facto standard for modules in Prolog but introduces new mechanisms for defining module-local language extensions, and also adds some restrictions that make global analysis more meaningful, among other features. The mechanisms for defining module-local language extensions are particularly relevant for our study because they allow a structured organization of the language extensions, separate compilation, and techniques to manipulate the order and the relations between parts of code.

The Ciao compilation process has main two steps. First, the module is loaded together with the interface (i.e., names of exported predicates) of all the imported predicates. Then, predicate names can be resolved (disambiguated) before the rest of the compilation happens. Focusing on higher-order support for the language, the module system allows handling errors statically when calling undefined predicates, or using incompatible arities, etc.

The Ciao module system is predicate-based, which means that, by default, only predicate symbols belong to different name spaces for each module, while functor symbols are global.[3] When dealing with meta-programming it is necessary to map

[3]There is a hiding mechanism to make atoms and function symbols module-local but we deal here with the (more frequent) case where global functor symbols and local predicate symbols are

functor symbols to predicate symbols in the given context of a module (i.e., using the module import relations and the locally defined predicates). Meta-predicate annotations allow predicates that deal with meta-data (data representing code, e.g., a term representing a predicate) specify which predicate arguments require special treatment. Those annotations contain which kind of meta-type is expected for each argument (only those relevant to our research are shown):

- **goal**, with or without argument instantiation.
- **predicate**, with the functor *pred(N)* stating that N is the number of arguments.

The annotations allow the Ciao compiler, not only to perform predicate name resolution but also to apply the necessary syntactic and semantic extensions.

2.2.2 Syntactic and semantic extensions

Ciao Prolog provides directives to declare code expansions at different targets:

- **sentence** translation: that is called to replace every sentence (directives, facts, clauses, ...) read by the compiler.
- **term** translation: to obtain the expansions of every term and sub-term
- **goal** translation: specially when dealing with meta-predicates it is essential to be able to differentiate between goals and sub-goals (e.g., when using higher-order)

Those expansion directives accept a user-provided predicate that defines the actual translation. In addition to the input to expand and the expanded result, there is an optional third argument to denote the module to which the code to be expanded belongs.

2.3 Functional notation: *fsyntax* package

The `fsyntax` package allows writing Prolog code with functional notation. Functional notation allows writing clauses in functional style, where a predicate head $p(x_1, \ldots, x_n)$ is written as $p(x_1, \ldots, x_{n-1}) := x_n$ to denote a dedicated argument for the result and for using that predicate in term positions, where the translation will expand them to generate the actual call. The following is the append predicate written using this notation:

used and the mapping is needed.

FunClause	::=	*FunName(Args)* := *term* :- *body*	(Body is optional)
Args	::=	$Term_1, \ldots, Term_n$	Arguments
Body	::=	$Goal_1, \ldots, Goal_n$	Body of a clause
Term	::=	*Functor(Args)*	Compound terms
		Const	Constants
		Var	Variables
		Body ? *Term*	Conditional term
		Term “|” *Term*	Disjunction of terms
		˜*Term*	Function application

Figure 2.2: Extended syntax for the functional notation

```
append([],L) := L.
append([X|Xs],L) := [X|~append(Xs,L)].
```

When we define a predicate in this functional style, it is expanded into a predicate with one more argument:

```
append([],L,L).
append([X|Xs],L,[X|_1]) :- append(Xs,L,_1).
```

Note that output of the expansion is still a valid Prolog program, that can be used as usual (e.g., running backwards):

```
?- [1,2,3,4,5,6] = ~append([1,2,3],X).

X = [4,5,6]
```

As mentioned before, the expansion adds one last variable to the clause head which is the result to unify. Thus, it transforms what is put after the `:=` and before the `:-` (if it is placed), to be the result variable. In Figure 2.2 we present a simplified grammar that defines the extended syntax added with this package.

In the previous example we can observe another feature of the package, the tilde ˜, that is related to function application. If we write `append(Xs,L)` inside a literal, e.g., `p(append(Xs,L))`, without the tilde, then `append(Xs,L)` will be a plain compound term with arity 2. However, if we use the tilde then `p(˜append(Xs,L))` is expanded to `append(Xs,L,V), p(V)`.

We can avoid the use of the tilde for particular predicates using the directive *:- fun_eval predicate_name/arity*. Also, we can avoid the use of the tilde for all predicates defined using functional syntax within the module, using the directive *:- fun_eval defined(true)*:

```
% :- fun_eval defined(true).
% With the above directive, there is no need for the next one, and
% neither for any other one in the module.

:- fun_eval append/2. % the arity is 2, the number of arguments that
                      % are desired to be evaluated as a function
                      % application
append([],L) := L.
append([X|Xs],L) := [X|append(Xs,L)].
```

Another interesting application of the `fsyntax` package is arithmetic evaluation. Arithmetic operations in Prolog are specified with the *is/2* predicate, which evaluates an arithmetic expression (a compound term with arithmetic functors) into a numeric value. The same mechanism that expands function applications in term positions can be used to evaluate arithmetic operations appearing in arbitraty term positions, without directly using *is/2*. This transformation can be activated with the flag *:- fun_ eval arith(true)*, which is not activated by default because, usually in Prolog, users use arithmetic functors for compound terms (like the convention of having tuples or pairs with `//2` or `-/2` that correspond to the integer division and substraction functors respectively). Let us illustrate this with the following examples:

```
func_not_evaluating := 1+2.
```

The previous code will be transformed into:

```
func_not_evaluating(1+2).
```

Now, if we try it with the arithmetic evaluation flag activated:

```
:- fun_eval(arith(true)).

func_evaluating := 1+2.
```

This will be the resulting code after the expansions:

```
func_evaluating(_1) :-
    _1 is 1+2
```

If we try to evaluate this with queries in the top level, the following result will be displayed:

```
?- X = ~func_not_evaluating.

X = 1+2 ?

yes
?- X = ~func_evaluating.

X = 3 ?

yes
```

The `fsyntax` package also provides a quoting functor (^) that prevents the functional expansion of a compound term. That is useful, for example, when the arithmetic evaluation flag is activated and we still want to use the functors that represent arithmetic operations to be just functors and not directly evaluated. The next example shows the behaviour of quotes:

```
:- fun_eval(arith(true)).

func_not_evaluating := ^(1+2).
```

The resulting transformed code is the same as in the first example, a fact with the compound term "1+2".

One advantage of the functional style programming provided by the `fsyntax` extension is that, as mentioned at the beginning of this section, the resulting code is still Prolog.

Note that is perfectly possible to combine `:=` and `:-` in a clause. This is useful to specify conditions on input variables, as a guard or *where* expressions in functional languages. For example:

```
parent(john) := michael.
parent(paul) := john.

grandparent(X) := ~parent(Y) :- Y = ~parent(X).
```

Obviously this can be translated into `grandparent(X) := ˜parent(˜parent(X)).` but the notation allows giving a variable name to the intermediate result. As another example:

```
% the predicate "number_codes(Num,Codes)" is true if the string Codes
    represents the number Num
string_to_int(S) := N :- S = ~number_codes(N).
```

There is another clear use of this for functional notation and that is the mentioned guards. Many functional languages have the so called guards and we literally have this construct natively without losing logic programming behaviour (e.g., backtracking):

```
fact(N) := 1 :- N = 0.
fact(N) := N * ~fact(N-1) :- N > 0.
```

To end up with the original functional syntax package we need to talk about the conditional and disjunctive functors (`?/2` and `|/2`). The standard pattern matching and case distinction from many programming languages is accomplished in Ciao with these two functors from the *fsyntax* package. Switch or case statements from imperative or functional programming are not necessary in Prolog and logic programming, since it is already subsumed by nondeterministic clause execution. This allows writing an expression that takes one value, or follows one execution path or another, depending on whether a condition succeeds. The `?/2` expression performs a *committed choice*, that means, to have an expression that once a condition is true the path continues and never backtracks before that point. This is obviously the behaviour of Prolog's cut `!/0`, but this syntax provides a syntax sugar that allows combining conditional and disjunctive expressions (in the same way as Prolog if-then-else notation). To show the utility of this syntax, next, examples of code using Ciao's functional notation that take advantage of guards are shown. First, factorial of a number:

```
fact(N) := N = 0 ? 1
        | N \= 0 ? N * ~fact(N-1).
```

And an easy encoding from 1 to 5 into roman numerals:

```
encode(N) := N = 1 ? "I"
    | N = 2 ? "II"
    | N = 3 ? "III"
    | N = 4 ? "IV"
    | N = 5 ? "V".
```

To see the behaviour that these functors have in contrast with the standard use of predicate clauses in plain Prolog we can take a look at an example that works nondeterministically. Consider the classic relationship database having the parents of some individuals. In logic programming it is possible to ask for all the possible solutions of a query:

```
% written in functional style, but is the same as having facts of the form
    "parent(X,Y).":
parent(john) := mary.
parent(john) := peter.
parent(paul) := john.

% in contrast, we have the previous functors notation that have functional
    behaviour:
parent_func(X) := X = john ? mary
                | X = john ? peter
                | X = paul ? john.
```

If we try the following queries in the top level and ask for more solutions it will give these results:

```
?- X = ~parent(john).

X = mary ? ;

X = peter ? ;

no
?- X = ~parent_func(john).

X = mary ? ;

no
```

There are some more utilities in this package, but this are the essential ones to code in functional syntax within Ciao. One could think of this package to have functional logic programming behaviour like Curry (Hanus, 2022), but, indeed, these are syntactic transformations. So, the semantics behind is the one of the logic programs that are obtained as result of the expansions.

We end this subsection with more illustrative examples. It is interesting for example to compare examples in Ciao's functional notation to a natural encoding in other relevant languages. For example we can compare the previous factorial code to a Haskell version:

```
fact :: Integer -> Integer
fact n | n == 0 = 1
       | n /= 0 = n * fact (n-1)
```

Or the roman numerals encoding to an OCaml version:

```
let encode n =
  match n with
  | 1 -> "I"
  | 2 -> "II"
  | 3 -> "III"
  | 4 -> "IV"
  | 5 -> "V";;
```

As can be seen, to code with Ciao functional notation is almost the same as the Haskell or OCaml versions.

Some additional examples:

Ciao:

```
func_name(Args) := def :-
    Var1_in_func_scope = def_var1,
    ...
    VarN_in_func_scope = def_varN.
```

Haskell:

```
func_name args = def where
    var1_in_func_scope = def_var1
    ...
    varN_in_func_scope = def_varN
```

OCaml:

```
let func_name args =
    let var1_in_func_scope = def_var1 in
    ...
    let varN_in_func_scope = def_varN in
    def
```

The standard pattern matching and its functional case distinction can be compared with these code examples:

Ciao:

```
func_name(Args1) := def1 :- expr1. % (the body is optional)
...
func_name(ArgsN) := defN :- exprN.
```

or we may want to do it through conditions with disjunctions to avoid backtracking:

Ciao:

```
func_name(Args) := expr1 ? def1
                 | ...
                 | exprN ? defN.
```

Haskell:

```
func_name args1 (| expr1) = def1
...
func_name argsN (| exprN) = defN
```

or:

```
func_name args1 (| expr1) = def1
                (| exprN) = defN
```

OCaml:

```
let func_name args =
    (let additionalVarForMatch in)
    match argsOrVarsInScope with
    | expr1 -> def1
    ...
    | exprN -> defN;;
```

Chapter 3

Extensions

This section is dedicated to showing and discussing the extensions proposed in this work. A goal of this proposal is that they can be implemented using the existing extension mechanisms of Ciao Prolog packages. This means that, instead of supporting the extensions by changing the base language (Prolog), they are implemented as syntactic transformations contained in a *library* (Cabeza and Hermenegildo, 1999). It is important to note that in this way such extensions are modular and can be enabled selectively on different modules of a given application. Moreover, the result of the translation is still Prolog code, so the *client* of a module using these extensions can be plain Prolog code that uses the module with the extensions normally by calling the exported predicates:[1]

```
:- fun_eval get_last/1.
get_last(L) := L = [X]      ? X
             | L = [_|Rest] ? get_last(Rest).

% The previous function is transformed into:
get_last(L,_1) :-
    L=[X],
    !,
    _1=X.
get_last(L,_1) :-
    L=[_2|Rest],
    !,
    get_last(Rest,_1).
```

As already mentioned before, such optional extensions help exploring new language design options, e.g., to reduce verbosity, code duplication, etc. with ideas from other languages.

[1]An easy way to inspect the output of the translations consists in looking at the output of the CiaoPP (The CiaoPP Program Processor) analysis framework (as said in the manual, the output includes transformations).

3.1 Closures

Almost every modern general-purpose programming language has some notion of higher-order support. Indeed, as mentioned in the previous chapter Prolog already supports higher-order in the form of meta-programming, and Ciao Prolog already provided native support for predicate abstractions and partial applications.

The extension described in this section allows a more convenient way of writing predicates within predicates (i.e., similarly to lambda expressions or anonymous functions), so that they can be used directly in higher-order calls. The proposal improves code readability and maintainability without introducing significant issues (though this depends on the implementation details that will discussed in the following section). The extension revisits the *hiord* proposal (Cabeza et al., 2004), that introduces a syntax and semantics for type-free pure logic programming in Ciao Prolog.

From meta-predicate to explicit closure notation. As explained before, *meta-predicate* declarations are used to declare which arguments of a predicate are expected to receive higher-order terms. In predicate-based module systems, those annotations are necessary to disambiguate predicate symbols, e.g., passing a local predicate as an argument to an imported predicate, so that the imported predicate is able to call the local predicate. In the case of Ciao, context is implicitly added during the predicate name resolution.

When a higher-order term appears statically in an argument, this process can be done statically at compile time with no performance penalty. However, when some meta-predicate annotations are missing, or the compiler is unable to propagate that information, runtime checks and runtime module expansions need to be introduced. In addition to runtime cost penalties, this make analysis of code involving higher-order predicates more difficult.

Let us see an example to motivate the *closure* proposal, based on the *hiord* approach. The following predicate `filter/3` implements a higher-order filter predicate:

```
:- meta_predicate filter(pred(1), ?, ?).
filter(_, [], []).
filter(P, [X|Xs], Ys) :-
    ( P(X) ->
        Ys = [X|Ys0]
    ; Ys = Ys0
    ),
    filter(P, Xs, Ys0).
```

That predicate accepts as its first argument a filter predicate, that is used to selectively keep or discard elements from the second list argument into the third one. As in functional programming and also other programming paradigms, such higher-order code avoids code repetition, since the same predicate can be used for many purposes (e.g., filtering even numbers, odd numbers, etc.).

Now, we will describe some usage examples of the predicate above to introduce the closure proposal. Let us write a predicate that filters even numbers from a list. The traditional Prolog solution requires code like the following:

```
is_even(X) :- X mod 2 =:= 0.

even_numbers_from_list(List, Result) :-
    filter(is_even, List, Result).
```

Note that if the `is_even/1` predicate is only used once, sometimes it would be more convenient to place it directly in the `filter/3` call. This is still possible in Ciao Prolog, as it already supports predicate abstractions in this form:

```
even_numbers_from_list(List, Result) :-
    filter((''(X) :- X mod 2 =:= 0), List, Result).
```

Although the current implementation generates slighlty worse code for that version, there is enough static information for the compiler to turn the predicate abstraction into an auxiliary predicate. Let us rewrite the code as follows:

```
even_numbers_from_list(List, Result) :-
    Even = (''(X) :- X mod 2 =:= 0),
    filter(Even, List, Result).
```

In this case, the variable `Even` is unified with a term that is only interpreted as a predicate abstraction when it reaches the `filter/3` call. Although a smart compiler could propagate this *meta-type* information backwards, it is quite easy to find cases where this approach is inefficient. Moreover, it makes increasingly difficult nesting predicate abstractions inside structures.

This motivates the curly brackets notation from the *hiord* approach. Rather than waiting until the term is used in a meta-predicate context, we opt for directly annotating the term:

```
even_numbers_from_list(List, Result) :-
    filter({''(X) :- X mod 2 =:= 0}, List, Result).
```

which can also be written, this time with no inefficiencies, as:

```
even_numbers_from_list(List, Result) :-
    Even = {''(X) :- X mod 2 =:= 0},
    filter(Even, List, Result).
```

Note that this differs from normal Prolog code since it requires treating the `{}/1` term in an special way. The idea in this proposal is we will reuse the `fsyntax` package mechanisms to do that.

Term	::=	...	(extend Figure 2.2)
		{ [$V_1, \ldots, V_n$] -> *PredAbs* }	Closure, positive sharing
		{ -[$V_1, \ldots, V_n$] -> *PredAbs* }	Closure, negative sharing
		{ *PredAbs* }	Closure, default sharing
PredAbs	::=	*PAclause*	A single clause
		PAclause. ... *PAclause*.	Multiple clauses
PAclause	::=	*PAhead* :- *Body*	(Body is optional)
		PAhead := *Term* :- *Body*	(Body is optional)
PAhead	::=	''(*Term*, ..., *Term*)	Anonymous head

Figure 3.1: Grammar for closures

Shared variables in closures. Before introducing the curly-bracket closure notation, let us consider the following example:

```
filter_greater(N, Xs, Ys) :-
    filter(greater(N), Xs, Ys).

greater(N,X) :- X > N.
```

The `filter_greater(N,Xs,Ys)` predicate uses the `filter/3` predicate to filter in `Ys` all elements from `Xs` that are greater than `N`, using the `greater/2` predicate. The filter predicate expects a predicate of arity 1. Passing the `greater(N)` term acts as a partial application, so that it is effectively a predicate of arity 1.

Using the existing predicate abstraction notation in Ciao Prolog this can also be written as the following, where the shared variables must be explicitly annotated:

```
filter_greater(N, Xs, Ys) :-
    filter(([N] -> ''(X) :- X > N), Xs, Ys).
```

This indicates that `N` is a shared variable from the context. The closure notation from this proposal, as we will see later, implements several heuristics to extract the shared variables by default. Thus, the code above can be written as:

```
filter_greater(N, Xs, Ys) :-
    filter({''(X) :- X > N}, Xs, Ys).
```

Syntax for closures. Figure 3.1 shows a simplified grammar for the closure proposal. Note that closures are allowed in term positions, and as described before, this extension uses functional notation (see Figure 2.2). Closures are written as anonymous clauses, where the head predicate name is an empty atom or pair of single quotes, surrounded by `{` and `}`. Before the head, closures are annotated with an optional list of shared variables, as follows:

1. Closure with positive sharing: the shared variables from the outer scope that will be shared with the local environment are the ones explicitly written.
2. Closure with negative sharing: all the variables from the outer scope will be shared in the local environment, except those explicitly written.
3. Closure with default sharing: all the variables from the outer scope will be shared in the local environment except those appear in the predicate abstraction's head.

The variable scoping rules for closures in this proposal are similar to closures in functional languages.

Let us see now some examples of the different sharing modes. The first one is an example of a closure without default variable sharing:

```
list_has_0(L) := Member(0,L) ? valid | invalid :-
    Member = {
        ''(X,[X|_]).
        ''(X,[_|Rest]) :- Member(X,Rest).
    }.
```

The predicate above just checks if `0` is an element of the input list. If so, it returns `valid`, otherwise it returns `invalid`. Instead of using a separate predicate, we define *member* as a closure. As usual, that predicate is recursive. Default variable sharing automatically shares the `Member` variable, thus it can be used inside the closure to perform the resursive call.

As stated above, the user can explicitly indicate which variables will be shared, and thus, belong to the parent scope. This is useful when the variables that are wanted to be shared only appear inside the closure definition. Below, we show an example with a shared variable that is explicitly necessary:

```
filter_1le5_2same(L) :=
    ~filter( { [Z] -> ''(X) :- arg(1,X,Y), Y < 5, arg(2,X,Z) }, L).
```

Followed by a query to show the behaviour:

```
?- Result = ~filter_1le5_2same(
    [f(1,the_value), g(6,the_value), x(12,another_value),
     h(3,yet_another_value), j(0,the_value),
     i(2,SomeVar)]).

Result = [f(1,the_value),j(0,the_value),i(2,the_value)],
SomeVar = the_value
```

The example predicate will filter a list of elements, and the filter function will select elements whose first argument is less than 5, and the second argument is equal to `Z`. Since that variable is *shared* with the parent context in the closure, all invocations will share the same value.

In addition to the previous usage, this positive sharing can be used also to avoid the sharing of the parent context. This means, that neither the X (in fact, it was not going to be shared anyways because of its occurrence in the head) nor the Y would have been inherited from the parent context even if they occurred in it.

Another example with an explicit variable sharing is shown below:

```
all_same_multiplication(L,X) :-
    maplist({ X -> ''(F) :- arg(1,F,Y), arg(2,F,Z), X is Y * Z }, L).
```

Having a list of pairs of numbers, if all of the pairs of numbers multiplied give the same result, then the variable X is unified with the result.

We finish with one more example of a closure that has no explicit sharing. If we wanted to name variables inside the scope with names that have been used, putting them in the head of a clause stops them from being shared between scopes:

```
varX_not_shared_between_scopes(X,Y) := ~Aux(X) :- Aux = { ''(X) := X + Y }.

varY_not_shared_between_scopes(X,Y) := ~Aux(X) :- Aux = { ''(Y) := X + Y }.
```

```
?- Result = ~varX_not_shared_between_scopes(1,2).

Result = 3 ?

yes
?- Result = ~varY_not_shared_between_scopes(1,2).

Result = 2 ?

yes
```

Translation rules for closures. We briefly describe in Figure 3.2 a simplified set of translation rules for closures. As said before, since closures appear as well at term positions, this extension is part of the *fsyntax* package. That extension needs to first identify goal positions, and then expand terms containing predicate calls in functional notation within those goals.

In this formalization, we make the assumptions that follow. First we assume that clauses (including predicate abstractions) are represented as terms. The variable *Ctx* refers to the context of the clause where the transformation is going to take place, so, it contains the variables of the clause. The function *ClVars* is a function that given

$$Tr : Term \rightharpoonup Term$$

$$Tr(\{Head\,\texttt{:-}\,Body\}) \quad = \quad Tr(\{\sigma\,\texttt{->}Head\,\texttt{:-}\,Body\})$$
$$\text{where } \sigma = [x \mid x \in ClVars(Ctx) \wedge x \notin VarSet(Head)]$$

$$Tr(\{-[v_1, \ldots, v_n]\,\texttt{->}Head\,\texttt{:-}\,Body\}) \quad = \quad Tr(\{\sigma\,\texttt{->}Head\,\texttt{:-}\,Body\})$$
$$\text{where } \sigma = [x \mid x \in ClVars(Ctx) \wedge x \notin \{v_1, \cdots, v_n\})]$$

$$Tr(\{[v_1, \ldots, v_n]\,\texttt{->}Head\,\texttt{:-}\,Body\}) \quad = \quad p(v_1, \ldots, v_n)$$
$$\text{where } Head = {}''(a_1, \cdots, a_m)$$
p is a new predicate symbol
"$p(v_1, \cdots, v_n, a_1, \cdots, a_m)\,\texttt{:-}\,Body$" is added to the queue of pending clauses

Figure 3.2: Translation rules for closures

the context of a clause returns its set of variables. The function *VarSet* is a function that returns the set of variables occuring in a term. Using those functions, the syntax transformation for closures (in term positions) will be a partial function *Tr*. This function is expected to return the transformed term, obtain a fresh predicate symbol, and add additional (auxiliary) clauses added to the compilation queue.[2] For simplicity, we assume here that clauses in predicate abstractions have already been treated by the functional notation transformation and have the form $Head\,\texttt{:-}\,Body$.

The first and second rules reduce default variable sharing and negative variable sharing as a positive variable sharing case. The third case deals with positive variable sharing. Positive variable sharing is compatible with the existing implementation of predicate abstractions in Ciao Prolog. Both negative (2) and default sharing (3) – new in this proposal – are based on the idea of automatically connecting variables from the parent scope. Negative sharing explicitly shows which variables will not be shared and default sharing removes only those that appear in the head of the predicate abstraction. The motivation behind this design decision for default sharing was its similarity with *lexical scoping* rules in functional and imperative programming. As we will see in the examples, it resulted in more idiomatic translations of code in functional style, while preserving the semantics of logic programming.

3.1.1 Implementation details

A mentioned in Chapter 2 Prolog already has mechanisms for supporting higher-order programming through its meta-programming facilities. By this we mean creating a term that represents a goal or a partial goal, extending the number of arguments, and calling it later. Predicate abstractions in Ciao Prolog are imple-

[2]For clarity, we use side effects and implicit arguments (Ctx) in those definitions. A more complete and pure formalization is left as future work.

mented on top of this mechanism, with the difference that closures in higher-order programming are not expected to be manipulated. That is, higher-order is in fact a restriction of meta-programming focused only on passing and returning procedures, and enabling flexible compositions, while preventing the programmatic inspection of closures.

The implementation of closures can be divided into two parts: the code expansion that is executed statically, which transforms closures into an internal term-based representation (callable terms), and the runtime support for closures, that executes those callable terms.

The compiler-side of the closure implementation is itself divided into two parts as well: functional notation and higher order. The support for higher-order programming in Ciao is provided by the *hiord* package, while functional notation is implemented in the *fsyntax* package. The *fsyntax* package is in charge of transforming code closures in term positions into finer grained *built-in* literals that the *hiord* package (which is not aware of functional syntax) can manipulate.

Built-ins for controlled meta-expansions. This proposal requires the addition of a new `'$meta_exp'(MetaType, MetaTerm, ExpandedTerm)` built-in predicate, where `MetaType` specifies a meta-type (exactly like those described in `meta_predicate` declarations) for the *meta-term* in the second argument, resulting in an expanded third argument. For example, if the meta-type is the atom `goal`, the compiler will introduce a static or dynamic meta-expansion. Let us see the following examples:

```
?- '$meta_exp'(goal, (X is 1+2), C1).
C1 = $:('arithmetic:is'(X,1+2)) ?

?- '$meta_exp'(pred(2), (''(X,Y) :- Y is X + 1), C2).
C2 = $:('PAEnv'([],'PA'([],''(_B,_A),'arithmetic:is'(_A,_B+1)))) ?

?- '$meta_exp'(pred(2), length, C3).
C3 = $:('PAEnv'(length,'PA'(length,''(_A,_B),'lists:length'(_A,_B)))) ?
```

The three queries use the `'$meta_exp'/3` built-in to obtain the closure representation of a goal (`C1` variable), a predicate abstraction of arity 2 (`C2` variable), and a partial application to a predicate of arity 2 (`C3` variable), respectively. All the closures, or callable terms, begin with the internal `'$:'/1` functor. Obviously, this internal representation is transparent to the user.

When representing goals, this structure just contains the goal. When representing closures, it begins with the `PAEnv` structure, which encodes the predicate abstraction head (arguments), shared variables, and the module expanded body. When this translation happens statically, no meta-expansion is necessary at runtime.

Besides this built-in, we also need the `call/N` family of predicates, which, as mentioned in Chapter 2 accepts as first argument a callable, followed by the necessary arguments (which can be zero). When using the `hiord` package, `call(X,A1,...,An)` can be written as `X(A1,...,An)`.

Closures via functional notation and meta-exp. Having this primitive for meta-expansion of closures, we can now describe how the rules presented in figure 3.2 are actually implemented. The idea is that the curly-bracket terms, which can appear at arbitrary term positions, are expanded as calls to `'$meta_exp'/3`. Let us see this illustrative example:

```
% This predicate uses closure notation
closure(Z) :-
    X = 1,
    Y = 2,
    P = { [X,Y] -> ''(Z) :- Z is X + Y },
    P(Z).

% This predicate is the result after the fsyntax's curly-brackets expansion
closure_metaexp(Z) :-
    X = 1,
    Y = 2,
    '$meta_exp'(pred(1), ( [X,Y] -> ''(Z) :- Z is X + Y ), P),
    P(Z).

% This predicate uses an auxiliary predicate
closure_aux(Z) :-
    X = 1,
    Y = 2,
    P = aux(X,Y) % 'aux', new predicate name for the closure
    call(P,Z). % this will result in calling "aux(X,Y,Z)"

aux(ShVar1,ShVar2,Arg1) :- Arg1 is ShVar1 + ShVar2.
```

The `closure/1` predicate unifies a variable with a closure that takes two variables from the context, and then calls it. The `closure_metaexp/1` version represents the same predicate, after the functional notation has expanded the curly-bracket term as a closure for a predicate of arity 1 (as seen from the predicate abstraction head), using the *meta-exp* builtin. The process is similar to that of expansion of arithmetic expressions or `fun_eval` terms, with an additional step to infer the meta-type from the shape of the closure. Finally, the `closure_aux/1` predicate represents the plain Prolog implementation of the same predicate, which is similar to what the internal representation achieves.

3.1.2 Differences with the Hiord proposal

The main difference w.r.t. the *Hiord-1* proposal (Cabeza et al., 2004) lies in the handling of shared variables. In this work, we initially opted for explicit sharing of variables but after studying closures in other languages, using realistic examples, identified that the code was unnecessarily verbose. Automatic sharing of variables from the context and a default sharing similar to *lexical scoping*, fixed this problem.

Finally, there is one last difference, the placeholder syntax sugar with the symbol `"#"` is not included. That proposal defines a placeholder syntax for simple abstractions with the mentioned atom (`#`), that indicates a *hole* for a predicate abstraction variable. For example, `{ # < # }` represents the `{ "(X,Y) :- X < Y }` predicate abstraction.

We opted for keeping this proposal as simple as possible and allow these kind of extensions as additional optional notation.

3.2 Closures in other languages

This extension is closely related with features available in other programming languages. The principal one is the concept of closure, lambda expressions, or anonymous functions. In fact, this extension is an approach to adding closures in an untyped logic programming language. However, similarly to lambda expressions, it can be used to implement other features like *where* expressions from functional languages.

In general, closures are defined as a record storing a function with its lexical scope for variable name binding; thus, it needs some support for functions as first-class citizens. This is the reason why it is closely related to higher-order and metaprogramming. Handling of environments (for local variables) can be more complex in other programming languages than in Prolog, where there is a unique context per clause and variables can be passed around freely by means of unification (e.g., no need to care about lifetimes).

3.2.1 Haskell

Haskell is a lazy pure functional language that treats closures and its contexts for local bindings for variables in closures in the same way. As explained in the STG Machine (Jones, 1993), the closures, when evaluated on demand (lazyness), are progressively replaced by their values. When the closure is created in the heap, it includes the code for creating free variables if needed, or directly bind values for them, and later when the closure needs to be evaluated it replaces its result value using the bounded values of its free variables:

```
foo :: Fractional a => a -> a -> (a -> a)
foo x y = (\z -> z + r)
          where r = x / y

f :: Fractional a => a -> a
f = foo 1 0

main = print (f 123)
```

A possible translation to Ciao of the code above could be:

```
foo(X,Y) := { ''(Z) := Z + R :- R = X / Y }.

main := ~Closure(123) :- Closure = ~foo(1,0).
```

The difference is due to absence of *currying*. Although it is possible to implement the equivalent of curried syntax for functions, this would incur in some performance penalties and incompatibilties with existing Prolog code. Thus our proposal relies on the existing mechanisms for partial applications (with similar advantages to curried syntax). This forces us to bound the free variables of the function named *foo* inside the context where the function is going to be used. Otherwise, if we try to define a function with no arguments *closure* as the evaluation of the function *foo* with two variables it will not understand it as a closure and when using it in the main with one argument it will say that there is no existence of a predicate *closure* with that number of arguments (because it is 0 the number of arguments, not the curryed one defined by the closure bounded *foo* with arguments 1 and 0 for the free variables X and Y).

In addition to the closures, another use for the extension is the encoding of *where* expressions defining auxiliary local functions. For example:

```
where_clause_with_closures :: [[Char]] -> [Char]
where_clause_with_closures (l:ls) = aux_func (aux_func_rec l) :
    where_clause_with_closures ls
  where
  aux_func num = case num of
                   0 -> '0'
                   1 -> 'X'
                   _ -> 'W'
  aux_func_rec list = case list of
                          [] -> 0
                          'X':rest -> 1 + aux_func_rec rest
                          _:rest -> 0 + aux_func_rec rest
where_clause_with_closures [] = []
```

A straightforward translation of the code above would be:

```
where_clause_with_closures([L|Ls]) := [~Aux_func(~Aux_func_rec(L)) | ~
   where_clause_with_closures(Ls)]
  :-
  Aux_func = { ''(Num) := Num = 0 ? '0'
                        | Num = 1 ? 'X'
                        | 'W' % If only one clause no need to put the dot
             },
  Aux_func_rec = {
                    ''([]) := 0.
                    ''([88|Rest]) := 1 + ~Aux_func_rec(Rest).
                    ''([_|Rest]) := 0 + ~Aux_func_rec(Rest).
                 }.
where_clause_with_closures([]) := [].
```

As seen above, local variables in the body of the clause (right after the rule neck (:-)) can be used that have the same effect as the *where* expressions.

As one last example, we will show the standard linear complexity reverse function written with a *where* clause in Haskell:

```
reverse :: [a] -> [a]
reverse xs = go [] xs where
  go :: [a] -> [a] -> [a]
  go acc    []  =      acc
  go acc (x:xs) = go (x:acc) xs
```

Note that there exists two different variables with the name *xs*, due to its occurrence in the pattern matching that is local to *go* function. If it were not in the pattern matching, then it would be inherited from the parent context and the variable *xs* would have the value from the pattern matching of *go* definition. Note that this rule is the same that we use in our proposal for the default variable sharing in closures, so we can simply write it as:

```
reverse(Xs) := ~Reverse([], Xs) :-
    Reverse = {
        ''(Acc, []) := Acc.
        ''(Acc, [X|Xs]) := ~Reverse([X|Acc], Xs).
    }.
```

Note that this is not possible using the traditional notation for predicate abstractions in Ciao.

3.2.2 OCaml

As another representative of functional programming languages, we will study in this section the OCaml language. OCaml (Objective Caml) is a strongly typed functional language derived from ML (Paulson, 1996). OCaml differs from Haskell in many aspects, for example it uses eager (strict) evaluation by default, and it allows mutable state and side effects, while Haskell uses lazy evaluation and it handles side effects through monads. Although the semantics are different, OCaml also allows closures in a similar way to Haskell.

In this case, closures save the content of variables by value. As explained in (OCaml), the "`let <formal> = <defines> in <expression>`" expressions define what is called the formal parameter to be a local variable in the body expression. It also allows the use of let and let-rec expressions for binding variables to functions. The syntax permits declaring local variable arguments for these functions but it can be seen as syntactic sugar for closures as in the next code:

```
let <function_name> <local_argument_names> = <definition> in <expression>

(* The previous let expression is the same as writting the following *)
let <function_name> = (fun <local_argument_names> -> <definition>) in
    <expression>
```

Using a *let* expression to define a function is very similar to the equivalent construct in Ciao in the default variable sharing, where variables in the body appearing in the parent scope are captured in the abstraction.

Let us see some examples of closures in OCaml and their equivalent code in Ciao:

```
let make_multiplier x =
    let inner_function y = x * y
    in inner_function

let multiply_by_two = make_multiplier 2

let result = multiply_by_two 5 (* result is 10 *)
```

The function *make_ multiplier* above returns a closure that multiplies its own argument by the argument of the function. The variable *result* is 10. The Ciao translation can be expressed as follows:

```
make_multiplier(X) := InnerFunction :-
  InnerFunction = {
    ''(Y) := X * Y
  }.

result := ~MultiplyByTwo(5) :- MultiplyByTwo = ~make_multiplier(2).
% a query with "X = ~result." unifies X with value 10
```

Similarly to Haskell in Section 3.2.1, OCaml allows multiple *where* expressions to define auxiliary local functions, which would have a similar translation in Ciao. Let us see the OCaml version:

```
let where_clause_with_closures lists =
  let aux_func num =
    match num with
    | 0 -> '0'
    | 1 -> 'X'
    | _ -> 'W'
  in
  let rec aux_func_rec l =
    match l with
    | [] -> 0
    | 'X' :: rest -> 1 + aux_func_rec rest
    | _ :: rest -> 0 + aux_func_rec rest
  in match lists with
    | [] -> []
    | l :: ls -> aux_func (aux_func_rec l) :: where_clause_with_closures ls
```

3.2.3 Python

Once we move to imperative languages there is a big difference in the treatment of variable scoping. The main difference is due to mutability of variables and destructive assignment. Let us consider an example of a closure capturing a mutable variable from its parent context:

```
def calculate():
    num = 1
    def inner_func():
        nonlocal num
        num += 2
        return num
    return inner_func

""" call the outer function """
odd = calculate()

""" call the inner function """
print(odd())
print(odd())
print(odd())

""" call the outer function again """
odd2 = calculate()
print(odd2())

""" output will be: 3 5 7 3 """
```

Although Ciao Prolog (as some other Prolog systems) does support backtrackable mutable variables, an idiomatic translation of this example is not simple, or produces code that would usually be written differently in the logic programming paradigm.

On the other hand, when closures do not mutate any variable from the context, closures in Python seem very similar to the closure examples presented previously for Haskell and OCaml (and thus have a similar translation to Ciao):

```python
def where_clause_with_closures():
    def aux_func(num):
        if num == 0:
            return '0':
        elif num == 1:
            return 'X'
        else:
            return 'W'
    def aux_func_rec(l):
        result = 0
        for c in l:
            if c == 'X':
                result += 1
            return result

    return lambda l: aux_func(aux_func_rec(l))

""" Imagine we have a list of string in a variable named list """
closure_variable = where_clause_with_closures()

result = []
for s in list:
    result.append(closure_variable(s))
```

3.3 Let notation

The functional paradigm is based essentially on computing everything as a result of a function. Let us introduce an example showing a typical "if-then-else" expression, and a recursive local function in a pure functional language:

```
partial def primeFactors (n : Nat) : List Nat :=
  loop n 2 [] |>.reverse
where
  loop (tgt candidate : Nat) (acc : List Nat) : List Nat :=
    if tgt <= 1 || candidate > tgt then
      acc
    else if tgt % candidate = 0 then
      loop (tgt / candidate) candidate <| candidate :: acc
    else
      loop tgt (candidate + 1) acc
```

In the translation of the code above to Ciao, if we want to keep a similar structure, rather than using if-then-else notation for goals (`... -> ... ; ...`), we can use the equivalent notation for conditional expressions in *fsyntax* (`... ? ... | ...`):

```
primeFactors(N) := ~reverse(~Loop(N,2,[])) :-
    Loop = {
        ''(Tgt,Candidate,Acc) := (Tgt =< 1; Candidate > Tgt) ? Acc
            | Tgt mod Candidate =:= 0 ? ~Loop(Tgt//Candidate, Candidate,
                [Candidate|Acc])
            | ~Loop(Tgt, Candidate+1, Acc)
    }.
```

As explained in section 2.3, the evaluable functors `?/2` and `|/2` are the equivalent in Ciao functional notation to functional conditional expressions.

However, one shortcoming of the conditional expression syntax in *fsyntax* is that it does not allow arbitrary goals before the final expression is computed, e.g., in the *then* or *else* branch. The goal of the *let* notation is having a concise syntax to write an expression that evaluates several goals before *returning* an expression. That is, an expression in term positions like:

```
{ let Variable <- ExprVar, Goal1, ..., GoalN, return(ExprRet) }
```

that defines its own context (where there are one or more locally defined variables), several goals to be executed, and finally the result is unified with the term appearing in the final *return* literal.

Like for the closure extension, this enables local name bindings for the variables that are involved in a certain way. The rules are simple:

1. The variables defined by the *let* construct are local to the curly brackets scope.
2. All the other variables from the outer context are inherited. That means, excepting the ones that have name collision with the local ones. In that case, they become local inside the scope.

Let us present a simple example using this notation that provides an evidence of the local name binding of the variables defined by the let construct and the variables inherited from the outer context:

```
simple_let_notation_example(X, Y, Result1, Result2) :-
    Result1 = { let X <- Y, return(X) },
    Result2 = X.
```

The next query in the top level will give this result:

```
?- simple_multiplication(2, 1, R1, R2).

R1 = 1,
R2 = 2 ?

yes
```

The rules are clear and define the usual semantics in functional let notation. This notation is useful, especially in functional style when an expresion involves several intermediate computations that must be executed before reaching a final result. Moreover, the let notation can ease maintainability by providing local context for some variables. This can be shown in the following code, and it is easy to see how much more complex constructions can be built:

```
% Imagine we have a predicate "get_last" defined that returns the
% last element of a list.

% With this simple predicate, we can see that the normal coding of it
% is simpler than using let:
add_lasts(L,L2) := ~get_last(L) + ~get_last(L2).

% Despite that, we can see that the let separates the values involved
% in the result from the context of the predicate "add_lasts"
% providing local treatment of the parts:
add_lasts(L,L2) := {
   let X <- ~get_last(L),
   let Y <- ~get_last(L2),
   return(X + Y)
}.
```

```
% One last example mixing the let notation with the closure one:
add_lasts(L,L2) := {
   let X <- ~Get_last(L),
   let Y <- ~Get_last(L2),
   return(X + Y)
} :-
    Get_last = {
        ''([X]) := X.
        ''([_|Rest]) := ~Get_last(Rest).
    }.
```

As said before, this notation allows a more straightforward use of a functional programming style. The next example shows a predicate in Prolog syntax and two translations using functional syntax and functional let notation. Having a list of numbers they return the sum of the adjacent cells of a given index (it will fail if the given index is 1 or less, or greater or equal than the length of the list):

```
% In plain Prolog
adjacent_cells_sum(List, Index, Result) :-
    NextIndex is Index + 1,
    nth(NextIndex, List, Next),
    PrevIndex is Index - 1,
    nth(PrevIndex, List, Previous),
    Result is Next + Previous.

% Prolog with fsyntax
adjacent_cells_sum(List, Index, Result) :-
    Next = ~nth(Index+1, List),
    Previous = ~nth(Index-1, List),
    Result is Next + Previous.

% Let notation
adjacent_cells_sum_func(List, Index) := {
  let Next <- ~nth(Index+1,List),
  let Previous <- ~nth(Index-1,List),
  return(Next + Previous)
}.
```

As we can see, the code in functional style is usually easier to read. In this small example the version using *let* notation just shows a very minor advantage of not duplicating the `Result` variable. Despite that, it allows using several block expressions with local variables without the need to do manual renamings.

3.3.1 Implementation

The implementation of this extension is also based on both the *fsyntax* package and the extension for closures. A simple translation identifies the *let* annotated curly brackets and expands them as a combination of *closure* and *call* expressions. For example:

```
simple_let_notation_example(X, Y, Result1, Result2) :-
    Result1 = { let X <- Y, return(X) },
    Result2 = X.
```

is translated to:

```
simple_let_notation_example(X, Y, Result1, Result2) :-
    Result1 = ~call({ ''(X, R) :- X = R }, Y),
    Result2 = X.
```

3.3.2 Let notation in other languages

let expressions originate in functional programming and lambda calculus (Selinger, 2008). Besides function abstractions and function applications, the let construct provides a shorthand notation for the application of an abstraction in an expression in a concise way. For example:

$$(\lambda x.E)\ a \longrightarrow \text{Let } x = a \text{ in } E$$

The let notation is related with the concept of local variable binding. A related construct in Lisp and OCaml is the *letrec* (Arvind et al., 1996), which allows declaring local recursive functions. Other languages, such as Lean or Haskell, do not distinguish between *let* and *letrec*.

In the rest of this section we will show some definitions of the factorial function written in different languages. First, in Lisp and OCaml, using letrec for the recursive of the function, later in Haskell and, finally, an equivalent translation in Ciao.

Lisp:

```
(letrec ((factorial (lambda (n)
                      (if (<= n 1)
                          1
                          (* n (factorial (- n 1)))))))
  (factorial 5))
```

OCaml:

```
let rec factorial n =
  if n == 0 then 1 else n * factorial (n - 1)
in factorial 5
```

Haskell:

```
factorial_5_with_let :: Integer
factorial_5_with_let =
  let
    factorial :: Integer -> Integer
    factorial 0 = 1
    factorial n = n * factorial (n - 1)
  in
    factorial 5
```

Finally the Ciao translation (note that the same can be written without *let*, just with a variable binding in the body):

```
factorial_5_with_let :=
    { let Factorial <- { ''(N) := 1 :- N =< 1.
                         ''(N) := N * ~Factorial(N-1).
                       },
      return(~Factorial(5))
    }.
```

Chapter 4

Other Related Work

As mentioned before, the Ciao Prolog design that has been our starting point already included facilities for higher-order support through the `hiord` approach as well as functional notation for relations through the `fsyntax` approach, all on top of the Prolog operational semantics. Chapter 2 already discussed two of the other approaches for adding higher-order in logic programming: Hilog (Chen et al., 1993) and λProlog (Nadathur and Miller, 1988).

In addition to these higher-order extensions to Prolog, other related work exist that is based on a more explicit combination of the semantics of the logic and functional paradigms. This is in fact a well-studied problem in the literature, with different approaches that differ on the evaluation mechanism. These approaches for marrying functional and logic programming typically extend first-order terms with applicative expressions (atoms, abstractions, and applications) augmented with logical variables. These approaches require more involved operational semantics. We can split them into narrowing and residuation. Rather than using resolution, narrowing is based on term reduction (Hullot, 1980). The most relevant systems based on narrowing are:

- ALF (Hanus and Schwab, 1991b,a), based on innermost narrowing with normalization.
- LPG (Bert et al., 1993), which extends and combines SLD-resolution with innermost narrowing.
- BABEL (Kuchen et al., 1990), based on lazy narrowing and provides some higher-order features.
- Curry (Hanus, 2013), which uses a lazy narrowing strategy with residuation for concurrent computations.
- TOY (Kuchen et al., 1992; López-Fraguas and Sánchez-Hernández, 1999), lazy narrowing, higher-order features (including higher-order logic variables), dise-

quality constraints (for constructed data terms), and non-deterministic functions.

Another approach to combining functional features to logic programs uses residuation (Aït-Kaci et al., 1986; Thom and Zobel, 1987). In residuation unification is delayed until variable instantiations make it possible to reduce unificands. Some relevant systems using residuation or an equivalent approach are:

- LIFE (Aït-Kaci, 1993; Ait-Kaci et al., 1986)
- Oz (Haridi et al., 1996)
- NU-Prolog (Thom and Zobel, 1987)
- Mercury (Henderson et al., 2014) (where the execution algorithm uses analysis and static rewriting to avoid delays).

The residuation approach is relatively easy to combine with the `hiord` and `fsyntax` approaches and can be seen as orthogonal and complementary to these Ciao packages and our work extending them.

Chapter 5

Conclusions

Designing a programming language that allows the programmer to express ideas to the computer in an efficient and comfortable manner is certainly a challenging task. Many modern languages have been influenced by the study of the theoretical frameworks and features of other programming languages in order to find useful solutions. As a result currently many high-level general-purpose programming languages combine features from multiple paradigms and in particular incorporate features from functional programming. Some characteristics of the functional programming paradigm that are typically attractive to carry over to other paradigms is the excellent support for higher order and the syntactic compactness. In this work we have studied these issues in the context of Prolog.

We have seen that there are multiple solutions for achieving higher order and other functional programming characteristics within logic programming. Many of these change the semantics to include part of the semantics of functional languages, such as the simply typed lambda calculus, or to add fragments of higher-order logics, such as for example lambda term unification. In this work we have concentrated instead on the approach taken by Ciao Prolog system, which also includes the possibility of programming in a functional style and using higher order, but without changing the underlying Prolog operational semantics. The Ciao approach is based on syntactic transformations, thanks to the language extension capabilities and module system of the Ciao language.

We have presented new extensions to the Ciao support for functional notation and higher order, while continuing in the line of not changing the underlying Prolog semantics. This has been done by extending the previous work in *Hiord*, which has been revisited and improved, and compared to similar solutions in other programming languages.

The transformational approach pioneered by Ciao is perhaps not always the perfect solution for every feature, but we believe that both the previous work done in the context of Ciao and our extensions prove that extending the semantics in a minimal and concise way only when needed (something that can be done with modularity) and through transformations, respects the language philosophy without adding unnecessary noise to the language, and is a desirable solution, provided of course the underlying language has the power and mechanisms that allow supporting the extensions. In summary, independently of the concrete extensions proposed, we believe this work also contributes to showing that the Ciao Prolog philosophy of starting with a powerful kernel language and extending it gradually and modularly continues to be an attractive solution for developing further the language in an efficient and flexible way.

Bibliography

AÏT-KACI, H. An Introduction to LIFE – Programming with Logic, Inheritance, Functions and Equations. In *Proceedings of the 1993 International Symposium on Logic Programming* (edited by D. Miller), 52–68. MIT Press, 1993.

AIT-KACI, H., NASR, R. and LINCOLN, P. E An Overview. Tech. Rep. AI-420-86-P, Microelectronics and Computer Technology Corporation, 9430 Research Boulevard, Austin, TX 78759, 1986.

APT, K. and VAN EMDEN, M. Contributions to the theory of logic programming. *Journal of the ACM*, Vol. 29(3), 841–863, 1982.

APT, K. R. Introduction to logic programming. In *Handbook of Theoretical Computer Science* (edited by J. van Leeuwen), 493–576. Elsevier, 1990.

ARVIND, MAESSEN, J.-W., NIKHIL, R. S. and STOY, J. A lambda calculus with letrecs and barriers. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, 19–36. Springer, 1996.

AÏT-KACI, H., LINCOLN, P. and NASR, R. Residuation: A paradigm for integrating logic and functional programming. Tech. rep., MCC, 1986.

BARENDREGT, H. *The Lambda Calculus*. North-Holland, Amsterdam, 1981.

BERT, D., ECHAHED, R. and REYNAUD, J.-C. Reference manual of the specification language lpg. Tech. rep., LGI-IMAG, Gernoble, 1993.

CABEZA, D. and HERMENEGILDO, M. V. A New Module System for Prolog. Technical Report CLIP8/99.0, Facultad de Informática, UPM, 1999.

CABEZA, D., HERMENEGILDO, M. V. and LIPTON, J. Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction. In *Ninth Asian Computing Science Conference (ASIAN'04)*, no. 3321 in LNCS, 93–108. Springer-Verlag, 2004. ISBN ISBN 3-540-24087-X. ISSN ISSN 0302-9743.

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

Clark, K. L. Negation as Failure. In *Logic and Data Bases* (edited by H. Gallaire and J. Minker), 293–322. Springer, 1978.

van Emden, M. H. and Kowalski, R. A. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, Vol. 23, 733–742, 1976.

Google Drive. Associated code folder. `https://drive.google.com/drive/folders/1kQQcok3Nw28Z-o58YLcLaOfzAL0g66rj`, 2024.

Hanus, M. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, 123–168. Springer LNCS 7797, 2013.

Hanus, M. From logic to functional logic programs. *Theory and Practice of Logic Programming*, Vol. 22(4), 538–554, 2022.

Hanus, M. and Schwab, A. ALF user's manual. FB Informatik, Univ. Dortmund, 1991a.

Hanus, M. and Schwab, A. The implementation of the functional-logic language ALF. FB Informatik, Univ. Dortmund, 1991b.

Haridi, S., Van Roy, P. and Smolka, G. An Overview of the Design of Distributed Oz. In *Proc. of the 1996 JISCLP Workshop on Multi-Paradigm Logic Programming*. T.U.Berlin, 1996.

Henderson, F., Conway, T., Somogyi, Z., Jeffery, D., Schachte, P., Taylor, S., Speirs, C., Dowd, T., Becket, R., , Brown, M. and Wang, P. *The Mercury Language Reference Manual. Version 14.01.1*. The University of Melbourne, 2014. Available from `https://mercurylang.org/information/doc-release/reference_manual.pdf`.

Hullot, J.-M. Canonical forms and unification. In *Proc. 5th Conference on Automated Deduction*, 318–334. Springer LNCS 87, 1980.

Jaffar, J. and Lassez, J.-L. Constraint Logic Programming. In *ACM Symposium on Principles of Programming Languages*, 111–119. ACM, 1987.

Jones, S. L. P. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine version 2.5. 1993.

Kowalski, R. and Kuehner, D. Linear resolution with selection function. *Artificial Intelligence*, Vol. 2(3), 227–260, 1971.

Kuchen, H., Loogen, R., Navarro, J. M. and Artalejo, M. R. Graph-based implementation of a functional logic language. In *European Symp. on Prog. (ESOP)*, LNCS 432, 271–290. 1990.

Kuchen, H., López-Fraguas, F., Moreno-Navarro, J. and Artalejo, M. R. Implementing a lazy functional logic language with disequality constraints. In *Joint International Confeence and Symposium on Logic Programming - JIC-SLP'92, Washington (USA)* (edited by K. Apt), 207–221. Association for Logic Programming and University of Maryland, The MIT Press, 1992.

Lloyd, J. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

López-Fraguas, F. and Sánchez-Hernández, J. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, 244–247. Springer LNCS 1631, 1999.

Marriot, K. and Stuckey, P. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

Nadathur, G. and Miller, D. An Overview of λProlog. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, 810–827. MIT Press, 1988.

Nilsson, N. J. Problem solving methods in artificial intelligence. 1971.

OCaml. The ocaml manual. `https://www.ocaml.org/manual/5.2/manual001.html`, 2024.

Paulson, L. C. *ML for the Working Programmer*. Cambridge University Press, 1996.

PROLOG. *PROLOG. ISO/IEC DIS 13211 — Part 1: General Core*. International Organization for Standardization, National Physical Laboratory, Teddington, Middlesex, England, 1994.

Robinson, J. A. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, Vol. 12(23), 23–41, 1965.

Selinger, P. Lecture notes on the lambda calculus. *arXiv preprint arXiv:0804.3434*, 2008.

Sterling, L. and Shapiro, E. *The Art of Prolog*, Ch. 3, 39. Logic Programming Series. MIT Press, fourth edn., 1987. *Numbered as Program 3.4; this number may be different in other reprints.*

Swift, T. and Warren, D. S. Xsb: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming*, Vol. 12(1-2), 157–187, 2012.

Tamaki, H. and Sato, M. OLD Resolution with Tabulation. In *Third International Conference on Logic Programming*, 84–98. Lecture Notes in Computer Science, Springer-Verlag, London, 1986.

The CiaoPP Program Processor. The ciaopp low-level interface. `https://ciao-lang.org/ciao/build/doc/ciaopp.html/ciaopp.html`, 2022.

Thom, J. and Zobel, J. *NU-Prolog Reference Manual*. Dept. of Computer Science, U. of Melbourne, 1987.

Van Hentenryck, P. Constraint logic programming. *The Knowledge Engineering Review*, Vol. 6(3), 151–194, 1991.

Warren, D. S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R. and Rossi, F. *Prolog - The Next 50 Years*. No. 13900 in LNCS. Springer, 2023.

Wastl, E. Advent of code. `https://adventofcode.com`, 2023.

Weslley College. Lambda calculus notes. `https://cs.wellesley.edu/~cs251/f20/notes/lambda.html#free-variables-in-the-lambda-calculus`, 2018.