The field of programming languages has seen significant evolution over the years with various paradigms emerging designed to address different kinds of computational problems and methodologies. In this chapter, we provide an overview of the logic programming paradigm and discuss , the primary logic language used in this thesis. In addition to this, we overview the field of specialization which helps us overcome some of the performance challenges that are encountered in logic programming.

# 1 Logic Programming Languages

Over the years, multiple logic programming languages have been developed, with [?] being the most widespread. It was the first successful attempt to enable declarative programming by means of writing programs in a subset of formal logic. At its core, uses Horn clauses, a semidecidable subset of first-order predicate logic. Each program formulates a set of facts and predicates that connect these facts. The evaluation of a program is done by a Selective Linear Definite clause resolution [?] (SLD resolution) of a query, often following the depth-first approach.

For years, logic programming was highly limited by hardware capabilities, leading to necessary compromises. One of them was an early removal of occurs-check from the unification algorithm [?]. This means that running a query "`? f(X, a(X)).`", given a program "`f(X, X).`", produces a nonsensical result "`X` $\mapsto$ `a(X)`". It is up to the user to ensure that a variable never occurs in a term it is unified with. Fortunately, a special sound unification predicate such as `unify_with_occurs_check` can be used to prevent such results.

Another compromise is linked to the implementation details and has more significant consequences. Logic languages are inherently nondeterministic, and evaluation on a deterministic computer requires decisions about how to explore the search space. was first designed for automatic theorem proving, an area in which a single solution to a query is generally sufficient. Thus, most implementations feature depth-first search, which often results in either non-termination or the generation of infinitely many similar answers to a query when an infinite branch of the search tree is explored. Additionally, non-relational constructs such as a cut and `copy-term` have been adopted for efficiency reasons. Unfortunately, these two aspects often limit a relation to a single mode and directly contradict the main idea of declarative programming making it impossible to write a program without considering the peculiarities of the language.

About twenty years after had been created, there was a resurgence of the logic programming paradigm with the emergence of new languages, including [1], [2], and others. and are stand-alone functional logic programming languages designed to combine the power of logic programming with the more mainstream functional programming. These languages have dedicated compilers which make

[1] The website of the programming language https://mercurylang.org/

[2] The website of the programming language https://curry.pages.ps.informatik.uni-kiel.de/curry-lang.org/

it difficult to interoperate with bigger systems typically written in a general-purpose language. Additionally, a prominent video games developer Epic Games invested into designing a new functional logic programming language [**?**] which is still under development at the time of writing this thesis.

In contrast, [3] is implemented as a lightweight embedded domain-specific language, enabling the power of logic programming in any general purpose language. features interleaving search [**?**] that guarantees that every solution to a query will be found, given enough time. Moreover, its extendible architecture allows for easy experimentation and addition of new features. The main design philosophy of is to adhere to the pure logic programming as much as possible, so any program can be called in any direction. Taking all these considerations into account, we chose as the main language for this research.

# 2 Specialization

The first specialization method, called supercompilation, was introduced by Turchin in 1986 [**?**]. It was designed for the Refal programming language [**?**], which was significantly different from the mainstream languages of the time. Since then, supercompilation has been adapted for various languages, expanding its utility across various programming paradigms [**?**, **?**]. Numerous modifications have also emerged, featuring alternative termination strategies, generalization, and splitting techniques [**?**, **?**, **?**].

Several optimizations rely on the information about program arguments known statically. These optimizations precompute the parts of the program that depend on the known arguments and produce a more efficient residual program. Such transformations are generally known as mixed computations, specialization, or partial evaluation. It was first introduced by Ershov [**?**] and was mostly aimed at imperative languages. A lot of effort has been extended to partial evaluation [**?**, **?**] since its first appearance, including the development of self-applicable partial evaluators.

In logic programming, a general framework called rules + strategies, or fold/unfold transformations, was introduced by Pettorossi and Proietti [**?**, **?**]. It serves as a foundational theory for many semantics-preserving transformations, including tupling, specialization, compiling control, and partial deduction. Unfortunately, this approach relies on user guidance for control decisions, its termination is not always guaranteed, and because of it its automation is complicated.

Specialization in logic programming is commonly referred to as partial deduction. It was introduced by Komorowski [**?**] and formalized by Lloyd and Shepherdson [**?**]. Comparing to fold/unfold transformations, partial deduction is less powerful, because it considers every atom on its own and does not track dependencies between variables. However, it is significantly easier to control and can be automated.

[3] The website of the programming language http://minikanren.org/

The main drawback of partial deduction is addressed by Leuschel with conjunctive partial deduction [**?**] in the ECCE system. This method makes use of the interaction between conjuncts for specialization, removing some repeating traversals of data structures as a result. We implemented this algorithm as a proof-of-concept for `miniKanren`, and found out that some of the specialization results were subpar. In some cases, the specialized programs performed worse than the original ones.

Partial evaluators are categorized into offline and online methods, depending on whether control decisions are made before or during the specialization stage. LOGEN is the implementation of the offline approach for logic programming, developed by Leuschel [**?**]. It includes an automatic binding-time analysis to derive annotations used to guide the specialization process. Offline specialization usually takes less time than online, and is capable to generate shorter and more efficient programs.

The fact that majority of implementations do not impose a type system may be seen as a disadvantage when it comes to optimizations. developed a strong static type and mode system that can be used in compilation [**?**, **?**]. Mode analysis embodies data-flow analysis that makes it possible to compile the same definition into several functions specialized for the given direction.

# 3 Relational Programming and

is not a single language, but a family of domain specific languages for logic programming. The core language is very small and can be embedded in any host language, be it functional, imperative, object-oriented or logic-based[4]. Furthermore, being designed for easy modification, features multiple extensions, including Constraint Logic Programming, nominal logic programming, and tabling. In this thesis, we focus on the core language shared between all languages in the family that can be implemented in as few as forty lines of code [**?**]. Let us now describe the syntax and semantics of the language and illustrate it with some examples.

## 3.1 Syntax

The basic block of the language is a goal which is analogous to a predicate and can be considered as a function that takes a state as an input and either fails or succeeds, producing a sequence of new (extended) states. Although the state may vary depending on which extensions a particular implementation features, in its simplest form it is just a substitution of logic variables encountered in a program. This substitution represents the answer to the user query and is used as the primary data structure describing the intermediate result of the computation.

A goal $\mathcal{G}$ can be constructed using one of four constructors: term unification ($==$), conjunction ($\&$), disjunction ($|$), or fresh variable introduction ($fresh$):

[4]See the list of implementations at http://minikanren.org/

see figure **??**. Term unification is the backbone of any logic language, and it succeeds if and only if its two arguments unify [**?**]. A term $\mathcal{T}$ can either be a variable $\mathcal{V}$, or created from a list of other terms with a constructor $\mathcal{C}$. Succeeding, unification can modify a substitution by adding new associations for the variables used in the terms being unified. When unification fails, no substitution is produced, and the computation halts.

r0.43 $\mathcal{T} : \mathrm{V}$
$\mid \mathcal{C}_k^o\ (\mathcal{T}_0, \ldots, \mathcal{T}_k)$
$\mathcal{G} : \mathcal{T} == \mathcal{T}$
$\mid \mathcal{G}_1\ \&\ \mathcal{G}_2\ \&\ldots\ \&\ \mathcal{G}_k$
$\mid \mathcal{G}_1 \mid \mathcal{G}_2 \mid \ldots \mid \mathcal{G}_k$
$\mid fresh\ (\mathcal{V}_1, \ldots, \mathcal{V}_k)\ in\ \mathcal{G}$
$\mathcal{D} : \mathcal{R}_k^o\ (\mathcal{V}_0, \ldots, \mathcal{V}_k) = \mathcal{G}$

The syntax of the core language

The constructor $fresh$ introduces new, or fresh, logic variables into scope. The last argument of the constructor is a goal, in which these variables are considered bound. Furthermore, introducing a variable with the same name as another in the wider scope shadows the previous variable.

The last two constructors, & and |, represent the conjunction and disjunction of goals respectively. With them, we express Boolean logic over the predicates. Notice, that unlike in other implementations, at least two goals should be joined with these constructors, and we do not impose a limit on the number of the goals in a single constructor. Moreover, we treat these operators as right-associative when evaluating a program, which means that the following equation holds: $g_1\ \&\ g_2\ \&\ g_3\ \&\ g_4 = g_1\ \&\ (g_2\ \&\ (g_3\ \&\ g_4))$. This interpretation aligns with the operational semantics of the language and simplifies its implementation.

Finally, to define a new relation $\mathcal{D}$, one specifies its name $\mathcal{R}_k^o$, variable arguments $\mathcal{V}_0, \ldots, \mathcal{V}_k$, and body $\mathcal{G}$, which must be a goal. According to the convention accepted in community, we add the superscript $^o$ to the name of a relation. Unlike in , we do not allow using the same variable twice in the argument list nor do we permit pattern matching there. Instead, explicit unifications are should be used in the body of the relation.

## 3.2 Example

Having introduced the syntax of the language, let us consider the example program shown in figure **??**. The relation addo relates its arguments x, y, and z in such a way that the property x + y = z holds.

[language=Haskell, basicstyle=] – addo :: [Nat] -¿ [Nat] -¿ [Nat] -¿ Goal addo x y z = (x == Zero  y == z) — (fresh x1, z1 in (x == Succ x1  z == Succ z1  addo x1 y z1));

Figure 1: The addition relation