

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 an Selective Linear Definite clause resolution [?] (SLD resolution) of a query, often following 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 reason. Unfortunately, these two aspects often limit a relation to a single mode and directly contradict the main idea of declarative programming: a program can no longer be written with disregard of the peculiarities of the language.

Recently, there has been a resurgence of the logic programming paradigm with the emergence of new languages, including ¹, ², ³, and others. Additionally, a prominent video games developer Epic Games invested into designing a new functional-logic programming language [?]. This new generation of logic languages combines the paradigms of logic and more mainstream functional programming. and are stand-alone logic-functional programming languages with dedicated compilers that makes it difficult to interoperate with bigger systems typically written in a general-purpose language.

In contrast, 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

¹The website of the programming language <https://mercurylang.org/>

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

³The website of the programming language <http://minikanren.org/>

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.

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.