universität innsbruck

Seminar Report

λProlog: Higher-Order Logic Programming

James Fox
james.fox@uibk.ac.at

15 July 2023

Supervisor: Dr Cezary Kaliszyk

Abstract

λProlog is a logic programming language that extends Prolog to the setting of simply-typed higher-order logic. At the term level, λProlog augments Prolog with λ-abstractions and quantification over higher-order variables. Formulas in λProlog are then based on the intuitionistic theory of higher-order hereditary Harrop formulas, which extend higher-order Horn clauses by allowing universal quantification and logical implication in goal clauses.
This report provides an introduction to λProlog, covering logical background, syntax, and proof search semantics. The language is subsequently placed in a broader programming context with a discussion of real-world applications of λProlog and a comparison to related languages.

Contents

1 Introduction

λProlog is a declarative logic programming language that extends Prolog with higher-order logic and strong typing [30]. In the paradigm of *logic programming*, computation is realised as logical deduction [28]. Logic *programs* are composed of *facts*, which represent true statements, and *rules*, which specify logical implications [42]. For example, a program may consist of the fact "Felix is a cat" and the rule "if something is a cat, then it is a mammal".

After defining a program, one can pose a *query* or *goal*, for which the logic programming language will attempt to construct a proof [42]. In the example above, one may wish to prove the goal "Felix is a mammal". Although no prior knowledge of logic programming is required, this report assumes that the reader is familiar with predicate and first-order logic, lambda calculus, and at least one functional programming language.

The remainder of this report commences with an overview of Prolog in Section 2, providing an introduction to the logic programming principles that underpin λProlog. Section 3 then describes the syntax, semantics, and applications of λProlog. In Section 4, λProlog is positioned within the broader programming landscape through a comparison with related programming languages. Finally, Section 5 concludes the report with an outlook on the future of λProlog.

2 Prolog: A Brief Introduction

Prolog is a declarative logic programming language. Developed by Alain Colmerauer and Phillipe Roussel in 1972 [20], Prolog is based on first-order predicate logic centred around Horn clauses [42]. Its foundation in formal logic makes Prolog particularly well-suited to applications such as formal language parsing and modelling state-transition frameworks [42]. This section provides an overview of the syntax and semantics of Prolog, with particular emphasis on language elements that are relevant to λProlog.

2.1 First-Order Horn Clauses

In Prolog, goal formulas and program clauses belong to a subset of first-order predicate logic known as first-order Horn clauses (FOHC) [42]. In FOHC, goal formulas G and program clauses D are described by the grammar

$$\texttt{G} ::= \top \mid \texttt{A} \mid \texttt{G} \wedge \texttt{G} \mid \texttt{G} \vee \texttt{G} \mid \exists \texttt{x}\ \texttt{G}$$
$$\texttt{D} ::= \texttt{A} \mid \texttt{G} \rightarrow \texttt{D} \mid \texttt{D} \wedge \texttt{D} \mid \forall \texttt{x}\ \texttt{D}$$

where A is a first-order atomic formula and quantification is over first-order term variables. Several logically equivalent characterisations of FOHC exist – this variant is favoured by Miller and Nadathur [28].

FOHC comprises a somewhat restricted subset of first-order predicate logic. Firstly, both implication and universal quantification are forbidden in goal formulas. Secondly, quantification is not permitted over higher-order variables. Finally, existential quantification and disjunction are both prohibited at the top level of program clauses.

At this point, one might question the rationale behind restricting the underlying logic of Prolog to Horn clauses. Although this restriction reduces the expressive power of the language, it also

provides certain computational advantages. In particular, it is possible to construct proofs over Horn clauses in a refutation-complete manner using SLD-resolution [4] which forms the basis of computation in Prolog [14]. Moreover, the semantics of FOHC are complete with respect to first-order logic [29] in a sense that will be formalised when discussing the semantics of λProlog in Subsection 3.3.

2.2 Syntax

In Prolog, the role of values is encoded in their initial letter. Terms that begin with lowercase letters represent constants,[1] while terms starting with uppercase letters represent variables that can be instantiated with specific values. For example, in Prolog syntax, the term `parent(anne, X)` represents a constant `parent` of arity 2 applied to a constant `anne` of arity 0 and a variable `X`. The intended interpretation is that `anne` is the parent of some child represented by the variable `X`.

Prolog formulas are constructed using built-in logical constants. Conjunction is represented by a comma, disjunction is represented by a semicolon, and `:-` denotes the "implied by" relation. Each clause is then terminated by a dot. With this notation, one can now write a Prolog program

```
parent(anne, bob).
parent(bob, cara).
grandparent(X,Y) :- parent(X, P), parent(P, Y).
```

which expresses that `anne` is the parent of `bob`, `bob` is the parent of `cara`, and that `X` is the grandparent of `Y` if `X` is the parent of a person `P`, who in turn is the parent of `Y`. As the variable `P` is free in the final clause, it is implicitly *universally* quantified over this clause.

2.3 Semantics

To enquire about the existence of any grandparents in this logical universe, one can pose a *query* to Prolog, indicated by the prefix `?-` in this report. This produces output

```
?- grandparent(X, Y).
X = anne,
Y = cara
```

which confirms that `anne` is indeed the grandparent of `cara`, as expected. During the proof search, the free variables `X` and `Y` are instantiated via *unification* which is decidable for first-order logic [25]. It is important to note that in the context of queries, free variables are implicitly *existentially* quantified, as opposed to the implicit universal quantification in the context of programs.

In addition to representing logical disjunction, the semicolon operator can be used within Prolog to request multiple solutions to a query. If a user typed a semicolon after the previous query, the complete output would then be

```
?- grandparent(X, Y).
X = anne,
Y = cara ;
false.
```

[1] In this report, "constants" refers to all non-variable values, including functions of arity greater than 0.

where `false` indicates that Prolog was unable to find any additional solutions.

This notion of falsity differs from the role that negation of truth plays in classical logic. If Prolog can not prove a statement to be true, then it is assumed to be false: this is known as *negation as failure*. Prolog provides a built-in predicate `not(p)` which evaluates to true if `p` is not provable.[2] For example, consider program

```
likes(jen, apples).
likes(nate, pears).
```

The goal `not(likes(jen, pears))` will return `true` as it is not provable that `jen` likes `pears`, even though this was not explicitly stated to be false.

2.4 Limitations

This section has provided a brief overview of the logical basis, syntax, and semantics of Prolog. However, the primary focus of this report is λProlog, which was developed as an extension of Prolog [30]. This raises the question: which logical features are absent from Prolog, thus inspiring the development of λProlog?

Firstly, Prolog restricts quantification to first-order variables. Prolog also lacks support for explicit universal or existential quantification: these are only available implicitly. Moreover, `FOHC` forbids implication and universal quantification in goals, which restricts the expressive power of Prolog. Finally, Prolog does not have native support for types. These limitations are all addressed in λProlog.

3 λProlog

λProlog is a declarative logic programming language that extends Prolog to the realm of simply-typed higher-order logic. Introduced by Gopalan Nadathur and Dale Miller in 1988 [30], λProlog builds upon several aspects of Prolog. Firstly, it extends the term language to the setting of Church's simply-typed lambda calculus ($\lambda^{\rightarrow}$) [8]. Secondly, while Prolog is based around Horn clauses, λProlog is based on the more expressive logic of higher-order hereditary Harrop formulas (`HOHH`) [30]. Therefore, λProlog extends Prolog both at the term level and at the formula level [17].

The remainder of this section begins with an introduction to $\lambda^{\rightarrow}$ and `HOHH`, followed by descriptions of the syntax and semantics of λProlog. An integral characteristic of λProlog is its foundation in *intuitionistic* logic, which does not assume the law of the excluded middle (LEM) as an axiom [28]. The reason for adopting intuitionistic logic as the logical basis of λProlog will become clear after introducing the semantics of the language. Finally, the section concludes with an overview of modern implementations, practical applications, and limitations of λProlog.

3.1 Logical Background

This subsection provides the logical background necessary to understand the syntax and semantics of λProlog. After an overview of the relevant aspects of $\lambda^{\rightarrow}$, the Hebrandt universes for `HOHH` are

[2] The notation `\+(p)` is preferred in some Prolog distributions due to the misleading association of `not` with logical negation [21].

introduced, followed by the complete definition of HOHH.

3.1.1 Simply-Typed Lambda Calculus

The term language of λProlog is based on Church's simply-typed lambda calculus ($\lambda^{\rightarrow}$) which extends the lambda calculus to a typed setting [8]. In $\lambda^{\rightarrow}$, every term has an associated *type*, every constant has a type *signature*, and constants can only be applied to terms which match their signature. This is reminiscent of the role of types in typed functional programming languages – refer to Chapter 4 of Miller and Nadathur [28] for formal definitions.

The rules of α-, β-, and η-rewriting can then be defined as follows (using curried notation) [28]:

- Replacing $\lambda x\ s$ by $\lambda y\ s[x/y]$, provided that y is free for x in s and y is not free in s, is called *α-rewriting*. The reflexive, symmetric, and transitive closure of this operation is called *α-conversion*.
- Replacing $(\lambda x\ s)\ t$ by $s[x/t]$, provided that t is free for x in s, is called *β-contraction*. The reflexive transitive closure of the union of α-rewriting and β-contraction is called *β-reduction*.
- Replacing $\lambda x\ (s\ x)$ by s, provided that x is not free in s, is called *η-contraction*. The reflexive transitive closure of η-contraction is called *η-reduction*.

Informally, α-rewriting captures the notion of bound variable renaming, β-contraction expresses function application, and η-contraction describes function abstraction. Two terms are then said to be λ-*convertible* if there exists a sequence of conversion steps that transform one term into the other. Equality in λProlog is defined modulo λ-conversion [28].

A term is in λ-*normal form* if no further β- or η-contractions can be applied. A higher-order *atomic* formula in λ-normal has the shape $(h\ t_1\ \ldots\ t_n)$ where h is a variable or (non-logical) constant and the t_i are terms. If h is a variable, then this atom is said to be *flexible*; otherwise, it is *rigid*. This distinction is necessary when defining the permissible shape of formulas in HOHH.

3.1.2 Herbrandt Universes

Let Σ be a signature that contains the logical constants $\top$, $\wedge$, $\vee$, $\forall$, $\exists$, and $\rightarrow$. Two sets, known as *Herbrand universes*, are used in defining the syntax and semantics of λProlog:

- $\mathcal{H}_1^\Sigma$ is defined as the set of all λ-normal terms over Σ that do not contain $\forall$ or $\rightarrow$.
- $\mathcal{H}_2^\Sigma$ is defined as the set of all λ-normal terms over Σ that do not contain $\rightarrow$.

$\mathcal{H}_2^\Sigma$ therefore directly extends $\mathcal{H}_1^\Sigma$ by allowing $\forall$ in terms.

3.1.3 Higher-Order Hereditary Harrop Formulas

Using the provided definitions, it is now possible to define the class of formulas known as *higher-order hereditary Harrop formulas* (HOHH), which serve as the foundation of λProlog [30, 28]. Goal formulas G and program clauses D for HOHH are defined as

```
G ::= ⊤ | A | G ∧ G | G ∨ G | ∃x G | D → G | ∀x G
D ::= A_r | G → D | D ∧ D | ∀x D
```

where A is a syntactic variable ranging over atomic formulas in $\mathcal{H}_2^\Sigma$, A_r ranges over rigid atoms in $\mathcal{H}_2^\Sigma$, and quantification is over terms in $\mathcal{H}_2^\Sigma$ of an appropriate type [28]. Note that HOHH extends FOHC by lifting atomic formulas and quantification to a higher-order setting, and also by allowing $\rightarrow$ and $\forall$ in goals.

To understand the requirement for rigidity in program clauses, consider the formula $\forall p\ p$. If such a formula were allowed as a program clause, this would result in an *inconsistent* program from which any arbitrary formula is provable [28]. However, in HOHH, the rigidity restriction of A_r prevents this formula from being used as a program clause, even though it is permissible as a goal. Requiring atomic program clauses to be rigid ensures logical consistency [28].

3.2 Syntax

The syntax of λProlog closely resembles that of Prolog, with variables denoted by uppercase letters and constants by lowercase letters. The logical constants "`,`", "`;`", and "`:-`" have the same meaning as in Prolog, and the concept of negation as failure is represented by the `not` predicate.

However, unlike Prolog, λProlog uses curried notation to represent function application [17]. For example, the Prolog expression `f(X, g(Y, Z))` is written as `f X (g Y Z)` in λProlog. Moreover, λProlog introduces several additional keywords to represent types and the logic of $\lambda^{\rightarrow}$, which are described in this subsection.

3.2.1 Types

λProlog is *strongly typed*, meaning that type checking can be performed statically once all constants and variables in an expression have been specified, thus improving runtime stability of programs [28]. However, as λProlog supports type polymorphism, additional type checking must be performed at runtime due to dynamic type inference [28].

λProlog provides pervasive (built-in) sorts for integers, real numbers, strings, and input/output streams [28]. An additional pervasive sort *o* is used to represent logical formulas. Elements with sort *o*, written as `o` in λProlog syntax, can be interpreted as part of the meta-logic. For example, logical predicates have target type `o` and the constant "`,`" has type `o -> o -> o`.

In λProlog, new type constructors for primitive types are defined using the keyword `kind`. Each kind has an associated *kind expression* of the shape

⟨`kind exp`⟩ `::= type | type ->` ⟨`kind exp`⟩

which specifies the order of the associated constructor [28]. For example, one can define a type constructor `pair` which takes two type expressions as arguments and returns a new type as

```
kind  pair  type -> type -> type.
```

where `->` associates to the left. The type constructor `pair` can now be used in type expressions. A *value constructors* declaration begins with the keyword `type` and is followed by a type expression in curried form connected by the right-associative operator `->`. For example,

```
type  pair_cons  A -> B -> pair A B.
```

defines a value constructor `pair_cons` which takes two arguments of types `A` and `B` and returns a value of type `pair A B`. The capital letters `A` and `B` are *type variables* which can be instantiated with an arbitrary type, allowing for polymorphism in type declarations. It is also possible to define multiple values with the same signature by separating the names by a comma. For a full description of type syntax in λProlog, refer to Chapter 1 of Miller and Nadathur [28].

3.2.2 Syntactical Extensions of Prolog

λProlog introduces several additional logical constants that are not present in Prolog, which are summarised in Table 1. The constants `pi` and `sigma` represent universal and existential quantification respectively, borrowing from the notation used by Church [8]. Note that these are polymorphic, allowing quantification over variables of an arbitrary type: quantification over a specific type τ will be written as $\forall_\tau$ and $\exists_\tau$ in the remainder of this report when the type can not be inferred from the context.

Constant	Type	Example	Meaning
`pi`	`(A -> o) -> o`	`pi X\ expr`	$\forall x$ `expr`
`sigma`	`(A -> o) -> o`	`sigma X\ expr`	$\exists x$ `expr`
`\`	`A -> B -> (A -> B)`	`X\ expr`	λx `expr`
`=>`	`o -> o -> o`	`expr1 => expr2`	`expr1` $\rightarrow$ `expr2`
`&`	`o -> o -> o`	`expr1 & expr2`	`expr1` $\wedge$ `expr2`

Table 1: Logical constants that are specific to λProlog and their corresponding meaning in logic notation. Bound variables may start with uppercase or lowercase letters [28].

The constant "=>" has the same logical meaning as "`:-`" read from right to left, and the constants "`&`" and "`,`" have identical meanings. Although this introduces some redundancy into the meta-logic of λProlog, the symbols `:-` and `&` are intended to represent implication and conjunction in *program* clauses, while `=>` and `,` are used at the top level of *goal* clauses [28]. This distinction is made to improve the readability of programs.

To illustrate this difference, consider implementing a predicate for list reversal. In λProlog, lists are constructed using the built-in constants `nil`, representing the empty list, and `::`, which prepends a new value to an existing list [28]. One can now define a binary predicate `reverse` which uses an auxiliary predicate `rev` to express that a list `K` is the reversal of list `L` as follows [28]:

```
type  reverse, rev  list A -> list A -> o.
reverse L K :-
   (rev nil K & (pi X\ pi L\ pi K\ rev (X::L) K :- rev L (X::K)))
      => rev L nil.
```

The reader is invited to confirm that this program is indeed in the language of `HOHH` and follows the syntactic conventions described above.

A further distinguishing feature of λProlog is that equality is considered modulo λ-conversion. For example, for any constant `f` of arity ≥ 2, the query

```
?- (X\ Y\ f X Y) = (Z\ f Z).
```

returns `true` as the terms $(\lambda x \lambda y\ (f\ x\ y))$ and $(\lambda z\ (f\ z))$ are λ-convertible.

3.3 Semantics

Thus far, this report has presented an overview of the logical foundations of λProlog without addressing how proofs are found. This section covers both theoretical and practical aspects of proof search and provides insights into how the challenges posed by higher-order unification and universal quantification are addressed.

The current state of an idealised logic programming interpreter can be described by its program signature, program clauses, and goal formula. Let Σ denote a signature, $\mathcal{P}$ denote a set of program clauses over Σ, and $\mathcal{G}$ denote a goal formula over Σ. The current proof state can be described as a *sequent* or triple $\Sigma; \mathcal{P} \longrightarrow \mathcal{G}$ encoding that one wishes to prove goal $\mathcal{G}$ from a given program [28]. For example, consider extending the program in Subsection 2.2 to a typed setting by introducing a single type `p` to represent people. This program would have initial proof state:

Σ = {`anne:p, bob:p, cara:p, parent:p -> p -> o, grandparent:p -> p -> o`}
$\mathcal{P}$ = {`parent(anne, bob), parent(bob, cara),`
`∀X ∀Y ∀P parent(X, P) ∧ parent(P, Y) → grandparent(X,Y)`}
$\mathcal{G}$ = {`grandparent(X,Y)`}

The proof search carried out by λProlog can be formalised as a *sequent calculus* which iteratively updates this proof state using inference rules which are described below.

3.3.1 Goal-Directed Search Semantics

Given a goal formula, λProlog seeks to derive a proof of this goal in a bottom-up manner. The proof search can be divided into two phases. Firstly, *right-introduction steps* (summarised in Table 2) are applied until an atomic formula is reached [28]. These inference rules capture the intended semantic meaning of each logical constant by interpreting each logical connective in a goal formula as a search instruction [29].

If the proof search reaches an atomic goal, λProlog applies *backchaining* steps to advance the proof search (see Table 3). Backchaining uses the logical structure of program clauses to further decompose the proof obligations. Unlike right-introduction steps that depend solely on the structure of the goal formula, backchaining incorporates the logical assumptions encoded in program clauses into the proof search.

A proof constructed using the rules in Table 2 and Table 3 is called a *uniform proof* in terminology introduced by Miller et al. [29, 28]. It is desirable that the semantic notion of uniform provability aligns with the concept of provability in the underlying logic: this notion of soundness and completeness is formalised via the proof-theoretic concept of an *abstract logic programming language*. This is an inference system in which every uniform proof can be represented as a proof in the underlying logic and vice versa [29]. Notably, `FOHC` and `HOHH` are abstract programming languages [29, 28] and therefore semantically complete in this sense.

At this point, it is possible to see why intuitionistic logic is in fact essential for the soundness of `HOHH`. Consider constructing a uniform proof of the formula $p \vee (p \rightarrow q)$ from the empty program. This is provable as a goal if and only if either p is provable from the empty program or q is provable from p [28]. Since neither of these is provable, this formula does not have a uniform proof. However, if one assumes the LEM, then $p \vee (p \rightarrow q)$ is equivalent to $p \rightarrow (p \vee q)$ which *does* have a uniform proof via the `AUGMENT` and `OR` rules followed by backchaining on p.

Name	Inference Rule	Meaning
TRUE	$\frac{}{\Sigma;\mathcal{P}\longrightarrow\top}$	The goal $\top$ is immediately provable
AND	$\frac{\Sigma;\mathcal{P}\longrightarrow B_1 \quad \Sigma;\mathcal{P}\longrightarrow B_2}{\Sigma;\mathcal{P}\longrightarrow B_1\wedge B_2}$	Goal $B_1 \wedge B_2$ is provable if both B_1 and B_2 are provable
OR	$\frac{\Sigma;\mathcal{P}\longrightarrow B_1}{\Sigma;\mathcal{P}\longrightarrow B_1\vee B_2} \quad \frac{\Sigma;\mathcal{P}\longrightarrow B_2}{\Sigma;\mathcal{P}\longrightarrow B_1\vee B_2}$	Goal $B_1 \vee B_2$ is provable if B_1 or B_2 is provable
AUGMENT	$\frac{\Sigma;\mathcal{P},B_1\longrightarrow B_2}{\Sigma;\mathcal{P}\longrightarrow(B_1\rightarrow B_2)}$	Goal $(B_1 \rightarrow B_2)$ is provable if B_2 is provable when B_1 is appended to the program clauses
INSTANCE	$\frac{\Sigma;\mathcal{P}\longrightarrow B[x/t] \quad \Sigma;\emptyset \Vdash t{:}\tau}{\Sigma;\mathcal{P}\longrightarrow\exists_\tau x\ B}$	Goal $\exists_\tau x\ B$ is provable if $B[x/t]$ is provable for some λ-term $t \in \mathcal{H}_1^\Sigma$ of type τ
GENERIC	$\frac{c{:}\tau,\Sigma;\mathcal{P}\longrightarrow B[x/c]}{\Sigma;\mathcal{P}\longrightarrow\forall_\tau x\ B}$	Goal $\forall_\tau x\ B$ is provable if $B[x/c]$ is provable for some fresh constant c of type τ

Table 2: Right-introduction rules [28]. During the proof search, λProlog matches λ-normal forms with the lower sequent to advance the proof search in a bottom-up manner. A comma is used to represent set union and it is assumed that substitutions avoid variable capture.

Name	Inference Rule	Meaning
Decide	$\frac{\Sigma;\mathcal{P}\xrightarrow{D}A}{\Sigma;\mathcal{P}\longrightarrow A}$	Select a formula $D \in \mathcal{P}$ to backchain on
Initial	$\frac{}{\Sigma;\mathcal{P}\xrightarrow{A'}A}$	If $A' \in \mathcal{P}$ and A and A' are α-convertible formulas, then A is derivable by backchaining on A'
→L	$\frac{\Sigma;\mathcal{P}\xrightarrow{D}A \quad \Sigma;\mathcal{P}\longrightarrow G}{\Sigma;\mathcal{P}\xrightarrow{G\rightarrow D}A}$	If A is derivable when backchaining on $D \in \mathcal{P}$ and G is provable, then A is derivable by backchaining on $G \rightarrow D$
∧L	$\frac{\Sigma;\mathcal{P}\xrightarrow{D_1}A}{\Sigma;\mathcal{P}\xrightarrow{D_1\wedge D_2}A} \quad \frac{\Sigma;\mathcal{P}\xrightarrow{D_2}A}{\Sigma;\mathcal{P}\xrightarrow{D_1\wedge D_2}A}$	If A is derivable when backchaining on D_1 or D_2, then it is derivable by backchaining on $D_1 \wedge D_2$
∀L	$\frac{\Sigma;\mathcal{P}\xrightarrow{D[x/t]}A \quad \Sigma;\emptyset \Vdash t{:}\tau}{\Sigma;\mathcal{P}\xrightarrow{\forall_\tau x\ D}A}$	If A is derivable for any λ-term t of type τ in $\mathcal{H}_1^\Sigma$, then A is derivable by backchaining on $\forall_\tau x\ D$

Table 3: Backchaining rules [28]. If the proof search reaches an atomic goal A, then λProlog selects a specific clause $D \in \mathcal{P}$ to backchain on, denoted by superscript $\xrightarrow{D}$.

It follows that the semantics defined above are unsound for `HOHH` if the LEM is assumed. In the first-order setting, `FOHC` is an abstract logic programming language for both classical and intuitionistic logic [29, 28].

3.3.2 Proof Search in Practice

The proof search described above is nondeterministic, as multiple inference steps might be applicable to a single sequent. λProlog addresses this by utilising a deterministic, depth-first proof search strategy in which backchaining is carried out on the most recently added clause in $\mathcal{P}$ [28]. As a result, the clauses in $\mathcal{P}$ are treated like an ordered list rather than a set.

The restriction to a depth-first proof search might result in non-termination in some cases. Consider, for example, the program consisting of a single clause

```
loop :- loop.
```

The query `p, loop.` will return `false` almost immediately since `p` is an atom with no defining clauses. However, the logically equivalent query `loop, p.` will not terminate in a depth-first search [28] due to recursive applications of the →L rule.

If the proof search reaches a dead end, λProlog *backtracks* to an earlier proof state (not to be confused with backchaining) and continues the proof search from there [28]. A user can prevent backtracking beyond a certain point using the cut operator "`!`", for example, to ensure termination or to enforce specific search semantics [28]. Other than this, λProlog offers no direct means of influencing search behaviour. This determinism may be seen as advantageous due to its transparency but is also inflexible if users desire more fine-grained control over proof search.

3.3.3 Unification

λProlog uses *unification* over higher-order terms to instantiate variables during the proof search process. First-order unification, as used in Prolog, is decidable and guarantees the existence of a most general unifier (mgu) if a unifier exists. However, higher-order unification satisfies neither of these properties [25], introducing undecidability at the core of λProlog.

In 1991, Miller identified a computationally "well-behaved" fragment of the term language, known as L_λ [25]. Unification on this fragment, referred to as *pattern unification*, is decidable and the existence of an mgu is assured if a unifier exists [25]. Although the theory of λProlog does not restrict the shape of unification supported, some implementations do choose to restrict unification to L_λ to ensure decidability (see Subsection 3.4).

It is important to note that unification in λProlog is *intensional* rather than extensional. That is, unification is carried out modulo λ-conversion, but does not consider the extensional definitions of constants. To illustrate this, consider the following definition of a predicate for term equality:

```
type eq_pred (A -> o) -> (A -> o) -> o.
eq_pred T T.
```

The query

```
?- eq_pred (X\ p X, q X) (X\ q X, p X).
```

will fail no matter how `p` and `q` are defined [28]. Even though the arguments of `eq_pred` in this query have the same extensional logical meaning, these arguments can not be unified intensionally.

3.3.4 Universal Quantification

λProlog incorporates a notion of variable scope to allow for clauses containing nested bound variables without variable capture [28]. However, the implementation of universal quantification presents further challenges. In λProlog, the interpretation of ∀ is *intensional* [30]. Rather than verifying that a universal goal holds for each element in a given domain, λProlog instantiates the universally quantified variable with an *eigenvariable* representing an arbitrary value and attempts to derive a proof of the resulting formula. This is logically stronger than the extensional interpretation but requires care to ensure sound implementation of the `GENERIC` rule (see Table 2).

For example, consider the goal `sigma X\ pi Y\ f X Y` corresponding to the logical expression $\exists x \forall y\ f(x, y)$. To solve this goal, the λProlog interpreter will first instantiate `X` with a fresh variable of the same type, say `X'`, due to the `INSTANCE` rule. This results in a new goal `pi Y\ f X' Y`. At this point, the interpreter will choose some fresh eigenvariable due to the `GENERIC` rule, say `Y'`, resulting in the updated goal `f X' Y'`. This variable `Y'` is never instantiated during computation [28]. However, it must additionally be ensured that `X'` is never instantiated to a term containing `Y'` in the remaining proof search [30].

Nadathur and Miller propose two solutions to avoid the unintended capture of universally quantified eigenvariables [27]. The first approach is to modify the unification procedure to have an awareness of the scope of each variable. The second solution is to modify the implementation of both the `GENERIC` *and* `INSTANCE` rules to keep track of the variables which are universally quantified within the current scope [30]. The implementation details for universal quantification are not prescribed by the language itself but rather are left to the discretion of individual systems.

3.4 Implementations

The report thus far has been concerned with the theoretical aspects of λProlog rather than concrete implementations. However, various distributions of λProlog are available, which are summarised in Table 4. Early implementations of λProlog did not impose restrictions on higher-order unification and instead utilised a semi-decidable procedure [38] described by Huet [18]. However, recent implementations are typically restricted to pattern unification, which is decidable but reduces the expressive power of the language.

Miller and Nadathur claim that *"The overwhelming majority of 'typical' λProlog program clauses are within the L_λ fragment of the `HOHH` language"* [28]. Although this reflects their empirical experiences, it may be seen as somewhat unsatisfying from a logical standpoint. In particular, the restriction to L_λ may lead to an inability to find proofs despite the existence of an appropriate unifier, especially for users who lack a deep familiarity with pattern unification.

3.5 Applications

λProlog has predominantly been used in theoretical research concerning formal specifications [40, 1, 22, 41] and theorem proving. Early work on theorem proving in λProlog was carried out by

[3] See Qi [38], www.lix.polytechnique.fr/~dale/lProlog/faq/implementations.html, and github.com/teyjus/teyjus/wiki/AboutTeyjus for more information about early development of λProlog. All websites accessed 11 July, 2023.

Name	Initial Release	Language	Unification Fragment	Execution Model
LP V2.7 [27]	1988	C-Prolog	-	Interpreted
eLP [11]	1989	Common Lisp	-	Interpreted
MALI [5]	~1993	C	-	Compiled
Terzo [43]	~1997	Standard ML of NJ	?	Interpreted
Teyjus V1 [31]	1999	C	-	Compiled
Teyjus V2 [38]	2009	OCaml and C	L_λ	Compiled
ELPI [10]	2015	OCaml	L_λ	Interpreted

Table 4: Comparison of λProlog implementations.[3] ELPI is technically a dialect as it introduces additional features [9, 15], but does implement unification on L_λ despite being optimised for a subset thereof [10].

Felty, a PhD student of Dale Miller [13, 12]. Appel then created a simple theorem prover in 1999 [2], later collaborating with Felty in 2004 [3].

Appel and Felty implemented their theorem prover by defining a *meta-language* for reasoning about proofs within the language of λProlog [3]. In this meta-language, types have kind `tp` and terms have kind `tm`. Formulas are then represented by a constant `form` of type `tp` and implication is written as an infix constant `imp` that takes two terms `s` and `t` as input arguments and returns a term `s imp t`. This is written in λProlog syntax as

```
kind  tp      type.
kind  tm      type.
type  form    tp.
type  imp     tm -> tm -> tm. infixr imp 7.
```

where the keyword `infixr` defines the relative binding strength of `imp` [3]. Note in particular, that `imp` refers to a value constructor in the meta-language and does not refer to the λProlog constant =>.

Appel noted that λProlog is especially well-suited for elegant and concise representations of proof systems, as the native typing of λProlog can be used to ensure well-typedness in the logical meta-language [2]. For example, the rule that `A imp B` is a well-typed formula if both `A` and `B` are well-typed formulas in the meta-language can be expressed concisely as

```
type  hastype   tm -> tp -> o.
hastype (A imp B) form :- hastype A form, hastype B form.
```

Since the invention of ELPI in 2015 [10], there has been a small resurgence of interest in λProlog for theorem proving. The creators of ELPI have implemented a simple interactive theorem prover for higher-order logic [9] and implemented the core components of a more general theorem prover by extending λProlog with programming constraints [15]. Kohlhase et al. also utilised ELPI in creating an early-stage theorem prover that simulates the Edinburgh Logical Framework (LF) [19]. This framework was then used to generate a tableau prover and model generator as a proof of concept in 2020.

3.6 Limitations

Although λProlog encodes a richer logic than Prolog, there are still inherent limitations on what can be expressed and proven within the language. Firstly, λProlog relies on higher-order unification in its proof search, which is undecidable in general. To address this, several implementations of λProlog deliberately restrict the fragment of unification supported, choosing decidability at the cost of expressivity. Secondly, the lack of control over the proof search may be viewed either as a limitation of the language or merely as a feature of declarative programming. Although backtracking can be prevented via the cut operator, users who require more control over proof search may find interactive theorem provers (ITPs) to be a more suitable solution.

The logical basis of λProlog in $\lambda^{\rightarrow}$ may also be seen as a limitation. More expressive logical systems exist which allow types to depend on values, known as *dependent typing*. Examples include the Edinburgh Logical Framework (LF) which is implemented in the Elf family of languages [34, 35, 36, 31] and the Calculus of Inductive Constructions (CIC) which forms the basis of the Coq theorem prover [15].

Looking beyond the language itself, the relatively small community and limited documentation surrounding λProlog present additional challenges. The creators of λProlog acknowledge that programming with higher-order unification initially has a steep learning curve [30, 28]. While Miller and Nadathur have written a book on the theory underlying λProlog [28] and code examples are available with most λProlog distributions, there is a dearth of introductory tutorials compared to more widely used languages. This lack of accessible resources is likely a factor in why λProlog remains largely confined to specialised research applications.

4 Higher-Order Programming: Context and Comparison

In this section, λProlog is brought into context within the wider programming landscape. The section begins with a high-level comparison of λProlog and the functional programming language Haskell, serving as a case study illustrating the paradigmatic differences between λProlog and functional programming. Following this, an overview is provided of languages that have been either inspired by or share similarities with λProlog.

4.1 Comparison: λProlog Versus Haskell

λProlog exhibits several similarities with Haskell: they both support the use of functions as arguments (higher-order programming), lambda abstractions, and feature strong typing with type polymorphism. Moreover, both languages cite the functional language ML as an influence [30, 24].

The most apparent differences between λProlog and Haskell stem from their intended purposes. The former is a specialised logic programming language with native support for logical inference and higher-order unification, but almost no support for complex data structures and data manipulation. Haskell, on the other hand, is a *general-purpose* programming language [24] that offers a broader range of programming capabilities but does not utilise logical inference as computation. In particular, computation in λProlog corresponds to a proof search that works *backwards* from a goal, using *unification* and backtracking to advance this search [28]. Evaluation

in Haskell proceeds forwards, using *pattern matching* to lazily evaluate a given expression by applying function definitions from left to right [24].

A significant difference between the two languages lies in their distinct concepts of equality. In λProlog, equality is *intensional* modulo λ-conversion, which is decidable [28]. For example, the goal `(X\ Y\ f X Y) = (Z\ f Z)` returns `true` for any constant `f` of arity ≥ 2 as these terms are λ-convertible. λProlog thus enables direct comparison of higher-order functions based on their term structure, even over infinite domains. By contrast, the intended semantics of equality in Haskell are *extensional*, which is undecidable in the general case [28]. The ability of λProlog to directly compare higher-order functional expressions therefore provides a more expressive means of reasoning about functions [28].

Finally, there is a clear discrepancy in the popularity of the two languages. As of July 2023, Haskell ranks as the 26th most popular programming language according to Google searches for tutorials[4] and holds the 36th position in the TIOBE index,[5] which measures the number of search engine results. In contrast, λProlog is not ranked in either popularity index or by Google Trends,[6] indicating that it is not widely searched for.

4.2 Beyond λProlog

Although $\lambda^{\rightarrow}$ can be simulated in other languages, λProlog stands out as one of the few languages that implement $\lambda^{\rightarrow}$ directly. There are, however, a number of related programming languages, which are described below.

4.2.1 Prolog Extensions

In addition to λProlog, other languages have extended the logic of Prolog to a higher-order setting. One such language is HiLog [6], which retains the first-order semantics of Prolog but expands the *syntax* to allow arbitrary terms (including variables) in the place of function calls. Although this provides greater syntactic flexibility, HiLog programs can be translated back into first-order predicate logic, indicating that this extension does not provide additional semantic strength [6].

Qu-Prolog (Quantifier Prolog) extends Prolog by introducing explicit quantification and variable binding [7]. It supports unification involving α-renaming and substitutions [32] and also allows optional type hints for bound variables in its modern form [39]. However, Qu-Prolog is not strongly typed and does not support unification relative to the full set of λ-conversion rules [31], making it computationally weaker than λProlog.

4.2.2 Languages Inspired by λProlog

Perhaps the most well-known language inspired by λProlog is Elf [34]. Pfenning proposed Elf in 1989 as a logic programming implementation of LF [34], having been inspired by the correspondence between λProlog and $\lambda^{\rightarrow}$ [34, 35]. Elf is primarily intended for use as a meta-language to reason about deductive systems such as programming languages [36].

[4] pypl.github.io/PYPL.html accessed 13 July, 2023.

[5] www.tiobe.com/tiobe-index/ accessed 13 July, 2023.

[6] trends.google.com/trends/explore?q=lambda%20prolog accessed 13 July, 2023.

Makam is a more recent language inspired by λProlog that has been under development since 2012.[7] It is implemented from scratch in OCaml and was designed as a refinement of λProlog specifically for prototyping new programming languages via formal specifications.

Efforts have also been made to extend λProlog to a logical system known as linear logic. Hodas, a PhD student of Miller, developed an extension of λProlog to linear logic called lolli in 1992 [16]. Miller later extended both λProlog and lolli to a language called Forum, which introduces the possibility of multiple conclusions in a classical (rather than intuitionistic) setting [26]. However, it appears that neither language has been actively maintained since the 1990s.[8]

4.2.3 Interactive Theorem Provers

Many interactive theorem provers (ITPs) support logical deduction in a higher-order setting. Isabelle is a proof assistant that implements several logical frameworks, including Higher-Order Logic (Isabelle/HOL). Isabelle/HOL, like λProlog, is based on $\lambda^{\rightarrow}$ [33]. However, while λProlog relies on a single deterministic proof search strategy, Isabelle offers a wide range of tactics and tacticals, granting users more control in guiding the proof search.[9]

Furthermore, several ITPs implement richer logics than $\lambda^{\rightarrow}$. For example, Twelf is an extension of Elf which implements LF [37], while Coq and Lean are based on the calculus of inductive constructions [15]. These systems thus support a more expressive logic than λProlog.

5 Conclusions

In summary, λProlog extends Prolog to the realm of strongly-typed higher-order intuitionistic logic. This extension is achieved by expanding the term language to $\lambda^{\rightarrow}$ and the formula language to `HOHH`. As a result, λProlog is capable of reasoning about a significantly more expressive logic than Prolog, while remaining firmly rooted in the declarative logic programming paradigm.

In theory, the semantics of λProlog are sound and complete, meaning that a goal is provable in `HOHH` if and only if it is derivable via a uniform proof. However, in practice, the deterministic depth-first proof search strategy employed by λProlog might not terminate. A more serious restriction is posed by the undecidability of higher-order unification. To address this, recent implementations of λProlog have restricted unification to L_{λ}, thus ensuring decidability at the cost of reduced expressivity. Future implementations of λProlog could further extend the supported unification fragment (see e.g. Libal & Miller [23]), but this undecidability will always remain an inherent limitation of λProlog.

Since its invention in 1988, λProlog has primarily been used in niche research applications. When compared with more widely used languages, one may well question the relevance, utility, and purpose of λProlog in a modern context. However, what sets λProlog apart is its distinctive position as a purely declarative logic programming implementation of $\lambda^{\rightarrow}$. This provides value

[7] github.com/astampoulis/makam accessed on 11 July, 2023.

[8] The language homepages are available at www.lix.polytechnique.fr/~dale/lolli/ and www.lix.polytechnique.fr/Labo/Dale.Miller/forum/ (accessed on 11 July, 2023).

[9] See also www.lix.polytechnique.fr/~dale/lProlog/faq/related.html accessed on 11 July, 2023.

in settings where reasoning involving higher-order logic is required, but more complex logical frameworks would introduce unnecessary complexity.

Since the introduction of ELPI in 2015 [10], there has been a growing number of research papers utilising λProlog for theorem proving [9, 15, 19]. Perhaps this, in turn, will lead to increased interest and further exploration of λProlog in the coming years; or perhaps λProlog will remain relatively unknown. Only time will reveal the lasting impact and ongoing relevance of λProlog in the field of higher-order logic programming.

References

[1] J. H. Andrews. Executing formal specifications by translation to higher order logic programming. In G. Goos, J. Hartmanis, J. Van Leeuwen, E. L. Gunter, and A. Felty, editors, *Theorem Proving in Higher Order Logics*, volume 1275, pages 17–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. Series Title: Lecture Notes in Computer Science.

[2] A. W. Appel. Lightweight Lemmas in λ Prolog. The MIT Press, 1999.

[3] A. W. Appel and A. P. Felty. Polymorphic lemmas and definitions in Lambda Prolog and Twelf. 2004. Publisher: arXiv Version Number: 1.

[4] K. R. Apt and M. H. Van Emden. Contributions to the Theory of Logic Programming. *Journal of the ACM*, 29(3):841–862, July 1982.

[5] P. Brisset and O. Ridoux. The Compilation of lambdaProlog and its execution with MALI. Jan. 1993.

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

[7] A. S. Cheng, P. J. Robinson, and J. Staples. Higher level meta programming in Qu-Prolog 3.0. Technical Report 90-01, Key Centre for Software Technology, Department of Computer Science, University of Queensland, Oct. 1990.

[8] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[9] C. Dunchev, C. S. Coen, and E. Tassi. Implementing HOL in an Higher Order Logic Programming Language. In *Proceedings of the Eleventh Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '16, New York, NY, USA, 2016. Association for Computing Machinery. event-place: Porto, Portugal.

[10] C. Dunchev, F. Guidi, C. Sacerdoti Coen, and E. Tassi. ELPI: Fast, Embeddable, λProlog Interpreter. In M. Davis, A. Fehnker, A. McIver, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 9450, pages 460–468. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. Series Title: Lecture Notes in Computer Science.

[11] C. Elliott, F. Pfenning, and D. Miller. eLP: Ergo Lambda Prolog v0.15, Feb. 1990.

[12] A. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, 1993.

[13] A. Felty and D. Miller. Specifying theorem provers in a higher-order logic programming language. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310, pages 61–80. Springer-Verlag, Berlin/Heidelberg, 1988. Series Title: Lecture Notes in Computer Science.

[14] J. H. Gallier. SLD-Resolution And Logic Programming (PROLOG). In *Logic for computer science: foundations of automatic theorem proving*, Dover books on computer science, pages 410–447. Dover Publications, Inc, Mineola, New York, second edition, 2015.

[15] F. Guidi, C. Sacerdoti Coen, and E. Tassi. Implementing type theory in higher order constraint logic programming. *Mathematical Structures in Computer Science*, 29(8):1125–1150, Sept. 2019.

[16] J. S. Hodas. Lolli: An extension of $$\lambda $$prolog with linear logic context management. *Workshop on the lambda-Prolog Programming Language*, May 1992.

[17] J. S. Hodas. *Logic programming in intuitionistic linear logic: theory, design, and implementation*. PhD thesis, University of Pennsylvania, University of Pennsylvania, 1994.

[18] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, June 1975.

[19] M. Kohlhase, F. Rabe, C. Sacerdoti Coen, and J. F. Schaefer. Logic-Independent Proof Search in Logical Frameworks: (Short Paper). In N. Peltier and V. Sofronie-Stokkermans, editors, *Automated Reasoning*, volume 12166, pages 395–401. Springer International Publishing, Cham, 2020. Series Title: Lecture Notes in Computer Science.

[20] R. A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, Jan. 1988.

[21] T. Lager. SWI Prolog Reference Manual. Reference Manual v9.0.4, Jan. 2023.

[22] C. C. Liang. Compiler Construction in Higher Order Logic Programming. In G. Goos, J. Hartmanis, J. Van Leeuwen, S. Krishnamurthi, and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages*, volume 2257, pages 47–63. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. Series Title: Lecture Notes in Computer Science.

[23] T. Libal and D. Miller. Functions-as-constructors higher-order unification: extended pattern unification. *Annals of Mathematics and Artificial Intelligence*, 90(5):455–479, May 2022.

[24] S. Marlow. *Haskell 2010 language report*. Cambridge, Apr. 2010.

[25] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In P. Schroeder-Heister, editor, *Extensions of Logic Programming*, volume 475, pages 253–281. Springer-Verlag, Berlin/Heidelberg, 1991. Series Title: Lecture Notes in Computer Science.

[26] D. Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, Sept. 1996.

[27] D. Miller and G. Nadathur. LP v2.7, July 1988.

[28] D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 1 edition, June 2012.

[29] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1-2):125–157, Mar. 1991.

[30] G. Nadathur and D. Miller. An Overview of Lambda-Prolog. 1988.

[31] G. Nadathur and D. J. Mitchell. System Description: Teyjus—A Compiler and Abstract Machine Based Implementation of λProlog. In G. Goos, J. Hartmanis, and J. Van Leeuwen, editors, *Automated Deduction — CADE-16*, volume 1632, pages 287–291. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. Series Title: Lecture Notes in Computer Science.

[32] P. Nickolas and P. J. Robinson. The Qu-Prolog unification algorithm: formalisation and correctness. *Theoretical Computer Science*, 169(1):81–112, Nov. 1996.

[33] L. C. Paulson, T. Nipkow, and M. Wenzel. Isabelle's Object-Logics. Technical report, University of Cambridge, 1997.

[34] F. Pfenning. Elf: a language for logic definition and verified metaprogramming. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, CA, USA, 1989. IEEE Comput. Soc. Press.

[35] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–182. Cambridge University Press, 1 edition, Sept. 1991.

[36] F. Pfenning. Elf: A meta-language for deductive systems: System description. In J. G. Carbonell, J. Siekmann, G. Goos, J. Hartmanis, and A. Bundy, editors, *Automated Deduction — CADE-12*, volume 814, pages 811–815. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994. Series Title: Lecture Notes in Computer Science.

[37] F. Pfenning and C. Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In G. Goos, J. Hartmanis, and J. Van Leeuwen, editors, *Automated Deduction — CADE-16*, volume 1632, pages 202–206. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. Series Title: Lecture Notes in Computer Science.

[38] X. Qi. An Implementation of the Language Lambda Prolog Organized around Higher-Order Pattern Unification. 2009. Publisher: arXiv Version Number: 1.

[39] P. J. Robinson. *Qu-Prolog 10.7 Reference Manual*. May 2022.

[40] E. J. Rollins and J. M. Wing. Specifications as search keys for software libraries: A case study using lambda prolog. Technical Report CMU-CS-90-159, Paris, Sept. 1990.

[41] M. Southern and G. Nadathur. A Lambda Prolog Based Animation of Twelf Specifications. 2014. Publisher: arXiv Version Number: 1.

[42] L. Sterling and E. Y. Shapiro. *The Art of Prolog (2nd Ed.): Advanced Programming Techniques*. MIT press, Cambridge, MA, USA, 1994.

[43] P. Wickline, D. Miller, C. Eliott, and F. Pfenning. Terzo v1.2b, Jan. 1999.