universität innsbruck

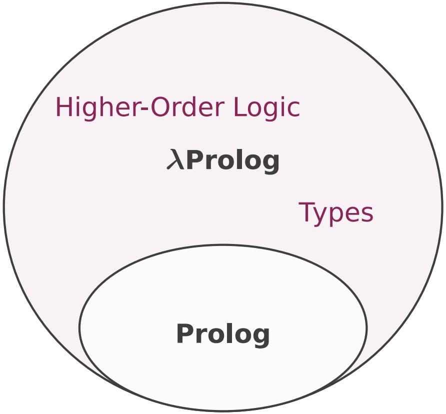
λProlog: Higher-Order Logic Programming

Specialisation Seminar

Jamie Fox

Introduction

- Prolog is a **logic programming** language
- λProlog extends Prolog



Introduction

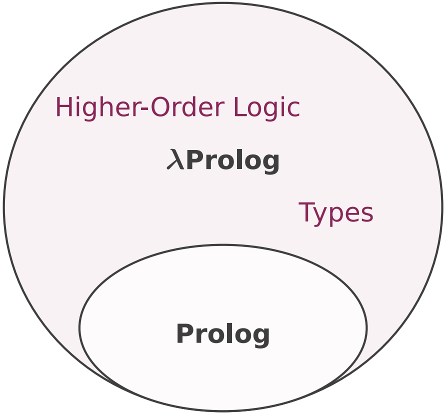
- Prolog is a **logic programming** language
- λProlog extends Prolog
- This talk covers...

  - Logic Programming

  - Prolog

  - λProlog

  - Applications



Predicate Logic In 60 Seconds

Terms

Entity	Examples
Variables	`X, Y, Person`
Constants	`p, cat, anne`
Predicates	`mammal(cat), parent(anne,Person), f(X,g(Y))`

Predicate Logic In 60 Seconds

Terms

Entity	Examples
Variables	`X, Y, Person`
Constants	`p, cat, anne`
Predicates	`mammal(cat), parent(anne,Person), f(X,g(Y))`

Logic Operations

$p \wedge q$	p and q
$p \vee q$	p or q
$p \leftarrow q$	p if q (“implied by”)
$\exists X\ p$	There exists an X such that p
$\forall X\ p$	For all X, p

Logic Programming

- Programming paradigm based on formal logic
- Programs consist of facts and rules

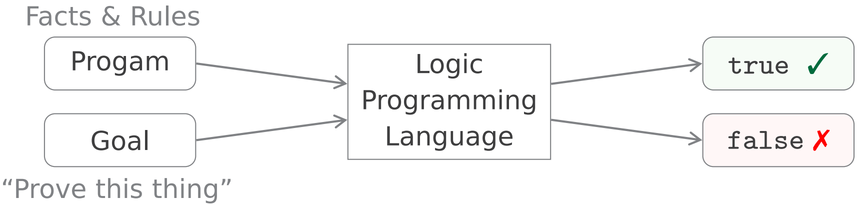
Example

Facts: $\texttt{p(X)}, \texttt{parent(anne, bob)}, \texttt{even(2)}$

Rules: $\texttt{sibling(X, Y)} \Longleftarrow \texttt{parent(P, X)} \land \texttt{parent(P, Y)}$

Logic Programming

- Programming paradigm based on formal logic
- Programs consist of facts and rules
- Typically have a logic program and try to prove a **goal**



Motivation

Why program using formal logic?

1. **Proof Formalisation**
   - Humans can make mistakes in proofs, computers (hopefully!) reason correctly
2. **Program Verification**
   - Testing can not *prove* that code works as intended – we need logic
3. **Modelling Algorithms**
   - Natural language processing, compiler design, constraint solving, ...

1. Introduction
2. Logic Programming
3. **Prolog**

   Background

   Horn Clauses

   Examples
4. λProlog
5. Applications

Prolog – Background

- Developed in France in $\sim$1972
- Prolog is a **declarative** language
    - Tell Prolog what to prove but not *how* to prove it

Prolog – Background

- Developed in France in $\sim$1972
- Prolog is a **declarative** language
  - Tell Prolog what to prove but not *how* to prove it
- Based on first-order logic restricted to **Horn clauses**

Prolog – Background

- Developed in France in ∼1972
- Prolog is a **declarative** language
  - Tell Prolog what to prove but not *how* to prove it
- Based on first-order logic restricted to Horn clauses

Horn Clauses

Define goal formulas $\mathtt{G}$ and program clauses $\mathtt{D}$ as follows:

$$\mathtt{G} ::= \top \mid \mathtt{A} \mid \mathtt{G} \land \mathtt{G} \mid \mathtt{G} \lor \mathtt{G} \mid \exists \mathtt{X}\ \mathtt{G}$$

$$\mathtt{D} ::= \mathtt{A} \mid \mathtt{G} \rightarrow \mathtt{D} \mid \mathtt{D} \land \mathtt{D} \mid \forall \mathtt{X}\ \mathtt{D}$$

where A is a first-order atomic formula and quantification is over variables.

Horn Clause Examples

Horn Clauses

Define goal formulas $\mathtt{G}$ and program clauses $\mathtt{D}$ as follows:

$$\mathtt{G} ::= \top \mid \mathtt{A} \mid \mathtt{G} \wedge \mathtt{G} \mid \mathtt{G} \vee \mathtt{G} \mid \exists \mathtt{X}\ \mathtt{G}$$

$$\mathtt{D} ::= \mathtt{A} \mid \mathtt{G} \rightarrow \mathtt{D} \mid \mathtt{D} \wedge \mathtt{D} \mid \forall \mathtt{X}\ \mathtt{D}$$

where A is a first-order atomic formula and quantification is over variables.

Formula	$\mathtt{D}$	$\mathtt{G}$
$p \vee (q \wedge r)$	✗	✓
$\exists X\ f(X) \wedge g(X)$	✗	✓

} No ∨ or ∃ at the top level in programs

Horn Clause Examples

Horn Clauses

Define goal formulas G and program clauses D as follows:

$$\mathtt{G} ::= \top \mid \mathtt{A} \mid \mathtt{G} \wedge \mathtt{G} \mid \mathtt{G} \vee \mathtt{G} \mid \exists \mathtt{X}\ \mathtt{G}$$

$$\mathtt{D} ::= \mathtt{A} \mid \mathtt{G} \rightarrow \mathtt{D} \mid \mathtt{D} \wedge \mathtt{D} \mid \forall \mathtt{X}\ \mathtt{D}$$

where A is a first-order atomic formula and quantification is over variables.

Formula	D	G	
$p \vee (q \wedge r)$	✗	✓	No ∨ or ∃ at the top level in programs
$\exists X\ f(X) \wedge g(X)$	✗	✓	
$\forall X\ f(X)$	✓	✗	No ∀ or → goals
$p \wedge q \rightarrow p$	✓	✗	

Horn Clause Examples

Horn Clauses

Define goal formulas G and program clauses D as follows:

$$\begin{aligned} \mathtt{G} &::= \top \mid \mathtt{A} \mid \mathtt{G} \wedge \mathtt{G} \mid \mathtt{G} \vee \mathtt{G} \mid \exists \mathtt{X}\ \mathtt{G} \\ \mathtt{D} &::= \mathtt{A} \mid \mathtt{G} \to \mathtt{D} \mid \mathtt{D} \wedge \mathtt{D} \mid \forall \mathtt{X}\ \mathtt{D} \end{aligned}$$

where A is a first-order atomic formula and quantification is over variables.

Formula	D	G	
$p \vee (q \wedge r)$	✗	✓	No ∨ or ∃ at the top level in programs
$\exists X\ f(X) \wedge g(X)$	✗	✓	
$\forall X\ f(X)$	✓	✗	No ∀ or → goals
$p \wedge q \to p$	✓	✗	
$\forall F\ g(F(x))$	✗	✗	No quantification over functions

Prolog – Read, Prove, Print Loop

- Each clause ends with a “.”
- Prolog demos also work in λProlog

Program

```
owns(alex, cat(fluffy)).      % alex owns a cat called fluffy
owns(alex, dog(frodo)).
owns(dave, cat(whiskers)).
```

Prolog – Read, Prove, Print Loop

- Each clause ends with a “.”
- Prolog demos also work in λProlog

Program

```
owns(alex, cat(fluffy)).      % alex owns a cat called fluffy
owns(alex, dog(frodo)).
owns(dave, cat(whiskers)).
```

Query

```
?- owns(alex, Pet).           % Does alex own any pets?
```

Prolog – Read, Prove, Print Loop

- Each clause ends with a “`.`”
- Prolog demos also work in λProlog

Program

```
owns(alex, cat(fluffy)).      % alex owns a cat called fluffy
owns(alex, dog(frodo)).
owns(dave, cat(whiskers)).
```

Query

```
?- owns(alex, Pet).           % Does alex own any pets?
Pet = cat(fluffy) .
```

Prolog – Read, Prove, Print Loop

- Each clause ends with a “.”
- Prolog demos also work in λProlog

Program

```
owns(alex, cat(fluffy)).        % alex owns a cat called fluffy
owns(alex, dog(frodo)).
owns(dave, cat(whiskers)).
```

Query

```
?- owns(alex, Pet).             % Does alex own any pets?
Pet = cat(fluffy) .

?- owns(dave, cat(X)).          % Does dave own any cats?
```

Prolog – Read, Prove, Print Loop

- Each clause ends with a "."
- Prolog demos also work in λProlog

Program

```
owns(alex, cat(fluffy)).      % alex owns a cat called fluffy
owns(alex, dog(frodo)).
owns(dave, cat(whiskers)).
```

Query

```
?- owns(alex, Pet).           % Does alex own any pets?
Pet = cat(fluffy) .

?- owns(dave, cat(X)).        % Does dave own any cats?
X = whiskers.
```

Prolog Demo

In Prolog clauses, “`:-`” means “$\leftarrow$” and “`,`” means “$\wedge$”

Program

```
% anne has children bob and cara
parent(anne, bob).
parent(anne, cara).
sibling(X, Y) :- parent(P, X), parent(P, Y), not(X = Y).
```

Prolog Demo

In Prolog clauses, “`:-`” means “←” and “`,`” means “$\wedge$”

Program

```
% anne has children bob and cara
parent(anne, bob).
parent(anne, cara).
sibling(X, Y) :- parent(P, X), parent(P, Y), not(X = Y).
```

Query

```
?- sibling(X,Y).
```

How many solutions are there?

Prolog – Multiple Solutions

Program

```
parent(anne, bob).
parent(anne, cara).
sibling(X, Y) :- parent(P, X), parent(P, Y), not(X = Y).
```

Query

```
?- sibling(X,Y).
X = bob,
Y = cara
```

Prolog – Multiple Solutions

Program

```
parent(anne, bob).
parent(anne, cara).
sibling(X, Y) :- parent(P, X), parent(P, Y), not(X = Y).
```

Query

```
?- sibling(X,Y).
X = bob,
Y = cara ;      % Prolog can output multiple solutions using ;
X = cara,
Y = bob
```

Prolog – Multiple Solutions

Program

```
parent(anne, bob).
parent(anne, cara).
sibling(X, Y) :- parent(P, X), parent(P, Y), not(X = Y).
```

Query

```
?- sibling(X,Y).
X = bob,
Y = cara ;       % Prolog can output multiple solutions using ;
X = cara,
Y = bob ;
false.
```

Negation As Failure

Program

```
parent(anne, bob).
parent(dave, cara).
sibling(X, Y) :- parent(P, X), parent(P, Y), not(X = Y).
```

Query

```
?- sibling(X,Y).
```

Negation As Failure

Program

```
parent(anne, bob).
parent(dave, cara).
sibling(X, Y) :- parent(P, X), parent(P, Y), not(X = Y).
```

Query

```
?- sibling(X,Y).
false.
```

Provability for first-order Horn clauses is undecidable.
So how did Prolog decide that there are no solutions?

Negation As Failure

Program

```
parent(anne, bob).
parent(dave, cara).
sibling(X, Y) :- parent(P, X), parent(P, Y), not(X = Y).
```

Query

```
?- sibling(X,Y).
false.
```

If Prolog fails to find a solution to a query then it is **assumed to be false**

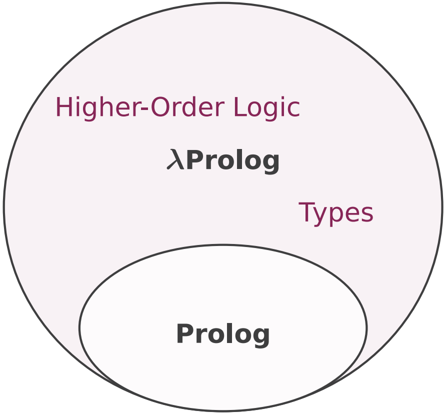
- e.g. we did not specify whether `parent(anne, cara)` is true or false, so Prolog assumes this to be false

What’s Missing?

1. **Higher-Order Logic**

   First order Horn clauses do not allow...

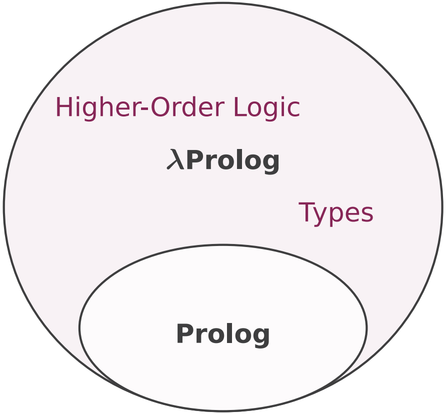
   - Quantification over functions
   - $\rightarrow$ and $\forall$ in goal clauses



What’s Missing?

1. **Higher-Order Logic**
   First order Horn clauses do not allow...
   - Quantification over functions
   - $\rightarrow$ and $\forall$ in goal clauses

2. **Typed Logic Programming**
   What if we want to restrict predicates to certain types of objects?



1. Introduction

2. Logic Programming

3. Prolog

4. λProlog

- Syntax
- Semantics
- Types
- Demo

5. Applications

λProlog Background

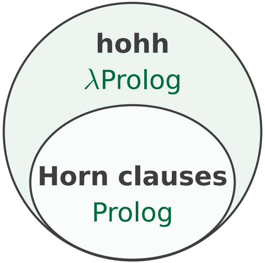
- Invented in 1987 by Dale Miller and Gopalan Nadathur

λProlog Background

- Invented in 1987 by Dale Miller and Gopalan Nadathur
- Adds higher-order logic, native lambda expressions, and polymorphic typing to Prolog

λProlog Background

- Invented in 1987 by Dale Miller and Gopalan Nadathur
- Adds higher-order logic, native lambda expressions, and polymorphic typing to Prolog
- Based on **higher-order hereditary Harrop** formulas (hohh)



Horn Clauses vs. hohh

hohh

Define goal formulas $\mathtt{G}$ and program clauses $\mathtt{D}$ for hohh as follows:

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

$$\mathtt{D} ::= \mathtt{A}_r \mid \mathtt{G} \rightarrow \mathtt{D} \mid \mathtt{D} \wedge \mathtt{D} \mid \forall \mathtt{X}\ \mathtt{D}$$

where elements of $\mathtt{A}$ are normal forms wrt λ-calculus.

Formula	$\mathtt{D}$	$\mathtt{G}$
$p \vee (q \wedge r)$	✗	✓
$\exists X\ f(X) \wedge g(X)$	✗	✓
$\forall X\ f(X)$	✓	✗
$p \wedge q \rightarrow p$	✓	✗
$\forall F\ g(F(x))$	✗	✗

Horn Clauses vs. hohh

hohh

Define goal formulas $\mathtt{G}$ and program clauses $\mathtt{D}$ for hohh as follows:

$$\mathtt{G} ::= \top \mid \mathtt{A} \mid \mathtt{G} \wedge \mathtt{G} \mid \mathtt{G} \vee \mathtt{G} \mid \exists \mathtt{X}\ \mathtt{G} \mid \mathtt{D} \to \mathtt{G} \mid \forall \mathtt{X}\ \mathtt{G}$$

$$\mathtt{D} ::= \mathtt{A}_r \mid \mathtt{G} \to \mathtt{D} \mid \mathtt{D} \wedge \mathtt{D} \mid \forall \mathtt{X}\ \mathtt{D}$$

where elements of $\mathtt{A}$ are normal forms wrt λ-calculus.

Formula	$\mathtt{D}$	$\mathtt{G}$
$p \vee (q \wedge r)$	✗	✓
$\exists X\ f(X) \wedge g(X)$	✗	✓
$\forall X\ f(X)$	✓	✗
$p \wedge q \to p$	✓	✗
$\forall F\ g(F(x))$	✗	✗

} Still no $\vee$ or $\exists$ at the top level in programs

Horn Clauses vs. hohh

hohh

Define goal formulas $\mathtt{G}$ and program clauses $\mathtt{D}$ for hohh as follows:

$$\mathtt{G} ::= \top \mid \mathtt{A} \mid \mathtt{G} \wedge \mathtt{G} \mid \mathtt{G} \vee \mathtt{G} \mid \exists \mathtt{X}\ \mathtt{G} \mid \mathtt{D} \to \mathtt{G} \mid \forall \mathtt{X}\ \mathtt{G}$$

$$\mathtt{D} ::= \mathtt{A}_r \mid \mathtt{G} \to \mathtt{D} \mid \mathtt{D} \wedge \mathtt{D} \mid \forall \mathtt{X}\ \mathtt{D}$$

where elements of $\mathtt{A}$ are normal forms wrt λ-calculus.

Formula	$\mathtt{D}$	$\mathtt{G}$
$p \vee (q \wedge r)$	✗	✓
$\exists X\ f(X) \wedge g(X)$	✗	✓
$\forall X\ f(X)$	✓	✓
$p \wedge q \to p$	✓	✓
$\forall F\ g(F(x))$	✗	✗

$\}$ Still no $\vee$ or $\exists$ at the top level in programs

$\}$ $\forall$ and $\to$ are now allowed in goals

Horn Clauses vs. hohh

hohh

Define goal formulas $\mathtt{G}$ and program clauses $\mathtt{D}$ for hohh as follows:

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

$$\mathtt{D} ::= \mathtt{A}_r \mid \mathtt{G} \rightarrow \mathtt{D} \mid \mathtt{D} \wedge \mathtt{D} \mid \forall \mathtt{X}\ \mathtt{D}$$

where elements of $\mathtt{A}$ are normal forms wrt λ-calculus.

Formula	D	G
$p \vee (q \wedge r)$	✗	✓
$\exists X\ f(X) \wedge g(X)$	✗	✓
$\forall X\ f(X)$	✓	✓
$p \wedge q \rightarrow p$	✓	✓
$\forall F\ g(F(x))$	✓	✓

} Still no $\vee$ or $\exists$ at the top level in programs

} $\forall$ and $\rightarrow$ are now allowed in goals

} Quantification is now allowed over functions

1. Introduction

2. Logic Programming

3. Prolog

4. λProlog

- Syntax
- Semantics
- Types
- Demo

5. Applications

Semantics

How does λProlog find solutions?

- λProlog uses a deterministic proof search
  1. Start from the goal query
  2. Work backwards using the current program state (facts and rules)
  3. If stuck:
     - a) Try to backtrack to an earlier proof state
     - b) If no more backtracking is possible, then return `false`

Semantics

How does λProlog find solutions?

- λProlog uses a deterministic proof search
- Variables are instantiated using unification
  - Decidable for first-order logic but undecidable for higher-order logic (beyond the scope of this talk)

Semantics

How does λProlog find solutions?

- λProlog uses a deterministic proof search
- Variables are instantiated using unification
- λProlog is based on **intuitionistic logic**
  - The logic taught in the 4th semester is **classical logic**
  - Classical logic assumes the Law of the Excluded Middle, i.e. that $p \vee \neg p$ always holds

Semantics

How does λProlog find solutions?

- λProlog uses a deterministic proof search
- Variables are instantiated using unification
- λProlog is based on **intuitionistic logic**
  - The logic taught in the 4th semester is **classical logic**
  - Classical logic assumes the Law of the Excluded Middle, i.e. that $p \vee \neg p$ always holds
  - Inference in λProlog is unsound for classical logic
  - Solution: get rid of the LEM!

Proof Search

- λProlog uses a deterministic, **depth-first** proof search

Program

```
loop :- loop.
```

Proof Search

- λProlog uses a deterministic, **depth-first** proof search
- Query 1 and query 2 are logically equivalent

Program

```
loop :- loop.
```

Query

```
?- fail, loop.      % query 1
```

Query

```
?- loop, fail.      % query 2
```

Proof Search

- λProlog uses a deterministic, **depth-first** proof search
- Query 1 and query 2 are logically equivalent
- Semantically, query 1 fails immediately while query 2 will not terminate

Program

```
loop :- loop.
```

Query

```
?- fail, loop.      % query 1
```

Query

```
?- loop, fail.      % query 2
```

Proof Search

- λProlog uses a deterministic, **depth-first** proof search
- Query 1 and query 2 are logically equivalent
- Semantically, query 1 fails immediately while query 2 will not terminate
- In practice: limit proof search depth

Program

```
loop :- loop.
```

Query

```
?- fail, loop.       % query 1
```

Query

```
?- loop, fail.       % query 2
```

1. Introduction

2. Logic Programming

3. Prolog

4. λProlog

Syntax

Semantics

Types

Demo

5. Applications

λProlog Types

λProlog is strongly typed

- Based on simply-typed λ calculus
- Built-in sorts (`int`, `real`, `string`, `list` ...)
- Formulas have special sort `o`

λProlog Types

λProlog is strongly typed

- Based on simply-typed λ calculus
- Built-in sorts (`int`, `real`, `string`, `list` ...)
- Formulas have special sort `o`

Example

- The predicate `sibling X Y` might have type `person -> person -> o`

λProlog Types

λProlog is strongly typed

- Based on simply-typed λ calculus
- Built-in sorts (`int`, `real`, `string`, `list` ...)
- Formulas have special sort `o`

Example

- The predicate `sibling X Y` might have type `person -> person -> o`
- A predicate `member X L` which encodes that `X` is a member of list `L` might have type `A -> (list A) -> o` where `A` is a type variable

λProlog Types

λProlog is strongly typed

- Based on simply-typed λ calculus
- Built-in sorts (`int`, `real`, `string`, `list` ...)
- Formulas have special sort `o`

Example

- The predicate `sibling X Y` might have type `person -> person -> o`
- A predicate `member X L` which encodes that `X` is a member of list `L` might have type `A -> (list A) -> o` where `A` is a type variable
- “`,`” (logical conjunction) has type `o -> o -> o`

λProlog Demo – 1

- Define new types with `kind`

Program

```
kind dragon_ty                  type.
kind cave_ty                    type.
```

λProlog Demo – 1

- Define new types with `kind`

Program

```
kind dragon_ty, cave_ty         type.
```

λProlog Demo – 1

- Define new types with `kind` and value constructors with `type`

Program

```
kind dragon_ty, cave_ty         type.
type asleep, dangerous          dragon_ty -> o.
```

λProlog Demo – 1

- Define new types with `kind` and value constructors with `type`

Program

```
kind dragon_ty, cave_ty         type.
type asleep, dangerous          dragon_ty -> o.
type safe, daytime              cave_ty -> o.
```

λProlog Demo – 1

- Define new types with `kind` and value constructors with `type`

Program

```
kind dragon_ty, cave_ty         type.
type asleep, dangerous          dragon_ty -> o.
type safe, daytime              cave_ty -> o.
type in                         dragon_ty -> cave_ty -> o.
```

λProlog Demo – 1

- Define new types with `kind` and value constructors with `type`
- Dragons are nocturnal
    - If it is daytime in a cave `C` which contains a dragon `D`, then the dragon is asleep

Program

```
kind dragon_ty, cave_ty         type.
type asleep, dangerous          dragon_ty -> o.
type safe, daytime              cave_ty -> o.
type in                         dragon_ty -> cave_ty -> o.

asleep D :- daytime C, in D C.
```

λProlog Demo – 1

- Define new types with `kind` and value constructors with `type`
- Dragons are nocturnal
  - If it is daytime in a cave `C` which contains a dragon `D`, then the dragon is asleep
- A cave is safe if all dangerous dragons in the cave are asleep

Program

```
kind dragon_ty, cave_ty         type.
type asleep, dangerous          dragon_ty -> o.
type safe, daytime              cave_ty -> o.
type in                         dragon_ty -> cave_ty -> o.

asleep D :- daytime C, in D C.
safe C  :- pi D\ (dangerous D,  in D C) => asleep D.
```

λProlog Demo – 2

Program

```
kind dragon_ty, cave_ty         type.
type asleep, dangerous          dragon_ty -> o.
type safe, daytime              cave_ty -> o.
type in                         dragon_ty -> cave_ty -> o.

asleep D :- daytime C, in D C.
safe C  :- pi D\ (dangerous D,  in D C) => asleep D.
```

Query

Logically, it follows that $\forall$`C daytime C` $\to$ `safe C`. We can ask λProlog:

```
?- pi C\ daytime C => safe C.
```

1. **Introduction**
2. **Logic Programming**
3. **Prolog**
4. **λProlog**
5. **Applications**
    - Comparison: λProlog vs. Haskell
    - Who Uses λProlog?
    - Alternatives to λProlog

λProlog vs. Haskell

Similarities

- Both λProlog and Haskell are based on λ calculus and higher-order logic
- Both support polymorphic types, abstraction via modules, and I/O streams

λProlog vs. Haskell

Similarities

- Both λProlog and Haskell are based on λ calculus and higher-order logic
- Both support polymorphic types, abstraction via modules, and I/O streams

Differences

1. **Purpose**
   - λProlog has native support for logical inference
   - Haskell is better for general-purpose calculation

λProlog vs. Haskell

Similarities

- Both λProlog and Haskell are based on λ calculus and higher-order logic
- Both support polymorphic types, abstraction via modules, and I/O streams

Differences

1. **Purpose**
   - λProlog has native support for logical inference
   - Haskell is better for general-purpose calculation
2. **Evaluation**
   - λProlog works backwards from a goal using unification
   - Haskell works forwards using matching-driven lazy evaluation

λProlog vs. Haskell

Similarities

- Both λProlog and Haskell are based on λ calculus and higher-order logic
- Both support polymorphic types, abstraction via modules, and I/O streams

Differences

1. **Purpose**
   - λProlog has native support for logical inference
   - Haskell is better for general-purpose calculation
2. **Evaluation**
   - λProlog works backwards from a goal using unification
   - Haskell works forwards using matching-driven lazy evaluation
3. **Popularity** – Haskell is much more widely used and better documented

Who Uses λProlog?

There are many implementations of λProlog... but what is it used for?

1. **Theorem Proving**
    - Modelling logic frameworks
    - Appel (1999), Appel and Felty (2004), Guidi et al. (2019), Kohlhase et al. (2020)

Who Uses λProlog?

There are many implementations of λProlog... but what is it used for?

1. **Theorem Proving**
   - Modelling logic frameworks
   - Appel (1999), Appel and Felty (2004), Guidi et al. (2019), Kohlhase et al. (2020)

2. **Program Verification**
   - Modelling program specifications
   - Andrews (1997), Southern and Nadathur (2014)

Who Uses λProlog?

There are many implementations of λProlog... but what is it used for?

1. **Theorem Proving**
   - Modelling logic frameworks
   - Appel (1999), Appel and Felty (2004), Guidi et al. (2019), Kohlhase et al. (2020)
2. **Program Verification**
   - Modelling program specifications
   - Andrews (1997), Southern and Nadathur (2014)
3. **Algorithm Design**
   - Modelling search algorithms and compiler design
   - Rollins and Wing (1990), Liang (2002)

Alternatives to λProlog

Languages Inspired by λProlog

- Elf – implements a more expressive logic (LF) but inspired by λProlog
- Makam – a dialect of λProlog for language prototyping (very niche)

Alternatives to λProlog

Languages Inspired by λProlog

- Elf – implements a more expressive logic (LF) but inspired by λProlog
- Makam – a dialect of λProlog for language prototyping (very niche)

Alternatives to λProlog

- HiLog – extends of Prolog to higher-order logic (less general than λProlog)
- Twelf – logically more general than λProlog (also based on LF)
- Interactive Theorem Provers (Isabelle, Coq, Lean) – more control over proof search

Conclusion

λProlog ...

- is a declarative logic programming language
- extends Prolog with higher-order logic and types
- can be used for theorem proving and program verification
- mostly used in research

Conclusion

λProlog ...

- is a declarative logic programming language
- extends Prolog with higher-order logic and types
- can be used for theorem proving and program verification
- mostly used in research

Further Reading

Programming with Higher-Order Logic, Miller and Nadathur (2012)
An Overview of Lambda-Prolog, Nadathur and Miller (1988)

Bibliography I

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

Appel, A. W. (1999). Lightweight Lemmas in λ Prolog. The MIT Press.

Appel, A. W. and Felty, A. P. (2004). Polymorphic lemmas and definitions in Lambda Prolog and Twelf. Publisher: arXiv Version Number: 1.

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

Bibliography II

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

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

Miller, D. and Nadathur, G. (2012). *Programming with Higher-Order Logic*. Cambridge University Press, 1 edition.

Nadathur, G. and Miller, D. (1988). An Overview of Lambda-Prolog.

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

Bibliography III

Southern, M. and Nadathur, G. (2014). A Lambda Prolog Based Animation of Twelf Specifications. Publisher: arXiv Version Number: 1.

Intuitionistic Logic in λProlog

The formula `p → (p ∨ q) ≡ p ∨ (p → q)` is a **tautology** in classical logic

Proof: If we assume the LEM, then $p \to q \equiv \neg p \vee q$ holds.
Then

$$\begin{aligned} p \to (p \vee q) &\equiv \neg p \vee (p \vee q) \\ &\equiv p \vee (\neg p \vee q) \\ &\equiv p \vee (p \to q) \end{aligned}$$

Intuitionistic Logic in λProlog

The formula $\mathtt{p} \rightarrow (\mathtt{p} \lor \mathtt{q}) \equiv \mathtt{p} \lor (\mathtt{p} \rightarrow \mathtt{q})$ is a tautology in classical logic

Example 1

Consider trying to prove goal $\mathtt{p} \rightarrow (\mathtt{p} \lor \mathtt{q})$ from the empty program.
This is provable if and only if $\mathtt{p} \lor \mathtt{q}$ is provable from $\mathtt{p}$, which is trivial.

Example 2

Consider trying to prove goal $\mathtt{p} \lor (\mathtt{p} \rightarrow \mathtt{q})$ from the empty program.
This is provable if and only if either $\mathtt{p}$ is provable from the empty program or if $\mathtt{q}$ is provable from $\mathtt{p}$.
Neither of these is possible!

Conclusion: λProlog semantics are **unsound** for classical logic