Type Inference for a Higher-Order Extension of Prolog

Emmanouil Koukoutos   Nikolaos S. Papaspyrou

School of Electrical and Computer Engineering
National Technical University of Athens

July 15, 2013

Introduction

Today, we are going to talk about

- The $\mathcal{H}$ framework for higher-order logic programming
- An extension thereof, *poly*$\mathcal{H}$

Higher-order logic programming

Since the invention of Prolog, logic programming has traditionally been first-order.

Attempts to introduce higher order features can be classified in two general categories:

Higher-order logic programming

Since the invention of Prolog, logic programming has traditionally been first-order.

Attempts to introduce higher order features can be classified in two general categories:

- **Intensional semantics**: Two predicates are equal if they have the same name
  - λProlog (Miller, Nadathur) $\rightarrow$ Teyjus
  - Hilog (Chen *et al.*) $\rightarrow$ XSB

Higher-order logic programming

Since the invention of Prolog, logic programming has traditionally been first-order.

Attempts to introduce higher order features can be classified in two general categories:

- **Intensional semantics**: Two predicates are equal if they have the same name
  - λProlog (Miller, Nadathur) $\rightarrow$ Teyjus
  - Hilog (Chen *et al.*) $\rightarrow$ XSB
- **Extensional semantics**: Two predicates are equal if they succeed for the same instances
  - "Definitional subset of Higher-order Horn Logic" (Wadge)
  - $\mathcal{H}$ (Chalalambidis *et al.*) $\rightarrow$ HOPES

The $\mathcal{H}$ framework

A formal basis to extend Prolog with higher-order predicates.

The $\mathcal{H}$ framework

A formal basis to extend Prolog with higher-order predicates.
In a Prolog-like syntax

```
closure(R, X, Y) :- R(X, Y).
closure(R, X, Y) :- R(X, Z), closure(R, Z, Y).
```

The $\mathcal{H}$ framework

A formal basis to extend Prolog with higher-order predicates.
In a Prolog-like syntax

```
closure(R, X, Y) :- R(X, Y).
closure(R, X, Y) :- R(X, Z), closure(R, Z, Y).

parent(trude, sally).
parent(tom, sally).
parent(tom, erica).
parent(mike, tom).
```

The $\mathcal{H}$ framework

A formal basis to extend Prolog with higher-order predicates.
In a Prolog-like syntax

```
closure(R, X, Y) :- R(X, Y).
closure(R, X, Y) :- R(X, Z), closure(R, Z, Y).

parent(trude, sally).
parent(tom, sally).
parent(tom, erica).
parent(mike, tom).

?- closure(parent, mike, X).
X = tom ;
X = sally ;
X = erica ;
false.
```

In $\mathcal{H}$ syntax (omitting type annotations)

$$\begin{array}{lcl} \texttt{closure} & \leftarrow & \lambda \texttt{R}.\ \lambda \texttt{X}.\ \lambda \texttt{Y}.\ \texttt{R X Y} \lor (\exists \texttt{Z}.\ \texttt{R X Z} \land \texttt{closure R Z Y}) \\ \texttt{ancestor} & \leftarrow & \texttt{closure parent} \end{array}$$

Types and Syntax of $\mathcal{H}$

Types in $\mathcal{H}$

$\mathtt{i}$: individual type

$\mathtt{o}$: boolean type

$\sigma ::= \mathtt{i} \mid (\mathtt{i} \rightarrow \sigma)$	*functional types*
$\rho ::= \mathtt{i} \mid \pi$	*argument types*
$\pi ::= \mathtt{o} \mid (\rho \rightarrow \pi)$	*predicate types*

Types in $\mathcal{H}$

$\mathtt{i}$: individual type

$\mathtt{o}$: boolean type

$$\sigma ::= \mathtt{i} \mid (\mathtt{i} \to \sigma)$$ *functional types*

$$\rho ::= \mathtt{i} \mid \pi$$ *argument types*

$$\pi ::= \mathtt{o} \mid (\rho \to \pi)$$ *predicate types*

Syntax of $\mathcal{H}$

$$E ::= \mathtt{true} \mid c_\rho \mid V_\rho \mid f_\sigma\ E_1 \ldots E_n \mid E_1\ E_2$$ *Expressions in $\mathcal{H}$*

$$\mid \lambda V_\rho .\ E \mid E_1 \wedge_\pi E_2 \mid E_1 \vee_\pi E_2$$

$$\mid E_1 \approx E_2 \mid \exists_\rho V .\ E$$

$$C ::= c_\pi \leftarrow_\pi E$$ *clauses in $\mathcal{H}$*

$$G ::= \mathtt{false} \leftarrow_{\mathtt{o}} E$$ *goals in $\mathcal{H}$*

Higher-order queries (1)

```
p(Q) :- Q(0), Q(s(0)).
nat(0).
nat(s(X)) :- nat(X).
```

Higher-order queries (1)

```
p(Q) :- Q(0), Q(s(0)).
nat(0).
nat(s(X)) :- nat(X).

?- p(nat).
Yes.
```

Higher-order queries (1)

```
p(Q) :- Q(0), Q(s(0)).
nat(0).
nat(s(X)) :- nat(X).

?- p(nat).
Yes.

?- p(Q).
```

Higher-order queries (1)

```
p(Q) :- Q(0), Q(s(0)).
nat(0).
nat(s(X)) :- nat(X).

?- p(nat).
Yes.

?- p(Q).
```

- `Q` may be a uncountable set!

Higher-order queries (1)

```
p(Q) :- Q(0), Q(s(0)).
nat(0).
nat(s(X)) :- nat(X).

?- p(nat).
Yes.

?- p(Q).
```

- `Q` may be a uncountable set!
- How do we examine uncountable candidate solutions?

Higher-order queries (1)

```
p(Q) :- Q(0), Q(s(0)).
nat(0).
nat(s(X)) :- nat(X).

?- p(nat).
Yes.

?- p(Q).
```

- `Q` may be a uncountable set!
- How do we examine uncountable candidate solutions?
- How do we return possibly uncountable solutions?

Higher-order queries (2)

The semantics of $\mathcal{H}$ ensures that *a predicate is a solution to a query, if and only it is a superset of a countable set of (finite) compact elements.*
So the proof procedure only has to examine countable solutions!

Higher-order queries (2)

The semantics of $\mathcal{H}$ ensures that *a predicate is a solution to a query, if and only it is a superset of a countable set of (finite) compact elements.*
So the proof procedure only has to examine countable solutions!

Additionally, there is a "representative" solution: In our example,
Q is a solution $\Leftrightarrow Q \supseteq \{0, s(0)\}$.

Higher-order queries (2)

The semantics of $\mathcal{H}$ ensures that *a predicate is a solution to a query, if and only it is a superset of a countable set of (finite) compact elements.*
So the proof procedure only has to examine countable solutions!

Additionally, there is a "representative" solution: In our example,
Q is a solution $\Leftrightarrow Q \supseteq \{0, s(0)\}$.

So

```
?- p(Q).
Q = \X. (X = 0) ; \Y. (Y = s(0)) ; R
```

HOPES: a higher-order logic programming language

HOPES implements $\mathcal{H}$ with a Prolog-like syntax.

Examples:

```
length( [], 0 ).
length( [X|T], s(N) ) :- length( T, N ).

all( R, [] ).
all( R, [ X | Xs ] ) :- R( X ), all( R, Xs ).

map( R, [], [] ).
map( R, [ X | Xs ], [ Y | Ys ] ) :-
    R( X, Y ), map( R, Xs, Ys ).
```

HOPES is incompatible with Prolog

- Prolog allows name aliases

HOPES is incompatible with Prolog

- Prolog allows name aliases
- But in a higher-order setting, we have a problem

```
p.
p(0).
q(X) :- X(p).
```

HOPES is incompatible with Prolog

- Prolog allows name aliases
- But in a higher-order setting, we have a problem

```
p.
p(0).
q(X) :- X(p).
```

What is `p` in the third line? `p/0`, `p/1`, an individual symbol, . . . ?

HOPES is incompatible with Prolog

- Prolog allows name aliases
- But in a higher-order setting, we have a problem

  ```
  p.
  p(0).
  q(X) :- X(p).
  ```

  What is `p` in the third line? `p/0`, `p/1`, an individual symbol, ...?
- HOPES allows partial application ...

HOPES is incompatible with Prolog

- Prolog allows name aliases
- But in a higher-order setting, we have a problem

  ```
  p.
  p(0).
  q(X) :- X(p).
  ```

  What is `p` in the third line? `p/0`, `p/1`, an individual symbol, …?
- HOPES allows partial application …
  but then what is `p(X)`?

HOPES is incompatible with Prolog

- Prolog allows name aliases
- But in a higher-order setting, we have a problem

  ```
  p.
  p(0).
  q(X) :- X(p).
  ```

  What is `p` in the third line? `p/0`, `p/1`, an individual symbol, ...?
- HOPES allows partial application ...
  but then what is `p(X)`?
  `p/1` or a partially applied `p/2`?

HOPES is incompatible with Prolog

- Prolog allows name aliases
- But in a higher-order setting, we have a problem

```
p.
p(0).
q(X) :- X(p).
```

  What is `p` in the third line? `p/0`, `p/1`, an individual symbol, . . . ?

- HOPES allows partial application . . .
  but then what is `p(X)`?
  `p/1` or a partially applied `p/2`?

Solution of HOPES: Name aliases disallowed!

HOPES is monomorphic

- HOPES offers type inference
- but monomorphic
- `closure` $:: (\mathtt{i} \to \mathtt{i} \to \mathtt{o}) \to \mathtt{i} \to \mathtt{i} \to \mathtt{o}$

HOPES is monomorphic

- HOPES offers type inference
- but monomorphic
- $\texttt{closure} :: (\texttt{i} \to \texttt{i} \to \texttt{o}) \to \texttt{i} \to \texttt{i} \to \texttt{o}$
- Why not
  $\texttt{closure} :: \forall \alpha.\ (\alpha \to \alpha \to \texttt{o}) \to \alpha \to \alpha \to \texttt{o}$

poly$\mathcal{H}$: A proposed Extension to $\mathcal{H}$

To solve these problems, we propose *poly*$\mathcal{H}$, an extension to $\mathcal{H}$, which is a more suitable theoretical base for a higher-order extension of Prolog.

poly$\mathcal{H}$: A proposed Extension to $\mathcal{H}$

To solve these problems, we propose *poly*$\mathcal{H}$, an extension to $\mathcal{H}$, which is a more suitable theoretical base for a higher-order extension of Prolog.

New elements introduced in *poly*$\mathcal{H}$:

- All constants are taken from the same alphabet
- But predicates use explicit arities: `p/1`
- Multiple parameters in λ-expressions;
  when a predicate is applied, all arguments must be given.
- Polymorphism (Hindley-Milner)

poly$\mathcal{H}$: An example

- Prolog-like syntax:

```
closure(R)(X, Y) :- R(X, Y).
closure(R)(X, Y) :- R(X, Z), closure(R)(Z, Y).
```

poly$\mathcal{H}$: An example

- Prolog-like syntax:

```
closure(R)(X, Y) :- R(X, Y).
closure(R)(X, Y) :- R(X, Z), closure(R)(Z, Y).
```

- *poly$\mathcal{H}$* syntax:

$$\texttt{closure}/1 \;\leftarrow\; \lambda(\texttt{R}).\, \lambda(\texttt{X},\texttt{Y}).\; \texttt{R}(\texttt{X},\texttt{Y}) \vee (\, \exists \texttt{Z}.\; \texttt{R}(\texttt{X},\texttt{Z}) \;\wedge \texttt{closure}/1(\texttt{R})(\texttt{Z},\texttt{Y})\,)$$

poly$\mathcal{H}$: An example

- Prolog-like syntax:

```
closure(R)(X, Y) :- R(X, Y).
closure(R)(X, Y) :- R(X, Z), closure(R)(Z, Y).
```

- *poly$\mathcal{H}$* syntax:

$$\texttt{closure}/1 \;\leftarrow\; \lambda(\texttt{R}).\, \lambda(\texttt{X},\texttt{Y}).\; \texttt{R}(\texttt{X},\texttt{Y}) \vee (\, \exists \texttt{Z}.\, \texttt{R}(\texttt{X},\texttt{Z}) \wedge \texttt{closure}/1(\texttt{R})(\texttt{Z},\texttt{Y})\,)$$

- Inferred type: $\texttt{closure}/1 :: \forall(\alpha, \phi).\, ((\alpha, \alpha) \to \phi) \to (\alpha, \alpha) \to \phi$
  α: argument type variable
  ϕ: predicate type variable

Syntax of *poly*$\mathcal{H}$

$$
\begin{array}{rcll}
E & ::= & V \mid c \mid c/m & \textit{expressions} \\
& \mid & E(E_1, \ldots, E_n) \mid \lambda(V_1, \ldots, V_n).\ E & \\
& \mid & E_1 \wedge E_2 \mid E_1 \vee E_2 & \\
& \mid & E_1 \approx E_2 \mid \exists V.\ E & \\
R & ::= & c/m \leftarrow E & \textit{rules} \\
G & ::= & R_1, \ldots, R_n & \textit{declaration groups} \\
P & ::= & G_1, \ldots, G_n & \textit{programs}
\end{array}
$$

Types in *poly$\mathcal{H}$*

ρ	$::=$	$\mathtt{i} \mid \alpha \mid \pi$	*argument types*
π	$::=$	$\mathtt{o} \mid \phi \mid (\rho_1, \ldots, \rho_n) \to \pi$	*monomorphic types*
τ	$::=$	$\forall(\alpha_1, \ldots, \alpha_m, \phi_1, \ldots, \phi_{m'}).\ \pi$	*polymorphic types*
Γ	$::=$	$\emptyset \mid \Gamma, V : \rho \mid \Gamma, c/m : \tau$	*type environments*

Type inference in *poly*$\mathcal{H}$

We give an algorithm in the style of Hindley-Milner, with **constraint generation** and **solving**.

$$C \ ::= \ \emptyset \ | \ C, \ \rho_1 = \rho_2 \qquad \textit{constraints}$$

Typing rules in *poly*$\mathcal{H}$

Programs

$$\frac{\Gamma_{i-1} \vdash G_i : \Gamma'_i \quad \text{for all } 1 \leq i \leq n \qquad \Gamma_i = \Gamma_{i-1} \cup \Gamma'_i \quad \text{for all } 1 \leq i \leq n}{\Gamma_0 \vdash G_1, \ \ldots \ , G_n : \Gamma_n}$$

Declaration groups

$$\frac{\begin{array}{c}\Gamma' = \{c/m : \mathsf{artyp}(m) \mid (c/m \leftarrow E) \in G\} \\ \Gamma \cup \Gamma' \vdash R_i \mid C_i \quad \text{for all } R_i \in G \qquad s = \mathsf{unify}(C_1 \cup \ldots \cup C_n)\end{array}}{\Gamma \vdash G : \mathsf{gen}(s(\Gamma'))}$$

$\mathsf{artyp}(0) = \mathsf{o}$

$\mathsf{artyp}(n) = (\alpha_1, \ \ldots \ , \alpha_n) \to \phi$, if $n \geq 1$, α_i, ϕ fresh

Typing rules in *poly$\mathcal{H}$*

Rules

$$\frac{\Gamma \vdash E : \rho \mid C \qquad (c/m : \pi) \in \Gamma}{\Gamma \vdash c/m \leftarrow E \mid \{\pi = \rho\} \cup C}$$

Expressions

$$\frac{(V : \rho) \in \Gamma}{\Gamma \vdash V : \rho \mid \emptyset} \qquad \frac{}{\Gamma \vdash c : \mathtt{i} \mid \emptyset} \qquad \frac{(c/m : \tau) \in \Gamma}{\Gamma \vdash c/m : \text{freshen}(\tau) \mid \emptyset}$$

Typing rules in *poly*$\mathcal{H}$

Expressions

$$\frac{\Gamma \vdash E_i : \rho_i \mid C_i \quad \text{for all } 1 \leq i \leq n}{\Gamma \vdash c(E_1, \ldots, E_n) : \mathtt{i} \mid \{\rho_i = \mathtt{i} \mid 1 \leq i \leq n\} \cup C_1 \cup \ldots \cup C_n}$$

$$\frac{\Gamma \vdash E : \rho \mid C \qquad E \not\equiv c \qquad \Gamma \vdash E_i : \rho_i \mid C_i \quad \text{for all } 1 \leq i \leq n \qquad \phi \text{ fresh}}{\Gamma \vdash E(E_1, \ldots, E_n) : \phi \mid \{\rho = (\rho_1, \ldots, \rho_n) \to \phi)\} \cup C \cup C_1 \cup \ldots \cup C_n}$$

$$\frac{\alpha_i, \phi \text{ fresh} \qquad \Gamma' = \{V_i : \alpha_i \mid 1 \leq i \leq n\} \qquad \Gamma \cup \Gamma' \vdash E : \rho \mid C}{\Gamma \vdash \lambda(V_1, \ldots, V_n).\, E : (\alpha_1, \ldots, \alpha_n) \to \phi \mid \{\phi = \rho\} \cup C}$$

Typing rules in *poly$\mathcal{H}$*

Expressions

$$\frac{\Gamma \vdash E_1 : \rho_1 \mid C_1 \qquad \Gamma \vdash E_2 : \rho_2 \mid C_2 \qquad \phi \text{ fresh}}{\Gamma \vdash E_1 \wedge E_2 : \phi \mid \{\phi = \rho_1, \phi = \rho_2\} \cup C_1 \cup C_2}$$

$$\frac{\Gamma \vdash E_1 : \rho_1 \mid C_1 \qquad \Gamma \vdash E_2 : \rho_2 \mid C_2 \qquad \phi \text{ fresh}}{\Gamma \vdash E_1 \vee E_2 : \phi \mid \{\phi = \rho_1, \phi = \rho_2\} \cup C_1 \cup C_2}$$

$$\frac{\Gamma \vdash E_1 : \rho_1 \mid C_1 \qquad \Gamma \vdash E_2 : \rho_2 \mid C_2}{\Gamma \vdash E_1 \approx E_2 : \texttt{o} \mid \{\rho_1 = \texttt{i}, \rho_2 = \texttt{i}\} \cup C_1 \cup C_2}$$

$$\frac{\Gamma, V : \alpha \vdash E : \rho \mid C \qquad \alpha, \phi \text{ fresh}}{\Gamma \vdash \exists V.\, E : \phi \mid \{\phi = \rho\} \cup C}$$

$$
\begin{array}{ll}
\text{unify}(\emptyset) = \text{id} & \\
\text{unify}(\{\rho_1 = \rho_2\} \cup C) = & \\
\quad \text{unify}(C) & \text{if } \rho_1 = \rho_2 \\
\quad \text{unify}([\alpha \mapsto \rho_2]C) \circ [\alpha \mapsto \rho_2] & \text{if } \rho_1 = \alpha \text{ and } \alpha \notin \rho_2 \\
\quad \text{unify}([\alpha \mapsto \rho_1]C) \circ [\alpha \mapsto \rho_1] & \text{if } \rho_2 = \alpha \text{ and } \alpha \notin \rho_1 \\
\quad \text{unify}([\phi \mapsto \pi]C) \circ [\phi \mapsto \pi] & \text{if } \rho_1 = \phi,\ \rho_2 = \pi \text{ and } \phi \notin \pi \\
\quad \text{unify}([\phi \mapsto \pi]C) \circ [\phi \mapsto \pi] & \text{if } \rho_2 = \phi,\ \rho_1 = \pi \text{ and } \phi \notin \pi \\
\quad \text{unify}(C \cup \left\{\rho_1^i = \rho_2^i \mid 1 \leq i \leq n\right\} \cup \{\pi_1 = \pi_2\}) & \text{if } \rho_1 = (\rho_1^1, \ldots, \rho_1^n) \to \pi_1 \\
& \quad \rho_2 = (\rho_2^1, \ldots, \rho_2^n) \to \pi_2 \\
\quad \text{type error} & \text{otherwise}
\end{array}
$$

Examples

$$
\begin{aligned}
\texttt{length/2} \;\leftarrow\;\; & \lambda(\texttt{L},\texttt{N}).\\
& \quad (\ \texttt{N} \approx \texttt{0} \wedge \texttt{L} \approx [\,]\ ) \vee \\
& \quad (\ \exists \texttt{H}.\ \exists \texttt{L}_2.\ \exists \texttt{N}_2.\\
& \qquad \texttt{L} \approx .(\texttt{H},\texttt{L}_2) \ \wedge\ \texttt{length/2}(\texttt{L}_2,\texttt{N}_2) \wedge \\
& \qquad \texttt{is/2}(\texttt{N}, +(\texttt{N}_2, \texttt{1}))\\
& \quad)\\
\texttt{map/1} \;\leftarrow\;\; & \lambda(\texttt{R}).\ \lambda(\texttt{L}_1,\texttt{L}_2).\\
& \quad (\ \texttt{L}_2 \approx [\,] \wedge \texttt{L}_1 \approx [\,]\ ) \vee \\
& \quad (\ \exists \texttt{Y}.\ \exists \texttt{Ys}.\ \exists \texttt{X}.\ \exists \texttt{Xs}.\\
& \qquad \texttt{L}_2 \approx .(\texttt{Y},\texttt{Ys}) \ \wedge\ \texttt{L}_1 \approx .(\texttt{X},\texttt{Xs}) \ \wedge\ \texttt{R}(\texttt{X},\texttt{Y}) \wedge \\
& \qquad \texttt{map/1}(\texttt{R})(\texttt{Xs},\texttt{Ys})\\
& \quad)\\
\texttt{o/2} \;\leftarrow\;\; & \lambda(\texttt{F},\texttt{G}).\ \lambda(\texttt{X},\texttt{Y}).\ \exists \texttt{Z}.\ (\ \texttt{F}(\texttt{X},\texttt{Z}) \wedge \texttt{G}(\texttt{Z},\texttt{Y})\ )
\end{aligned}
$$

Conclusion

We defined *poly*$\mathcal{H}$, a framework for logic programming which

- extends the $\mathcal{H}$ framework by Charalambidis *et al.*
- is suitable as a basis for a higher-order extension of Prolog
- supports (Hindley-Milner) polymorphism

Conclusion

We defined *poly*$\mathcal{H}$, a framework for logic programming which

- extends the $\mathcal{H}$ framework by Charalambidis *et al.*
- is suitable as a basis for a higher-order extension of Prolog
- supports (Hindley-Milner) polymorphism

A prototype implementation, *poly*HOPES, offering a Prolog-like syntax, is under development at
`https://github.com/acharal/hopes/tree/polyhopes`

Future work

- Principality of types
- Semantics of polymorphic goals
- Implement a complete extension of Prolog

The End

Thank you for your attention!