Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

A Higher-Order Extension of Prolog with Polymorphic Type Inference

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΕΜΜΑΝΟΥΗΛ ΚΟΥΚΟΥΤΟΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Επικ. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2013

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

A Higher-Order Extension of Prolog with Polymorphic Type Inference

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΕΜΜΑΝΟΥΗΛ ΚΟΥΚΟΥΤΟΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Επικ. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 12η Ιουλίου 2013.

....	....	....
Νικόλαος Σ. Παπασπύρου Επικ. Καθηγητής Ε.Μ.Π.	Κωστής Σαγώνας Αν. Καθηγητής Ε.Μ.Π.	Δημήτρης Φωτάκης Λέκτορας Ε.Μ.Π.

Αθήνα, Ιούλιος 2013

....

Εμμανουήλ Κουκουτός

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

στη μητέρα μου

Περίληψη

Από όταν σχεδιάστηκε η Prolog, ο λογικός προγραμματισμός έγινε ένα από τα επικρατέστερα μοντέλα προγραμματισμού. Ο λογικός προγραμματισμός είναι παραδοσιακά πρώτης τάξης. Ωστόσο, έχουν υπάρξει κάποιες προσπάθειες να επεκταθεί με χαρακτηριστικά υψηλότερης τάξης.

Μία πρόσφατη εργασία των Χαραλαμπίδη κ.α. προτείνει μία καινοτόμα εναλλακτική στις προηγούμενες προσπάθειες. Οι συγγραφείς ορίζουν ένα θεωρητικό πλαίσιο, $\mathcal{H}$, με στατικό σύστημα τύπων και εκτατή σημασιολογία. Σε αντίθεση με προηγούμενα πλαίσια υψηλότερης τάξης με εκτατή σημασιολογία, το δικό τους επιτρέπει μη αρχικοποιημένες μεταβλητές υψηλότερης τάξης. Παρέχουν επίσης μία πρωτότυπη υλοποίηση, υπό το όνομα Hopes.

Σε αυτή τη διπλωματική εργασία, συνεχίζουμε τη δουλειά των Χαραλαμπίδη κ.α. για να υπερβούμε δύο μειονεκτήματά της:

- Η Hopes είναι αυτή τη στιγμή ασύμβατη με την Prolog, το εκ των πραγμάτων σημείο αναφοράς για το λογικό προγραμματισμό. Πιο συγκεκριμένα, απλά προγράμματα σε Prolog που θεωρητικά ανήκουν στο υποσύνολο πρώτης τάξης της $\mathcal{H}$ δεν γίνονται δεκτά από την Hopes.
- Η Hopes δεν υποστηρίζει πολυμορφικά κατηγορήματα.

Για να υπερβούμε αυτά τα μειονεκτήματα, επανασχεδιάζουμε την $\mathcal{H}$, στοχεύοντας σε μια υλοποίηση που θα μπορεί να χειριστεί ένα υπερσύνολο υψηλότερης τάξης της συνηθισμένης Prolog, και την επεκτείνουμε με ένα *πολυμορφικό σύστημα συμπερασμού τύπων*. Ονομάζουμε το καινούριο μας πλαίσιο *poly*$\mathcal{H}$. Στη συνέχεια προτείνουμε μία γλώσσα προγραμματισμού βασισμένη στο πλαίσιο αυτό, την *poly*Hopes, η οποία είναι σχεδιασμένη για να αποτελέσει μια επέκταση υψηλότερης τάξης της Prolog, και για την οποία παρέχουμε μια πρωτότυπη υλοποίηση.

Λέξεις κλειδιά

Συμπερασμός τύπων, πολυμορφισμός, λογικός προγραμματισμός, προγραμματισμός υψηλής τάξης, Prolog

Abstract

Since the design of Prolog, logic programming has been one of the most prominent programming paradigms. Logic programming has traditionally been first-order. However, there have been some attempts to extend it with higher-order features.

A recent work by Charalambidis *et al.* proposes an innovative alternative to the previous attempts. Its authors define a framework, $\mathcal{H}$, with a static typing discipline and an extensional semantics. Unlike previous extensional higher-order frameworks, their framework allows for uninstantiated higher-order variables. They also provide a prototype implementation, Hopes.

In this thesis, we build on this work of Charalambidis *et al.* to overcome two drawbacks:

- Hopes is currently incompatible with Prolog, the de-facto standard in logic programming. More specifically, simple Prolog programs which theoretically belong in the first-order subset of $\mathcal{H}$ are not accepted by Hopes.
- Hopes supports no polymorphic predicates.

To overcome these drawbacks, we redesign $\mathcal{H}$, aiming at an implementation that could handle a higher-order proper superset of ordinary Prolog, and extend it with a *polymorphic type inference* system. The new framework is named *poly*$\mathcal{H}$. Then, we propose a surface language over our framework, *poly*Hopes, which is designed to be a higher-order extension of Prolog, and of which we provide a prototype implementation.

Key words

Type inference, polymorphism, logic programming, higher-order programming, Prolog

Ευχαριστίες

Καταρχήν, θα ήθελα να ευχαριστήσω τον επιβλέποντα αυτής της διπλωματικής, Νικόλαο Παπασπύρου, για την πολύτιμη καθοδήγησή του κατά τη διάρκεια εκπόνησης της διπλωματικής μου. Επίσης τους καθηγητές μου Δημήτρη Φωτάκη και Κωστή Σαγώνα, και όσους άλλους συμβάλλουν θετικά στη λειτουργία του Πολυτεχνείου. Επίσης, θα ήθελα να ευχαριστήσω τον Άγγελο Χαραλαμπίδη, υποψήφιο διδάκτορα του ΕΚΠΑ, για τις παρατηρήσεις του και τη βοήθειά του. Ακόμα, ευχαριστώ τους γονείς μου, που πάντα με κινητοποιούσαν και με υποστήριζαν στις σπουδές μου. Τέλος, ένα μεγάλο ευχαριστώ στους συμφοιτητές και τους φίλους μου, που δουλέψαμε και ζήσαμε μαζί αυτά τα μοναδικά χρόνια.

Εμμανουήλ Κουκουτός,

Αθήνα, 12η Ιουλίου 2013

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-1-13, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2013.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Contents

List of Figures

Chapter 1

Introduction

1.1 Objectives

This thesis aims at defining and implementing a higher-order logic programming language supporting polymorphic type inference, which is also a proper superset of Prolog. It mostly builds on recent work by Charalambidis *et al.* [Char10, Char13b]. The work presented here mainly addresses the front-end of the system (syntax, type checking), while the language semantics and proof procedure are borrowed (with the necessary adjustments) from the original work in [Char10].

1.2 Motivation

Since its conception, logic programming has been one of the most prominent programming paradigms. Prolog, the first complete logic programming system to be developed in 1972, has been the *de facto* standard in logic programming since. Prolog (PROgramming in LOGic) was based on work by Alain Colmerauer and Robert Kowalski, who developed a procedural implementation of implications in first-order logic in 1972. The use of Prolog as a practical programming language was given great momentum by the development of a compiler by David Warren in Edinburgh in 1977.

In the meantime, the other prominent declarative paradigm, functional programming, was being developed. Functional languages implement a superset of a (usually typed) lambda calculus. As is the case in lambda calculus, functional programming allows for higher-order functions – that is, functions that accept other functions as arguments or return them as return values.

Logic programming, on the other hand, has traditionally been first-order, following the trend Prolog has set. However, there have been some attempts to incorporate higher-order features into logic programming, mainly λProlog [Mill86], HiLog [Chen93], a framework developed by W. Wadge which he calls "the definitional subset of higher-order Horn logic" [Wadg91] and more recently, the $\mathcal{H}$ framework by Charalambidis *et al.* [Char10, Char13b]. Of these, HiLog is the only one which attempts to define a proper extension to Prolog, while the others are independent languages. HiLog, though, only offers an extended syntax. It does not offer any static type checking discipline, and its semantics collapses to this of first-order logic.

The most recent work [Char10] is an innovative alternative to the previous attempts. It defines a language $\mathcal{H}$ with a static typing discipline and an extensional semantics, which collapses to the traditional semantics of logic programming when restricted to the first-order subset of $\mathcal{H}$. $\mathcal{H}$ also defines a proof procedure that is proved to be complete and sound. Unlike previous attempts to define extensional higher-order logic programming, their framework allows for uninstantiated higher-order variables, namely predicate variables that can be returned as a reply for a query. Their framework is implemented as a simple interpreter, HOPES [Char13a].

In this thesis, we build on this work of Charalambidis *et al.* to overcome two drawbacks:

- HOPES is currently incompatible with Prolog, the de-facto standard in logic programming. More specifically, simple Prolog programs which theoretically belong in the first-order subset of $\mathcal{H}$ are not accepted by HOPES. This is partly due to language design decisions; e.g., Charalambidis *et al.* make use of currying for passing multiple parameters to predicates.

- $\mathcal{H}$ requires type annotations, which is largely inconsistent with Prolog's untyped nature. On the other hand, HOPES infers predicate types but does not support polymorphism.

To overcome these drawbacks, we proceed in three fronts:

- We redesign $\mathcal{H}$, aiming at an implementation that could handle a higher-order proper superset of ordinary Prolog.

- We extend $\mathcal{H}$ with a *polymorphic type inference* system which abolishes the need for explicit type annotations in predicates. Our type system supports Hindley-Milner (or ML-style) parametric polymorphism for higher-order predicates. We are naming the revised framework *poly*$\mathcal{H}$.

- We propose a surface language for our framework, called *poly*HOPES, which is designed to be a higher-order extension of Prolog with polymorphic type inference. *poly*HOPES is influenced by the syntax of Erlang [Eric13, Cesa09], a functional language which was in turn originally inspired by Prolog. We also provide a prototype implementation of *poly*HOPES.

1.3 Outline of the Thesis

The rest of the thesis is organized as follows: First of all, Chapter 2 gives a brief overview of $\mathcal{H}$ and its implementation, HOPES, focusing on the features that are primarily addressed in this thesis. Chapter 3 proposes an alternative framework, *poly*$\mathcal{H}$, that is better suited to offer the basis for a higher-order language that is compatible with Prolog. Chapter 4 presents such a language, *poly*HOPES, while Chapter 5 explains how the structures of the surface language *poly*HOPES correspond to those of *poly*$\mathcal{H}$ and Chapter 6 contains illustrative examples. Finally, we conclude the thesis and suggest possible directions for further research on the subject in Chapter 7.

Chapter 2

An overview of $\mathcal{H}$

Higher-order logic programming comes in two general categories: *intensional* and *extensional*. The former considers two predicates equal if they have the same name; the latter considers them equal if they succeed for the same instances. The former category is more developed, and some systems based on this principles have been built, notably XSB [XSB12] and Teyjus [Teyj08]. The latter is less developed, and no working systems had been built until recently. In [Char10], the authors build upon the work of Wadge [Wadg91] and present $\mathcal{H}$, a complete theoretical framework for extensional higher-order logic programming, including a denotational semantics and a proof procedure which is shown to be complete and sound. The most innovative feature of $\mathcal{H}$ is that, although it does not offer higher-order unification, it allows uninstantiated higher-order variables to appear in program clauses and queries.

2.1 An Intuitive Approach to $\mathcal{H}$

The language $\mathcal{H}$, defined by Charalambidis *et al.*, can be used as the formal basis to extend Prolog with higher-order predicates [Char10, Char13b], i.e., predicates taking other predicates as arguments. The following predicate, written in a Prolog-like syntax, defines the transitive closure of a relation `R`:

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

Now, if `parent` is defined as:

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

then the following goal will find all the descendants of `mike`:

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

In the syntax of $\mathcal{H}$, the definition of `closure` is written as follows (omitting type annotations). Notice also that the *partial application* of predicates and *currying* can be used to define `ancestor`:

$$\begin{array}{lll} \texttt{closure} & \leftarrow & \lambda \mathsf{R}.\ \lambda \mathsf{X}.\ \lambda \mathsf{Y}.\ \mathsf{R\ X\ Y} \vee (\exists \mathsf{Z}.\ \mathsf{R\ X\ Z} \wedge \texttt{closure}\ \mathsf{R\ Z\ Y}) \\ \texttt{ancestor} & \leftarrow & \texttt{closure parent} \end{array}$$

We mentioned before that the $\mathcal{H}$ framework also accepts uninstantiated higher-order variables. So what would happen if the user had inserted the query

```
?- closure(R, mike, erica).
```

instead?

To illustrate this, let us consider the following simpler example, borrowed from [Char13b] and written in a Prolog-like syntax:

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

where `s(X)` stands for the successor of a natural number X. Understood under an extensional semantics, `p` is the set of all relations that succeed at least for `0` and `s(0)`. Now consider the query

```
?- p(Q).
```

At first sight, this query seems unreasonable, since there are uncountably many relations that succeed for `0` and `s(0)` (otherwise said, that are supersets of `{0, s(0)}`). However, 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 relations*. That means that the proof procedure of $\mathcal{H}$ can only examine those finite relations as candidate solutions and enumerate those that satisfy the query. Additionally, there is a solution that is representative of all others (intuitively, that is a subset of all others). In our concrete example, every acceptable solution `Q` is a superset of `{0, s(0)}` and vise versa. So, `{0, s(0)}` is representative of all solutions and will be return as a result to the user. To be accurate, the answer will be returned in a notation compatible with the syntax of the language, namely as a disjunction of λ-abstractions:

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

where `';'` is the (higher-order) disjunction operator, and `Q'` represents any other solution (so `Q` is any superset of `{0, s(0)}`).

Now let us continue with a more formal approach to $\mathcal{H}$.

2.2 Types in $\mathcal{H}$

The language $\mathcal{H}$ is based on a simple type system that supports two base types: `o`, the boolean domain, and `i`, the domain of individuals (data objects). The composite types are partitioned into three classes: functional (assigned to function symbols), predicate (assigned to predicate symbols) and argument (assigned to parameters of predicates).

$$\begin{array}{lll} \sigma ::= \mathtt{i} \mid (\mathtt{i} \to \sigma) & & \textit{functional types} \\ \rho ::= \mathtt{i} \mid \pi & & \textit{argument types} \\ \pi ::= \mathtt{o} \mid (\rho \to \pi) & & \textit{predicate types} \end{array}$$

Functional types, as is the tradition in logic, represent mappings from vectors of individuals to new individuals. As you can see, there are no type variables or polymorphic types in $\mathcal{H}$.

2.3 Syntax of $\mathcal{H}$

Below we give a slightly simplified version of the $\mathcal{H}$ syntax. In the following, V denotes variables, c denotes argument (predicate or individual) constants of every argument type, f denotes functional symbols of every functional type $\sigma \neq \mathtt{i}$, and n is a natural number with $n \geq 1$. Note that, because individual constants can be considered the subset of functional constants with arity 0, we will sometimes refer to both categories of symbols as "functional" symbols.

Expressions of $\mathcal{H}$ consist of the `true` constant, variables (V), predicate/individual constants (c), functional applications, predicate applications, lambda abstractions, conjunctions and disjunctions, unifications, and existentials.

$$\begin{array}{rcl} E & ::= & \mathtt{true} \mid c_\rho \mid V_\rho \mid f_\sigma\ E_1 \ldots\ E_n \mid E_1\ E_2 \mid \lambda V_\rho.\ E \\ & \mid & E_1 \wedge_\pi E_2 \mid E_1 \vee_\pi E_2 \mid E_1 \approx E_2 \mid \exists_\rho V.\ E \end{array} \qquad \textit{Expressions in } \mathcal{H}$$

Notice that constants, variables, as well as all connectives in $\mathcal{H}$ need type annotations (except for $\approx$ (unification), which works only on expressions of type $\mathtt{i}$). In the rule for functional application, f must be of the passing functional type, or $\sigma = \mathtt{i}^n \to \mathtt{i}$.

A *clause* in $\mathcal{H}$ is a predicate constant, followed by the reverse implication symbol and a defining expression:

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

The defined predicate, the $\leftarrow$ connective and the expression must be of the same predicate type.

Finally, a goal in $\mathcal{H}$ is a defined as

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

We will not go into details about the typing rules of $\mathcal{H}$ here, as the typing rules are generally straightforward. The type system of our own revision of the framework, *poly*$\mathcal{H}$, will be analyzed in detail in Section 3.2.

2.4 Semantics of $\mathcal{H}$

In Section 2.1 we mentioned that the semantics of $\mathcal{H}$ allow the construction of a proof procedure which returns meaningful answers to queries with uninstantiated higher-order variables. In this section we will quickly cover the semantics of $\mathcal{H}$ to clarify how this is possible. We are keeping the analysis as simple as possible, as our aim is not to cover the semantics of $\mathcal{H}$ in depth, but to present the most original ideas found in [Char13b] that allow higher-order queries. Notably, we are simplifying notation, unfortunately losing on the formality of our analysis. For a more formal and detailed analysis, please refer to the original work.

To begin with, let us define *lattices* and *algebraic lattices*: A lattice is a partially ordered set (poset) in which every subset has a least upper bound (lub) and a greatest lower bound (glb). An algebraic lattice is a lattice $\mathcal{L}$ with the additional property that each of its elements is the lub of a set of *compact* (intuitively, simple) elements of $\mathcal{L}$. If additionally these compact elements are countable, the lattice is called an *ω-algebraic lattice*. We will write $\mathcal{K}(\mathcal{L})$ to denote the set of compact elements of the algebraic lattice $\mathcal{L}$, and $\mathcal{K}(\mathcal{L})_x$ to denote the set C of compact elements y with $y \sqsubseteq x$. It can be shown that $x = \bigsqcup C$.

Let $\mathcal{D}$ be a nonempty set functioning as the domain of interpretations of expressions of $\mathcal{H}$. The individual type in $\mathcal{H}$ is interpreted as $\mathcal{D}$: $[\![\mathtt{i}]\!] = \mathcal{D}$. We can consider $\mathcal{D}$ to be an algebraic lattice under the trivial partial order $\sqsubseteq_{\mathtt{i}}$ such that $d \sqsubseteq_{\mathtt{i}} d$, for all $d \in D$. Also, the boolean type forms an algebraic lattice with $false \sqsubseteq_{\mathtt{o}} true$, $false \sqsubseteq_{\mathtt{o}} false$ and $true \sqsubseteq_{\mathtt{o}} true$. In fact, the semantics of types are defined in such a way that all predicate types in $\mathcal{H}$ form ω-algebraic lattices.

This is achieved by introducing the following constraint to the semantics of types: the semantics of a type $(\pi_1 \rightarrow \pi_2)$ is not the set of functions from $[\![\pi_1]\!]$ to $[\![\pi_2]\!]$, as one may expect, but only $\mathcal{K}([\![\pi_1]\!]) \overset{m}{\rightarrow} [\![\pi_2]\!]$, where $\overset{m}{\rightarrow}$ denotes monotonic functions. Under this constraint, all predicate types in $\mathcal{H}$ are shown to be ω-algebraic lattices.

However, it is not obvious how the semantics of expressions can be defined to ensure that all expressions are indeed interpreted as elements of the domain described above. To achieve that, $\mathcal{H}$ interprets λ-abstractions and (predicate) applications in a non-standard way. Parameters of λ-abstractions range only over the compact elements of their type. The semantics of application is defined as

$[\![(E_1 E_2)]\!] = \bigsqcup_b ([\![E_1]\!](b))$,

with b ranging over $\mathcal{K}(type(E_2))_{[\![E_2]\!]}$.

Due to this property of the domain, it is shown in [Char10] that each program has a minimum Herbrand model as in the first order case and an immediate consequence operator can be defined that is indeed continuous. That means that for every goal G, x is a solution of G only if every element in $\mathcal{K}(type(x))_x$ is also a solution of G. Also, if $\mathcal{C}$ is the set of all compact elements that satisfy G, then a predicate p is a solution of G if and only if $\bigsqcup \mathcal{C} \sqsubseteq p$. Consequently, we can represented all solutions of a query by enumerating the (countable) compact elements $\mathcal{C}$ comprising it.

2.5 Hopes

The ideas covered in the previous sections have been implemented in a prototype interpreter by A. Charalambidis, Hopes [Char13a]. The interpreter offers a Prolog-like extended syntax, of which we have already seen some examples. Also, it offers a type inference mechanism, to avoid the explicit type annotations mandatory in $\mathcal{H}$, and some useful syntactic sugar. Hopes currently implements no arithmetic, so one has to rely on successor notation to define simple predicates on integers.

To get a feel of how this language looks like, let us consider some examples. Let's start with some simple examples given in [Char13a]:

```
nat(0).
nat(s(X)) :- nat(X).
even(0).
even( s(s(X)) ) :- even(X).
```

The predicate `nat` defines natural numbers in successor notation, and `even` succeeds only for even naturals.

Hopes also offers support for lists. From a theoretical point of view, list notation is just syntactic sugar for an individual constant, `nil` or `[]`, along with the functional symbol `cons` or `'.'`, which constructs a new list from a two other individuals (head and tail). So we could define a predicate calculating the length of a list as in Prolog:

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

In $\mathcal{H}$ syntax, the above predicate would be written (omitting type annotations) as

$$\texttt{length} \;\leftarrow\; \lambda \texttt{L}.\, \lambda \texttt{N}'.\, (\texttt{L} \approx [\,] \wedge \texttt{N}' \approx 0) \vee (\exists \texttt{X}.\, \exists \texttt{T}.\, \exists \texttt{N}.\, \texttt{L} \approx .(\texttt{X}, \texttt{T}) \wedge \texttt{N}' \approx \texttt{s}(\texttt{N}) \wedge \texttt{length}\; \texttt{T}\; \texttt{N})$$

Let us now write two examples of higher order predicates: `all(R, L)` succeeds if R succeeds for all elements of the list `L`. `map` is the predicate version of the functional programming favorite:

```
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 ).
```

Take in account, however, that since functional symbols only work on entities of type `i`, lists in Hopes are monomorphic and cannot contain predicates, as is the case in functional programming.

Partial application is allowed in Hopes. For example, `all(even)` is a predicate of type $(\texttt{i} \rightarrow \texttt{o})$ that succeeds if its argument is a list of even numbers.

2.6 Shortcomings of $\mathcal{H}$ and Hopes

Although $\mathcal{H}$ is a very innovative and well-defined framework, with some nice attributes such as completeness and soundness, it does have some drawbacks, originating mostly from the difficulty of combining a Prolog-like language with higher-orderness.

One of the major difficulties in introducing higher-order features to Prolog is the language's liberty with name aliases. Prolog is an *untyped* language. Multiple predicates with the same name can be defined, which are then told apart by their arities; they cannot be confused because only full application is allowed. The same name can be used in an expression both as a predicate and functional (or individual) symbol; Prolog always knows which entity the name refers to, because predicate names appear only outside all parentheses in rule bodies, while functional (or individual) symbols appear only inside parentheses, as predicate arguments. However, in a higher-order setting, aliasing creates many problems which cannot easily be resolved. Consider, for example, the following simple program:

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

What does `p` refer to, in the third line? The predicate of arity 0 defined in the first line, the predicate of arity 1 defined in the second line, or an individual constant? Hopes bypasses this problem by disallowing name aliases completely: no synonym predicates are allowed, and a name defining a predicate cannot then be used as a functional symbol. This, however, is incompatible with ordinary Prolog.

Another problem in this context is created by partial application: If we allowed partial application along with name aliasing without taking measures, it would be impossible to tell if the predicate in `p(X)` is `p/1` or a partially applied `p/2`, `p/3`, etc. Hopes, liberated from aliasing, offers a simple partial application mechanism which ignores arity, as we saw earlier. If we are to design a Prolog-compatible higher-order language, this mechanism must be redesigned.

Finally, $\mathcal{H}$ offers no parametric polymorphism. The original implementation of Hopes provides a type inference mechanism to avoid explicit type annotations, but this mechanism, like the underlying type system for $\mathcal{H}$, is monomorphic. For example, the type of `closure` from Section 2.1 would be inferred in Hopes as: $(\texttt{i} \rightarrow \texttt{i} \rightarrow \texttt{o}) \rightarrow \texttt{i} \rightarrow \texttt{i} \rightarrow \texttt{o}$. This type is not, however, the most general possible. Why not allow `closure` to handle relations on any argument type?

In the rest of this thesis, we are trying to address these issues. We begin by defining an alternative framework in the following chapter.

Chapter 3

poly$\mathcal{H}$: A Framework for Higher-Order Logic Programming

In this chapter we present *poly$\mathcal{H}$* (*polymorphic $\mathcal{H}$*), our alternative framework for higher-order logic programming with extensional semantics. *poly$\mathcal{H}$* builds on $\mathcal{H}$, but is more suited to be a formal basis of a proper superset of Prolog. Additionally, it upgrades $\mathcal{H}$'s monomorphic type system to one supporting let-polymorphism (in the style of ML).

In brief, the most significant differences between $\mathcal{H}$ and *poly$\mathcal{H}$* are the following:

- In *poly$\mathcal{H}$*, individual, functional and predicate constants are taken from the same alphabet, as in Prolog. The problem with name aliases is resolved by using explicit arities when referring to predicate constants; e.g., we write `p/1`. To remain compatible with Prolog, the implementation *poly*Hopes will often be able to infer a predicate's arity when the predicate is applied. Additionally, *poly*Hopes will be able to differentiate predicates from functional symbols by their position. Some extra measures need to be taken to allow the programmer to override the default position-based interpretation if needed; this will be explained in detail in Chapters 4 and 5.
- In *poly$\mathcal{H}$* we allow multiple parameters in λ-expressions; however, when such a predicate is applied, all arguments must be given. In Prolog-like syntax, the predicate `closure` could be defined as follows, allowing for the partial application of its first argument:

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

  The type system is adjusted accordingly to describe the types of the predicates defined with multiple parameters. E.g. a first-order predicate that takes two parameters of type `i` would be of type $(\mathtt{i}, \mathtt{i}) \to \mathtt{o}$.
- The major addition in *poly$\mathcal{H}$* w.r.t. $\mathcal{H}$ is polymorphism. In *poly$\mathcal{H}$*, the `closure` predicate, written as:

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

  would have its type inferred as $\forall(\alpha, \phi).\, ((\alpha, \alpha) \to \phi) \to (\alpha, \alpha) \to \phi$, where α denotes an argument type variable and ϕ denotes a predicate type variable, as will be explained in §3.2.

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

The syntax of *poly$\mathcal{H}$* is defined as follows, where *V* denotes variables, *c* denotes constants, *m* and *n* are natural numbers with $m \geq 0$ and $n \geq 1$. Expressions consist of variables (*V*), functional/individual symbols (*c*), predicate symbols which are always annotated by their arity (c/m), applications, lambda abstractions, conjunctions and disjunctions, unifications, existentials, and liftings.

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

Figure 3.1: The syntax of expressions in *poly*$\mathcal{H}$

Each predicate is defined by exactly one rule. For simplifying the type inference algorithm, we assume that the rules in a program are organized in *declaration groups*. Each group contains a set of rules defining predicates that are (necessarily) mutually recursive; rules in a group may use only the predicates defined in previous groups or in this one. Given a program in a language like Prolog or Haskell, where these declaration groups are not given explicitly by the user, they can be deduced automatically with a simple dependency analysis [Marl10].

The `true` and `false` logical constants of $\mathcal{H}$ are not mentioned separately here, but they are present in our language too and are written `true/0` and `false/0`.

Notice that application allows for multiple arguments and, likewise, λ-abstraction allows for multiple parameters. Additionally, $\wedge$, $\vee$, $\exists$ and $\leftarrow$ now lack type annotations, and *poly*$\mathcal{H}$ is able to infer their (possibly polymorphic) types.

The reader may wonder at this point what this peculiar $\uparrow$ (lift) connective means. $\uparrow$ is itself polymorphic and maps a boolean value to a value of a predicate type; specifically, for each predicate type π, $\uparrow(\texttt{true}) = \top_\pi$ and $\uparrow(\texttt{false}) = \bot_\pi$. In other words, $\uparrow(\texttt{true})$ is a predicate of any type that succeeds given any set of parameters, and $\uparrow(\texttt{false})$ fails no matter what parameters it is given. At this point, one may question the usefulness of such a structure, but it will be become apparent when we try to convert our surface language to *poly*$\mathcal{H}$ in Chapter 5.

Additionally, we define goals as

$$\begin{array}{lcll}
\textit{Goal} & ::= & \texttt{false/0} \leftarrow_o E & \qquad\qquad\qquad\qquad \textit{goals}
\end{array}$$

In this case, we are using a monomorphic version of the reverse implication connective, as we demand that the expression functioning as a goal is of the boolean type. Also, until it becomes clear what a polymorphic goal could mean, all subexpressions of a goal must be monomorphic.

3.2 The Type System of *poly*$\mathcal{H}$

3.2.1 Types and Environments

The syntax of types and environments is defined in Figure 3.2, where α denotes argument type variables, ϕ denotes monomorphic type variables, m, m' and n are natural numbers with $m, m' \geq 0$ and $n \geq 1$. We distinguish three kinds of types: *argument* types, for characterizing arguments of predicates, as well as (monomorphic) *predicate* and *polymorphic* types, for characterizing predicates. Notice that we allow two kinds of polymorphism: over argument types and over (monomorphic) predicate types. The reason is that `i` is not an acceptable type for some polymorphic expressions, e.g. the arguments of $\wedge$. Environments contain two kinds of mappings: from argument variables to argument types ($V : \rho$) and from predicate symbols to (possibly) polymorphic types ($c/m : \tau$).

We will use π as an abbreviation for $\forall().\,\pi$ when $m = m' = 0$, in other words, we will treat predicate types as a subset of polymorphic types. Also, we will sometimes write ρ instead of (ρ) when $\rho \in \{\texttt{i}, \alpha, \texttt{o}, \phi\}$.

$$\begin{array}{llll}
\rho & ::= & \mathtt{i} \mid \alpha \mid \pi & \textit{argument types} \\
\pi & ::= & \mathtt{o} \mid \phi \mid (\rho_1, \ldots, \rho_n) \to \pi & \textit{(monomorphic) predicate types} \\
\tau & ::= & \forall(\alpha_1, \ldots, \alpha_m, \phi_1, \ldots, \phi_{m'}).\ \pi & \textit{polymorphic types} \\
\mathit{\Gamma} & ::= & \emptyset \mid \mathit{\Gamma}, V : \rho \mid \mathit{\Gamma}, c/m : \tau & \textit{type environments}
\end{array}$$

Figure 3.2: Types in *poly$\mathcal{H}$*

As a last comment on types, we have dropped the functional type for simplicity. Individual symbols, when in the functor of an application, assume the role of functional symbols.

3.2.2 Typing Rules

The type system of *poly$\mathcal{H}$* uses ML-style polymorphism [Miln78, Dama82]. We describe below an algorithm in the style of Hindley-Milner type inference, consisting of two steps: constraint generation and unification [Pier02, ch. 22].

We define constraints to be sets of equations between argument types.

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

The typing relation for programs is quite simple. Starting from an initial type environment Γ_0, which should contain all built-in predicates, we iterate through each declaration group in turn and augment the environment by adding all the predicates defined in the group. The final environment (Γ_n) contains all predicates defined in the program.

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

The typing relation for declaration groups is where constraints are first generated and then solved, using unification. A type environment Γ' is formed, containing the predicates defined in the declaration group. Each predicate is associated in Γ' with the most general type that is consistent with its arity m. This is accomplished with the auxiliary function $\mathsf{artyp}(m)$ which is defined as follows: $\mathsf{artyp}(0) = \mathtt{o}$; and $\mathsf{artyp}(n) = (\alpha_1, \ldots, \alpha_n) \to \phi$, if $n \ge 1$, where α_i and ϕ are fresh type variables.

Note that such a definition of artyp can never create a predicate of type $\forall(\phi).\phi$, the most general type imaginable. This is deliberate, as such a predicate could accept any number of arguments and would thus have no fixed arity. Additionally, we see that arity affects the type of a predicate only as far as the first set of arguments is concerned.

After we have built Γ', all rules are type checked in the environment $\Gamma \cup \Gamma'$ and, for each rule R_i, a set of constraints C_i is generated. The union of all constraints is then solved, using the function unify which will be defined in §3.2.3. If unification succeeds, a substitution s is obtained and applied to Γ', then the result is *generalized*.

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

Function gen takes an environment where the types of all predicates are monomorphic but may contain free type variables. These variables must be generalized, thus resulting in polymorphic predicate types.

The result of gen is a "generalized" environment, in which monomorphic types π have been replaced with polymorphic types $\forall(\alpha_1, \ldots, \alpha_m, \phi_1, \ldots, \phi_{m'}).\, \pi$, where $\alpha_1, \ldots, \alpha_m$ and $\phi_1, \ldots, \phi_{m'}$ are all the free type variables that occur in π.

The typing of rules is also straightforward. The only constraint here is that the type of the defining expression E must be the same as the type of the predicate in the environment.

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

Variables, constant symbols and predicate symbols generate no additional constraints and their typing rules contain no surprises. However, notice that whenever a polymorphic predicate symbol is used, the type variables in its type must be replaced by *fresh* type variables.

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

The auxiliary function freshen converts (possibly) polymorphic predicate types to monomorphic, by introducing fresh type variables; i.e., it replaces $\forall(\alpha_1, \ldots, \alpha_m, \phi_1, \ldots, \phi_{m'}).\, \pi$ with $s(\pi)$, where s is a substitution which maps $\alpha_1, \ldots, \alpha_m$ and $\phi_1, \ldots, \phi_{m'}$ to fresh variables $\alpha'_1, \ldots, \alpha'_m$ and $\phi'_1, \ldots, \phi'_{m'}$, respectively.

There are two typing rules for application. The first handles the application of functional symbols, whereas the second handles all other kinds of application. Notice that functional symbols take arguments of type `i` and return a result of type `i`.

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

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

In the typing rule for abstraction, the environment is extended with the function's arguments. The constraint requires that the type of the abstraction's body is a predicate type.

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

The remaining typing rules correspond to conjunction, disjunction, unification, existentials and liftings. Notice that unification takes place between operands of type `i`, whereas conjunction, disjunction, existentials and liftings are essentially polymorphic on any predicate type.

$$\frac{\begin{array}{c}\Gamma \vdash E_1 : \rho_1 \mid C_1 \qquad \Gamma \vdash E_2 : \rho_2 \mid C_2 \\ \phi \text{ is a fresh type variable}\end{array}}{\Gamma \vdash E_1 \wedge E_2 : \phi \mid \{\phi = \rho_1, \phi = \rho_2\} \cup C_1 \cup C_2} \qquad \frac{\begin{array}{c}\Gamma \vdash E_1 : \rho_1 \mid C_1 \qquad \Gamma \vdash E_2 : \rho_2 \mid C_2 \\ \phi \text{ is a fresh type variable}\end{array}}{\Gamma \vdash E_1 \vee E_2 : \phi \mid \{\phi = \rho_1, \phi = \rho_2\} \cup C_1 \cup C_2}$$

$$\frac{\Gamma \vdash E_1 : \rho_1 \mid C_1 \qquad \Gamma \vdash E_2 : \rho_2 \mid C_2}{\Gamma \vdash E_1 \approx E_2 : \mathtt{o} \mid \{\rho_1 = \mathtt{i}, \rho_2 = \mathtt{i}\} \cup C_1 \cup C_2} \qquad \frac{\begin{array}{c}\Gamma, V : \alpha \vdash E : \rho \mid C \\ \alpha \text{ and } \phi \text{ are fresh type variables}\end{array}}{\Gamma \vdash \exists V.\, E : \phi \mid \{\phi = \rho\} \cup C}$$

$$\frac{\begin{array}{c}\Gamma \vdash E : \rho \mid C \\ \phi \text{ is a fresh type variable}\end{array}}{\Gamma \vdash \uparrow(E) : \phi \mid \{\rho = \mathtt{o}\} \cup C}$$

Finally, we give the rule for goals. It demands that the body of the goal is of the boolean type:

$$\frac{\Gamma \vdash E : \rho \mid C}{\Gamma \vdash \texttt{false/0} \leftarrow_o E \mid \{\rho = \texttt{o}\} \cup C}$$

3.2.3 The Unification Algorithm

The algorithm that we use for solving type constraints is a simple adaptation of standard Hindley-Milner unification. A substitution s is a function mapping an argument type ρ to an argument type ρ', by replacing free occurrences of type variables (ϕ or α) with types (π and ρ, respectively). Composition of substitutions, denoted as $s_1 \circ s_2$ is defined as $(s_1 \circ s_2)(\rho) = s_1(s_2(\rho))$. By id we denote the identity substitution, leaving types unchanged. By $[\phi \mapsto \pi]$ and $[\alpha \mapsto \rho]$ we denote singleton substitutions that replace all free occurrences of a single type variable with a specific type. Abusing notation a bit, we extend the application of substitutions to constraints and environments: we denote by $s(C)$ and $s(\Gamma)$ the result of applying substitution s to all types in C and Γ, respectively.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{unify}(\emptyset) \;=\; \mathsf{id}} \\
\multicolumn{2}{l}{\mathsf{unify}(\{\rho_1 = \rho_2\} \cup C) \;=} \\
\quad \mathsf{unify}(C) & \text{if } \rho_1 = \rho_2 \\
\quad \mathsf{unify}([\alpha \mapsto \rho_2]C) \circ [\alpha \mapsto \rho_2] & \text{if } \rho_1 = \alpha \text{ and } \alpha \text{ does not appear in } \rho_2 \\
\quad \mathsf{unify}([\alpha \mapsto \rho_1]C) \circ [\alpha \mapsto \rho_1] & \text{if } \rho_2 = \alpha \text{ and } \alpha \text{ does not appear in } \rho_1 \\
\quad \mathsf{unify}([\phi \mapsto \pi]C) \circ [\phi \mapsto \pi] & \text{if } \rho_1 = \phi,\, \rho_2 = \pi \text{ and } \phi \text{ does not appear in } \pi \\
\quad \mathsf{unify}([\phi \mapsto \pi]C) \circ [\phi \mapsto \pi] & \text{if } \rho_2 = \phi,\, \rho_1 = \pi \text{ and } \phi \text{ does not appear in } \pi \\
\quad \mathsf{unify}(C \cup \{\rho_1^i = \rho_2^i \mid 1 \leq i \leq n\} \cup \{\pi_1 = \pi_2\}) & \text{if } \rho_1 = (\rho_1^1, \ldots, \rho_1^n) \to \pi_1 \text{ and } \rho_2 = (\rho_2^1, \ldots, \rho_2^n) \to \pi_2 \\
\quad \text{type error} & \text{otherwise}
\end{array}
$$

Figure 3.3: Unification

Our unification algorithm differs from that of Robinson [Robi65], in that it handles two kinds of type variables. Since argument type variables are more general (and thus carry less information) they are substituted out first; the predicate type variables can then only be unified with a predicate type. The use of a different unification algorithm poses the need for a different proof of the principality of the resulting types than the one used in [Dama82]. This, however, is outside the scope of this thesis and is left for future work.

Chapter 4

*poly*Hopes: a Higher-Order Extension of Prolog

So far, our discussion has concerned the theoretical foundation of a higher order logic programming language, but we have not yet explained how our framework is associated with a superset of Prolog. Indeed, the examples of *poly*$\mathcal{H}$ code given in the previous chapter do not resemble Prolog at all. In this chapter, we present *poly*Hopes (*polymorphic* Hopes), a prototype interpreter for a higher-order extension of Prolog that attempts to satisfy the Prolog standard in its integrity, while integrating higher-order features in a smooth way. The design of this language is partly influenced by the syntax of Erlang [Eric13, Cesa09], a functional language which was in turn originally inspired by Prolog.

In this chapter, we will be using the `verbatim` font for terminal grammar symbols of the *poly*Hopes language, and the *italic* font for nonterminal.

4.1 Lexical Conventions of *poly*Hopes

*poly*Hopes tries to imitate Prolog's lexical conventions as much as possible.

Comments are enclosed in `/*` and `*/`. An one-line comment starts with `%`. The lexical structure of variables and constants is given below. *upper*, *lower* and *alphaNum* denote uppercase letters, lowercase letters and alphanumeric characters respectively. *char* stands for any character, and *escaped* for an escaped character sequence. Braces denote choice of one of the enclosed characters. As in Prolog, constants (or atoms) can be graphical or alphanumeric identifiers, the `!` (cut) constant, or any string within single quotes. The only reserved word in *poly*Hopes is `pred`. Additionally, `\~`, `=>` and `|` are reserved identifiers.

V	::=	(_ | *upper*)(_ | *alphaNum*)*	*variables*
c	::=	*graphicIdent* | *alphaIdent* | *stringLiteral* | `!`	*constants*
graphicIdent	::=	{`.#$&*+-;/:<=>?@^~|`}+	*graphic identifiers*
alphaIdent	::=	*lower* (*alphanum* | _)*	*identifiers*
stringLiteral	::=	`'`(*char* | *escaped*)*`'`	*string literals*

Figure 4.1: Lexical conventions in *poly*Hopes

4.2 The Syntax of *poly*Hopes

4.2.1 Syntax of Expressions

The syntax of expressions is given below. Again, *V* stands for variables, *c* for constants, *num* for unsigned integer or floating point constants, n, m for natural numbers with $m \geq 1$. *prefixop*, *postfixop* and *infixop* are operators (special constants – see below). An expression enclosed in square brackets ([ ]) is optional.

$$\begin{array}{lcll} E & ::= & V \mid c \mid num \mid \texttt{pred}\, c\, [\texttt{/}m] \mid E\texttt{(}E_1\texttt{, ... ,}\, E_n\texttt{)} & \textit{expressions} \\ & \mid & \textit{prefixop}\ E \mid E\ \textit{postfixop} \mid E_1\ \textit{infixop}\ E_2 & \\ & \mid & \texttt{\textbackslash\~{}(}V_1\texttt{, ... ,}\, V_n\texttt{)=>}\, E \mid \textit{list} \mid \texttt{(}E\texttt{)} & \\ \textit{list} & ::= & \texttt{[]} \mid \texttt{[}E_1\texttt{, ... ,}\, E_n\, [\, \texttt{|}(V \mid \textit{list})\,]\texttt{]} & \textit{lists} \end{array}$$

Figure 4.2: The syntax of expressions in *poly*HOPES

Following Prolog's style, predicate and functional symbols are taken from the same alphabet, and the interpreter takes up the task to decide which constant symbol refers to which entity. However, the default behavior sometimes has to be overridden, and the programmer needs to force a constant to be interpreted as a predicate to ensure that higher-orderness is indeed available. Thus, `pred` $c[/n]$ stands for a predicate constant with the name c. The programmer has the option of specifying the predicate's arity themselves; if they do not, it is inferred from the context. The reader may object that the necessity of this construct admits *poly*HOPES's incompatibility with Prolog. Hopefully, we will convince them otherwise in Chapter 5. Erlang uses a similar construct to pass functions as parameters, albeit with the `fun` keyword instead [Eric13, Cesa09].

Operators in *poly*HOPES are just constants that have been declared as such through the special `op` directive, as done in ordinary Prolog. For example,

```
:- op(500, yfx, '+').
```

tells the interpreter that + can now be used as an infix operator with precedence 500 and left associativity.

*poly*HOPES has no special syntax for conjunction, disjunction, or unification, as these are just ordinary operators, written `','`, `';'` and `'='` respectively. When a comma-separated list of expressions is expected (in the arguments of an application or the initial elements of a list) a comma (`,`) is interpreted as a delimiter rather than an operator. The user can of course still use it as an operator by enclosing the expression in parentheses. Also, there is nothing equivalent to the existential or lift structures of $poly\mathcal{H}$ in the surface language; these structures will be created during the transformation to $poly\mathcal{H}$ as needed.

One minor difference from ordinary Prolog is that we restrict list tails to be either a list or a variable. Anything else would (certainly) create an *improper list* (a list whose tail is not itself another list).

Finally, the delimiters of λ-abstraction have been chosen so as to not coincide with standard Prolog operators.

4.2.2 Syntax of Sentences and Programs

A sentence is either a clause, a goal or a directive. A program is a nonempty set of sentences.

$$\begin{array}{lcll} \textit{Sentence} & ::= & \textit{Clause} \mid \textit{Goal} \mid \textit{Dir} & \textit{sentences} \\ \textit{Program} & ::= & \textit{Sentence}^{+} & \textit{programs} \end{array}$$

Figure 4.3: Sentences and programs in *poly*HOPES

A clause in *poly*HOPES partly defines a predicate. Multiple clauses may contribute to the definition of the same predicate. Clauses allow predicates to be defined with parameter notation, as done in Prolog, in addition to the λ-abstraction notation. Clauses contain a clause head and an optional clause body. The body, if present, is a proper expression. The head, however, has a more restricted syntax:

Clause	::=	*Head* [*Impl Body*] `.`	*clauses*
Head	::=	*AppHead* | `c/n`	*clause head*
AppHead	::=	*c* | *AppHead*`(`E_1`, ... ,` E_n`)`	
Impl	::=	`:-` | `<-`	*reverse implication*
Body	::=	*E*	*clause body*

Figure 4.4: The syntax of clauses in *poly*Hopes

A clause head in *poly*Hopes is more complex than in ordinary Prolog. The first possibility to define it, *AppHead*, allows for more than one sets of arguments. This is necessary to define higher-order predicates. Each of these arguments will later be used as a parameter of a λ-abstraction if it is a variable, or it will be unified with such a parameter if it is a more complex structure (a *pattern*). And since unification is provided only for the individual type, there is no point of having predicates in clause heads. Therefore, λ-abstractions and `pred` structures are disallowed in clause heads.

We have the choice to use either `<-` or `:-` as the reverse implication symbol in a clause. The first one corresponds to the $\leftarrow$ connective in *poly*$\mathcal{H}$. The second one is a monomorphic version thereof, $\leftarrow_{\mathtt{o}}$, which demands that both the head and the body of the clause are of type `o`. The first symbol is more flexible, but the second one has the same semantics as in Prolog and keeps better compatibility. Also, it is often more intuitive to have the clauses one defines return `true` or `false` than a higher-order predicate.

If the programmer wants to define a predicate with the use of `<-` and without using any parameters (e.g. with partial application of another predicate), they have the option to do so by using just the predicate name along with its arity (the second option to define a clause head).

We will call a clause without an empty body a *fact*, and one with a nonempty body a *rule*.

A goal is just a boolean typed expression introduced with the symbol `?-`. A directive is an expression introduced with `:-` .

Goal	::=	`?-` *E*`.`	*goal*
Dir	::=	`:-` *E*`.`	*directive*

Figure 4.5: Goals and directives in *poly*Hopes

Directives do not have the same semantics as other expressions. They are not typed and are used only to provide the compiler with additional information on how to handle the program, e.g. which constants may be used as operators, or which predicates may accept new clauses dynamically.

4.3 Implementation of *poly*Hopes

An implementation of *poly*Hopes is under development and publicly available at [Kouk13]. So far it has had some successes, such as successfully parsing and type checking a 10.000-line program in Prolog, with some minor problems that will be better described in Section 7.2.

Chapter 5

Transforming *poly*HOPES to *poly*$\mathcal{H}$

In the previous two chapters, we defined *poly*$\mathcal{H}$, a language with a simple, elegant syntax and a well-defined type system, and *poly*HOPES, which is syntactically a superset of Prolog, but for which we have yet not specified its meaning.

In this chapter, we are establishing how

- we decide the meaning (predicate or functional) and arity of a constant,
- we split program sentences into dependency groups and finally
- we map each syntactic structure of *poly*HOPES to the corresponding one of *poly*$\mathcal{H}$.

5.1 Inferring the Meaning and Arity of Constants

As mentioned before, *poly*HOPES follows Prolog's footsteps and takes both its functional and predicate constants from the same alphabet. So the first thing we have to do is decide the meaning of constants in our syntax tree, keeping in mind that we would like to keep our language compatible with Prolog if possible.

In Prolog, a simple rule can determine what a constant refers to: since Prolog is first-order, the "topmost functor" of a clause or goal body, that is, the constant outside all parentheses and besides the `:-` operator, is interpreted as a predicate.

More formally, by *topmost functor* of an expression *E* we are referring to the constant we find by starting with the syntax tree of *E* and moving to the functor of an application or the operator part of an operator application until we end up with a single constant (if we do).

On the other hand, any constant that is found inside an argument is interpreted as a functional symbol; otherwise we would have a predicate passed to another predicate as an argument, which is forbidden in first-order logic.

This rule, though, must admit some exceptions: constants in the scope of some specific operators should be considered predicates themselves. For a start, we will choose these *special operators* to be `','/2` (conjunction) and `';'/2` (disjunction). We could also include `'->'/2` (implication) and `'\+'/1` (negation-as-failure), once we have determined how they can be defined in our language. Bear in mind though that all special operators can be encountered as functional symbols too (e.g. in the expression `p(q;r)`); in that case, their arguments are no predicates either.

Unfortunately, the above assumption about Prolog is not always valid. For example,

```
p :- X, q(s(X)).
```

is legal Prolog. According to what we have said so far, `s` must be considered a functional symbol here, thus `X` is of type `i`. However, we can see that `X` also appears as an operand of the `','` predicate, and must thus also have a predicate type; it follows that the above program is mistyped. However, our thoughts are still of some value, as the above clause can actually be written equivalently as

```
p :- call(X), q(s(X)).
```

with `call/1` being the familiar Prolog *meta-predicate*, which in *poly*HOPES has type $\texttt{i} \rightarrow \texttt{o}$. Similarly, something like

```
w(s(X)) :- \+ X.
```

can be written

```
w(s(X)) :- \+ call(X).
```

So with minimal syntactic changes to a Prolog program and no changes at all to its semantics, we can ensure that our assumption is indeed correct. From now on, let us assume that these changes have been made to Prolog programs, whether by hand or by an automated script.

This is a good starting point as to how to analyze our *poly*HOPES source and extract information about constants, while respecting the compatibility with Prolog. To the above observations, we have to add that the topmost functor of the body of a λ-abstraction is naturally considered a predicate as well.

However, the point of higher-orderness is that we sometimes do want to pass predicates as arguments. Fortunately, the programmer can overcome the default behavior by using the `pred` construct: `pred c/n` always denotes the predicate c/n. Notice that a Prolog file, which lacks this syntactic extension, is still interpreted correctly, because predicates are not found in arguments and thus this structure is unneeded.

To sum it up, a constant or operator is interpreted as a predicate when

- It is the topmost functor of a clause body, clause head, goal, or λ-abstraction body,
- It is the topmost functor of an operand of a special operator, and this special operator is interpreted as a predicate, or
- It is defined with the aid of the `pred` structure.

When we have decided that a constant is actually a predicate, we must also specify its arity. This part is easier: if a constant is explicitly given the arity n by the programmer (in a clause head or a `pred` construct), then its arity is n. If it does not have its arity explicitly specified, we have to infer it automatically: if a constant is the functor of an application with n arguments, or an operator with n operands, it has arity n. Otherwise, its has arity 0.

We can run through the syntax tree of all expressions of a *poly*HOPES program and annotate each constant and operator with a flag (predicate or functional) and a natural number (arity), which we will later extract and use in further analysis, as seen in the following sections.

5.2 Dependency Analysis

Now that we have found which constants in our clauses are really predicates, it is time to do a dependency analysis to separate the program into *dependency groups*. Dependency groups are minimal sets of (necessarily) mutually recursive predicates. As you may recall from Chapter 3, constraint generation and unification takes place in each such group separately in dependency order, and the order of the predicates within each group is irrelevant. This is necessary to achieve maximum polymorphism.

To explain why it is not optimal to generate and solve constraints for the entire program at once, let us consider the following example, written in *poly*$\mathcal{H}$:

$$\begin{aligned}&\texttt{apply/2} && \leftarrow \lambda(\texttt{X},\texttt{Y}).\ \texttt{X}(\texttt{Y})\\ &\texttt{isZero/1} && \leftarrow \lambda(\texttt{X}).\ \texttt{apply/2}(\lambda(\texttt{Y}).\texttt{Y} \approx \texttt{0},\ \texttt{X})\end{aligned}$$

Here, if we skip the dependency analysis and try to solve constraints generated from both rules together, we will end up with `apply/2` :: $(\texttt{i} \rightarrow \texttt{o},\ \texttt{i}) \rightarrow \texttt{o}$, which is not the most general type possible.

Some ideas found in this section are borrowed from Haskell, another language that forms dependency groups automatically [Marl10].

5.3 Mapping *poly*$\mathcal{H}$ to *poly*HOPES

Having done the preparatory work, we are now ready to transform our program written in *poly*HOPES to *poly*$\mathcal{H}$.

5.3.1 Transforming Expressions

Let us begin with the mapping of expressions. In Figure 5.1 we define a function, transformE, that takes an expression in *poly*HOPES and returns one in *poly*$\mathcal{H}$. Essentially, what it does is remove syntactic sugar (operators, lists etc.), introduce the special $\wedge$, $\vee$ and $\approx$ structures, and transform constants that are interpreted as predicates in *poly*HOPES to the respective *poly*$\mathcal{H}$ predicate constants. Numbers are left as they are (they are simply a subset of individual symbols). The empty list is left as $[\,] \in \texttt{i}$, and the "cons" functional symbol is written as a dot (.), as in Prolog. We do not give a separate rule for "cut" (!), as it behaves like any other constant.

The functions arity(e) and isPred(e) extract the information that we annotated on the syntax tree earlier. Finally, transformApp is a function that maps an application of the predicates `','/2`, `';'/2` and `'='/2` to their special meanings, and returns an ordinary application in all other cases. Thus

$$\begin{aligned}\mathsf{transformApp}(\texttt{','/2}, \{E_1, E_2\}) &= E_1 \wedge E_2\\ \mathsf{transformApp}(\texttt{';'/2}, \{E_1, E_2\}) &= E_1 \vee E_2\\ \mathsf{transformApp}(\texttt{'='/2}, \{E_1, E_2\}) &= E_1 \approx E_2\\ \mathsf{transformApp}(E, \{E_1,\ \dots\ , E_n\}) &= E(E_1,\ \dots\ , E_n),\ \text{for all other functors } E\end{aligned}$$

5.3.2 Transforming Clauses

Transformation of clauses is the tricky part of converting our source code to *poly*$\mathcal{H}$ code correctly. The first thing we have to do is to find which predicate is defined by each clause. If the clause is in the form

$$c(E_{1,1},\ \dots\ , E_{1,n_1}) \dots (E_{k,1},\ \dots\ , E_{k,n_k})\ \ \texttt{<-}\ \ \mathit{Body}.$$

then we can see that the arity of the predicate is n_1 so we are defining c/n_1. If the clause is in the form

$$
\begin{array}{lll}
\mathsf{transformE}(V) & = V & \\
\mathsf{transformE}(c) & = c/n & \text{, if } \mathsf{isPred}(c) \text{ and } n = \mathsf{arity}(c) \\
 & = c & \text{, otherwise} \\
\mathsf{transformE}(num) & = num & \text{, } (num \in \mathtt{i}) \\
\mathsf{transformE}(\mathtt{pred}\ c[/n]) & = c/n' & \text{, if } n' = \mathsf{arity}(\mathtt{pred}\ c[/n]) \\
\mathsf{transformE}(E\mathtt{(}E_1\mathtt{,}\ \ldots\ \mathtt{,}\ E_n\mathtt{)}) & = \mathsf{transformApp}(E', \{E'_1,\ \ldots\ , E'_n\}), & E'_i = \mathsf{transformE}(E_i), \\
 & & E' = \mathsf{transformE}(E) \\
\mathsf{transformE}(\mathit{prefixop}\ E) & = \mathsf{transformApp}(op', \{E\}) & \text{, } op' = \mathsf{transformE}(\mathit{prefixop}) \\
 & & E' = \mathsf{transformE}(E) \\
\mathsf{transformE}(E\ \mathit{postfixop}) & = \mathsf{transformApp}(op', \{E\}) & \text{, } op' = \mathsf{transformE}(\mathit{postfixop}) \\
 & & E' = \mathsf{transformE}(E) \\
\mathsf{transformE}(E_1\ \mathit{infixop}\ E_2) & = \mathsf{transformApp}(op', \{E'_1, E'_2\}) & \text{, } op' = \mathsf{transformE}(\mathit{infixop}) \\
 & & E'_i = \mathsf{transformE}(E_i) \\
\mathsf{transformE}(\mathtt{\backslash\tilde{}(}V_1\mathtt{,}\ \ldots\ \mathtt{,}\ V_n\mathtt{)=>}E) & = \lambda(V_1,\ \ldots\ , V_n).E' & \text{, } E' = \mathsf{transformE}(E) \\
\mathsf{transformE}(\mathtt{[]}) & = [\,] & \\
\mathsf{transformE}(\mathtt{[}E_1\mathtt{,}\ \ldots\ \mathtt{,}\ E_n\ \mathtt{|}\ \mathit{tail}\mathtt{]}) & = .(E_1, .(E_2, \ldots\ .(E_n, \mathit{tail}')...)) & \text{, } \mathit{tail}' = \mathsf{transformE}(\mathit{tail}) \\
\mathsf{transformE}(\mathtt{[}E_1,\ \ldots\ , E_n\mathtt{]}) & = .(E_1, .(E_2, \ldots\ .(E_n, [\,])...)) & \\
\mathsf{transformE}(\ \mathtt{(}E\mathtt{)}\ ) & = \mathsf{transformE}(E) &
\end{array}
$$

Figure 5.1: Transforming a *poly*HOPES expression to *poly*$\mathcal{H}$

c/n `<-` *Body*.

then we are defining c/n, and if we have

c `<-` *Body*.

it would be $c/0$.

We can break the transformation down to three tasks:

- Transform parameters to lambda abstractions
- Insert extra unification clauses where needed
- Insert existential quantifications if needed

λ-abstraction parameters can only be (unique) variables. Therefore, there are two cases where we cannot directly use a clause argument as a parameter in the resulting λ-abstraction: if the parameter appears a second time, or if it is not a variable; in the latter case, it is called a *pattern*. In both cases, what is implied is an additional unification: in the first case, the two synonym variables have to be unified, and in the second, the parameter of the λ-abstraction has to be unified with the original pattern in the argument. So if our initial clause is

$$\mathit{Clause}\ ::=\ c(E_{1,1},\ \ldots\ , E_{1,n_1}) \ldots (E_{k,1},\ \ldots\ , E_{k,n_k})\ \mathtt{<\text{-}}\ \mathit{Body}\ .$$

it will be transformed to

$$\lambda(V_{1,1},\ \ldots\ ,V_{1,n_1}).\ \ldots\ \lambda(V_{k,1},\ \ldots\ ,V_{k,n_k}).\ \mathit{Body}'$$

where

$$V_{i,j} \quad = \quad \begin{cases} E_{i,j} & \text{if } E_{i,j} \text{ is a } \textbf{variable} \text{ encountered for the first time,} \\ \text{fresh variable} & \text{otherwise} \end{cases}$$

Body′, in turn, should contain a conjunction of unifications of each newly introduced variable with its respective pattern (or variable) in the original clause. However, to make sure there is no type mismatch between these unifications (of type `o`) and the clause body (of any predicate type), we should *lift* the unifications before conjoining them with the original body, using the special connective $\uparrow$:

$$\mathit{Body}' \quad \equiv \quad \mathsf{transformE}(\mathit{Body}) \wedge (\ \uparrow(U_1 \wedge U_2 \wedge \ldots \wedge U_e)\ )$$

where

$$\{U_i\} \quad = \quad \{V_{i,j} \approx \mathsf{transformE}(E_{i,j}) \mid V_{i,j} \text{ is a freshly introduced variable}\}$$

In case our clause is a fact (it has no body), we take the initial $\mathit{Body} = \texttt{true/0}$.

If the original *poly*Hopes clause uses the monomorphic `:-` connective, we must somehow restrict the type of *Body* to be boolean. If our implementation supports no annotations that could help the interpreter know about the limitation, we can achieve that in a less elegant way, by inserting a conjunction with `true/0`. In that case, since the type of the original *Body* is surely `o`, we can drop $\uparrow$. So in that case

$$\mathit{Body}' \quad \equiv \quad \mathsf{transformE}(\mathit{Body}) \wedge (U_1 \wedge U_2 \wedge \ldots \wedge U_e) \wedge (\texttt{true/0})$$

However, our final expression is still incomplete, as it may still have some free variables. So we have to introduce an existential for each variable that is still left free. The most natural way to do it is between the *Body*′ and the newly introduced λ-abstractions.

So to sum it all up, let us define a function, transformClause, which takes a clause and returns an expression in *poly*$\mathcal{H}$ which partly defines the predicate in question:

$$\mathsf{transformClause}(\mathit{Clause}) \quad = \quad \lambda(V_{1,1},\ \ldots\ ,V_{1,n_1}).\lambda(V_{k,1},\ \ldots\ ,V_{k,n_k}).\exists V^E_1.\ldots\exists V^E_f.\mathit{Body}'$$

with *Body*′ and $V_{i,j}$ as defined above, and $\{V^E_1,\ldots,V^E_f\}$ are the free variables in *Body*′ excluding $\{V_{i,j}\}$.

Finally, if $\mathit{Clause}_1,\ldots,\mathit{Clause}_n$ define the predicate c/n, the respective predicate in *poly*$\mathcal{H}$ will be defined by a disjunction of all Clause_i:

$$c/n \ \leftarrow \ \mathsf{transformClause}(\mathit{Clause}_1) \vee \ldots \vee \mathsf{transformClause}(\mathit{Clause}_n)$$

5.3.3 Transforming Rules and Directives

A rule is transformed by simply calling transformE on the expression defining it. In the current version of *poly*HOPES, directives do not translate to *poly*$\mathcal{H}$. Even though they have a similar syntax to the other two kinds of sentences, they are just used as untyped compiler directives.

5.4 Examples of Transformations

To make the procedure more concrete, let us now present some examples of how predicate definitions written in the *poly*HOPES source language are transformed into *poly*$\mathcal{H}$. Let us begin with our original *poly*$\mathcal{H}$ example:

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

Firstly, we have to infer the meaning and arities of constants in the bodies of the rules. Here we have only two constants, both in the second clause: firstly, `','` is the topmost functor of a clause body, so it is a predicate. It is applied on two operands, so it stands for `','/2`. Then, `closure` is the topmost functor of `closure(R)(Z, Y)`, an operand of the `','/2` special predicate. Because it is additionally applied on one operand, `R`, it stands for `closure/1`.

Now let us proceed with the transformation. To avoid needless repetition, let's call the two clauses $Clause_1$ and $Clause_2$ respectively. $Clause_1$ is relatively simple: we have to add no extra unifications, but we are using the monomorphic `:-`. So here

$$Body'_1 \quad = \quad \mathtt{R}(\mathtt{X},\mathtt{Y}) \wedge (\mathtt{true}/0)$$

and

$$\text{transformClause}(Clause_1) \quad = \quad \lambda(\mathtt{R}).\ \lambda(\mathtt{X},\mathtt{Y}).\ \mathtt{R}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{true}/0$$

$Clause_2$ is a bit more complicated. Firstly, we see that we have an application of `','/2`, which will become a $\wedge$ connective. Secondly, `Z` is now a variable which is not bound in a λ-abstraction, so we have to insert an extra existential. So working as before we get

$$\text{transformClause}(Clause_2) \quad = \quad \lambda(\mathtt{R})\ .\lambda(\mathtt{X},\ \mathtt{Y}).\ \exists\mathtt{Z}.\ \mathtt{R}(\mathtt{X},\mathtt{Z}) \wedge \mathtt{closure}/1(\mathtt{R})(\mathtt{Z},\mathtt{Y}) \wedge \mathtt{true}/0$$

Finally, we disjoin the two clauses defining `closure/1` to create its complete definition:

$$\begin{aligned}\mathtt{closure}/1 \ \leftarrow\ & (\ \lambda(\mathtt{R})\ .\lambda(\mathtt{X},\mathtt{Y}).\ \mathtt{R}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{true}/0\ ) \vee \\ & (\ \lambda(\mathtt{R})\ .\lambda(\mathtt{X},\mathtt{Y}).\ \exists\mathtt{Z}.\ \mathtt{R}(\mathtt{X},\mathtt{Z}) \wedge \mathtt{closure}/1(\mathtt{R})(\mathtt{Z},\mathtt{Y}) \wedge \mathtt{true}/0\ )\end{aligned}$$

The only difference that this predicate has to the one given in Chapter 3 is the extra `true/0` conjunctions which restrict its return type to `o` instead of the more general (but less intuitive) $\forall\phi.\phi$.

Now if we define

```
ancestor/1 <- closure(pred parent/2).
```

we can easily see that the defined predicate is `ancestor/1`, and the constants referred to in the body are predicates (one because of its position, and the other because of the `pred` keyword). Also, `pred parent/2` has explicitly been given the arity 2. So the resulting definition in *poly*$\mathcal{H}$ would be

$$\texttt{ancestor/1} \leftarrow \texttt{closure/1}(\texttt{parent/2})$$

Finally, let us define a more "exotic" predicate which showcases the other tools we have at our disposal. Let's say a programmer with strange taste defines

```
strange(s(N)) <- pred closure/1.
```

Here the transformation of the body is trivial. The parameter of the head, however, is a pattern. Thus it will be replaced with a fresh variable (say $\texttt{N}'$) and force us to add an extra unification to the resulting body. Notice that here, the use of the $\uparrow$ connective is necessary, because the body of the clause is not boolean. Also, the resulting expression ends up with a free variable ($\texttt{N}$), which has to be existentially quantified:

$$\texttt{strange/1} \leftarrow \lambda(\texttt{N}').\exists \texttt{N}.\texttt{closure/1} \wedge (\uparrow (\texttt{N}' \approx \texttt{s}(\texttt{N})))$$

Chapter 6

Examples

To illustrate how our language looks like, we now give some more examples written in *poly*Hopes. Some are accompanied by their respective version in *poly*$\mathcal{H}$. Also, we are indicating a possible behavior of the `call/1` meta-predicate.

6.1 First-order List Predicates

Firstly, let us illustrate that we can express familiar first-order list predicates in *poly*Hopes just as we can in Prolog. Notice that we are defining two (distinct) synonym predicates with different arity, `reverse/2` and `reverse/3`.

```
length([], 0).
length([X|L], N2) :- length(L, N), N2 is N+1.

append( [], L, L ).
append( [ X | L1 ], L2, [ X | L3 ] ) :- append( L1, L2, L3 ).

reverse( L, R ) :- reverse( L, [], R ).
reverse( [], R, R ).
reverse( [ X | Xs ], L, R ) :- reverse( Xs, [ X | L ], R ).
```

These predicates are written in *poly*$\mathcal{H}$

$$
\begin{array}{lll}
\texttt{length/2} & \leftarrow & (\ \lambda(\texttt{V}_1,\texttt{V}_2).\ \texttt{V}_2 \approx \texttt{0} \wedge \texttt{V}_1 \approx [\,]\ ) \vee \\
 & & (\ \lambda(\texttt{V}_1,\texttt{N}_2).\ \exists\texttt{X}.\exists\texttt{L}.\exists\texttt{N}.\ \texttt{V}_1 \approx .(\texttt{X},\texttt{L}) \wedge \texttt{length/2}(\texttt{L},\texttt{N}) \wedge \texttt{is/2}(\texttt{N}_2,+(\texttt{N},\texttt{1}))\ ) \\
\texttt{append/3} & \leftarrow & (\ \lambda(\texttt{V}_1,\texttt{L},\texttt{V}_2).\ \texttt{L} \approx \texttt{V}_2 \wedge \texttt{V}_1 \approx [\,]\ ) \vee \\
 & & (\ \lambda(\texttt{V}_1,\texttt{L}_2,\texttt{V}_2).\ \exists\texttt{X}.\exists\texttt{L}_3.\exists\texttt{L}_1. \\
 & & \quad \texttt{V}_2 \approx .(\texttt{X},\texttt{L}_3)\ \wedge\ \texttt{V}_1 \approx .(\texttt{X},\texttt{L}_1)\ \wedge\ \texttt{append/3}(\texttt{L}_1,\texttt{L}_2,\texttt{L}_3) \\
 & &) \\
\texttt{reverse/2} & \leftarrow & \lambda(\texttt{L},\texttt{R}).\ \texttt{reverse/3}(\texttt{L},[\,],\texttt{R}) \\
\texttt{reverse/3} & \leftarrow & (\ \lambda(\texttt{V}_1,\texttt{R},\texttt{V}_2).\ \texttt{R} \approx \texttt{V}_2 \wedge \texttt{V}_1 \approx [\,]\ ) \vee \\
 & & (\ \lambda(\texttt{V}_1,\texttt{L},\texttt{R}).\ \exists\texttt{X}.\exists\texttt{Xs}.\ \texttt{V}_1 \approx .(\texttt{X},\texttt{Xs}) \wedge \texttt{reverse/3}(\texttt{Xs},.(\texttt{X},\texttt{L}),\texttt{R})\ )
\end{array}
$$

For better readability, we have indexed the numbers in the variable names, and written newly introduced variables as $\texttt{V}_i$. Also, we have skipped superficial conjunctions with `true/0`.

The type checker outputs the following types for these predicates:

```
length/2  :: (i, i) -> o
append/3  :: (i, i, i) -> o
reverse/2 :: (i, i) -> o
reverse/3 :: (i, i, i) -> o
```

6.2 Higher-order List Predicates

The point of *poly*HOPES is to offer a higher level of abstraction than Prolog through higher-orderness. So let us now see some familiar higher-order predicates from functional programming implemented in *poly*HOPES.

Of course we could start with none other than `map/1`, written here in curried form:

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

The type of `map/1` is found by the type checker to be

```
map/1 :: ((i, i) -> o) -> (i, i) -> o.
```

In *poly*$\mathcal{H}$, `map/1` is written

$$\begin{aligned}
\texttt{map/1} \leftarrow\; & (\ \lambda(\texttt{R}).\ \lambda(\texttt{V}_1,\texttt{V}_2).\ \texttt{V}_2 \approx [\,] \wedge \texttt{V}_1 \approx [\,]\ ) \vee \\
& (\ \lambda(\texttt{R}).\ \lambda(\texttt{V}_1,\texttt{V}_2).\ \exists\texttt{Y}.\ \exists\texttt{Ys}.\ \exists\texttt{X}.\ \exists\texttt{Xs}. \\
& \qquad \texttt{V}_2 \approx .(\texttt{Y},\texttt{Ys}) \wedge \texttt{V}_1 \approx .(\texttt{X},\texttt{Xs}) \wedge \texttt{R}(\texttt{X},\texttt{Y}) \wedge \texttt{map/1}(\texttt{R})(\texttt{Xs},\texttt{Ys}) \\
&)
\end{aligned}$$

Now using `map/1` we could write a predicate, `samelength/2`, which, as the name suggests, succeeds for predicates of the same length. We are giving it in two forms, one defined with partial application, and one in a more traditional way. To do that we are using an auxiliary anonymous predicate (defined through a λ-abstraction) that succeeds for every pair of inputs. Hopefully, this example, though curiously defined, will showcase the usefulness of `<-`:

```
sameLength/2     <- map(\~(X,Y) => true).
sameLength2(X,Y) :- map(\~(X,Y) => true)(X,Y).
```

Another familiar function on lists are folds. Folds start with an initial element and traverse through a list, each time performing an operation on the previous result and the current list element, and return the final result. A generic version of a left fold for logic programming is given below:

```
foldl(F, Z)([],Z).
foldl(F, Y0)([X|Xs], Z) :-
    F(Y0, X, Y1),
    foldl(F, Y1)(Xs, Z).
```

As most predicates, `foldl` must be changed to adapt to logic programming: instead of returning the result as a value, `foldl/2` accumulates the result into an accumulator parameter. Expanding the folded list, `foldl(F,Z1)([X1, X2, ..., Xn], Z)` is equivalent to

```
F( Z1, X1, Z2 ),
F( Z2, X2, Z3 ),
...
F( Zn, Xn, Z  ).
```

Splitting the arguments this way (two in the first set and two in the second) just seems like a logical choice: this way, `foldl(F,Z1)` is a predicate that succeeds if its second argument is the result of folding `F` over its first argument, with initial value `Z1`. The programmer could of course split them any other way they choose.

The type of `foldl/2` is

```
foldl/2 :: ((i, i, i) -> o, i) -> (i, i) -> o
```

Now we can define a predicate that sums a list of numbers:

```
'+'(X,Y,Z) :- Z is X+Y.
sum(L,Sum) :- foldl(pred '+'/3, 0)(L,Sum).
```

6.3 A Comment on List Predicates

List predicates showcase a weakness of our system. Since our language supports no polymorphic data structures and lists are just elements of the individual type, we have to restrict ourselves to monomorphic list predicates. The Prelude of Haskell, for example, defines a function `and` that takes a list of boolean values and returns their conjunction:

```
-- Haskell code
and = foldr (&&) True
```

In this case, the argument of `and` is a list of boolean values. It is impossible to define such a predicate in *poly*Hopes using the monomorphic fold.

However, in comparison to their functional counterparts, these list predicates have the advantage of supporting backtracking, thus also functioning non-deterministically.

For example, we can define a predicate that checks whether all elements of a list satisfy a condition `R`:

```
all(_)([]).
all(R)([H|T]) :- R(H), all(R)(T).
```

Then if we define `isDigit` to succeed for 0, 1, . . . , 9,

```
?- length(List,5), all(pred isDigit/1)(List).
```

would return all digit sequences of length 5.

6.4 Generic Higher-order Predicates

One type of predicates our language supports well are the ones based on predicate application itself. `closure`, which we have seen many times through the thesis, is one example. In the following, the interpreter denotes argument type variables with `a1`, `a2`, . . . and predicate type variables with `t1`, `t2`, ... Also, it skips (the always assumed) $\forall$ notation:

```
closure(R)(X, Y) :- R(X, Y).
closure(R)(X, Y) :- R(X, Z), closure(R)(Z, Y).
% closure/1 :: ((a1, a1) -> o) -> (a1, a1) -> o
```

As other examples, we can reproduce some functions from the Haskell prelude: `curry` and `uncurry` take a predicate defined without currying and return one with currying and vise versa, and `flip` flips the arguments of a predicate.

```
curry(P)(X)(Y)  <- P(X,Y).
uncurry(P)(X,Y) <- P(X)(Y).
flip(F)(X)(Y) <- F(Y)(X).
% curry/1   :: ((a1, a2) -> t3) -> a1 -> a2 -> t3
% uncurry/1 :: (a1 -> a2 -> t3) -> (a1, a2) -> t3
% flip/1    :: (a1 -> a2 -> t3) -> a2 -> a1 -> t3
```

Another interesting predicate to define would be predicate composition. We first define it to be a highly associative operator. We use two dots to not get confused with the synonym list cons functional symbol and delimiter:

```
:- op(60, xfy, '..').

'..'(F,G)(X,Z) :- F(X,Y), G(Y,Z).
% ../2 :: ((a1, a2) -> o, (a2, a3) -> o) -> (a1, a3) -> o
```

In *poly*$\mathcal{H}$, composition is written as

$$\texttt{../2} \;\leftarrow\; \lambda(\texttt{F},\texttt{G}).\,\lambda(\texttt{X},\texttt{Z}).\,\exists\texttt{Y}.\,(\,\texttt{F}(\texttt{X},\texttt{Y}) \wedge \texttt{G}(\texttt{Y},\texttt{Z}) \wedge \texttt{true/0}\,)$$

Now if we define the successor predicate as

```
succ(X,Y) :- Y is X+1.
```

then

```
add2/2 <- pred succ/2 .. pred succ/2.
```

adds 2 to a given number.

6.5 The `call/1` meta-predicate

In Section 5.1 we mentioned `call/1` as a *meta-predicate* of Prolog. In [SWI10], a meta-predicate is defined as *a predicate that calls other predicates dynamically, modifies a predicate or reasons about properties of a predicate*. `call/1` is mentioned specifically as a *meta-call predicate*.

Since meta-predicates manipulate predicates, they often violate the assumption that we made in Section 5.1 about Prolog, that predicates can never be encountered as predicate arguments. It follows that meta-predicates need special treatment in *poly*HOPES. We already saw how we treat the meta-predicates `','/2` and `';'/2`: we consider their arguments to be predicates by default as well. Now we will address how `call/1` behaves in our language.

`call(X)` in Prolog tries to call `X` as a predicate, and succeeds if and only if `X` also succeeds. This function is necessary in Prolog to simulate higher-order features in an otherwise first-order environment, as well as to call terms that have been constructed dynamically through `functor/3` and `arg/3`.

The last function can sometimes be useful even in a higher-order setting. In *poly*HOPES, `call/1` is a built-in predicate of type $\texttt{i} \to \texttt{o}$.

We will informally give the behavior of `call/1`. Since it functions on expressions of type `i`, we will only show what it does on constants, functional applications, and variables. Firstly, trying to call a constant without arguments should result in the respective predicate of arity 0. So $\texttt{call/1}(c)$ would be evaluated by the interpreter to $c/\texttt{0}$. A functional application, on the other hand, would require its functor to become a predicate of the correct arity. Thus $\texttt{call/1}(c(E_1,\ E_2,\ \ldots, E_n))$ is evaluated to $c/n(E_1,\ E_2,\ \ldots, E_n)$. Exceptionally, if we have a "special" operator, we have to make sure we output the correct structure, and possibly propagate `call/1` to the arguments of that structure. E.g. $\texttt{call}(;(E_1, E_2))$ must become $(\texttt{call/1}(E_1) \vee \texttt{call/1}(E_2))$. Finally, $\texttt{call/1}(V)$ throws an instantiation error.

Chapter 7

Concluding Remarks

7.1 Contribution

In this thesis we have presented a framework for higher-order logic programming with extensional semantics, *poly*$\mathcal{H}$. We have also presented a higher-order extension of Prolog, *poly*HOPES, which, though yet imperfect, is to our knowledge the first attempt to create an extension of Prolog with a polymorphic type discipline. A prototype implementation of *poly*HOPES, written in Haskell, is publicly available at [Kouk13].

7.2 Limitations and Future Work

Our framework, though it has made progress in integrating a polymorphic type inference mechanism into the untyped nature and unusual syntax of Prolog, still has not managed to solve all problems associated with this combination. These problems need to be addressed if a higher-order language which satisfies the standard of Prolog in its integrity is to be developed.

Specifically, as explained in Section 5.1, there are some cases where Prolog directly calls expressions that can be shown to have type `i`. This can be corrected if we wrap these expressions in `call/1`. This, however, currently demands intervention to the original source by the programmer. It would be interesting to develop a technique to insert these calls automatically, and be able to compile Prolog code directly.

Additionally, it is not yet clear how the system can integrate dynamic predicates. Prolog offers two predicates, `assert/1` and `retract/1`, which dynamically add or remove clauses in runtime. It is not clear how this these new clauses will be type checked against the old ones in a typed environment. Similarly, negation-as-failure needs to be integrated to our system. Also, *poly*HOPES currently demands that goals are monomorphic. What a polymorphic variable would mean in a goal is debatable.

On the more practical side, more work needs to be done on the interpreter. Firstly, one has to make sure there are no details of Prolog syntax that we are failing to capture: for example, clause heads defined in operator notation, though allowed in Prolog, are currently not parsed by our system. Secondly, the code needs to be optimized, as some stages such as parsing are considerably slow. Perhaps Haskell is not an adequate implementation language, and a faster language like C needs to be used, as is done in modern industrial-strength Prolog systems.

Also, the back-end of the interpreter is not yet ready, and those parts that are significantly different from the original $\mathcal{H}$ have to be sorted out. To mention two examples, the types of predicates in the environment have to be instantiated each time these are called, and polymorphic goals have to be addressed, possibly through some defaulting discipline. For both of these issues there is experience that can be gained from interpreted functional languages.

Finally, there is work to be done concerning the meta-language of *poly*$\mathcal{H}$. The semantics of types and expressions has not been formally specified, although it would probably resemble this of $\mathcal{H}$ closely. Also, we lack a proof of the principality of types for the type inference algorithm that we have provided.

Bibliography

[Cesa09] Francesco Cesarini and Simon Thompson, *Erlang Programming: A Concurrent Approach to Software Development*, O'Reilly, 2009.

[Char10] Angelos Charalambidis, Konstantinos Handjopoulos, Panos Rondogiannis and William W. Wadge, "Extensional Higher-Order Logic Programming", in *Proceedings of the 12th European Conference on Logics in Artificial Intelligence*, vol. 6341 of *Lecture Notes in Computer Science*, pp. 91–103, Helsinki, Finland, September 2010, Springer-Verlag.

[Char13a] Angelos Charalambidis, "HOPES: Higher-Order Prolog with Extensional Semantics", Available from https://github.com/acharal/hopes, 2013. [Online; accessed March 13, 2013].

[Char13b] Angelos Charalambidis, Konstantinos Handjopoulos, Panos Rondogiannis and William W. Wadge, "Extensional Higher-Order Logic Programming", *ACM Transactions On Computational Logic*, vol. 14, no. 3, 2013. To appear.

[Chen93] Weidong Chen, Michael Kifer and David S. Warren, "HiLog: A foundation for higher-order logic programming", *The Journal of Logic Programming*, vol. 15, no. 3, pp. 187–230, 1993.

[Dama82] Luis Damas and Robin Milner, "Principal type-schemes for functional programs", in *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 207–212, ACM, 1982.

[Eric13] Ericsson AB, "Erlang/OTP Documentation", Available from http://www.erlang.org/doc/, 2013. [Online; accessed March 3, 2013].

[Kouk13] Emmanouil Koukoutos, Angelos Charalambidis and Nikolaos S. Papaspyou, "poly-HOPES implementation", Available from https://github.com/acharal/hopes/tree/polyhopes, 2013. [Online].

[Marl10] Simon Marlow, editor, *Haskell 2010 Language Report*, Available from http://www.haskell.org/definition/haskell2010.pdf, 2010. [Online; accessed 10-April-2013].

[Mill86] Dale Miller and Gopalan Nadathur, "Higher-order logic programming", in *Third International Conference on Logic Programming*, pp. 448–462, Springer, 1986.

[Miln78] Robin Milner, "A theory of type polymorphism in programming", *Journal of Computer and System Sciences*, vol. 17, pp. 348–375, 1978.

[Pier02] Benjamin Pierce, *Types and Programming Languages*, MIT Press, 2002.

[Robi65] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle", *J. ACM*, vol. 12, no. 1, pp. 23–41, January 1965.

[SWI10] "SWI-Prolog manual, Section 5.4", Available from http://www.swi-prolog.org/pldoc/refman/., 2010. [Online; accessed 20-June-2013].

[Teyj08] "Teyjus: a λProlog Implementation", Available from http://teyjus.cs.umn.edu, 2008. [Online; accessed 20-June-2013].

[Wadg91] William W. Wadge, "Higher-order Horn logic programming", in *Proceedings of the International Symposium on Logic Programming (ISLP)*, pp. 289–303, 1991.

[XSB12] "XSB homepage", Available from http://xsb.sourceforge.net, 2012. [Online; accessed 20-June-2013].