A Case Study on Polymorphic Type Inference using Prolog (draft submitted to APLAS '15)

Ki Yung Ahn and Andrea Vezzosi

Portland State University,
Portland, OR, USA
kya@pdx.edu

Chalmers University of Technology,
Gothenburg, Sweden
vezzosi@chalmers.se

Abstract. A concise, declarative, and machine executable specification of the Hindley–Milner type system (HM) can be formulated using logic programming languages such as Prolog. Modern functional language implementations such as the Glasgow Haskell Compiler supports more extensive flavors of polymorphism beyond Milner's theory of type polymorphism in late '70s. In this case study, we progressively extend the Prolog specification to include more advanced type system features. An interesting development is that extending dimensions of polymorphism beyond HM resulted in a multi-staged solution: resolve the typing relations first, while delaying to resolve kinding relations, and then resolve the delayed kinding relations. Our case study shows that Prolog is an effective tool for prototyping type inference with rich features of polymorphism, and that logic programming could have been even more effective for specifying type inference if it were equipped with better theories and tools for staged resolution of different relations at different levels.

Keywords: Hindley–Milner, functional language, type system, type inference, unification, parametric polymorphism, higher-kinded polymorphism, type constructor polymorphism, kind polymorphism, algebraic datatype, nested datatype, logic programming, Prolog, delayed goals

1 Introduction

When implementing the type system of a programming language, we often face a gap between the design described on paper and the actual implementation. Sometimes there is no formal description but only an ambiguous description in English (or in other natural languages). Even when there is a mathematical description of the type system on paper, there can be a gap between the type system design and implementation. Language designers and implementers can suffer from this gap because it is difficult to determine whether a problem originated from a flaw in the design or a bug in the implementation. Having a declarative (i.e., structurally similar to the design) and flexible (i.e., easily extensible)

machine executable specification is extremely helpful, especially in early prototyping stage or when experimenting with possible extensions to the language type system. Jones' attempt of *Typing Haskell in Haskell* [12] is an exemplary work that demonstrates the value of such a concise, declarative, and machine executable specification: 90+ pages of Hugs type checker implementation in C code specified in only 400+ lines readable Haskell.

In our case study, we use Prolog to specify advanced polymorphic features of modern functional languages. Logic programming languages like Prolog are natural candidates for the purpose of specifying type systems that support type inference: the syntax and semantics are designed to represent logical inference rules (which is how type systems are typically formalized) and they offer native support for unification (which is the basic building block of type inference algorithms). As a result, Prolog specifications are usually more succinct than a specification using a functional language.

Our contributions are:

- A succint and declarative specification of a type system with several dimensions of polymorphism in less than 30 lines of Prolog (§2).
- An easily extensible specifiation for pragmatic use (as demonstrated in §3): We believe our specification is usable without expert knowledge in logic programming because we only used built-ins and fairly basic library predicates, not relying on any specialized frameworks.
- Two-staged inference for types and kinds using delayed goals (§2.3): We discovered that kind inference can be delayed after type inference (it is in some sense quite natural) and exploited the fact to get the most out of Prolog's native unification in both type and kind inference (which helped our specification to be more succinct).
- A Motivating example for calling support for a dual-view on variables in logic programming: Type variables are viewed as unification variables in type inference and as concrete atoimc variables in kind inference in our specification. Better way of organizing this idea is desirable for even better specification.

We give a step-by-step tutorial style explanation of our specification in §2, gradually extending from the Prolog specification of the simply-typed lambda calculus. In §3, we demonstrate that our method of specification is flexible for extensions with other language features. All our Prolog specification in §2 and §3 are tested on SWI Prolog 7.2 and its source code is available online.[1] We contemplate on more challenging language features such as GADTs and term indicies in §4, discuss related work in §5, and summarize our discussion in §6.

2 Polymorphic type inference specifications in Prolog

We start from a Prolog specification for the type system of simply-typed lambda calculus (STLC) (§2.1) and gradually add more features of polymorphism. We

[1] `https://github.com/kyagrd/HMtyInferUsingProlog`

discuss the specification for HM (§2.2), which extends STLC with type polymorphism, and then extend the specification to support type constructor polymorphism and kind polymorphism (§2.3).

The following two lines must be loaded into the Prolog system before running the Prolog specifications.

```
:- set_prolog_flag(occurs_check,true).
:- op(500,yfx,$).
```

In the first line, `set_prolog_flag(occurs_check,true)` sets Prolog's unification operator `=` to perform occurs check. For example, `X -> Y = Y` will fail because of occurs check. In the second line, `op(500,yfx,$)` declares `$` as a left associative infix operator, which is used to represent application operator in the object language syntax. For instance, `E1 $ E2` is an application of `E1` to `E2`.

2.1 STLC

The type system of STLC can be specified in 4 lines (excluding empty lines) of pure[2] Prolog (Fig. 1). The `type` predicate is self explanatory; its three rule definition almost literally transcribes the three typing rules of the STLC. For instance, the third rule `type(C,X$Y,B) :- type(C,X,A->B), type(C,Y,A)` is a transcription of the typing rule for applications $\frac{C \vdash X : A \to B \qquad C \vdash Y : A}{C \vdash X\ Y : B}$ in STLC. The `first` predicate implements looking up a binding (`X:T`) in the typing context (`C`) by first occurrence of the variable name (`X`). In case of repeated variable names, choosing the first occurrence in the context reflects shadowing of bound variables, e.g. the term $(\lambda x.\lambda x.x)$ will have only one type schema, the one of $(\lambda x.\lambda y.y)$. We assume variable names are given as atoms.[3] For example, `var(x)` is a variable expression whose name is represented by the atom `x`.

```
type(C,var(X),       T) :- first(X:T,C).
type(C,lam(X,E),A -> B) :- type([X:A|C], E,  B).
type(C,X $ Y,        B) :- type(C,X,A -> B), type(C,Y,A).

first(K:V,[K1:V1|Xs]) :- K=K1 -> V=V1 ; first(K:V, Xs).
```

Fig. 1: STLC in Prolog

[2] Prolog implementations usually support features beyond first-order predicate logic with unification, which are considered impure parts of Prolog. For example, control flow operations such as cut (`!/0`) or meta-logical operations such as variable test (`var/1`) are ISO standard Prolog built-ins.

[3] In Prolog, identifiers that start with an uppercase letter or an underscore (e.g., `X`, `Xs`, or `_G1`) are variables that could be matched with other terms. Identifiers that startwith a lowercase letter (e.g., `x`, `xs`, or `i_am_atom`) and appear alone (i.e., not as a function symbol such as `f` in `f(X,Y)`), are atoms, which are distinct form atoms with other names or complex terms (e.g., `f(X,Y)`).

One great merit of this specification is that it also serves as a machine executable reference implementation. We can use Prolog to run the specification for type checking, as shown in the following query:

```
?- type([], lam(x,var(x)), A -> A).
true .
```

and also for type inference, as shown the following query:

```
?- type([], lam(f,lam(x,var(f)$var(x))), T).
T = ((_G1861->_G1862)->_G1861->_G1862) .
```

In the following sections, we discuss how to add polymorphic features to the specification. The specifications with the extended features also serves as machine executable reference implementations, which are able to perform both type checking and type inference.

2.2 HM

HM supports rank-1 polymorphic types (a.k.a. type schemes) in contrast to STLC, which only support monomorphic types. Polymorphic variable bindings are introduced by let expressions in HM. For example, the following let expression introduces a polymorphic binding for the identity function ($id : \forall X.X \to X$):

$$\mathsf{let}\ id = (\lambda x.x)\ \mathsf{in}\ id\ id\,.$$

This let expression is well-typed in HM because the polymorphic type can be instantiated differently in each occurrence of id in the application expression. We distinguish different occurrences by superscripts:

$$\frac{id : \forall X.X \to X \vdash id^1 : (A \to A) \to (A \to A) \qquad id : \forall X.X \to X \vdash id^2 : A \to A}{id : \forall X.X \to X \vdash id^1\ id^2 : A \to A}\,.$$

If id had been bound monomorphically, the application would be ill-typed because the type of id would have to unify with its domain:

$$\frac{id : A \to A \vdash id^1 : A \to A \qquad id : A \to A \vdash id^2 : A \to A}{id : A \to A \vdash id^1\ id^2 : \texttt{ERROR: cannot unify } A \doteq A \to A}\,.$$

In the specification of HM (Fig. 2), a monomorphic type is represented as `mono(A)`, which corresponds to simply `A` in the STLC specification (Fig. 2), whereas a polymorphic type has the form `poly(C,A)`. A typing context in HM may contain two kinds of bindings: monomorphic bindings (`X:mono(A)`) and polymorphic bindings (`X:poly(C,A)`).

The four rules defining the `type` predicate are almost literal transcriptions of the typing rules of HM. The first rule for finding a type (`T1`) of a variable expression (`var(X)`) amounts to instantiating (`instantiate(T,T1)`) the type (`T`) in the binding (`X:T`) that first matches the variable bound in the typing context

```
type(C,var(X),      T1) :- first(X:T,C), instantiate(T,T1).
type(C,lam(X,E),A -> B) :- type([X:mono(A)  |C], E,  B).
type(C,X $ Y,        B) :- type(C,X,A -> B), type(C,Y,A).
type(C,let(X=E0,E1), T) :- type(C,               E0, A),
                           type([X:poly(C,A)|C], E1, T).

instantiate(poly(CXT,T),T1) :- copy_term(t(CXT,T),t(CXT,T1)).
instantiate(mono(T),    T).

first(K:V,[K1:V1|Xs]) :- K=K1 -> V=V1 ; first(K:V, Xs).
```

Fig. 2: HM in Prolog

(`C`). The two rules for lambda and application expressions are essentially the same as the corresponding rules in STLC. The last rule for let expressions introduce polymorphic bindings. Note that a polymorphic biding `poly(C,A)` refers to the typing context `C` of the let expression.

The `instantiate` predicate cleverly implements the idea of polymorphic instantiation in HM. The built-in predicate `copy_term` makes a copy of the first argument and unifies it with the second argument. The copied version is identical to the original term except all the Prolog variables have been substituted with freshly generated variables. The instantiation a polymorphic type `poly(CTX,T)` is implemented as `copy_term(t(CTX,T),t(CTX,T1))`. Firstly, a copied version of `t(CTX,T)` is made. Say `t(CTX2,T2)` is the copied version with all variables in both `CTX` and `T` are freshly renamed in `CTX2` and `T2`. Secondly, `t(CTX2,T2)` is unified with `t(CTX,T1)`, which amounts to `CTX2=CTX` and `T2=T1`. Because `CTX2` is being unified with the original context `CTX`, all freshly generated variables in `CTX2` are unified with the original variables in `CTX`. Therefore, only the variables in `T` that do not occur in its binding context `CTX` shall effectively be freshly instantiated in `T1`. This exactly captures generalization and instantiation of polymorphic types in HM.

The Prolog specification of HM is only 8 lines (excluding empty lines).

2.3 HM + Type Constructor Polymorphism + Kind Polymorphism

Modern functional languages such as Haskell support rich flavours of polymorphism beyond type polymorphism. For example, consider a generic tree datatype

$$\textbf{data}\ \mathit{Tree}\ c\ a\ =\ \mathit{Leaf}\ a\ \mid\ \mathit{Node}\ (c\ (\mathit{Tree}\ c\ a))$$

where c determines the branching structure at the node and a determines the type of the value hanging on the leaf. For instance, the tree datatype above instantiates to a binary tree when c instantiates to a pair constructor and a rose tree when c instantiates to a list constructor. The type system of Haskell infers that c has kind $* \to *$ and a has kind $*$. That is, c is a unary type constructor that takes an argument of kind $*$ and a is type. The type system of Haskell is

also able to infer types for polymorphic functions defined over *Tree*s, which may involve polymorphism over type constructors, such as c, as well as over types, such as a.

The Prolog specification in Fig. 3 implements type constructor polymorphism and kind polymorphism, in only 26 lines (excluding empty lines). We get kind polymorphism for free because we can reuse the same `instantiate` predicate for kinds as well as types; this is a delightful side-effect of Prolog not being statically typed. Here, we focus our discussion on the modifications to support type constructor polymorphism.

Supporting type constructor variables of arbitrary kinds introduces the possibility of ill-kinded type (constructor) formation (e.g., $F\,G$ when $F : * \to *$ but $G : * \to *$ or $A \to B$ when $A : * \to *$). In our Prolog specification, we use the atomic symbol `o` to represent the kind usually notated with $*$ (e.g., in Haskell) because `*` is predefined as a built-in infix operator in Prolog. The kind predicate transcribes the kinding rules for well-formed kinds: a type constructor variable must be bounded in the kinding context (`KC`), a function type `A->B` is a well-formed type (of kind `o`) when both `A` and `B` are well-formed types (of kind `o`), and a type constructor application `F$G` is well-formed when `F` is an arrow kind (`K1->K2`) whose codomain matches with the kind (`K1`) of its argument `G`. A type that satisfy these three kinding rules are well-formed kinds, or well-kinded.

The typing rules (`type`) need some modification from the rules of HM, in order to invoke checks for well-kindedness using the kinding rules (`kind`). We discuss the modification in three steps.

First step is to have the typing rules take an additional argument for kinding context (`KC`) along with the typing context (`C`). Because the kinding rules require a kinding context. The typing rules should keep track of the kinding context in order to invoke `kind` from `type`. Thus, we add another argument to `type` for the kinding context (i.e., from `type(C,...)` to `type(KC,C,...)`).

The second step is to invoke well-kindedness checks from the necessary places among the typing rules. We follow the formulation of Pure Type Systems [7], a generic theory of typed lambda calculi, which indicates that well-kindednss checks are required at the formation of function types, that is, in the typing rule for lambda expressions (the second rule of `type`). A straightforward modification for this typing rule would be

```
type(KC,C,lam(X,E),A->B) :- type(KC,C,[X:mono(A)|C],E,B),
                            kind(KC,A->B,o).
```

This second step modification is intuitive as a specification, but rather fragile as a reference implementation. For instance, a simple type inference query for the identity function fails

```
?- type([],[],lam(x,var(x)),T).
ERROR: Out of local stack
```

In STLC and HM specifications, such query would have successfully inferred its type as `T = A -> A`.

```
kind(KC,var(Z),K1) :- first(Z:K,KC), instantiate(K,K1).
kind(KC,F $ G, K2) :- kind(KC,F,K1 -> K2), kind(KC,G,K1).
kind(KC,A -> B,o)  :- kind(KC,A,o), kind(KC,B,o).

type(KC,C,var(X),     T1,G,G ) :- first(X:T,C), instantiate(T,T1).
type(KC,C,lam(X,E), A->B,G,G1) :- type(KC,[X:mono(A)|C],E,B,G0,G1),
                                  G0 = [kind(KC,A->B,o)|G]. % delay goal
type(KC,C,X $ Y,       B,G,G1) :- type(KC,C,X,A->B,G, G0),
                                  type(KC,C,Y,A,   G0,G1).
type(KC,C,let(X=E0,E1),T,G,G1) :- type(KC,C,              E0,A,G, G0),
                                  type(KC,[X:poly(C,A)|C],E1,T,G0,G1).

instantiate(poly(C,T),T1) :- copy_term(t(C,T),t(C,T1)).
instantiate(mono(T),  T).

first(K:V,[K1:V1|Xs]) :- K=K1 -> V=V1 ; first(K:V,Xs).

infer_type(KC,C,E,T) :-
  type(KC,C,E,T,[],Gs0), % 1st stage of typing in this line
  copy_term(Gs0,Gs),     % 2nd stage of kinding from here
  findall(Ty, member(kind(_,Ty,_),Gs), Tys),
  free_variables(Tys, Xs), % collect all free tyvars in Xs
  maplist(variablize,Xs),  % concretize tyvar with var(t) where t fresh
  findall(A:K, member(var(A),Xs), KC1), % kind binidngs for var(t)
  appendKC(Gs,KC1,Gs1),  % extend kinding context with new kind bindings
  maplist(call,Gs1).     % run all goals in Gs1

variablize(var(X)) :- gensym(t,X).

appendKC([],_,[]).
appendKC([kind(KC,X,K)|Gs],KC1,[kind(KC2,X,K)|Gs1]) :-
  append(KC1,KC,KC2), appendKC(Gs,KC1,Gs1).
```

Fig. 3: HM + type constructor polymorphism + kind polymorphism in Prolog (without pattern matching).

There are mainly two reasons for the erratic behavior. Firstly, there is not enough information at the moment of well-kindedness checking. At the invocation of `kind`, the only available information is that it is a function type `A -> B`. Whether `A` or `B` is a variable, a type constructor application, or a function type may be determined later on, when there are other parts of expression to be type checked or type inferred. Secondly, we have a conflicting view on type variables at the typing level and at the kinding level. At the typing level, we think of type variables as unification variables, implemented by Prolog variables in order to exploit the unification natively supported in Prolog. At the kinding level, on the contrary, we think of type variables as concrete names that can be looked up in the kinding context (just like term variables in the typing context).

The last step of the modification addresses the erratic behavior of the second step. A solution for these two problems mentioned above is to stage the control flow: first, get as much information as possible at the typing level, and then, concretize Prolog variables with atomic names for the rest of the work at the kinding level. Instead of directly invoking `kind` in the second rule of `type`, we collect the list of all the necessary well-kindedness checks in a list to be handled later. This programming technique is known as *delayed goals* in logic programming. You can also think of it as building up a to-do list or a continuation. We add two additional arguments to `type` to manage delayed goals. That is, `type` has six arguments, `type(KC,C,E,A,T,Gs,Gs1)`. The fifth (input) argument `Gs` is the current list of delayed goals before invoking `type`. The last (output) argument `Gs1` obtains the list of delayed goals after invoking `type`. These two arguments are threaded through the recursive definition of the `type` predicate.[4]

The `infer_type` predicate implements the two-staged solution, as follows:

1. The first line of the definition is the first stage at the typing level. For example, when inferring a type for the identity function,

   ```
   ?- type([],[],lam(x,var(x)),T,[],Gs0).
   T = (_G1643->_G1643),
   Gs0 = [kind([], (_G1643->_G1643), o)] .
   ```

   It infers the most generic type (`_G1643->_G1643`) of the identity function and generates one delayed goal `kind([], (_G1643->_G1643), o)`.
2. From the second line, we invoke this delayed goal after preprocessing of concertizing the type variables. In the second line, we make a copied version of the delayed goals using `copy_term` in order to decouple the variables of the first stage from the variables of the second stage. After the second line, `Gs` contains a copied version of `Gs0` with freshly renamed variables, say `Gs = [kind([], (_G2211->_G2211), o)]`.

3,4. The third and fourth line of the definition collects all the type variables in `Gs` into `Xs`, that is, `Xs = [_G2211]`, continuing with our example for the identity function.

[4] Experienced Prolog programmer would use *DCG Grammer rules* [17] to abstract away such extra arguments for accumulated differences. Here, we choose to stick to basic constructs to be more accessible, relying less on Prolog idioms and libraries.

5. The fifth line `maplist(variablize,Xs)` instantiates the Prolog variables collected in `Xs` into concrete type variables with fresh names. `variablize` implements fresh concrete variable generation using a symbol generation library predicate `gensym`. Invoking `gensym(t,X)` generates atoms whose fresh names that start with `t`. For instance, `X=t1`, `X=t2`, and so on. After fifth line, `Xs = [var(t1)]` and `Gs = [kind([], (var(t1)->var(t1)), o)]` because the `variablize` predicate has instantiated `_G2211` as `_G2211=var(t1)`.

6,7. Freshly generated concrete type variables need to be registered to the kinding context before invoking `kind` because the first kinding rule requires that type variables must be bound in the typing context. The sixth line creates monomorphic type bindings for all the variable names in `Xs`, and collects them into `KC1`. Continuing with our identity example, `KC1 = [t1:mono(K1)]` after the sixth line. The seventh line extends each kinding context in the dealyed goal `Gs` with the bindings (`KC1`) for the freshly generated variables, where the goals with extended contexts are collected in `Gs1`. Ater seventh line, we have `Gs1 = [kind([t1:mono(K1)], (var(t1)->var(t1)), o)]`.

8. Finally, we perform the delayed well-kindedness checks by calling all the goals in `Gs1`. We can execute the well-kindedness check for our identity example by querying

```
?- kind([t1:mono(K1)], (var(t1)->var(t1)), o).
K1 = o
```

The well-kindedness check succeeds by inferring that the freshly generated type variable `var(t1)` must have kind `o`.

3 Supporting Other Language Features

The purpose of this section is to demonstrate that our Prolog specification for polymorphic features is extensible for supporting other orthogonal features in functional languages including general recursion (§3.1), pattern matching over algebraic datatypes (§3.2), and recursion schemes over non-regular algebraic datatypes with user provided annotations (§3.3). The specification for the pattern-matching and the recursion schemes in this section are extensions that build upon the specification in §2.3.

Discussions on the details of the Prolog code is kept relatively brief, compared to the previous section, because our main purpose here is to demonstrate that supporting these features does not significantly increase the size and the complexity of our specification. Readers with further interest are encouraged to experiment with our specifications available online.

3.1 Recursive Let-bindings

Adding recursive let-bindings is obvious. Simply add a monomorphic binding for the let-bound variable (`X`) when inferring the type of the expression (`E0`) defining the let-bound value, as follows:

```
type(KC,C,letrec(X=E0,E1),T,G,G1) :- type(KC,[X:mono(A)  |C],E0,A,G, G0),
                                     type(KC,[X:poly(C,A)|C],E1,T,G0,G1).
```

We could also allow polymorphic recursion by type annotations on the let-bound variable, like we will do for Mendler-style iteration over non-regular datatypes §3.3.

3.2 Pattern Matching for Algebraic Datatypes

In Fig.4 (on p11), we specify pattern matching expressions without the scrutinee, which is also known as pattern-matching lambdas. A pattern lambda is a function that awaits an expression to be passed in as an argument to pattern-match with its value. For example, let $\{p_1 \to e_1; \cdots ; p_n \to e_n\}$ be a pattern-matching lambda. Then, the application $\{p_1 \to e_1; \cdots ; p_n \to e_n\}\, e$ corresponds to a pattern matching expression in Haskell **case** e **of** $\{p_1 \to e_1; \cdots ; p_n \to e_n\}$.

We represent pattern-matching lambdas in Prolog as a list of clauses that match each pattern to a body, for instance, `['Nil'-->E1, 'Cons'(x,xs)-->E2]` where `E1` and `E2` are expressions of the bodies. For simplicity, we implement the most simple design of non-nested patterns. That is, a pattern is either an atom that represents a nullary data constructor, such as `'Nil'`, or a complex term with n-ary function symbol that represents n-ary data constructor and n variables as arguments, such as `'Cons'(x,xs)`. Atoms and function symbols normally start with lowercase letters in Prolog. However, Prolog allows other names to become atoms and function symbols when those names are single-quoted. Here, we are using the convention such that names of type constructors and data constructors start with uppercase letters while names of term variables (including pattern variables) start with lowercase letters. We also add a delayed well-kindedness goal because patten lambdas introduce function types (`A-> T`), just like ordinary lambda expressions.

The specification above for non-nested pattern-matching lambdas is only 16 lines. To sum up, the specification for HM extended with type constructor polymorphism, kind polymorphism, and non-nested pattern matching for algebraic datatypes is in $26+16 = 42$ lines of Prolog, which is also an executable reference implementation of the type system.

3.3 Recursion Schemes for Non-Regular Algebraic Datatypes

Consider the following two recursive datatype declarations in Haskell:

$$\begin{aligned} &\textbf{data}\ \mathit{List}\ a\ \ = N_L \mid C_L\ a\ (\mathit{List}\ a) \\ &\textbf{data}\ \mathit{Bush}\ a\ = N_B \mid C_B\ a\ (\mathit{Bush}\ (\mathit{Bush}\ a)) \end{aligned}$$

List is a homogeneous list, which is either empty or en element tailed by a *List* that contains (zero or more) elements of *the same type as the prior element. Bush* is a list-like structure that is either empty or has en element tailed by a *Bush* that contains (zero or more) elements *but their type* $(\mathit{Bush}\ a)$ *is different from the*

```
type(KC,C,Alts,A->T,G,G1) :- type_alts(KC,C,Alts,A->T,G,G1),
                             G0 = [kind(KC,A->T,o) | G]. % delayed goal

type_alts(KC,C,[Alt],          A->T,G,G1) :- % assume at least one clause
  type_alt(KC,C,Alt,A->T,G,G1).              % empty list not allowed
type_alts(KC,C,[Alt,Alt2|Alts],A->T,G,G1) :-
  type_alt(KC,C,Alt,A->T,G,G0),
  type_alts(KC,C,[Alt2|Alts],A->T,G0,G1).

type_alt(KC,C,P->E,A->T,G,G1) :- % assume non-nested single depth pattern
  P =.. [Ctor|Xs], upper_atom(Ctor), % when P='Cons'(x,xs) then Xs=[x,xs]
  findall(var(X),member(X,Xs),Vs),   %    Vs = [var(x),var(xs)]
  foldl_ap(var(Ctor),Vs,PE),         %    PE = var('Cons')$var(x)$var(xs)
  findall(X:mono(Tx),member(X,Xs),C1,C), % extend C with bindings for Xs
  type(KC,C1,PE,A,G,G0), % type of the pattern as an expression is A
  type(KC,C1,E,T,G0,G1). % type of the body is T

upper_atom(A) :- atom(A), atom_chars(A,[C|_]), char_type(C,upper).
lower_atom(A) :- atom(A), atom_chars(A,[C|_]), char_type(C,lower).
```

Fig. 4: A Prolog specification of non-nested pattern-matching lambdas (coverage checking not included).

```
kind(KC,mu(F), K)  :- kind(KC,F,K->K).

type(KC,C,in(N,E),T,G,G1) :- type(KC,C,E,T0,G,G1),
                             unfold_N_ap(1+N,T0,F,[mu(F)|Is]),
                             foldl_ap(mu(F),Is,T).
type(KC,C,mit(X,Alts),mu(F)->T,G,G1) :-
  is_list(Alts), gensym(r,R),
  KC1 = [R:mono(o)|KC], C1 = [X:poly(C,var(R)->T)|C],
  type_alts(KC1,C1,Alts,F$var(R)->T,G,G1).
type(KC,C,mit(X,Is-->T0,Alts),A->T,G,G1) :-
  length(Is,N), length(Es,N), foldl_ap(mu(F),Es,A),
  is_list(Alts), gensym(r,R), foldl_ap(var(R),Is,RIs),
  KC1 = [R:mono(K)|KC], C1 = [X:poly(C,RIs->T0)|C],
  G0 = [kind(KC,F,K->K), kind(KC,A->T,o) | G], % delayed goal
  foldl_ap(F,[var(R)|Is],FRIs),
  type_alts(KC1,C1,Alts,FRIs->T,G0,G1).

unfold_N_ap(0,E,    E,[]).
unfold_N_ap(N,E0$E1,E,Es) :-
  N>0, M is N-1, unfold_N_ap(M,E0,E,Es0), append(Es0,[E1],Es).

foldl_ap(E, []     , E).
foldl_ap(E0,[E1|Es], E) :- foldl_ap(E0$E1, Es, E).
```

Fig. 5: A Prolog specification for Mendler-style iteration.

type of the prior element (a). The recursive component (*List* a in C_L a (*List* a)), which is the tail of a list, has exactly the same type argument (a) as the datatype being defined (*List* a in **data** *List* $a = \ldots$).

Every recursive component of *List*, which is the tail of a list, has exactly the same type argument (a) as the *List* containing the tail. In other words, the types of recursive occurrences in *List* is always the same, or *regular*. When all recursive occurrences in a (recursive) datatype are regular, as in *List*, it is called a *regular datatype*. The recursive component of *Bush*, on the contrary, has different type argument (*Bush* a) from the type argument (a) of its containing *Bush*. That is, a bush headed by an element of type a is tailed by a bush whose head element is of type *Bush* a. A recursive datatype that has a non-regular recursive occurrence, as in *Bush*, is called a *non-regular datatype*, also known as a *nested datatype* [8] because the types of recursive components typically become nested as the recursion goes deeper (e.g., $Bush(Bush(\cdots(Bush(Bush\ a))\cdots))$).

In order to define interesting and useful recursive functions over non-regular datatypes, one needs polymorphic recursion, whose type inference is known to be undecidable without the aid of user supplied type annotations. In Fig. 5 (on p5), we specify a subset of a functional language that supports a recursion scheme, which naturally generalize from regular datatypes to non-regular datatypes. In particular, we specify the Mendler-style iteration [13, 1] supported in the Nax language [2]. In Nax, all recursive constructs, both at the type level and at the term level, are defined using the primitives provided by the language, avoiding uncontrolled general recursion.

The `mu(F)` appearing in the Prolog specification corresponds to the recursive types μF constructed by the fixpoint type operator μ applied to a base structure F, which is not recursive by itself. Here, we require that F is either a type constructor introduced by a (non-recursive) datatype declaration or partial application of such a type constructor. We add a kinding rule for the fixpoint type operator by adding another rule of the `kind` predicate for `mu(F)`. The Mendler-style iteration over regular datatypes (`mit(X,Alts)`) does not need any type annotation. The Mendler-style iteration over non-regular datatypes (`mit(X,Is-->T0,Alts)`) needs an annotation (`Is-->T0`) to guide the type inference because it is likely to rely on polymorphic recursion. The specification for Mendler-style iteration relies on pattern-matching lambdas discussed in the previous subsection. Once we have properly set up the kinding context and typing context for the name of recursive call (`X`), the rest amounts to inferring types for pattern-matching lambdas. Pointers to further details on Mendler-style recursion [18, 1, 3] and Nax [2] are available in the references section at the end of this paper.

The specification for Mendler-style iteration in Fig. 5 is only 20 lines, including pattern-matching lambdas $16 + 20 = 36$ lines, including HM with type constructor polymorphism and kind polymorphism the total is $26+16+20 = 62$ lines of Prolog (excluding empty lines). The complete specification, and also a reference implementation, of all the features discussed so far fits in only 62 lines of Prolog.

A missing part from a typical functional language type system, which we have not discussed in this paper, is the initial phase of populating the kinding context and typing context from the list of algebraic datatype declarations prior to type checking the expressions using them. With fully functioning basic building blocks for kind inference (`kind`) and type inference (`type`), inferring kinds of type constructor names and inferring types for their associated data constructors should be straightforward.

4 Future Work

We plan to continue our work on two additional features. One feature is generalized algebraic datatypes (GADTs) and the other is enriching the kind syntax, as in the Nax language, to support term-indicies in datatypes, especially in GADTs.

GADTs add complexity of introducing local constraints within a pattern-matching clause, which should not escape the scope of the clause, unlike global unification constraints in HM. It would be interesting to see whether Prolog's built-in support for handling unification variables and symbols could help us express the concept of local constraints as elegantly as we expressed polymorphic instantiation in §2. In addition, more user supplied annotations will be required to guide the type inference, even when recursion is not involved, because GADT type inference is undecidable [10] regardless of the use of recursion. In Nax, pattern-matching lambdas can have annotations, which are just like the annotations on the Mender-style iteration in §3.3, to aid type inference.

The kind structure needed for type constructor (i.e. higher-kinded) polymorphism is exactly the kinds supported in the higher-order polymorphic lambda calculus, known as System F_ω [11]. Type constructors in F_ω can have types as arguments. For example, the type constructor *List* for lists has kind $* \to *$, which means that it needs one type argument to be fully applied as a type (e.g. *List* `Nat` : $*$).

kind in System F_ω	$\kappa ::= * \mid \kappa \to \kappa$
kind in System F_i	$\kappa ::= * \mid \kappa \to \kappa \mid \{A\} \to \kappa$

To support terms, as well as types, to be applied to type constructors as arguments, the kind structure needs to be extended. System F_i [4], which Nax is based on, extends the kind structure with $\{A\} \to \kappa$ to support term indices in types. This extension to kinds allows type constructors with term indicies such as $\mathit{Vec} : * \to \{\mathit{Nat}\} \to *$ for vectors (a.k.a. length indexed lists). For instance, *Vec Bool* $\{8\}$ is a type of boolean vectors of length 8. There are two ramifications of term indices being involved in type inference:

- the unification is modulo equivalence of terms: For instance, the type system should consider *Vec Bool* $\{n\}$ and *Vec Bool* $\{(\lambda x.x)\ n\}$ as equivalent types.
- Type inference/checking and kind inference/checking invokes each other: A typing rule had to invoke a kinding rule to support type constructor polymorphism (§2.3). In the extended kind structure, types can appear in kinds (A in $\{A\} \to \kappa$), therefore, kinding rules need to invoke typing rules.

Extending our specification with term indicies, while continuing to serve as a reference implementation, would be an interesting future work that might involve resolving possible challenges from these two ramifications.

5 Related Work

The idea of using logic programming to specify type inference is not new. Odersky, Sulzmann, and Wehr [15] defined a general framework called HM($\mathcal{X}$) for specifying extensions of HM (e.g., records, type classes, intersection types) and Alves and Florido [5] implemented HM($\mathcal{X}$) using Prolog with constraint handling rules (CHR). Testing a type system extension in the HM($\mathcal{X}$) framework provides a certain level of confidence that the extension would work well with type polymorphism in HM. Testing an extension by extending our specification provides additional confidence that the extension would work well with type constructor polymorphism and kind polymorphism, as well as with type polymorphism.

The concept of delayed goals have been used in many different contexts in logic programming. An AILog[5] textbook [16], introduces delaying goals as one of the useful abilities of meta-interpreter. Many Prolog systems, such as SWI or SICStus, provide built-in support for delaying a goal until certain conditions are met using the predicates such as `freeze` or `when`. In our specification supporting type constructor polymorphism and kind polymorphism, we could not simply use `freeze` or `when` because we pre-process the collected delayed goals (see `variableze` in §2.3).

There are experimental Prolog implementations that support peculiar features such as λProlog [14] supporting (restricted version of) higher-order unification and αProlog [9] supporting nominal abstraction in a purely logical setting. Such features may help us specify universal quantifications and fresh name generations more elegantly, but there is a trade-off for not having the pragmatic support (e.g. built-ins for extra-logical features, documentation, and graphical debugging) of more widely-used and well-maintained systems such as SWI or SICStus.

Recently, there has been research on type inference using logic programming with non-standard semantics (e.g., corecursive, coinductive, or coalgebraic) for object-oriented languages (e.g., featherweight Java) but functional languages were left for future work [6].

6 Conclusions

We defined in Prolog a type inference implementation for a non-trivial type system by making extensive use of Prolog's native support for unification, allowing us to obtain a concise and runnable specification. In particular we use logical variables to represent type (or kind) variables in type (or kind) schemas.

[5] A logical inference system for designing intelligent agents.

We overcame the conflicting view of type variables between type- and kind-checking by delaying kind-checking constraints until after type checking is completed and type variables can be made ground and so can be used as indexes into the kinding context. We believe this could have been simplified if there was either a language level support or well tailored framework for establishing a dual-view on the variables in logic programming.

We also showed the flexibility of this technique by successively extending the language with more advanced language features like recursive let, pattern matching and Mendler-style iteration.

The use of extra-logical built-ins like `copy_term/2` makes so we can exploit Prolog's logical variables not only for unification but also to represent binding of quantified type (or kind) variables, greatly simplifying the handling of implicit polymorphism.

Logical variables can however also limit the predictability of how the program is going to behave, since a predicate running on inputs that are not ground enough can get stuck.

Bibliography

[1] A. Abel, R. Matthes, and T. Uustalu. Generalized iteration and coiteration for higher-order nested datatypes. In *FoSSaCS*, volume 2620 of *LNCS*, pages 54–69. Springer, 2003.

[2] K. Y. Ahn. *The Nax language*. PhD thesis, Department of Computer Science, Portland State University, PO Box 751, Portland, OR, 97207 USA, November 2014. URL `http://archives.pdx.edu/ds/psu/13198`.

[3] K. Y. Ahn and T. Sheard. A hierarchy of Mendler-style recursion combinators: Taming inductive datatypes with negative occurrences. In *ICFP '11*, pages 234–246. ACM, 2011.

[4] K. Y. Ahn, T. Sheard, M. Fiore, and A. M. Pitts. System Fi: a higher-order polymorphic lambda calculus with erasable term indices. In *Proceedings of the 11th international conference on Typed lambda calculi and applications*, TLCA '13, 2013.

[5] S. Alves and M. Florido. Type inference using Constraint Handling Rules. In M. Hanus, editor, *WFLP '01: Proc. 10th Intl. Workshop on Functional and (Constraint) Logic Programming, Selected Papers*, volume 64 of *Electronic Notes in Theoretical Computer Science*, pages 56–72. Elsevier, Nov. 2002.

[6] D. Ancona, G. Lagorio, and E. Zucca. Type inference by coinductive logic programming. In S. Berardi, F. Damiani, and U. de'Liguoro, editors, *TYPES*, volume 5497 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008. ISBN 978-3-642-02443-6. URL `http://dx.doi.org/10.1007/978-3-642-02444-3`.

[7] H. Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[8] Bird and Meertens. Nested datatypes. In *MPC: 4th International Conference on Mathematics of Program Construction*. LNCS, Springer-Verlag, 1998.

[9] J. Cheney and C. Urban. Alpha-Prolog: A logic programming language with names, binding, and alpha-equivalence. In B. Demoen and V. Lifschitz, editors, *Logic Programming, 20th International Conference*, volume 3132 of *LNCS*, pages 269–283. Springer, 2004.

[10] A. Degtyarev and A. Voronkov. Simultaneous regid E-unification is undecidable. In H. K. Büning, editor, *CSL*, volume 1092 of *Lecture Notes in Computer Science*, pages 178–190. Springer, 1995. ISBN 3-540-61377-3. URL `http://dx.doi.org/10.1007/3-540-61377-3_38`.

[11] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[12] M. P. Jones. Typing Haskell in Haskell. In *ACM Haskell Workshop* , informal proceedings, Oct. 1999.

[13] R. Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. PhD thesis, Ludwig-Maximilians Universität, May 1998.

[14] G. Nadathur and D. J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 287–291, Trento, 1999. Springer.

[15] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, Jan. 1999. ISSN 1074-3227.

[16] D. Poole and A. K. Mackworth. *Artificial Intelligence - Foundations of Computational Agents*. Cambridge University Press, 2010. ISBN 978-0-521-51900-7. URL `http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521519007`.

[17] SWI-Prolog team. SWI-Prolog reference manual (section 4.2). `http://www.swi-prolog.org/pldoc/doc_for?object=manual`, 2014. [Online; accessed June 2015].

[18] V. Vene. *Categorical Programming with Inductive and Coinductive Types*. PhD thesis (Diss. Math. Univ. Tartuensis 23), Dept. of Computer Science, Univ. of Tartu, Aug. 2000.

Appendix: Bug fixes after submission to APLAS 2015

Andrea spotted a bug in Fig. 3. In Fig. 3, there was a line in the last `type` predicate rule for mit

```
G0 = [kind(KC1,F,K->K), kind(KC1,A->T,o) | G], % delayed goal
```

Andrea spotted that the latter `KC1` should be `KC` (i.e., without R in it). After submission, Ki Yung noticed that both `KC1` had to be `KC`. After this bug fix, kind checking instantiation for type schemes started to work fine (which didn't at the submission, so we left it out). Now, we can answer inquiries from reviewers why we didn't have kind checking in type scheme instantiation: "Well, we fixed the bug right after submission".

Ki Yung just fixed it on in LaTeX file and in the Prolog code (`HMmicronax.pl`) but have not updated Figures and corresponding text in the draft with the correct version of type instantiation. Maybe we should do this after the APLAS review comes in. The correct instantiation for type scheme should be defined as:

```
inst_type(KC,poly(C,T),T1,G,G1) :- copy_term(t(C,T),t(C,T1)),
    free_variables(T,Xs), free_variables(T1,Xs1), zip(Xs,Xs1,Ps),
    findall([kind(KC,X,K), kind(KC,X1,K)],(member((X,X1),Ps),X\==X1),Gs),
    flatten(Gs,G0), append(G0,G,G1).
inst_type(KC,mono(T),T,G,G).

zip([X|Xs],[Y|Ys],[(X,Y)|Zs]) :- zip(Xs,Ys,Zs).
zip([],[],[]).
```

Instantiation for kind schemes can still use the same `instantiate` predicate because there are no kind constructor variables, so all kinds have sort $\Box$.