A Case Study on Polymorphic Type Inference using Prolog

Ki Yung Ahn and Andrea Vezzosi

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

Charmers University of Technology,
Gothenburg, Sweden
vezzosi@charmers.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 be even more effective for specifying type inference when 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

Typing Haskell in Haskell

TODO page limit is 18 pages. I think there is adequate space left. Main matter (§?? and §??) is written so no more significant changes, b ut clarification and grammar typo fix etc. Other sections are still ongoing.

TODO Write future work briefly, related work discussions, and write a good intro/motivation, and then submit!

type constructor polymorphism (a.k.a. higher-kinded polymorphism)

TODO mention that all our Prolog specification runs in SWI Prolog 7.2. and available online point to github repo

logic programming languages such as Prolog are natural candidates
based on Hindley–Milner type inference
oustidein [?]
TODO TODO cite these
unification based
TODO
TODO
TODO
TODO
TODO

2   Polymorphic type inference specifications in Prolog

We start from a Prolog specification for the type system of simply-typed lambda
calculus (STLC) (§??) and gradually add more features of polymorphism. We
discuss the specification for HM (§??), which extends STLC with type polymor-
phism, and then extend the specification to support type constructor polymor-
phism and kind polymorphism (§??).

All Prolog specifications (Figs. ??, ??, and ??) in this section are tested in
SWI-Prolog 7.2. 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 unifica-
tion operator = to perform occurs check. For example, `X -> Y = Y` will fail be-
cause 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[1] Prolog (Fig. ??). The `type` predicate is self explanatory; its three rule
definition almost literally transcribes the three typing rules of the STLC. For in-
stance, the third rule `type(C,X$Y,B) :- type(C,X,A->B), type(C,Y,A)` is a tran-
scription of the typing rule for applications $\frac{C \vdash X : A \to B \quad C \vdash Y : A}{C \vdash X\,Y : B}$ in STLC.
The `first` predicate implements looking up a binding (`X:T`) in the typing con-
text (`C`) by first occurrence of the variable name (`X`). We assume variable names

---

[1]   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.

are given as atoms.[2] 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

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 ($id : \forall X.X \to X$):

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

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:

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

---

[2]   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)`).

If *id* had been bound monomorphically, it would be ill-typed because

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

```
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 [?]

In the specification of HM (Fig. ??), a monomorphic type is emphasized as `mono(A)`, which corresponds to simply `A` in the STLC specification (Fig. ??), 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 (`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 has 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 flavour of polymorphism beyond type polymorphism. For example, consider a Haskell function

$$mkToSet \; :: \; (c \; a \rightarrow List \; a) \rightarrow (c \; a \rightarrow Set \; a)$$
$$mkToSet \; toList \; = \; list2set \; \circ \; toList$$

which takes a function as an argument ($toList$) that maps a data structure (c) of a-values into a $List$ of a-values and produces a function that maps the data structure into a Set of a-values. Assuming we are given the function that converts a list to a set, $list2set :: List \; a \rightarrow Set \; a$, we can define $mkToSet$ by composing $list2set$ with the first argument $toList$. The type of $mkToSet$ is polymorphic over two variables c and a. Note that c is a polymorphic variable over type constructors, such as $List$ or Set, that takes one type argument. That is, the type of $mkToSet$ is polymorphic over a type constructor ($c :: * \rightarrow *$) as well as over a type ($a :: *$); this can be illustrated more clearly using explicit universal quantification and kind annotation in the type signature, as follows:[3]

$$mkToSet \; :: \; \forall \, (c :: * \rightarrow *) \, (a :: *) \, . \, (c \; a \rightarrow List \; a) \rightarrow (c \; a \rightarrow Set \; a) \, .$$

We gave explicit type signatures above for the sake of smooth explanation, but the type of $mkToSet$ can be completely inferred from its definition, without the type signature of $mkToSet$. In this subsection, we extended the Prolog specification of HM, in the previous subsection to support type constructor polymorphism.

The Prolog specification in Fig. **??** 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 $F : * \rightarrow *$ but $G : * \rightarrow *$ or $A \rightarrow B$ when $A : * \rightarrow *$). 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`),

---

[3]   The Glasgow Haskell Compiler support language extensions to write such explicit annotations in Haskell scripts.

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
kinding contexts. 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,...)`).

```
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).

The second step is to invoke well-kindedness checks from the necessary places among the typing rules. A generic theory of typed lambda calculus [TODO cite Pure Type System] 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`.

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* [TODO cite delayed goals related] 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.

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.

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 (§**??**), pattern matching over

algebraic datatypes (§??), and recursion schemes over non-regular algebraic datatypes with user provided annotations (§??). The specification for the pattern-matching and the recursion schemes in this section are extensions that build upon the specification in §??.

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).
```

3.2   Pattern Matching for Algebraic Datatypes

In Fig.??, 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.

```
type(KC,C,Alts,A->T,G,G1) :- type_alts(KC,C,Alts,A->T,G,G1),
                             GO = [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,GO),
  type_alts(KC,C,[Alt2|Alts],A->T,GO,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,GO), % type of the pattern as an expression is A
  type(KC,C1,E,T,GO,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,TO,G,G1),
                             unfold_N_ap(1+N,TO,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-->TO,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->TO)|C],
  GO = [kind(KC1,F,K->K), kind(KC1,A->T,o) | G], % delayed goal
  foldl_ap(F,[var(R)|Is],FRIs),
  type_alts(KC1,C1,Alts,FRIs->T,GO,G1).

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

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

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

3.3 Recursion Schemes for Non-Regular Algebraic Datatypes

Consider the following two recursive datatype declarations in Haskell:

$$\textbf{data } List \ a \ = \ N_L \mid C_L \ a \ (List \ a)$$
$$\textbf{data } Bush \ a \ = \ N_B \mid C_B \ a \ (Bush \ (Bush \ a))$$

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* (*Bush a*) *is different from the 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* [**?**] 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.**??**, 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 [**? ?**] supported in the Nax language [**?**]. 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-->TO,Alts)`) needs an annotation (`Is-->TO`) 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 [? ? ?] and Nax [?] are available in the references section at the end of this paper.

The specification for Mendler-style iteration 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 §??. 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 [?] 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 §??, to aid type inference.

The kind structure needed for type constructor polymorphism is exactly the kinds supported in the higher-order polymorphic lambda calculus, known as System F_ω [TODO cite Fw]. 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 [?] extends extend 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 $Vec : * \to \{Nat\} \to *$ 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 (§**??**). 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. **?**] defined a general framework called $\text{HM}(\mathcal{X})$ for specifying extensions of HM (e.g., records, type classes, intersection types) and **?**] implemented $\text{HM}(\mathcal{X})$ using Prolog with constraint handling rules (CHR).
    TODO
The situation I understand is that work around HM(X) does not consider type constructor polymorphism.
    TODO αProlog TODO
TODO
TODO
TODO
TODO
TODO
TODO

6   Conclusions

TODO observe specification
    TODO clearer picture of how type inference and kind inference interact with each other
    TODO although not pure but does not explicit rely on destructive updates So there is a degree of being declarative here
    TODO
TODO
TODO
TODO
TODO
TODO

TODO
TODO
TODO
TODO
TODO
TODO
TODO
TODO
TODO
TODO