Rapid Prototyping of Language Implementations via Higher-Order Logic Programming with Makam

Functional Pearl

ANONYMOUS AUTHOR(S)

TODO This is the abstract of the paper.

ACM Reference format:
Anonymous Author(s). 2017. Rapid Prototyping of Language Implementations via Higher-Order Logic Programming with Makam. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 26 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

TODO This is the introduction. We will cite the work by Miller and Nadathur (1988) here.

2 STARTING OUT SIMPLE

We will start with encoding a version of the simply typed lambda calculus in λProlog. We define two new meta-types to represent the two sorts of our object language: terms and types. We also define the `typeof` relation that corresponds to the typing judgement of the language.

```
term   : type.
typ    : type.
typeof : term → typ → prop.
```

Defining the basic forms of the λ-calculus is very easy, thanks to the support for higher-order abstract syntax in higher-order logic programming. We can reuse the meta-level function type in order to implement object-level binding. The reason is that the meta-level function space is *parametric* – that is, the body of a function is a value that can just mention the argument as-is, instead of being a computation that can inspect the specific value of an argument. Therefore, a meta-level function exactly represents an object-level binding of a single variable, without introducing *exotic terms*.

```
app    : term → term → term.
lam    : typ → (term → term) → term.
arrow  : typ → typ → typ.
```

Encoding the typing rule for application as a λProlog *clause* for the `typeof` relation is a straightforward transliteration of the pen-and-paper version.

```
typeof (app E1 E2) T' :-
  typeof E1 (arrow T T'),
  typeof E2 T.
```

DOI: 10.1145/nnnnnnn.nnnnnnn

In logic programming, the goal of a rule is written first, followed by the premises; the `:-` operator can be read as “is implied by,” and comma is logical conjuction. We use capital letters for unification variables.

The rule for lambda functions is similarly straightforward:

```
typeof (lam T1 E) (arrow T1 T2) :-
  (x:term → typeof x T1 → typeof (E x) T2).
```

There are three things of note in the premise of the rule. First, we introduce a fresh term variable `x`, through the form `x:term →`, which can be read as universal quantification. Second, we introduce a new assumption through the form `typeof x T →`, which essentially introduces a new rule for the `typeof` relation locally; this pattern can be read as logical implication. Third, in order to get to the body of the lambda function to type-check it, we need to apply it to the fresh variable `x`.

With these definitions, we have already implemented a type-checker for the simply typed lambda calculus, as we can issue queries for the `typeof` relation to Makam:

```
typeof (lam _ (fun x ⇒ x)) T' ?
>> Yes:
>> T' := arrow T T
```

One benefit of using λProlog instead of rolling our own type-checker is that the occurs check is already implemented in the unification engine. As a result, a query that would result in an ill-formed cyclical type with a naive implementation of unification fails as expected.

```
typeof (lam _ (fun x ⇒ app x x)) T' ?
>> Impossible.
```

Other than supporting higher-order abstract syntax, λProlog also supports polymorphic types and higher-order predicates, in a matter akin to traditional functional programming languages. For example, we can define the polymorphic `list` type, and an accompanying `map` higher-order predicate, as follows:

```
list : type → type.

nil : list A.
cons : A → list A → list A.

map : (A → B → prop) → list A → list B → prop.
map P nil nil.
map P (cons X XS) (cons Y YS) :- P X Y, map P XS YS.
```

Using the meta-level list type, we can encode object-level constructs such as tuples and product types directly:

```
tuple : list term → term.
product : list typ → typ.
```

Similarly we can use the `map` predicate to define the typing relation for tuples.

```
typeof (tuple ES) (product TS) :-
  map typeof ES TS.
```

Executing a query with a tuple yields the correct result:

```
typeof (lam _ (fun x ⇒ lam _ (fun y ⇒ tuple (cons x (cons y nil))))) T ?
>> Yes:
>> T := arrow T1 (arrow T2 (product (cons T1 (cons T2 nil))))
```

So far we have only introduced the predicate `typeof` for typing. In the same way, we can introduce a predicate for evaluating terms, capturing the dynamic semantics of the language.

```
eval : term → term → prop.
```

Most of the rules are straightforward, following standard practice for big-step semantics. We assume a call-by-value evaluation strategy.

```
eval (lam T F) (lam T F).
eval (tuple ES) (tuple VS) :- map eval ES VS.
```

For the beta-redex case, function application for higher-order abstract syntax gives us capture-avoiding substitution directly:

```
eval (app E E') V'' :-
  eval E (lam _ F), eval E' V', eval (F V') V''.
```

3 MULTIPLE BINDING

As we've seen, single-variable binding as in the lambda abstraction can be handled easily through higher-order abstract syntax. Let us now explore how to encode other forms of binding.

As a first example, we will introduce multiple-argument functions as a distinct object-level construct, as opposed to using currying. A first attempt at encoding such a construct could be to introduce a `list` of term variables at the same time, as follows:

```
lammany : (list term → term) → term.
```

However, this type does not correspond to the construct we are trying to encode. The type `list term → term` introduces a fresh local variable for the `list` type, as opposed to a number of fresh local variables for the `term` type. Since the HOAS function space is parametric, it not even possible to refer to the potential elements of the fresh `list` – we can only refer to the fresh list in full.

Instead, we would like a type that represents all types of the form:

- `term` (binding no variables)
- `term → term` (binding a single variable)
- `term → (term → term)` (binding two variables)
- `term → (term → (term → term))` (binding three variables) etc.

We can encode such a type inductively in λProlog, as follows:

```
bindmanyterms : type.
bindbase : term → bindmanyterms.
bindnext : (term → bindmanyterms) → bindmanyterms.
```

Furthermore, we can generalize the type that we are binding over, and the type of the body, leading to a polymorphic type of the form:

```
bindmany : type → type → type.
bindbase : B → bindmany A B.
bindnext : (A → bindmany A B) → bindmany A B.
```

With these, `lammany` can be encoded as:

```
lammany : bindmany term term → term.
```

(As an aside: here we have allowed binding zero variables for presentation reasons. We could disallow binding zero variables by changing the base case to require an argument of type `A → B` instead of a `B`, similar to how we can specify lists with at least one element inductively by replacing the `nil` constructor with a constructor that requires an element.)

How do we work with the `bindmany` type? For the built-in single binding type, we used three operations:

- variable substitution, encoded through HOAS function application

- introducing a fresh variable, through the predicate form `x:term → ...`
- introducing a new assumption, through the predicate form `P → ...`

We can define three equivalent operations as predicates, for the multiple binding case:

– *a generalization of application*, for substituting all the variables in a `bindmany` .

```
applymany : bindmany A B → list A → B → prop.
applymany (bindbase Body) [] Body.
applymany (bindnext F) (HD :: TL) Body :-
  applymany (F HD) TL Body.
```

– *local introduction of multiple fresh variables at once* within a predicate P; a list of the variables is passed to it.

```
intromany : bindmany A B → (list A → prop) → prop.
intromany (bindbase _) P :- P [].
intromany (bindnext F) P :-
  (x:A → intromany (F x) (fun tl ⇒ P (x :: tl))).
```

– *local introduction of a number of assumptions* of the form `P X Y` within a predicate `Q`. This is intended to be used, for example, for introducing assumptions for predicates such as `typeof`, taking a list of term variables and a list of types, in the same order.

```
assumemany : (A → B → prop) → list A → list B → prop → prop.
assumemany P [] [] Q :- Q.
assumemany P (X :: XS) (Y :: YS) Q :- (P X Y → assumemany P XS YS Q).
```

These predicates are in exact correspondence with the operations we have available for the built-in HOAS function type – save for application being a predicate rather than a term-level construct – so we are able to reap the benefits of HOAS representations for multiple bindings as well.

For convenience, it is also useful to define another predicate that gives access to both the variables introduced in a `bindmany` and the body of the construct as well. This predicate combines `intromany`, for introducing the variables, with `applymany`, for getting the body of the construct, and is defined as follows:

```
openmany : bindmany A B → (list A → B → prop) → prop.
openmany F P :-
  intromany F (pfun xs ⇒ [Body] applymany F xs Body, P xs Body).
```

This definition employs two notational idiosyncrasies of Makam, the λProlog dialect we are using:

`pfun` is syntactic convenience for anonymous predicate literals, allowing us to use the normal syntax for propositions that we use elsewhere, i.e. in clause premises. It is otherwise entirely equivalent to the `fun` construct for anonymous functions.

The square-bracket notation, used above in `[Body]`, introduces a new metavariable; it therefore can be read as existential quantification. Metavariables are allowed to capture all the free variables in scope at the points where they are introduced. For most of them, introduced implicitly in each clause, we capture the free variables in scope when their clauses are executed. In this case, however, it is necessary that `Body` can capture the fresh variables introduced by the `intromany` predicate too, hence the explicit metavariable introduction.

We can now define the typing rule for `lammany` using these predicates, as follows:

```
arrowmany : list typ → typ → typ.

typeof (lammany F) (arrowmany TS T') :-
  openmany F (fun xs body ⇒
    assumemany typeof xs TS (typeof body T')).
```

For example, the following query returns:

```
typeof (lammany (bindnext (fun x ⇒ bindnext (fun y ⇒ bindbase (tuple [x, y]))))) T ?
>> Yes:
>> T := arrowmany [T1, T2] (product [T1, T2])
```

Adding the corresponding appmany construct for simultaneous application is straightforward. We can use the applymany predicate defined above to encode simultaneous substitution for the evaluation rule.

```
appmany : term → list term → term.

typeof (appmany E ES) T' :-
  typeof E (arrowmany TS T'),
  map typeof ES TS.

eval (lammany F) (lammany F).

eval (appmany E ES) V' :-
  eval E (lammany F),
  map eval ES VS,
  applymany F VS E',
  eval E' V'.
```

We can use the bindmany type to encode other constructs, such as mutually recursive definitions, like the let rec construct of ML. In that case, we can capture the right binding structure by introducing a number of variables simultaneously, accessible both when giving the (same number of) definitions and the body of the construct.

We can therefore encode a let rec construct of the form:

```
let rec f = f_def and g = g_def in body
```

as

```
letrec (bindnext (fun f ⇒ bindnext (fun g ⇒ bindbase ([f_def, g_def]))))
       (bindnext (fun f ⇒ bindnext (fun g ⇒ bindbase body)))
```

The type-checking rule would be as follows:

```
letrec : bindmany term (list term) → bindmany term term → term.

typeof (letrec XS_Defs XS_Body) T' :-
  openmany XS_Defs (pfun xs defs ⇒
    assumemany typeof xs TS (map typeof defs TS)
  ),
  openmany XS_Body (pfun xs body ⇒
    assumemany typeof xs TS (typeof body T')
  ).
```

Still, even though this encoding matches the binding structure correctly, it is unsatisfying, as it does not guarantee that the same number of variables are introduced in both cases and that the same number of definitions are given. Though this requirement is enforced at the level of the typing rules, it would be better if we could enforce it at the syntax level. This would require some sort of dependency, though, which at first does not seem possible to do in λProlog.

4 DEPENDENT BINDING

The type system of λProlog can be viewed as a particular subset of System F_ω: namely, it is the simply typed lambda calculus extended with prenex polymorphism and simple type constructors of the form `type → type → ... → type`. (As an aside, `prop` can be viewed as a separate sort, but we take the view that it is just a distinguished extensible type.)

As is well-known from Haskell even before the addition of kind definitions, type promotion, and type-in-type, this subset of System F_ω is enough to model some form of dependency. For example, we can introduce two types for modelling natural numbers, then define vectors as a GADT using them:

```
natZ : type.
natS : type → type.

vector : type → type → type.
vnil : vector natZ A.
vcons : A → vector N A → vector (natS N) A.
```

In fact, λProlog naturally supports pattern-matching over such constructors as well, through *ad-hoc polymorphism*, where polymorphic type variables are allowed to be instantiated at *runtime* rather than at type-checking time. The mechanism through which ad-hoc polymorphism works in λProlog is simple: before performing unification at the term-level, we perform unification at the type level first, therefore further determining any uninstantiated type variables. Therefore, when we check to see whether the current goal matches the premise of a rule, type unification can force us to distinguish between different types. Based on these, the standard example of `map` for vectors is as follows:

```
vmap : [N] (A → B → prop) → vector N A → vector N B → prop.
vmap P vnil vnil.
vmap P (vcons X XS) (vcons Y YS) :- P X Y, vmap P XS YS.
```

The notation `[N]` in the type of `vmap` means that the type argument `N` is ad-hoc/not-parametric. Non-specified type arguments are parametric by default, so as to match standard practice in languages like ML and Haskell, and to catch type errors that allowing unqualified ad-hoc polymorphism would permit. For example, consider the following erroneous definition for `fold`, where the arguments for `P` in the `cons` case are flipped.

```
foldl : (B → A → B → prop) → B → list A → B → prop.
foldl P S nil S.
foldl P S (cons HD TL) S'' <- P HD S S', foldl P S' TL S''.
```

If ad-hoc polymorphism is allowed for `A` and `B`, the definition is well-typed. However, the erroneous call to `P` forces the types `A` and `B` to be unified, and therefore the `fold` predicate is unnecessarily restricted to only work when the two types are the same. Having to specify ad-hoc polymorphism explicitly helps us avoid such mistakes.

Though this support for ad-hoc polymorphism was part of the original λProlog design, we have not found extensive coverage of its implications in the literature. Furthermore, it is not supported well by standard implementations of λProlog (like Teyjus), which was one of the reasons that prompted us to work on Makam.

Armed with GADTs of this form, we can now introduce dependently typed binding forms, where the number of variables that are being bound is reflected in the type. One approach uses a type of the form `dbind A N B`, standing for a dependently typed binding of `N` fresh variables of type `A` into a body of type `B`. `N` will be instantiated with `natZ` and `natS` as above.

```
dbind : type → type → type → type.
```

```
dbindbase : B → dbind A natZ B.
dbindnext : (A → dbind A N B) → dbind A (natS N) B.
```

Another possibility, avoiding the need for introducing type-level natural numbers, is to use a more standard type as the dependent parameter: the type of tuples that would serve as substitutions for the introduced variables. The type would then become:

```
dbind : type → type → type → type.

dbindbase : B → dbind A unit B.
dbindnext : (A → dbind A T B) → dbind A (A * T) B.
```

The definitions for helper predicates, such as `intromany`, `applymany`, etc., follow the case for `bindmany` closely, only with more precise types. We first define a helper type `subst A T` that is equivalent to the type of tuples `T` we expect, which is not strictly necessary but allows for more precise types:

```
subst : type → type → type.
nil : subst A unit.
cons : A → subst A T → subst A (A * T).
```

The predicates are now defined as follows. First, their types are:

```
intromany : [T] dbind A T B → (subst A T → prop) → prop.
applymany : [T] dbind A T B → subst A T → B → prop.
openmany : [T] dbind A T B → (subst A T → B → prop) → prop.
```

Note that we are reusing the same predicate names as before. Makam allows overloading for all variable names; expected types are taken into account for resolving variables and disambiguating between them, as has been long known to be possible in the bi-directional type-checking literature. Type ascription is used when variable resolution is ambiguous. We also sometimes avoid overloading for constructors; having unambiguous types for constructors means that they can be used to resolve ambiguity between overloaded predicates easily. However, here we reuse the `nil` and `cons` constructors for `subst` so that we can use the sugared form for list-like datatypes (using `[]` and `::`).

```
intromany (dbindbase F) P :- P nil.
intromany (dbindnext F) P :-
  (x:A → intromany (F x) (pfun t ⇒ P (x :: t))).

applymany (dbindbase Body) [] Body.
applymany (dbindnext F) (X :: XS) Body :- applymany (F X) XS Body.

openmany F P :-
  intromany F (pfun xs ⇒ [Body] applymany F xs Body, P xs Body).
```

Also, we define predicates analogous to `map` and `assumemany` for the `subst` type:

```
assumemany : [T T'] (A → B → prop) → subst A T → subst B T' → prop → prop.
assumemany P [] [] Q :- Q.
assumemany P (X :: XS) (Y :: YS) Q :- (P X Y → assumemany P XS YS Q).

map : [T T'] (A → B → prop) → subst A T → subst B T' → prop.
map P [] [].
map P (X :: XS) (Y :: YS) :- P X Y, map P XS YS.
```

(Here we have not captured the relationship between the type of tuples `T` and `T'` precisely, namely that one structurally matches the other with `As` replaced by `Bs`. With a more complicated presentation, we could capture it by adding another argument of a dependent type.)

Using this type, we can define `letrec` as follows:

```
letrec : dbind term T (subst term T) → dbind term T term → term.
```

This encoding captures the binding structure of the construct precisely: we need the same number of definitions as the number of variables we introduce, and the body of the construct needs exactly the same number of variables bound.

The typing rule is entirely similar to the one we had previously:

```
typeof (letrec Defs Body) T' :-
  openmany Defs (pfun xs defs ⇒
    assumemany typeof xs TS (map typeof defs TS)
  ),
  openmany Body (pfun xs body ⇒
    assumemany typeof xs TS (typeof body T')
  ).
```

4.1 Patterns

We can also use the same ‘dependency’ trick for other, more complicated forms of binding. One such example, which we sketch below, is linear ordered binding as in the case of patterns. The point is that having explicit support in our metalanguage only for single-variable binding, as is standard in HOAS, together with the two kinds of polymorphism we have shown, is enough. Using them, we can encode complicated binding forms, that often require explicit support in other meta-linguistic settings (e.g. Needle + Knot, Unbound, etc.)

At the top level, a single type argument is needed for patterns, representing the list of variables that it uses in the order that they are used. Each variable can only be used once, so at the level of patterns, there is not really a notion of binding: pattern variables are “introduced” at their point of use. However, the list of variables that we build up can be reused in order to be actually bound into a term – e.g. the body of a pattern-matching branch.

(Single-variable binding is really a way to introduce a “point” in an AST that we can “refer back to” from its children; or the means to introduce sharing in the notion of ASTs, allowing to refer to the same “thing” a number of times. There is no sharing going on inside patterns, though; hence no binding constructs are needed for encoding the patterns themselves.)

Each sub-pattern that makes up a pattern needs to depend on two arguments, in order to capture the linearity – the fact that variables “go away” after their first uses. The first argument represents all the variables that can be used, initially matching the type argument of the top-level pattern; the second argument represents the variables that “remain” to be used after this sub-pattern is traversed. We use “initially” and “after” to refer to the order of visiting the sub-patterns in a structural, depth-first traversal of the pattern. The “difference” between the types corresponds to the variables that each particular sub-pattern uses.

To make the presentation cleaner, we will introduce a single type for patterns that has both arguments, with the requirement that for top-level arguments, no variables remain.

```
patt : type → type → type.
```

(Probably hidden: add natural numbers so that we can have a simple example of patterns.)

```
nat : typ.
zero : term.
succ : term → term.
typeof zero nat.
typeof (succ N) nat :- typeof N nat.
eval zero zero.
eval (succ E) (succ V) :- eval E V.
```

The pattern for zero does not use any variables; the pattern `succ P` for successor uses the same variables that `P` does.

```
patt_zero : patt T T.
patt_succ : patt T T' → patt T T'.
```

A single pattern variable declares/uses exactly itself.

```
patt_var : patt (term * T) T.
```

A wildcard pattern does not use any variables.

```
patt_wild : patt T T.
```

n-ary tuples require a type for pattern lists:

```
pattlist : type → type → type.
patt_tuple : pattlist T T' → patt T T'.

nil : pattlist T T.
cons : patt T1 T2 → pattlist T2 T3 → pattlist T1 T3.
```

We can now encode a single-branch "case-or-else" statement as follows:

```
case_or_else : term → patt T unit → dbind term T term → term → term.
```

The first argument is the scrutinee; the second is the pattern; the third is the branch body, where we bind the same number of variables as the ones used in the pattern; and the last argument is the `else` case.

The typing relation for patterns is defined as follows: given a pattern and a list of types for the variables that remain after the pattern, yield a list of types for all the variables that are available, plus the type of the pattern.

```
typeof : [T T' Ttyp T'typ] patt T T' → subst typ T'typ → subst typ Ttyp → typ → prop.

typeof patt_var S' (cons T S') T.
typeof patt_wild S S T.
typeof patt_zero S S nat.
typeof (patt_succ P) S' S nat :-
  typeof P S' S nat.

typeof :
  [T T' Ttyp T'typ] pattlist T T' → subst typ T'typ → subst typ Ttyp → list typ → prop.

typeof (patt_tuple PS) S' S (product TS) :-
  typeof PS S' S TS.

typeof [] S S [].
typeof (P :: PS) S3 S1 (T :: TS) :-
  typeof PS S3 S2 TS, typeof P S2 S1 T.

typeof (case_or_else Scrutinee Pattern Body Else) T' :-
  typeof Scrutinee T,
  typeof Pattern nil TS T,
  openmany Body (pfun xs body ⇒
     (assumemany typeof xs TS (typeof body T'))
  ),
  typeof Else T'.
```

In order to define evaluation rules, we could define a predicate that models unification between patterns and terms. However, we can do better than that: we can re-use the existing support for unification from the metalanguage! In that case, all we need is a way to convert a pattern into a term, with pattern variables replaced by *meta-level metavariables*. The metavariables are introduced at each conversion rule as needed, and will get instantiated to the right terms if unification with the scrutinee succeeds.

```
patt_to_term : [T T'] patt T T' → term → subst term T' → subst term T → prop.
patt_to_term patt_var X Subst (X :: Subst).
patt_to_term patt_wild _ Subst Subst.
patt_to_term patt_zero zero Subst Subst.
patt_to_term (patt_succ PN) (succ EN) Subst' Subst :- patt_to_term PN EN Subst' Subst.

pattlist_to_termlist : [T T'] pattlist T T' → list term → subst term T' → subst term T → prop.

patt_to_term (patt_tuple PS) (tuple ES) Subst' Subst :-
  pattlist_to_termlist PS ES Subst' Subst.

pattlist_to_termlist [] [] Subst Subst.
pattlist_to_termlist (P :: PS) (T :: TS) Subst3 Subst1 :-
  pattlist_to_termlist PS TS Subst3 Subst2,
  patt_to_term P T Subst2 Subst1.

eval (case_or_else Scrutinee Pattern Body Else) V :-
  patt_to_term Pattern TermWithUnifvars [] Unifvars,
  if (eq Scrutinee TermWithUnifvars)  (* reuse unification from the meta-language *)
  then (applymany Body Unifvars Body', eval Body' V)
  else (eval Else V).
```

Two new things here: if-then-else has the semantics described in the LogicT monad paper. eq is a predicate that forces its arguments to be unified, defined simply as:

```
eq : A → A → prop.
eq X X.
```

Here is an example of pattern matching: predecessor for natural numbers.

```
(eq _PRED
  (lam _ (fun n ⇒
    case_or_else n
      (patt_succ patt_var) (dbindnext (fun pred ⇒ dbindbase pred))
      zero
      )),
 typeof _PRED T,
 eval (app _PRED zero) PRED_OF_ZERO,
 eval (app _PRED (succ (succ zero))) PRED_OF_TWO) ?
>> Yes:
```

```
>> T := arrow nat nat
>> PRED_OF_ZERO := zero
>> PRED_OF_TWO := succ zero
```

5 MORE ML-LIKE LANGUAGE

Let us now proceed to encode more features of a programming language like ML using the techniques we have seen so far.

First we add polymorphism, therefore extending our simply typed lambda calculus to System F. We will only consider the explicit polymorphism case for the time being, leaving type inference for later.

We need a type for quantification over types, as well as term-level constructs for functions over types and instantiating a polymorphic function with a specific type.

```
forall : (typ → typ) → typ.
lamt : (typ → term) → term.
appt : term → typ → term.
```

The typing rules are similarly straightforward.

```
typeof (lamt E) (forall T) :-
  (a:typ → typeof (E a) (T a)).

typeof (appt E T) (TF T) :-
  typeof E (forall TF).
```

One thing to note is that in a pen-and-paper version, we would need to define a new context that keeps track of type variables that are in scope (typically named Δ), and an auxiliary judgement of the form $\Delta \vdash \tau$ wf that checks that all type variables used in τ are in scope. Here we get type well-formedness for free. Furthermore, if we had to keep track of further information about type variables (e.g. their kinds), we could have added an assumption of the form `kindof a K →`. Since the local assumption context can carry rules for any predicate, no extra declaration or change to the existing rules would be needed, as would be required in the pen-and-paper version in order to incorporate the new Δ context.

With these additions, we can give a polymorphic type to the identity function:

```
typeof (lamt (fun a ⇒ lam a (fun x ⇒ x))) T ?
```

Moving on towards a more ML-like language, we would like to add the option to declare algebraic datatypes. We must first introduce a notion of top-level programs, each composed of a series of declarations of types and terms, as well as a predicate to check that a program is well-formed:

```
program : type.
wfprogram : program → prop.
```

Let us add `let` definitions as a first example of a program component, each introducing a term variable that can be used in the rest of the program:

```
let : term → (term → program) → program.

wfprogram (let E P) :-
  typeof E T,
  (x:term → typeof x T → wfprogram (P x)).
```

We also need a "last" component for the program, typically a main expression:

```
main : term → program.
```

```
    wfprogram (main E) :-
      typeof E _.
```

Let us now proceed to algebraic datatypes. A datatype has a name, a number of type parameters, and a list of constructors; constructors themselves have names and lists of arguments:

```
    typeconstructor : type → type.
    constructor : type.

    ctor_declaration : type → type.
    nil : ctor_declaration unit.
    cons : list typ → ctor_declaration T →
             ctor_declaration (constructor * T).
    datatype_declaration : type → type → type.
    datatype_declaration :
      (typeconstructor Arity → dbind typ Arity (ctor_declaration Constructors)) →
      datatype_declaration Arity Ctors.

    datatype :
      datatype_declaration Arity Constructors →
      (typeconstructor Arity → dbind constructor Constructors program) →
      program.
```

The datatype introduces a type constructor, as well as a number of constructors, in the rest of the program. Here we use dependency to carry the arity of the type constructor in its meta-level type, avoiding the need for a well-formedness predicate for types. Of course, in situations where object-level types are more complicated, we would need to incorporate kind checking into our predicates.

Let us now proceed to well-formedness for datatype declarations. We will need two auxiliary predicates: one that keeps information about a constructor – which type it belongs to, what arguments it expects; and another one that abstracts over the type variables used in the datatype declaration, creating a polymorphic type for the type of the constructor, which can be instantiated with different types at different places.

```
    constructor_info :
      typeconstructor Arity → constructor → dbind typ Arity (list typ) → prop.

    constructor_polytypes : [Arity Ctors PolyTypes]
      subst typ Arity →
      ctor_declaration Ctors → subst (dbind typ Arity (list typ)) PolyTypes → prop.

    constructor_polytypes _ [] [].
    constructor_polytypes TypVars (CtorType :: CtorTypes) (PolyType :: PolyTypes) :-
      applymany PolyType TypVars CtorType,
      constructor_polytypes TypVars CtorTypes PolyTypes.
```

One interesting part intereaction is in the two `applymany` calls: these are used in the opposite direction than what we have used it so far, getting `TypVars` and `CtorType` as inputs and producing `PolyType` as an output. We need to be careful, though, to make sure that `PolyType` cannot capture the `TypVars` variables:

```
    wfprogram (datatype (datatype_declaration ConstructorDecls) Program') :-
      (dt:(typeconstructor T) → ([PolyTypes]
        openmany (ConstructorDecls dt) (pfun tvars constructor_decls ⇒ (
          constructor_polytypes tvars constructor_decls PolyTypes)),
        openmany (Program' dt) (pfun constructors program' ⇒
          assumemany (constructor_info dt) constructors PolyTypes
          (wfprogram program')))).
```

In order to be able to refer to datatypes and constructors, we will need type- and term-level formers.

```
tconstr : typeconstructor T → subst typ T → typ.
constr : constructor → list term → term.

typeof (constr Constructor Args) (tconstr TypConstr TypArgs) :-
  constructor_info TypConstr Constructor PolyType,
  applymany PolyType TypArgs Typs,
  map typeof Args Typs.
```

We will also need patterns:

```
patt_constr : constructor → pattlist T T' → patt T T'.

typeof (patt_constr Constructor Args) S' S (tconstr TypConstr TypArgs) :-
  constructor_info TypConstr Constructor PolyType,
  applymany PolyType TypArgs Typs,
  typeof Args S' S Typs.
```

As an example, we will define lists and their append function:

```
wfprogram
  (datatype
    (datatype_declaration (fun llist ⇒ dbindnext (fun a ⇒ dbindbase (
    [ [] (* nil *) ,
      [a, tconstr llist [a]] (* cons of a * list a *) ]))))
  (fun llist ⇒ dbindnext (fun lnil ⇒ dbindnext (fun lcons ⇒ dbindbase (
  (main
    (letrec
      (dbindnext (fun append ⇒ dbindbase (
      [ lamt (fun a ⇒ lam (tconstr llist [a]) (fun l1 ⇒ lam _ (fun l2 ⇒
        case_or_else l1
          (patt_constr lcons [patt_var, patt_var])
            (dbindnext (fun hd ⇒ dbindnext (fun tl ⇒ dbindbase (
            constr lcons [hd, app (app (appt append _) tl) l2]))))
          l2))) ])))
      (dbindnext (fun append ⇒ dbindbase (
    (app (app (appt append _)
      (constr lcons [zero, constr lnil []]))
      (constr lcons [zero, constr lnil []]))
)))))))))) ?
```

The semantics, if needed:

```
patt_to_term (patt_constr Constructor Args) (constr Constructor Args') S' S :-
  pattlist_to_termlist Args Args' S' S.
```

```
eval (constr C Args) (constr C Args') :-
  map eval Args Args'.

eval : program → program → prop.

eval (let E P') P'' :-
  eval E V, eval (P' V) P''.

eval (datatype D P') (datatype D P'') :-
  (dt:(typeconstructor T) →
    intromany CS (pfun cs ⇒ ([P'c P''c]
    applymany (P' dt) cs P'c,
    applymany (P'' dt) cs P''c,
    eval P'c P''c))).

eval (main E) (main V) :-
  eval E V.
```

Example:

```
(eq _PROGRAM (

    (datatype
      (datatype_declaration (fun llist ⇒ dbindnext (fun a ⇒ dbindbase (
      [ [] (* nil *) ,
        [a, tconstr llist [a]] (* cons of a * list a *) ]))))
      (fun llist ⇒ dbindnext (fun lnil ⇒ dbindnext (fun lcons ⇒ dbindbase (

    (main (constr lcons [zero, constr lnil []]))

    )))))),

 wfprogram _PROGRAM,
 eval _PROGRAM FINAL) ?
```

6 ADDING TYPE SYNONYMS

Let us proceed to add type synonyms:

```
type_synonym : dbind typ T typ → (typeconstructor T → program) → program.

type_synonym_info : typeconstructor T → dbind typ T typ → prop.

wfprogram (type_synonym Syn Program') :-
  (t:(typeconstructor T) →
   type_synonym_info t Syn →
   wfprogram (Program' t)).
```

Simple enough. How to typecheck them though? We need something like the conversion rule:

$$\frac{\Gamma \vdash e : \tau \qquad \tau =_{\delta} \tau'}{\Gamma \vdash e : \tau'}$$

Here $=_{\delta}$ means equality up to expanding type synonyms.

We will need a type equality predicate:

```
teq : typ → typ → prop.
```

A naive attempt at the conversion rule would be:

```
typeof E T :- typeof E T', teq T T'.
```

However, it is easy to see that this rule leads to divergence: it does a recursive call to itself.

We can do a bit better. We only need to use the conversion rule in cases where we already know something about the type T of the expression, but our typing rules do not match that type. In bi-directional typing parlance, instead of analyzing the type T of the expression E, we want to synthesize the type starting from a new meta-variable T', and then check that the two types are equal using teq. So we need to change our rule to only apply in the case where T starts with a concrete type constructor, rather than when it is an uninstantiated meta-variable.

It is typical for a logic programming language to have a predicate that only succeeds when a specific term is uninstantiated (usually called var). In Makam this is called refl.isunif – the refl namespace prefix standing for the fact that we call these kinds of predicates "reflective", as they give us extra-logical information about the form of a term (sometimes referred to as "meta-predicates" in Prolog parlance). Our second attempt thus looks as follows:

```
typeof E T :- not(refl.isunif T), typeof E T', teq T T'.
```

Upon further consideration, we see that this rule leads to an infinite loop as well: since teq should be reflective, for every proof of typeof E T' through the other rules, a new proof using this rule will be discovered, which will lead to another proof for it, etc. One way to fix it is to make sure that this rule is only used once at the end, if typing using the other rules fails:

```
typeof, typeof_cases, typeof_conversion : term → typ → prop.
typeof E T :-
  if (typeof_cases E T)
  then success
  else (typeof_conversion E T).
typeof_cases (app E1 E2) T' :-
  typeof E1 (arrow T1 T2),
  typeof E2 T1.
...
typeof_conversion E T :-
  not(refl.isunif T), typeof_cases E T', teq T T'.
```

However, this would require changing every typing rule we had. Instead, we can do a trick, to force the rule to only fire once for each expression E, remembering the fact that we have used the rule already:

```
already_in : [A] A → prop.
typeof E T :-
  not(refl.isunif T),
  not(already_in (typeof E)),
  (already_in (typeof E) → typeof E T'),
  teq T T'.
```

Also, we need to make sure that we also take the conversion rule into account for patterns:

```
typeof (P : patt A B) S' S T :-
  not(refl.isunif T),
  not(already_in (typeof P)),
  (already_in (typeof P) → typeof P S' S T'),
  teq T T'.
```

Now let's go and define the actual `teq` predicate. A first approach would be to just write out each case individually:

```
teq (arrow T1 T2) (arrow T1' T2') :- teq T1 T1', teq T2 T2'.
teq (product TS) (product TS') :- map teq TS TS'.
teq (arrowmany TS T) (arrowmany TS' T') :- teq T T', map teq TS TS'.
teq nat nat.
teq (forall T) (forall T') :- (x:typ → teq x x → teq (T x) (T' x)).
teq (tconstr TC Args) (tconstr TC Args') :- map teq Args Args'.
teq (tconstr TC Args) T' :-
  type_synonym_info TC Syn,
  applymany Syn Args T,
  teq T T'.
teq T' (tconstr TC Args) :-
  type_synonym_info TC Syn,
  applymany Syn Args T,
  teq T' T.
```

Only the two last cases are important; the rest is boilerplate that performs structural recursion through the type. Can we do better than that?

Let us ruminate on a possible solution. We want to handle the case where we have a constructor applied to a number of arguments generically, so roughly something like:

```
teq (Constructor Arguments) (Constructor Arguments') :-
  map teq Arguments Arguments'.
```

What we mean here, taking the `arrow` rule as an example, is that `Constructor` would match with `arrow`, and `Arguments` would get instantiated with the list of arguments of the constructor. One thing to be careful about though is that the types of arguments are not all the same. As a result, instead of a homogeneous list, we need a heterogeneous list. This is simple to represent using the existential type, dyn:

```
dyn : type.
dyn : A → dyn.
```

So the type of `Arguments` should be `list dyn` rather than `list typ`. The type of `teq` will need to be changed, so that we can apply it for any different type, rather than just `typ`:

```
teq : [A] A → A → prop.
```

Furthermore, since we have a heterogeneous list, we need a `map` that uses polymorphic recursion: it needs take a polymorphic function as an argument, so that it can be used at different types for different elements of the list.

```
dyn.map : (forall A. [A] A → A → prop) → list dyn → list dyn → prop.
```

This is in contrast to a type like `[A] (A → A → prop) → list dyn → list dyn → prop`, which would instantiate the type `A` to the type of the first element of the list, making further applications to different types impossible.

Makam currently does not provide higher-rank types as the above type suggests – nor do any λProlog implementations that we are aware of. Instead, it provides a way to side-step this issue, through a predicate that replaces polymorphic type variables with fresh variables, allowing it to be instantiated with new types. This is called dyn.call, and dyn.map can be defined in terms of that:

```
dyn.call : [B] (A → A → prop) → B → B → prop.
dyn.map : (A → A → prop) → list dyn → list dyn → prop.
dyn.map P [] [].
dyn.map P (HD :: TL) (HD' :: TL') :- dyn.call P HD HD', dyn.map P TL TL'.
```

(dyn.call is itself defined in terms of a more fundamental predicate dyn.duphead that creates a fresh version of a single polymorphic constructor with fresh type variables.)

Based on these, the only thing missing is a way to actually check whether a term is a ground term that can be decomposed into a constructor and a list of arguments. Makam provides this in the form of the refl.headargs predicate:

```
refl.headargs : B → A → list dyn → prop.
```

(Other Prolog implementations also provide predicates towards the same effect; for example, SWI-Prolog provides compound_name_arguments which is quite similar. Though such predicates are not typical in other λProlog implementations, they should not be viewed as a hack: we could always define these within the language if we maintained a discipline, where we added a rule to refl.headargs for every constructor that we introduce. For example:

```
arrowmany : list typ → typ → typ.
refl.headargs (arrowmany TS T) [arrowmany, [dyn TS, dyn T]].
```

Maybe taking extra care to check that we are not instantiating a unification by using refl.isunif.)

We are now ready to proceed to defining the boilerplate generically. We will do this as a reusable higher-order predicate for structural recursion, that we will use to implement teq. We will define it in open recursion style, providing the predicate to use on recursive calls as an argument:

```
structural_recursion : [B] (A → A → prop) → B → B → prop.

structural_recursion Rec X Y :-
  refl.headargs X Constructor Arguments,
  dyn.map Rec Arguments Arguments',
  refl.headargs Y Constructor Arguments'.
```

We also need to handle built-in types, such as the meta-level int and string types, in case they are used as an argument with other constructors:

```
structural_recursion Rec (X : string) (X : string).
structural_recursion Rec (X : int) (X : int).
```

And last, we need to handle the case of the meta-level function type as well:

```
structural_recursion Rec (X : A → B) (Y : A → B) :-
  (x:A → structural_recursion Rec x x → structural_recursion Rec (X x) (Y x)).
```

We are done! Now we can define teq using structural_recursion, through an auxiliary predicate called teq_aux. We only need to define the non-trivial cases for it, using structural_recursion for the rest, while tying the open recursion knot at the same time:

```
teq_aux : [A] A → A → prop.

teq T T' :- teq_aux T T'.
```

```
teq_aux T T' :-
  structural_recursion teq_aux T T'.

teq_aux (tconstr TC Args) T' :-
  type_synonym_info TC Synonym,
  applymany Synonym Args T,
  teq_aux T T'.

teq_aux T' (tconstr TC Args) :-
  type_synonym_info TC Synonym,
  applymany Synonym Args T,
  teq_aux T' T.
```

Other than minimizing the boilerplate, the great thing about using `structural_recursion` is that no adaptation needs to be done when we add any new constructor to our `typ` datatype – even if that uses new types that we have not defined before. For example, we did not have to take any special provision to handle types we defined earlier such as `dbind` – everything works out thanks to the reflective predicates we are using. (Mention something about the expression problem?)

The one form of terms that `structural_recursion` does not handle are uninstantiated unification variables. We find that leaving that as something that we handle whenever we define a new predicate that uses `structural_recursion` works fine. In this case, `teq` is only supposed to be used with ground terms, so it is fine if we fail when we encounter a unification variable.

Let's try an example out:

```
ascribe : term → typ → term.
typeof (ascribe E T) T :- typeof E T.

wfprogram (
  (type_synonym (dbindnext (fun a ⇒ dbindbase (product [a, a])))
  (fun bintuple ⇒

  main (lam (tconstr bintuple [product [nat, nat]])
            (fun x ⇒
    case_or_else x
   (patt_tuple [patt_tuple [patt_wild, patt_wild], patt_tuple [patt_wild, patt_wild]])
    (dbindbase (tuple []))
    (tuple [])
  ))
))) ?
>> Yes.
```

Let's make sure we don't diverge on type error:

```
ascribe : term → typ → term.
typeof (ascribe E T) T :- typeof E T.

wfprogram (
  (type_synonym (dbindnext (fun a ⇒ dbindbase (product [a, a])))
  (fun bintuple ⇒
  main (lam (tconstr bintuple [product [nat, nat]])
            (fun x ⇒
    case_or_else x
    (patt_tuple [patt_tuple [patt_wild], patt_tuple [patt_wild, patt_wild]])
    (dbindbase (tuple []))
    (tuple [])
  ))
))) ?
>> Impossible.
```

7 CONTEXTUAL TYPES

Let us now add one more meta level: make our object language a meta-language as well! That is, we will add the ability to our object language to manipulate terms of a different object language, in a type-safe manner. This is similar, for example, to heterogeneous meta-programming in MetaHaskell; however, the setting we have in mind is closer to metalanguages where the object language is a logic, similar to Beluga (where the object language is LF) and VeriML (where the object language is the λHOL logic).

We will follow the construction of (cite my dissertation), but using a simpler object language. We will first define the notion of *dependent objects*. These are objects that are external to the language that we have seen so far, but we will add a standard dependently typed subsystem to our language over them. (Similar to the DML construction/ the DML generalization by Licata and Harper.) Dependent objects are classified through *dependent classifiers*:

```
depobject, depclassifier : type.
depclassify : depobject → depclassifier → prop.
```

We also have a (perhaps non-trivial) substitution operation for terms containing a variable of type `depobject`:

```
depsubst : [A] (depobject → A) → depobject → A → prop.
```

(In the simple case this could just be the built-in function application:

```
depsubst F X (F X).
```

We could have something more complicated than that though.)

Now let's add the standards: dependent functions and dependent products:

```
lamdep : depclassifier → (depobject → term) → term.
appdep : term → depobject → term.
packdep : depobject → term → (depobject → typ) → term.
unpackdep : term → (depobject → term → term) → term.

pidep : depclassifier → (depobject → typ) → typ.
sigdep : depclassifier → (depobject → typ) → typ.

typeof (lamdep DC EF) (pidep DC TF) :-
  (x:depobject → depclassify x DC → typeof (EF x) (TF x)).

typeof (appdep E DO) T' :-
  typeof E (pidep DC TF),
  depclassify DO DC,
  depsubst TF DO T'.

typeof (packdep DO E TYPF) (sigdep DC TYPF) :-
  depclassify DO DC,
  depsubst TYPF DO T',
  typeof E T'.

typeof (unpackdep E F) T' :-
  typeof E (sigdep DC TYPF),
  (do:depobject → x:term → depclassify do DC → typeof x (TYPF do) →
   typeof (F do x) T').
```

OK, let's now add a very simple object language – the simply typed lambda calculus. Let's go again… or actually, let's just import what we have already in a separate namespace:

```
%import "01-base-language" as object.

%extend object.
intconst : int → term.
intplus : term → term → term.

tint : typ.

typeof (intconst _) tint.
typeof (intplus E1 E2) tint :- typeof E1 tint, typeof E2 tint.
%end.
```

(Note: we're just importing the previous literate development into a different namespace. Unfortunately I can't import the further developments right now, probably some issue with the importing code, but I think it's fine to skip for now. We could go with just defining a new language anew though.)

Now let's turn these into dependent objects:

```
term : object.term → depobject.
typ : object.typ → depobject.

typ : object.typ → depclassifier.
ext : depclassifier.

depclassify (term E) (typ T) :- object.typeof E T.
```

To classify types, we'll need to make sure they're well-formed. For the time being, all types are well-formed by construction, but let's prepare for the future:

```
%extend object.
wftype : typ → prop.
wftype_cases : [A] A → A → prop.

wftype T :- wftype_cases T T.
wftype_cases T T :- structural wftype_cases T T.
%end.
```

```
depclassify (typ T) ext :- object.wftype T.
```

Let’s proceed to substitution:

```
depsubst_aux, depsubst_cases : [A] depobject → depobject → A → A → prop.
depsubst F X Res :- (x:depobject → depsubst_aux x X (F x) Res).
depsubst_aux Var Replace Where Result :-
  if (depsubst_cases Var Replace Where Result)
  then (success)
  else (structural_recursion (depsubst_aux Var Replace) Where Result).
```

Alright. Now let’s see what we can do:

```
typeof
  (lamdep ext (fun t ⇒
  (packdep t (tuple []) (fun _ ⇒ product [])))) T ?
```

We can only use the dependent variables as they are, so not much use. The whole point of this though is being able to refer to dependent variables within the object terms:

```
%extend object.
metaterm : depobject → term.
metatyp : depobject → typ.

typeof (metaterm E) T :-
  refl.isnvar E,
  depclassify E (typ T).

wftype_cases (metatyp T) (metatyp T) :-
  refl.isnvar T,
  depclassify T ext.
%end.

depsubst_cases Var (term Replace) (object.metaterm Var) Replace.
depsubst_cases Var (typ Replace) (object.metatyp Var) Replace.
```

OK, now let’s see if we can do better:

```
typeof
  (lamdep ext (fun t ⇒
  (packdep
     (term (object.lam (object.metatyp t) (fun x ⇒ x)))
     (tuple []) (fun _ ⇒ product [])))) T ?
```

We can also handle the case of non-closed terms, using contextual types:

```
%extend object.
subst : type → type.
subst : list A → subst A.

ctx : type → type.
ctx : subst typ → bindmany term A → ctx A.

openctx : ctx A → (subst term → subst typ → A → prop) → prop.
```

```
applyctx : ctx A → subst term → A → prop.

openctx (ctx Types Binds) P :-
  openmany Binds (pfun vars body ⇒
    P (subst vars) Types body
  ).

applyctx (ctx _ Binds) (subst Args) Result :-
  applymany Binds Args Result.

map : (A → B → prop) → subst A → subst B → prop.
map P (subst L) (subst L') :- map P L L'.
%end.

openterm : object.ctx object.term → depobject.
ctxtyp : object.subst object.typ → object.typ → depclassifier.

depclassify (openterm CtxE) (ctxtyp Typs T) :-
  object.openctx CtxE (pfun vars typs e ⇒ [Units]
    object.map (pfun t u ⇒ object.wftype t) typs (Units : object.subst unit),
    object.map eq typs Typs,
    object.typeof e T).
```

And one last step: let’s reify open terms back into the language:

```
%extend object.
metaterm : depobject → subst term → term.

typeof (metaterm E ES) T :-
  refl.isnvar E,
  depclassify E (ctxtyp Typs T),
  object.map object.typeof ES Typs.
%end.

depsubst_cases Var (openterm CtxE) (object.metaterm Var Subst) Result :-
  object.applyctx CtxE Subst E,
  depsubst_aux Var (openterm CtxE) E Result.
```

Let’s try the final thing:

```
(eq _FUNCTION
  (lamdep ext (fun t1 ⇒
    (lamdep ext (fun t2 ⇒
   (lamdep (ctxtyp (object.subst [object.metatyp t1]) (object.metatyp t2)) (fun x_e ⇒
   (packdep (openterm (object.ctx (object.subst []) (bindbase (object.lam _ (fun x ⇒
      object.tuple [object.metaterm x_e #SUBST, object.intconst 5]
    ))))) (tuple []) (fun _ ⇒ product []))))))))),
 typeof _FUNCTION FUNCTION_TYPE,
 typeof
  (appdep (appdep
    _FUNCTION
    (typ object.tint))
    (typ (object.product [object.tint])))
 APPLIED_TYPE) ?
>> Yes:
>> SUBST := fun t1 t2 x_e x ⇒ subst (cons x nil),
>> FUNCTION_TYPE :=
>>  pidep ext (fun t1 ⇒
>>  pidep ext (fun t2 ⇒
>>  pidep (ctxtyp (object.subst (cons (object.metatyp t1) nil)) (object.metatyp t2))
>>  (fun x_e ⇒
>>    sigdep
>>      (ctxtyp (subst nil)
>>       (arrow
>>         (object.metatyp t1)
>>         (product (cons (object.metatyp t2) (cons tint nil)))))
>>     (fun _ ⇒ product nil)))),
>> APPLIED_TYPE :=
>>  pidep (ctxtyp
>>    (object.subst (cons object.tint nil))
>>    (object.product (cons object.tint nil)))
>>  (fun x_e ⇒
>>    sigdep (ctxtyp
>>      (subst nil)
>>      (arrow
>>        object.tint
>>        (product (cons (object.product (cons object.tint nil)) (cons tint nil)))))
>>    (fun _ ⇒ product nil))
```

Note that we can infer both the type of the lambda abstraction and the substitution itself. Getting to that point in the VeriML implementation took months!

Mention that adding polymorphic contexts and dependent pattern matching as in VeriML is also possible, but we won't show it here.

8 HINDLEY-MILNER POLYMORPHISM

(Text is very much WIP.)

Let's now do Hindley-Milner let-polymorphism:

```
let : term → (term → term) → term.
```

Easy so far.

The inference rule looks like this:

$$\frac{\Gamma \vdash e : \tau \qquad \vec{a} = \mathrm{fv}(\tau) - \mathrm{fv}(\Gamma) \qquad \Gamma, x : \forall \vec{a}.\tau \vdash e' : \tau'}{\Gamma \vdash \mathrm{let}\ x = e\ \mathrm{in}\ e' : \tau'}$$

(We have not added any side-effectful operations, so no need to add a value restriction.)

This is easy to transcribe in Makam, assuming a predicate for generalizing a type:

```
generalize : typ → typ → prop.

typeof (let E F) T' :-
  typeof E T,
  generalize T Tgen,
  (x:term → typeof x Tgen → typeof (F x) T').
```

So we need to do the following:

- something that picks out free-variables from a term – or, in our setting, uninstantiated meta-variables
- something that picks out free-variables from the local context
- a way to turn something that includes meta-variables into a `forall` type

This predicate picks out the first metavariable of a certain type it finds. It uses `generic.fold` which is another generic operation, defined similarly to `structural_recursion`, but which performs a fold over arbitrary types.

```
findunif : [A B] option B → A → option B → prop.
findunif (some X) _ (some X).
findunif none (X : A) (some (X : A)) :- refl.isunif X.
findunif In X Out :- generic.fold findunif In X Out.

findunif : [A B] A → B → prop.
findunif T X :- findunif none T (some X).
```

Note that the second rule, the important one, will only match when we encounter a metavariable of the same type as the one we require, as we do type specialization.

Now let's add something, that given a specific meta-variable and a specific term, replaces the meta-variable with the term. We will see later why this is necessary. Here we will need another reflective predicate, `refl.sameunif` that succeeds when its two arguments are the same exact metavariable. As opposed to unifying two metavariables, this allows us to "pick out" occurrences of a specific metavariable.

```
replaceunif : [A B] A → A → B → B → prop.
replaceunif Which ToWhat Where Result :-
  refl.isunif Where,
  if (refl.sameunif Which Where)
  then (eq (dyn Result) (dyn ToWhat))
  else (eq Result Where).
replaceunif Which ToWhat Where Result :-
  not(refl.isunif Where),
  structural_recursion (replaceunif Which ToWhat) Where Result.
```

A last auxiliary predicate will allow us to check whether a specific metavariable exists within a term:

```
hasunif : [A B] B → bool → A → bool → prop.
hasunif _ true _ true.
hasunif X false Y true :- refl.sameunif X Y.
hasunif X In Y Out :- generic.fold (hasunif X) In Y Out.

hasunif : [A B] A → B → prop.
hasunif Term Var :- hasunif Var false Term true.
```

We are now ready to implement `generalize`. Base case: there exist no unification variables within a type:

```
generalize T T :-
  not(findunif T X).
```

Recursive case: there exists at least one unification variable. We will pick out that unification variable, abstract over it and repeat the process to pick out any remaining ones. We will check whether we are allowed to generalize by getting something that holds all typs in the current variable environment – that is, all Ts for any typeof x T local assumptions – and making sure that the current unification variable does not occur in that. Getting the types in the environment is done through the get_types_in_environment predicate, and we will leave the type of its result abstract for the time being.

```
get_types_in_environment : [A] A → prop.

generalize T Res :-
  findunif T X,
  (x:typ → (replaceunif X x T (T' x), generalize (T' x) (T'' x))),
  get_types_in_environment Types,
  if (hasunif Types X)
  then (eq Res (T'' X))
  else (eq Res (forall T'')).
```

What can get_types_in_environment be? We could change all our typing rules to add a list argument that holds all the types that we put in the context, and thread it through all our predicates. However, again using reflective predicates, there is an easier way to do that: we can simply get all the local assumptions for the typeof predicate for terms, which will exactly correspond to the local assumptions for the current set of free variables:

```
get_types_in_environment Assumptions :-
  refl.assume_get (typeof : term → typ → prop) Assumptions.
```

We're done!

Example, easy:

```
typeof (let (lam _ (fun x ⇒ x)) (fun id ⇒ id)) T ?
>> Yes:
>> T := forall (fun a ⇒ arrow a a)
```

Another example, where the problem of naive generalization shows up:

```
typeof (let (lam _ (fun x ⇒ let x (fun y ⇒ y)))
            (fun z ⇒ z)) T ?
>> Yes:
>> T := forall (fun a ⇒ arrow a a)
```

(Just checking the issue where we don't remove all unification variables in the context – this is a hack, if we need to do this we can show the above in two steps instead:)

```
(get_types_in_environment [] →
  typeof (let (lam _ (fun x ⇒ let x (fun y ⇒ y)))
            (fun z ⇒ z)) T) ?
>> Yes:
>> T := forall (fun a ⇒ arrow a (forall (fun b ⇒ b)))
```

9 CONCLUSION

TODO We conclude the paper.

REFERENCES

Dale Miller and Gopalan Nadathur. 1988. An overview of λProlog. In *ICLP*.