Prototyping a Functional Language using Higher-Order Logic Programming: A Functional Pearl on Learning the Ways of λProlog/Makam

ANTONIS STAMPOULIS, Originate Inc., USA
ADAM CHLIPALA, MIT CSAIL, USA

We demonstrate how the framework of *higher-order logic programming*, as exemplified in the λProlog language design, is a prime vehicle for rapid prototyping of implementations for programming languages with sophisticated type systems. We present the literate development of a type checker for a language with a number of complicated features, culminating in a standard ML-style core with algebraic datatypes and type generalization, extended with staging constructs that are generic over a separately defined language of terms. We add each new feature in sequence, with little to no changes to existing code. Scaling the higher-order logic programming approach to this setting required us to develop approaches to challenges like complex variable binding patterns in object languages and performing generic structural traversals of code, making use of novel constructions in the setting of λProlog, such as GADTs and generic programming. For our development, we make use of Makam, a new implementation of λProlog, which we introduce in tutorial style as part of our (quasi-)literate development.

CCS Concepts: • **Software and its engineering → Constraint and logic languages**; *Formal language definitions*; Software prototyping;

Additional Key Words and Phrases: higher-order logic programming, programming language prototyping, metaprogramming

ACM Reference Format:
Antonis Stampoulis and Adam Chlipala. 2018. Prototyping a Functional Language using Higher-Order Logic Programming: A Functional Pearl on Learning the Ways of λProlog/Makam. *Proc. ACM Program. Lang.* 2, ICFP, Article 93 (September 2018), 30 pages. https://doi.org/10.1145/3236788

1 IN WHICH OUR HEROES SET OUT ON A ROAD TO PROTOTYPE A TYPE SYSTEM

HAGOP (Student). (…) So yes, I think my next step should be writing a toy implementation of the type system we have in mind, so that we can try out some examples and see what works and what does not.

ROZA (Advisor). Definitely – trying out examples will help you refine your ideas, too.

HAGOP. Let's see, though; we have the simply typed λ-calculus, some ML core features, a staging construct, and contextual types like in Nanevski et al. [2008]… I guess I will need a few weeks?

ROZA. That sounds like a lot. Why don't you use some kind of metalanguage to implement the prototype?

Authors' addresses: Antonis Stampoulis, Originate Inc. New York, New York, USA, antonis.stampoulis@gmail.com; Adam Chlipala, MIT CSAIL, Cambridge, Massachusetts, USA, adamc@csail.mit.edu.

2475-1421/2018/9-ART93
https://doi.org/10.1145/3236788

HAGOP. You mean a tool like Racket [Felleisen et al. 2015], PLT Redex [Felleisen et al. 2009], the K Framework [Roşu and Şerbănuţă 2010] or Spoofax [Kats and Visser 2010]?

ROZA. Yes, all of those should be good choices. I was thinking though that we could use higher-order logic programming… it's a formalism that is well-suited to what we want to do, since we will need all sorts of different binding constructs, and the type system we are thinking about is quite advanced.

HAGOP. Oh, so you mean λProlog [Miller and Nadathur 2012] or LF [Harper et al. 1993].

ROZA. Yes. Actually, a few years back a friend of mine worked on this new implementation of λProlog just for this purpose – rapid prototyping of languages. It's called Makam. It should be able to handle our MetaML-like language nicely, and we should not need to spend more than a few hours on it!

HAGOP. Sounds great! Anything I can read up on Makam then?

ROZA. Not much, unfortunately… But I know the language and its standard library quite well, so let's work on this together; it'll be fun. I'll tell you all the details along the way too – I know how you always want to understand things in depth!

(Our heroes install Makam from `https://github.com/astampoulis/makam` *and figure out how to run the REPL.)*

2 IN WHICH OUR READERS GET A PREMONITION OF THINGS TO COME

Section 3 serves as a tutorial to λProlog/Makam, showing the basic usage of the language to encode the static and dynamic semantics of the simply typed lambda calculus. *Section 4* explores the question of how to implement multiple-variable binding, culminating in a reusable polymorphic datatype. *Sections 5 and 6* present a novel account of how GADTs are directly supported in λProlog thanks to the presence of ad-hoc polymorphism and showcase their use for accurate encodings of mutually recursive definitions and pattern matching. *Section 7* describes a novel way to define operations by structural recursion in λProlog/Makam while only giving the essential cases, motivating them through the example of encoding a simple conversion rule. The following sections make use of the presented features to implement polymorphism and algebraic datatypes (*Section 8*), heterogeneous staging constructs with contextual typing (*Section 9*) and Hindley-Milner type generalization (*Section 10*). We then summarize and compare to related work.

We encourage readers to skim through the paper as a first pass, focusing on the highlighted code. These highlights provide a rough picture of the Makam code needed to implement a typechecker for a small ML-like language, along with a few key definitions from the standard library.

3 IN WHICH OUR HEROES GET THE EASY STUFF OUT OF THE WAY

HAGOP. OK, let's just start with the simply typed lambda calculus to see how this works. Let's define just the basics: application, lambda abstraction, and the arrow type.

ROZA. Right. We will first need to define the two meta-types for these two sorts:

```
term : type.
typ : type.
```

HAGOP. Oh, so `type` is the reserved keyword for the meta-level kind of types, and we'll use `typ` for our object-level types?

ROZA. Exactly. And let's do the easy constructors first:

```
app : term → term → term.
arrow : typ → typ → typ.
```

HAGOP. So we add constructors to a type at any point; we do not list them out when we define it like in Haskell. But how about lambdas? I have heard that λProlog supports higher-order abstract syntax [Pfenning and Elliott 1988], which should make those really easy to add, too, right?

ROZA. Yes, functions at the meta level are parametric, so they correspond exactly to single-variable binding – they cannot perform any computation such as pattern matching on their arguments and thus we do not have to worry about exotic terms. So this works fine for Church-style lambdas:

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

HAGOP. I see. And how about the typing judgment, $\Gamma \vdash e : \tau$?

ROZA. Let's add a *predicate* for that: a new constructor for the type of `prop`ositions. It will relate a `term` e to its `typ`, τ. Only, no Γ, there is an implicit context of assumptions that will serve the same purpose.

```
typeof : term → typ → prop.
```

HAGOP. I see. We now need to give the rules that make up the predicate, right? Let me see if I can get the typing rule for application. I know that in Prolog we write the conclusion of a rule first, and the premises follow the :– sign. Does something like this work?

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

ROZA. Yes! That's exactly right. Makam uses capital letters for unification variables.

HAGOP. I will need help with the lambda typing rule, though. What's the equivalent of extending the context as in $\Gamma,\ x : \tau$?

ROZA. Simple: we introduce a fresh constructor for terms and a new typing rule for it:

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

HAGOP. Hmm, so '`x:term →`' introduces the fresh constructor standing for the new variable, and '`typeof x T1 →`' introduces the new assumption? Oh, and we need to get to the body of the lambda function in order to type-check it, so that's why you do `E x`.

ROZA. Yes. Note that the introductions are locally scoped, so they are only in effect for the recursive call '`typeof (E x) T2`'.

HAGOP. Makes sense. So do we have a type checker already?

ROZA. We do! We can issue *queries* of the `typeof` predicate to type-check terms. Observe:

```
typeof (lam _ (fun x ⇒ x)) T ?
>> Yes:
>> T := arrow T1 T1.
```

HAGOP. Cool! So `fun` for metalevel functions, underscores for unification variables we don't care about, and ? for queries. But wait, last time I implemented unification in my toy STLC implementation it was easy to make it go into an infinite loop with $\lambda x.xx$. Does that work?

ROZA. Well, you were missing the occurs-check. λProlog unification includes it:

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

HAGOP. Right. So let's see, what else can we do? How about adding tuples to our language? Can we use something like a polymorphic list?

ROZA. Sure, λProlog has polymorphic types and higher-order predicates. Here's how lists are defined in the standard library:

```
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.
```

HAGOP. Nice! I guess that's why you wanted to go with λProlog for doing this instead of LF, since you cannot use polymorphism there?

ROZA. Indeed. We will see, once we figure out what our language should be, one thing we could do is monomorphize our definitions to LF, and then we could even use Beluga [Pientka and Dunfield 2010] to do all of our metatheoretic proofs. Or maybe we could use Abella [Baelde et al. 2014] directly.

HAGOP. Sounds good. So, for tuples, this should work:

```
tuple : list term → term.
product : list typ → typ.
typeof (tuple ES) (product TS) :- map typeof ES TS.
```

ROZA. Yes, and there is syntactic sugar for `cons` and `nil` too:

```
typeof (lam _ (fun x ⇒ lam _ (fun y ⇒ tuple [x, y]))) T ?
>> Yes:
>> T := arrow T1 (arrow T2 (product [T1, T2])).
```

HAGOP. So how about evaluation? Can we write the big-step semantics too?

ROZA. Why not? Let's add a predicate and do the two easy rules:

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

HAGOP. OK, let me try my hand at the beta-redex case. I'll just do call-by-value. I know that when using HOAS, function application is exactly capture-avoiding substitution, so this should be fine:

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

ROZA. Exactly! See, I told you this would be easy!

4 IN WHICH OUR HEROES ADD PARENTHESES AND DISCOVER HOW TO DO MULTIPLE BINDING

HAGOP. Still, I feel like we've been playing to the strengths of λProlog.... Yes, single-variable binding, substitutions, and so on work nicely, but how about any other form of binding? Say, binding multiple variables at the same time? We are definitely going to need that for the language

we have in mind. I was under the impression that HOAS encodings do not work for that – for example, I was reading Keuchel et al. [2016] recently, and I remember an observation to that end.

ROZA. That's not really true; having first-class support for single-variable binders should be enough. But let's try it out, maybe adding multiple-argument functions for example – I mean uncurried ones. Want to give it a try?

HAGOP. Let me see. We want the terms to look roughly like this:

```
lammany (fun x ⇒ fun y ⇒ tuple [y, x])
```

For the type of `lammany`, I want to write something like this, but I know this is wrong.

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

ROZA. Yes, that doesn't quite work. It would introduce a fresh variable for `lists`, not a number of fresh variables for `terms`. HOAS functions are parametric, too, which means we cannot even get to the potential elements of the fresh `list` inside the `term`.

HAGOP. Right. So I don't know, instead we want to use a type that stands for `term → term`, `term → term → term`, and so on. Can we write `term → ... → term`?

ROZA. Well, not quite, but we have already seen something similar, a type that roughly stands for `term * ... * term`, and we did not need anything special for that....

HAGOP. You mean the `list` type?

ROZA. Exactly. What do you think about this definition?

```
bindmanyterms : type.
bindnil : term → bindmanyterms.
bindcons : (term → bindmanyterms) → bindmanyterms.
```

HAGOP. Hmm. That looks quite similar to lists; the parentheses in `cons` are different. `nil` gets an extra `term` argument, too....

ROZA. Yes... So what is happening here is that `bindcons` takes a single argument, introducing a binder; and `bindnil` is when we get to the body and don't need any more binders. Maybe we should name them accordingly.

HAGOP. Right, and could we generalize their types? Maybe that will help me get a better grasp of it. How is this?

```
bindmany : type → type → type.
body : Body → bindmany Variable Body.
bind : (Variable → bindmany Variable Body) → bindmany Variable Body.
```

ROZA. This looks great! That is exactly what's in the Makam standard library, actually. And we can now define `lammany` using it:

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

HAGOP. I see. So our example term from before would be:

```
lammany (bind (fun x ⇒ bind (fun y ⇒ body (tuple [y,x]))))
```

This `bindmany` datatype is quite interesting. Is there some reference about it?

ROZA. Not that I know of, at least where it is called out as a reusable datatype – though the construction is definitely part of PL folklore. After I started using this in Makam, I noticed

similar constructions in the wild, for example in MTac [Ziliani et al. 2013], for parametric-HOAS implementation of telescopes in Coq.

HAGOP. Interesting. So how do we work with `bindmany`? What's the typing rule like?

ROZA. The rule is written like this, and I'll explain what goes into it:

```
arrowmany : list typ → typ → typ.
typeof (lammany XS_E) (arrowmany TS T) :-
  openmany XS_E (fun xs e ⇒ assumemany typeof xs TS (typeof e T)).
```

HAGOP. Let me see if I can read this… `openmany` somehow gives you fresh variables `xs` for the binders, plus the body `e` of the `lammany`; and then the `assumemany typeof` part is what corresponds to extending the Γ context with multiple typing assumptions?

ROZA. Yes, and then we typecheck the body in that local context that includes the fresh variables and the typing assumptions. But let's do one step at a time. `openmany` is simple; we iterate through the nested binders, introducing one fresh variable at a time. We also substitute each bound variable for the current fresh variable, so that when we get to the body, it only uses the fresh variables we introduced.

```
openmany : bindmany A B → (list A → B → prop) → prop.
openmany (body Body) Q :- Q [] Body.
openmany (bind F) Q :- (x:A → openmany (F x) (fun xs ⇒ Q (x :: xs))).
```

HAGOP. I see. I guess `assumemany` is similar, introducing one assumption at a time?

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

ROZA. Yes, exactly! Just a note, though – λProlog typically does not allow the definition of `assumemany`, where a non-concrete predicate like `P X Y` is used as an assumption, because of logical reasons. Makam allows this form statically, and so does ELPI [Dunchev et al. 2015], another λProlog implementation, though there are instantiations of `P` that will fail at run-time[1].

HAGOP. I see. But in that case we could just manually inline `assumemany typeof` instead, so that's not a big problem, just more verbose. But can I try our typing rule out?

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

Great, I think I got the hang of this. Let me try to see if I can add a multiple-argument application construct `appmany` and its evaluation rules.

```
appmany : term → list term → term.
typeof (appmany E ES) T :-
  typeof E (arrowmany TS T), map typeof ES TS.
eval (appmany E ES) V :-
  eval E (lammany XS_E'), map eval ES VS, (...).
```

How can I do simultaneous substitution of all of the `XS` for `VS`?

[1]The logical reason why this is not allowed in λProlog is that it violates the property of existence of uniform proofs; see Nadathur and Qi [2005] for more information. An example of a goal that will fail at runtime is one that starts with a logical-or (denoted as "`;`") as an assumption, like "`(typeof X T1; typeof X T2) → ...`".

ROZA. You'll need another standard-library predicate for `bindmany`, which iteratively uses HOAS function application to perform a number of substitutions:

```
applymany : bindmany A B → list A → B → prop.
applymany (body B) [] B.
applymany (bind F) (X :: XS) B :- applymany (F X) XS B.
```

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

HAGOP. I see, that makes sense. Can I ask you something that worries me, though – all these fancy higher-order abstract binders, how do we… make them concrete? Say, how do we print them?

ROZA. That's actually quite easy. We just add a concrete name to them, a plain old `string`. Our typing rules etc. do not care about it, but we could use it for parsing concrete syntax into our abstract binding syntax, or for pretty-printing.… All those are stories for another time, though; let's just say that we could have defined `bind` with an extra `string` argument, representing the concrete name; and then `openmany` would just ignore it.

```
bind : string → (Var → bindmany Var Body) → bindmany Var Body.
```

HAGOP. Interesting. I would like to see more about this, but maybe some other time. I thought of another thing that could be challenging: mutually recursive `let rec`s?

ROZA. Sure. Let's take this term for example:

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

If we write this in a way where the binding structure is explicit, we would bind `f` and `g` simultaneously and then write the definitions and the body in that scope:

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

HAGOP. I think I know how to do this then! How does this look?

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

ROZA. Exactly! Want to try writing the typing rules?

HAGOP. Maybe something like this?

```
typeof (letrec XS_DefsBody) T' :-
  openmany XS_DefsBody (fun xs (defs, body) ⇒
    assumemany typeof xs TS (map typeof defs TS),
    assumemany typeof xs TS (typeof body T')).
```

ROZA. Almost! You have used the syntax we use for writing rule premises in the `fun` argument of `openmany`; the Makam grammar only allows that with the syntactic form `pfun` instead, which is used to write anonymous predicates. Since this `pfun` argument will be a predicate and can thus perform computation, you are also able to destructure parameters like you did here on `(defs, body)` – that doesn't work for normal functions in the general case, since they need to treat arguments parametrically. This works by performing unification of the parameter with the given term – so `defs` and `body` need to be capitalized so that they are understood to be unification variables.

```
typeof (letrec XS_DefsBody) T' :-
  openmany XS_DefsBody (pfun XS (Defs, Body) ⇒
    assumemany typeof XS TS (map typeof Defs TS),
    assumemany typeof XS TS (typeof Body T')).
```

HAGOP. Ah, I see[2]. Let me ask you something, though: one thing I noticed with our representation of `letrec` is that we have to be careful so that the number of binders matches the number of definitions we give. Our typing rules disallow that, but I wonder if there's a way to have a more accurate representation for `letrec` which includes that requirement?

ROZA. Funny you should ask that… Let me tell you a story.

5 IN WHICH THE LEGEND OF THE GADTS AND THE AD-HOC POLYMORPHISM IS RECOUNTED

Once upon a time, our republic lacked one of the natural wonders that it is now well-known for, and which is now regularly enjoyed by tourists and inhabitants alike. I am talking of course about the Great Arboretum of Dangling Trees, known as GADTs for short. Then settlers from the far-away land of the Dependency started coming to the republic and started speaking of Lists that Knew Their Lengths, of Terms that Knew Their Types, of Collections of Elements that were Heterogeneous, and about the other magical beings of their home. And they set out to build a natural environment for these beings on the republic, namely the GADTs that we know and love, to remind them of home a little. And their work was good and was admired by many.

A long time passed, and dispatches from another far-away land came to the republic, written by authors whose names are now lost in the sea of anonymity, and I fear might forever remain so. And the dispatches went something like this.

ANONYMOUS AUTHOR. … In my land of λProlog that I speak of, the type system is a subset of System F_ω that should be familiar to you – the simply typed lambda calculus, plus prenex polymorphism, plus simple type constructors of the form `type * ... * type → type`. There is also a limited form of rank-2 polymorphism, allowing types of the form `forall A T`, which are inhabited by unapplied polymorphic constants through the notation `@foo`. There is a `prop` sort for propositions, which is a normal type, but also a bit special: its terms are not just values but are also computations, activated when queried upon.

However, the language of this land has a distinguishing feature, called Ad-Hoc Polymorphism. You see, the different rules that define a predicate in our language can *specialize* their type arguments. This can be used to define polymorphic predicates that behave differently for different types, like this, where we are essentially doing a `typecase` and we choose a rule depending on the *type* of the argument (as opposed to its value):

```
print : [A] A → prop.
print (I: int) :- (... code for printing integers ...)
print (S: string) :- (... code for printing strings ...)
```

[2]There is a subtlety here, having to do with the free variables that a unification variable can capture. In λProlog, a unification variable is allowed to capture all the free variables in scope at the point where it is introduced, as well as any variables it is explicitly applied to. `Defs` and `Body` are introduced as unification variables when we get to execute the `pfun`; otherwise, all unification variables used in a rule get introduced when we check whether the rule fires. As a result, `Defs` and `Body` can capture the `xs` variables that `openmany` introduces, whereas `T'` cannot. In λProlog terms, the `pfun` notation of Makam desugars to existential quantification of any (capitalized) unification variables that are mentioned while destructuring an argument, like the variables `Defs` and `Body`.

The local dialects Teyjus [Gacek et al. 2015; Nadathur and Mitchell 1999] and Makam include this feature, while it is not encountered in other dialects like ELPI [Dunchev et al. 2015]. In the Makam dialect in particular, type variables are understood to be parametric by default. In order to make them ad-hoc and allow specializing them in rules, we need to denote them using the `[A]` notation.

Of course, this feature has both to do with the statics as well as the dynamics of our language: and while dynamically it means something akin to a `typecase`, statically, it means that rules might specialize their type variables, and this remains so for type-checking the whole rule.

But… aha! Is it not type specialization during pattern matching that is an essential feature of the GADTs of your land? Maybe that means that we can use Ad-Hoc Polymorphism not just to do `typecase` but also to work with GADTs in our land? Consider this! The venerable List that Knows Its Length:

```
zero : type. succ : type → type.
vector : type → type → type.
vnil : vector A zero.
vcons : A → vector A N → vector A (succ N).
```

And now for the essential `vmap`:

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

In each rule, the first argument already specializes the `N` type – in the first rule to `zero`, in the second, to `succ N`. And so erroneous rules that do not respect this specialization would not be accepted as well-typed sayings in our language:

```
vmap P vnil (vcons X XS) :- ...
```

And we should note that in this usage of Ad-Hoc Polymorphism for GADTs, it is only the increased precision of the statics that we care about. Dynamically, the rules for `vmap` can perform normal term-level unification and only look at the constructors `vnil` and `vcons` to see whether each rule applies, rather than relying on the `typecase` aspects we spoke of before.

Coupling this with the binding constructs that I talked to you earlier about, we can build new magical beings, like the *Bind that Knows Its Length*:

```
vbindmany : (Var: type) (N: type) (Body: type) → type.
vbody : Body → vbindmany Var zero Body.
vbind : (Var → vbindmany Var N Body) → vbindmany Var (succ N) Body.
```

(Whereby I am using notation of the Makam dialect in my definition of `vbindmany` that allows me to name parameters, purely for the purposes of increased clarity.)

In the `openmany` version for `vbindmany`, the rules are exactly as before, though the static type is more precise:

```
vopenmany : [N] vbindmany Var N Body → (vector Var N → Body → prop) → prop.
vopenmany (vbody Body) Q :- Q vnil Body.
vopenmany (vbind F) Q :-
  (x:A → vopenmany (F x) (fun xs ⇒ Q (vcons x xs))).
```

We can also showcase the *Accurate Encoding of the Letrec*:

```
vletrec : vbindmany term N (vector term N * term) → term.
```

And that is the way that it turns out that our land of λProlog supports GADTs, without needing the addition of any feature. It is all thanks to the existing support for Ad-Hoc Polymorphism, which has been a staple since the days of yore [Nadathur and Miller 1988]. Who knew!

6 IN WHICH OUR HERO HAGOP ADDS PATTERN MATCHING ON HIS OWN

(Roza had a meeting with another student, so Hagop took a small break and is now back at his office. He is trying to work out on his own how to encode patterns. He is fairly confident at this point that having explicit support for single-variable binding is enough to model most complicated forms of binding, especially when making use of polymorphism and GADTs.)

HAGOP. OK, so let's implement simple patterns and pattern-matching like in ML… First let's determine the right binding structure. For a branch like:

```
| cons(hd, tl) → ... hd .. tl ...
```

the pattern introduces 2 variables, `hd` and `tl`, which the body of the branch can refer to. But we can't really refer to those variables in the pattern itself, at least for simple patterns[3]…. So there's no binding going on really within the pattern; instead, once we figure out how many variables a pattern introduces, we can do the actual binding all at once, when we get to the body of the branch:

```
branch(pattern,
       bind [# of variables in pattern].( .. body of the branch .. ))
```

So we could write the above branch in Makam like this:

```
branch (patt_cons patt_var patt_var)
       (bind (fun hd ⇒ bind (fun tl ⇒ body (.. hd .. tl ..))))
```

We do have to keep the order of variables consistent somehow, so `hd` here should refer to the first occurrence of `patt_var`, and `tl` to the second. Based on these, I am thinking that the type of `branch` should be something like:

```
branch : (Pattern: patt N) (Vars_Body: vbindmany term N term) → ...
```

Wait, before I get into the weeds let me just set up some things. First, let's add a simple base type, say `nats`, to have something to work with as an example. I'll prefix their names with `o` for "object language," so as to avoid ambiguity. And I will also add `case_or_else`, standing for a single-branch pattern-match construct. It should be easy to extend to a multiple-branch construct, but I want to keep things as simple as possible. I'll inline what I had written for `branch` above into the definition of `case_or_else`.

```
onat : typ. ozero : term. osucc : term → term.
typeof ozero onat. typeof (osucc N) onat :- typeof N onat.
eval ozero ozero. eval (osucc E) (osucc V) :- eval E V.

case_or_else : (Scrutinee: term)
  (Patt: patt N) (Vars_Body: vbindmany term N term)
  (Else: term) → term.
```

Now for the typing rule – it will be something like this:

[3]There are counterexamples, like for or-patterns in some ML dialects, or for dependent pattern matching, where consequent uses of the same variable perform exact matches rather than unification. We choose to omit the handling of cases like those in the present work for presentation purposes.

```
typeof (case_or_else Scrutinee Pattern Vars_Body Else) BodyT :-
  typeof Scrutinee T, typeof_patt Pattern T VarTypes,
  vopenmany Vars_Body (pfun Vars Body ⇒
    vassumemany typeof Vars VarTypes (typeof Body BodyT)),
  typeof Else BodyT.
```

Right, so when checking a pattern, we'll have to determine both what type of scrutinee it matches, as well as the types of the variables that it contains. We will also need `vassumemany` that is just like `assumemany` from before but which takes `vector` arguments instead of `list`.

```
typeof_patt : [N] patt N → typ → vector typ N → prop.
vassumemany : [N] (A → B → prop) → vector A N → vector B N → prop → prop.
(...)
```

Now, I can just go ahead and define the patterns, together with their typing relation, `typeof_patt`. Let me just work one by one for each pattern.

```
patt_var : patt (succ zero).
typeof_patt patt_var T (vcons T vnil).
```

OK, that's how we'll write pattern variables, introducing a single variable of a specific `typ` into the body of the branch. And the following should be good for the `onat`s I defined earlier.

```
patt_ozero : patt zero.
typeof_patt patt_ozero onat vnil.

patt_osucc : patt N → patt N.
typeof_patt (patt_osucc P) onat VarTypes :- typeof_patt P onat VarTypes.
```

A wildcard pattern will match any value and should not introduce a variable into the body of the branch.

```
patt_wild : patt zero.
typeof_patt patt_wild T vnil.
```

OK, and let's do patterns for our n-tuples.... I guess I'll need a type for lists of patterns too.

```
patt_tuple : pattlist N → patt N.
typeof_patt (patt_tuple PS) (product TS) VarTypes :-
  typeof_pattlist PS TS VarTypes.
pattlist : (N: type) → type.
pnil : patt zero.
pcons : patt N → pattlist N' → pattlist (N + N').
```

Uh-oh... don't think I can do that `N + N'`, really. In this `pcons` case, my pattern basically looks like `(P, ...PS)`; and I want the overall pattern to have as many variables as `P` and `PS` combined. But the GADTs support in λProlog seems to be quite basic. I do not think there's any notion of type-level functions like plus[4]....

However... maybe I can work around that, if I change `patt` to include an "accumulator" argument, say `NBefore`. Each constructor for patterns will now define how many pattern variables it adds to

[4]Since GADTs in λProlog have not been considered in the past, we only present what is already supported by the existing language design and by many λProlog implementations in the present work. We are exploring extensions to λProlog to support type-level computation as part of future work.

that accumulator, yielding `NAfter`, rather than defining how many pattern variables it includes… like this:

```
patt, pattlist : (NBefore: type) (NAfter: type) → type.
patt_var : patt N (succ N).
patt_ozero : patt N N.
patt_osucc : patt N N' → patt N N'.
patt_wild : patt N N.
patt_tuple : pattlist N N' → patt N N'.
pnil : pattlist N N.
pcons : patt N N' → pattlist N' N'' → pattlist N N''.
```

Yes, I think that should work. I have a little editing to do in my existing predicates to use this representation instead. For top-level patterns, we should always start with the accumulator being `zero`…

```
case_or_else : (Scrutinee: term)
  (Patt: patt zero N) (Vars_Body: vbindmany term N term)
  (Else: term) → term.
```

I also have to change `typeof_patt`, so that it includes an accumulator argument of its own:

```
typeof_patt : [NBefore NAfter] patt NBefore NAfter → typ →
  vector typ NBefore → vector typ NAfter → prop.
```

```
typeof (case_or_else Scrutinee Pattern Vars_Body Else) BodyT :-
  typeof Scrutinee T, typeof_patt Pattern T vnil VarTypes,
  vopenmany Vars_Body (pfun Vars Body ⇒
    vassumemany typeof Vars VarTypes (typeof Body BodyT)),
  typeof Else BodyT.
```

All right, let's proceed to the typing rules for patterns themselves:

```
typeof_patt patt_var T VarTypes VarTypes' :-
  vsnoc VarTypes T VarTypes'.
```

OK, here I need `vsnoc` to add an element to the end of a vector. That should yield the correct order for the types of pattern variables; I am visiting the pattern left-to-right after all.

```
vsnoc : [N] vector A N → A → vector A (succ N) → prop.
vsnoc vnil Y (vcons Y vnil).
vsnoc (vcons X XS) Y (vcons X XS_Y) :- vsnoc XS Y XS_Y.
```

The rest is easy to adapt….

(Our hero finishes adapting the rest of the rules for `typeof_patt`*, which are available in the unabridged version of this story.)*

Let me see if this works! I'll try out the predecessor function:

```
typeof (lam _ (fun n ⇒ case_or_else n
  (patt_osucc patt_var) (vbind (fun pred ⇒ vbody pred))
  ozero)) T ?
>> Yes:
>> T := arrow onat onat.
```

Great! Time to show this to Roza.

7 IN WHICH OUR HEROES REFLECT ON STRUCTURAL RECURSION

ROZA. Your pattern-matching encoding looks good! You seem to be getting the hang of this. How about we do something challenging then? Say, type synonyms?

HAGOP. Type synonyms? You mean, introducing type definitions like `type natpair = nat * nat`? That does not seem particularly tricky.

ROZA. I think we will face a couple of interesting issues with it, the main one being how to do *structural recursion* in a nice way. But first, let me write out the necessary pen-and-paper rules, so that we are on the same page. We'll do top-level type definitions, so let's add a top-level notion of programs c and a well-formedness judgment '$\vdash c$ wf' for them. (We could do modules instead of just programs, but I feel that would derail us a little.) We also need an additional environment Σ to store type definitions:

$$\textsc{WfProgram-Main}\ \frac{\emptyset;\ \Sigma \vdash e : \tau}{\Sigma \vdash (\text{main } e)\ \text{wf}} \qquad \textsc{WfProgram-Type}\ \frac{\Sigma,\ \alpha = \tau \vdash c\ \text{wf}}{\Sigma \vdash (\text{type } \alpha = \tau\ ;\ c)\ \text{wf}}$$

$$\textsc{Typeof-Conv}\ \frac{\Gamma; \Sigma \vdash e : \tau \qquad \Sigma \vdash \tau =_\delta \tau'}{\Gamma; \Sigma \vdash e : \tau'} \qquad \textsc{TypEq-Def}\ \frac{\alpha = \tau \in \Sigma}{\Sigma \vdash \alpha =_\delta \tau} \ \ldots$$

HAGOP. Right, we will need the conversion rule, so that we identify types up to expanding their definitions; that's δ-equality… And I see you haven't listed out all the rules for that, but those are mostly standard.

ROZA. Still, there are quite a few of those rules. Want to give transcribing this to Makam a try?

HAGOP. Yes, I got this. I'll add a new `typedef` predicate; I will only use it for local assumptions, to correspond to the Σ context of type definitions. I will also do the well-formed program rules:

```
typedef : (NewType: typ) (Definition: typ) → prop.
program : type.
main : term → program.
lettype : (Definition: typ) (A_Program: typ → program) → program.

wfprogram : program → prop.
wfprogram (main E) :- typeof E T.
wfprogram (lettype T A_Program) :-
  (a:typ → typedef a T → wfprogram (A_Program a)).
```

Well, I can do the conversion rule and the type-equality judgment too…. I will name that `typeq`. I'll just write the one rule for now, which should be sufficient for a small example:

```
typeq : (T1: typ) (T2: typ) → prop.
typeof E T :- typeof E T', typeq T T'.
typeq A T :- typedef A T.

wfprogram (lettype (arrow onat onat) (fun a ⇒
          (main (lam a (fun f ⇒ (app f ozero)))))) ?
>> (Complete silence)
```

ROZA. Time to `Ctrl-C` out of the infinite loop?

HAGOP. Oh. Oh, right. I guess we hit a case where the proof-search strategy of Makam fails to make progress?

ROZA. Correct. The loop happens when the new `typeof` rule gets triggered: it has `typeof E T'` as a premise, but the same rule still applies to solve that goal, so the rule will fire again, and so on. Makam just does depth-first search right now; until my friend implements a more sophisticated search strategy, we need to find another way to do this.

HAGOP. I see. I guess we should switch to an algorithmic type system then.

ROZA. Yes. Fortunately we can do that with relatively painless edits and additions. Consider this: we only need to use the conversion rule in cases where we already know something about the type `T` of an expression `E`, but the typing rules require that `E` has a type `T'` of some other form. That was the case above – for `E = f`, we knew that `T = a`, but the typing rule for `app` required that `T' = arrow T1 T2` for some `T1`, `T2`.

HAGOP. Oh. In that case we could try running the typing rules *without* the existing typing information that we have, like `T = a`? We would get a new type `T'` that way and we can then check whether it matches the original `T` type that we expect, up to δ-equality.

ROZA. Exactly. So we need to change the rule you wrote to apply only in the case where `T` starts with a concrete constructor (so we already know something about it), rather than when it is an uninstantiated unification variable. We will then check whether the resulting type `T'` is equal to `T`, using our `typeq` predicate.

HAGOP. Is that even possible? Is there a way in λProlog to tell whether something is a unification variable?

ROZA. There is! Most Prolog dialects have a predicate that does that – it's usually called `var`. In Makam it is called `refl.isunif`, the `refl` namespace prefix standing for *reflective* predicates. So, here's how we can write it instead, where I'll also use logical negation[5]:

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

HAGOP. Interesting. If we ever made a paper submission out of this, some reviewers would not be happy about this `typeof` rule. But sure. Oh, and we should add the conversion rule for `typeof_patt`, but that's almost the same as for terms. (…) I'll do `typeq` next.

```
typeq A T' :- typedef A T, typeq T T'.
typeq T' A :- typedef A T, typeq T T'.
```

ROZA. I like how you added the symmetric rule, but… this is subtle, but if `A` is a unification variable, we don't want to unify it with an arbitrary synonym. So we need to check that `A` is concrete somehow[6]:

```
typeq A T' :- not(refl.isunif A), typedef A T, typeq T T'.
typeq T' A :- not(refl.isunif A), typedef A T, typeq T T'.
```

HAGOP. I see what you mean. OK, I'll continue on to the rest of the cases….

```
typeq (arrow T1 T2) (arrow T1' T2') :-
  typeq T1 T1', typeq T2 T2'.
typeq (arrowmany TS T) (arrowmany TS' T') :-
  map typeq TS TS', typeq T T'.
```

ROZA. Writing boilerplate is not fun, is it?

[5] Makam follows Kiselyov et al. [2005] closely in terms of the semantics for logical if-then-else and logical negation.
[6] Though not supported in Makam, an alternative to this in other languages based on higher-order logic programming would be to add a `mode (i o)` declaration for `typedef`, so that `typedef A T` would fail if `A` is not concrete.

HAGOP. It is not. I wish we could just write the first two rules that you wrote; they're the important ones, after all. All the others just propagate the structural recursion through. Also, whenever we add a new constructor for types, we'll have to remember to add a `typeq` rule for it.…

ROZA. Right. Let's just use some magic instead.

(Roza changes the type definition of `typeq` *to* `typeq : [Any] (T1: Any) (T2: Any) → prop`*, and adds a few lines:)*

```
typeq A T' :- not(refl.isunif A), typedef A T, typeq T T'.
typeq T' A :- not(refl.isunif A), typedef A T, typeq T T'.
typeq T T' :- structural_recursion @typeq T T'.
```

HAGOP. … What just happened. Is `structural_recursion` some special Makam trick I don't know about yet?

ROZA. Indeed. There is a little bit of trickery involved here, but you will see that there is much less of it than you would expect, upon close reflection. `structural_recursion` is just a normal standard-library predicate like any other; it essentially applies a polymorphic predicate "structurally" to a term. Its implementation will be a little special, of course. But let's just think about how you would write the rest of the rules of `typeq` generically, to perform structural recursion.

HAGOP. OK. Well, when looking at two `typ`s together, we have to make sure that their constructors are the same and also that any `typ`s they contain as arguments are recursively `typequal`. So something like this:

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

ROZA. Right. Note, though, that the types of arguments might be different than `typ`. So even if we start comparing two types at the top level, we might end up having to compare, say, two lists of types that they contain – as will be the case for `arrowmany` for example.

HAGOP. I see! That's why you edited `typeq` to be polymorphic above; you have extended it to work on *any type* (of the metalanguage) that might contain a `typ`.

ROZA. Exactly. Now, the list of `Arguments` – can you come up with a type for them?

HAGOP. We can use the GADT of heterogeneous lists for them; not all the arguments of each constructor need to be of the same type!

```
typenil : type. typecons : (T: type) (TS: type) → type.
hlist : (TypeList: type) → type.
hnil : hlist typenil. hcons : T → hlist TS → hlist (typecons T TS).
```

ROZA. Great! We will need a heterogeneous `map` for these lists, too. We'll need a polymorphic predicate as an argument, since we'll have to use it for `Arguments` of different types:

```
hmap : [TS] (P: forall A (A → A → prop)) (XS: hlist TS) (YS: hlist TS) → prop.
hmap P hnil hnil.
hmap P (hcons X XS) (hcons Y YS) :- forall.apply P X Y, hmap P XS YS.
```

As I mentioned before, the rank-2 polymorphism support in Makam is limited, so you have to use `forall.apply` explicitly to instantiate the polymorphic `P` predicate accordingly and call it.

HAGOP. Let me try out an example of that:

```
change : [A]A → A → prop. change 1 2. change "foo" "bar".
hmap @change (hcons 1 (hcons "foo" hnil)) YS ?
```

```
>> Yes:
>> YS := hcons 2 (hcons "bar" hnil).
```

Looks good enough. So, going back to our generic rule – is there a way to actually write it? Maybe there's a reflective predicate we can use, similar to how we used `refl.isunif` before to tell if a term is an uninstantiated unification variable?

ROZA. Exactly – there is `refl.headargs`. It relates a concrete term to its decomposition into a constructor and a list of arguments[7]. This does not need an extra-logical feature save for `refl.isunif`, though: we could define `refl.headargs` without any special support, if we maintained a discipline whenever we add a new constructor, roughly like this:

```
refl.headargs : (Term: TermT) (Constr: ConstrT) (Args: hlist ArgsTS) → prop.

arrowmany : (TS: list typ) (T: typ) → typ.
refl.headargs Term Constructor Args :-
  not(refl.isunif Term), eq Term (arrowmany TS T),
  eq Constructor arrowmany, eq Args (hcons TS (hcons T hnil)).
```

By the way, `eq` is a standard-library predicate that simply attempts unification of its two arguments:

```
eq : A → A → prop. eq X X.
```

HAGOP. I see. I think I can write the generic rule for `typeq` now, then!

```
typeq T T' :-
  refl.headargs T Constr Args,
  refl.headargs T' Constr Args',
  hmap @typeq Args Args'.
```

ROZA. That looks great! Simple, isn't it? You'll see that there are a few more generic cases that are needed, though. Should we do that? We can roll our own reusable `structural_recursion` implementation – that way we will dispel all magic from its use that I showed you earlier! I'll give you the type; you fill in the first case:

```
structural_recursion : [Any]
  (RecursivePred: forall A (A → A → prop))
  (X: Any) (Y: Any) → prop.
```

HAGOP. Let me see. Oh, so, the first argument is a predicate – are we doing this in open-recursion style? I see. Well, I can adapt the case I just wrote above.

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

ROZA. Nice. Now, here you assume that `X` and `Y` are both concrete terms. What happens when `X` is concrete and `Y` isn't, or the other way around? Hint: you can use this `happly` predicate, to apply a list of arguments to a constructor and thus reconstruct a term:

```
happly : [Constr Args Term] Constr → hlist Args → Term → prop.
happly Constr hnil Constr.
```

[7] Other versions of Prolog have predicates toward the same effect; for example, SWI-Prolog [Wielemaker et al. 2012] provides '`compound_name_arguments`', which is quite similar.

```
happly Constr (hcons A AS) Term :- happly (Constr A) AS Term.
```

HAGOP. How about this? This way, we will decompose the concrete `X`, perform the transformation on the `Arguments`, and then reapply the `Constructor` to get the result for `Y`.

```
structural_recursion Rec X Y :-
  refl.headargs X Constructor Arguments,
  hmap Rec Arguments Arguments',
  happly Constructor Arguments' Y.
```

ROZA. That is exactly right. You need the symmetric case, too, but that's entirely similar. Also, there is another type of concrete terms in Makam: meta-level functions! It does not make sense to destructure functions using `refl.headargs`, so it fails in that case, and we have to treat them specially:

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

HAGOP. Ah, I see! Here you *are* actually relying on the `typecase` aspect of ad-hoc polymorphism, right? To check if `X` and `Y` are of the meta-level function type.

ROZA. Exactly. And you know what, that's all there is to it! So, we've minimized the boilerplate, and we won't need any adaptation when we add a new constructor – even if we make use of all sorts of new and complicated types.

HAGOP. That's right: we do not need to do anything special for the binding forms we defined, like `bindmany`.... quite a payoff for a small amount of code! But, wait, isn't `structural_recursion` missing a case: what happens if both `X` and `Y` are uninstantiated unification variables?

ROZA. You are correct; it would fail in that case. But in my experience, it's better to define how to handle unification variables as needed, in each new structurally recursive predicate. In this case, we should never get into that situation based on how we have defined `typeq`.

(Roza and Hagop try out a few examples and convince themselves that this works OK and no endless loops happen when things don't typecheck correctly.)

8 IN WHICH OUR HEROES BREAK INTO SONG AND ADD MORE ML FEATURES

(Our heroes need a small break, so they work on a couple of features while improvising on a makam[8]. Roza is singing lyrics from the folk songs of her land, and Hagop is playing the oud. Their friend Lambros from the next office over joins them on the kemençe.)

"You can skim this chapter / or skip it all the same.
It's mostly for completeness / since ML as a name
requires some poly-lambdas / as well as ADTs
so here we are dotting our i's / and crossing all our t's.

System F is easy / but later we might do
some type generalizing / like Hindley-Milner too
but if you're feeling tired / I told you just before
you can take a mini-break / like Lambros from next door."

[8]Makam is the system of melodic modes of traditional Arabic and Turkish music that is also used in the Greek rembetiko. It is comprised of a set of scales, patterns of melodic development, and rules for improvisation.

```
tforall : (typ → typ) → typ.
polylam : (typ → term) → term.
polyapp : term → typ → term.
```

```
typeof (polylam E) (tforall T) :- (a:typ → typeof (E a) (T a)).
typeof (polyapp E T) T' :- typeof E (tforall TF), eq T' (TF T).
```

> "The algebraic datatypes / caused all sorts of trouble
> in the previous version / and since it was a double-
> blind submission process / reviewers quite diverse
> wonder who's the lunatic / who writes papers in verse."

```
datadef : type. datatype_bind : (Into: type) → type.
datatype : (Def: datadef) (Rest: datatype_bind program) → program.
```

> "The types seem fairly easy / the constructors might be hard.
> So let's go step-by-step for now / or we'll be here next March.
> We won't support the poly-types / to keep the system simple,
> and arguments to constructors? / They'll all take just a single."

```
   data nattree = Leaf of nat | Node of (nattree * nattree) ; rest
⇝  data nattree = [ ("Leaf", nat), ("Node", nattree * nattree) ] ; rest
⇝  data nattree = [ nat, nattree * nattree ] ; λLeaf. λNode. rest
⇝  data (λnattree. [nat, nattree * nattree]) ; λnattree. λLeaf. λNode. rest
⇝  data (λnattree. [nat, nattree * nattree]) ;
     λnattree. bind (λLeaf. bind (λNode. body rest))
⇝  datatype (mkdatadef (fun nattree ⇒ [nat, nattree * nattree]))
     (bind_datatype (fun nattree ⇒
       bind (fun leaf ⇒ bind (fun node ⇒ body rest))))
```

> "Sometimes it just is better / to avoid all those words.
> Just squint your eyes a little bit; / Hagop will strum some chords."

```
constructor : type.
mkdatadef : (typ → list typ) → datadef.
bind_datatype : (typ → bindmany constructor A) → datatype_bind A.
```

> "We are avoiding GADTs / they're good but up the ante.
> And we have MetaML to do / (in prose 'cause I'm no Dante.)
> We're almost there, we need to add / the `wfprogram` clause.
> But first we'll need an env. with types that / `constructors` expose."

```
constructor_typ : (DataType: typ) (C: constructor) (ArgType: typ) → prop.
```

> "We go through the constructors / populating our new `prop`.
> `DT` stands for datatype / – the page is just too cropped."

```
wfprogram (datatype (mkdatadef DT_ConstrArgTypes)
                    (bind_datatype DT_Constrs_Rest)) :-
  (dt:typ → openmany (DT_Constrs_Rest dt) (pfun Constrs Rest ⇒
    assumemany (constructor_typ dt) Constrs (DT_ConstrArgTypes dt)
    (wfprogram Rest))).
```

> "That's it, it's almost over / there's our wf-programs.
> We can't use the constructors, though / remember `terms` and `patts` ?"

```
constr : (C: constructor) (Arg: term) → term.
patt_constr : (C: constructor) (Arg: patt N N') → patt N N'.
typeof (constr C Arg) Datatype :-
  constructor_typ Datatype C ArgType, typeof Arg ArgType.
typeof_patt (patt_constr C Arg) Datatype S S' :-
  constructor_typ Datatype C ArgType, typeof_patt Arg ArgType S S'.
```

> "That's all, there's no example / please, download Makam.
> Trust me: you can run this code / and check that all tests pass."

9 IN WHICH OUR HEROES TACKLE A NEW LEVEL OF META, CONTEXTS AND SUBSTITUTIONS

HAGOP. I'm fairly confident by now that Makam should be able to handle the research idea we want to try out. Shall we get to it?

ROZA. Yes, it is time. So, what we are aiming to do is add a facility for type-safe, heterogeneous metaprogramming to our object language, similar to MetaHaskell [Mainland 2012]. This way we can manipulate the terms of a *separate* object language in a type-safe manner.

HAGOP. Exactly. For the research language we have in mind, we aim for our object language to be a formal logic, so our language will be similar to Beluga [Pientka and Dunfield 2010] or VeriML [Stampoulis 2013]. We will also need dependent functions and pattern-matching over the object language… But we don't need to do all of that; let's just do a basic version for now, and I can do the rest on my own.

ROZA. Sounds good. First, let's agree on some terminology, because a lot of words are getting overloaded a lot. Let us call *objects* o any sorts of terms of the object language that we will be manipulating. And, for lack of a better word, let us call *classes* c the "types" that characterize those objects through a typing relation of the form $\Psi \vdash_o o : c$. It is unfortunate that these names suggest object-orientation, but this is not the intent.

HAGOP. I see what you are saying. Let's keep the objects simple – to start, let's just do the terms of the simply typed lambda calculus (STLC). In that case our classes will just be the types of STLC. The objects are run-time entities: essentially, our programs will be able to "compute" objects. So we need a way to return (or "lift") an object o as a meta-level value $\langle o \rangle$.

ROZA. We are getting into many levels of meta – there's the metalanguage we're using, Makam; there's the object language we are encoding, which is now becoming a metalanguage in itself, let's call that Heterogeneous Meta ML Light (HMML?); and there's the "object-object" language that HMML is manipulating. One option would be to have the object-object language be the full HMML metalanguage itself, which would lead us to a homogeneous, multi-stage language like MetaML [Taha and Sheard 1997]. But, I agree, we should keep the object-object language simple: the STLC will suffice.

HAGOP. Great. How about we try to do the standard example of a staged `power` function? Here's a rough sketch, where I'm using `~I` for antiquotation:

```
let rec power (n: onat): < stlc.arrow stlc.onat stlc.onat > =
  match n with
    0 ⇒ < stlc.lam (fun x ⇒ 1) >
  | S n' ⇒ letobj I = power n' in
      < stlc.lam (fun x ⇒ stlc.mult (stlc.app ~I x) x) >
```

ROZA. It's a plan. So, let's get to it. Should we write some of the system down on paper first?

HAGOP. Yes, that would be very useful. For this example, we will need the lifting construct $\langle \cdot \rangle$ and the `letobj` form. Here are their typing rules, which depend on an appropriately defined typing judgment $\Psi \vdash_o o : c$ for objects. In our case, this will initially match the $\Delta \vdash \hat{e} : \hat{t}$ typing judgment for STLC (I'll use hats for terms of STLC, to disambiguate them from terms of HMML). We use i for variables standing for objects, which we will call *indices*. And we will need a way to antiquote indices inside STLC terms, which means that we will have to *extend* the STLC terms as well as their typing judgment accordingly. We store indices in the Ψ context, so the STLC typing judgment will end up having the form $\Psi; \Delta \vdash \hat{e} : \hat{t}$. Last, I'll also write down the evaluation rules of the new constructs, as they are quite simple.

OB-OB-SYNTAX

$\hat{e} ::= \lambda x : \hat{t}.\hat{e} \mid \hat{e}_1\ \hat{e}_2 \mid x \mid n \mid \hat{e}_1 * \hat{e}_2 \mid \mathbf{aq}(i)$

$\hat{t} ::= \hat{t}_1 \to \hat{t}_2 \mid \hat{\text{nat}}$

$o ::= \hat{e} \qquad c ::= \hat{t}$

HMML-SYNTAX

$e ::= \ldots \mid \langle o \rangle \mid \texttt{letobj}\ i = e\ \texttt{in}\ e'$

$\tau ::= \ldots \mid \langle c \rangle$

TYPEOF-LIFTOBJ

$$\frac{\Psi \vdash_o o : c}{\Gamma; \Psi \vdash \langle o \rangle : \langle c \rangle}$$

TYPEOF-LETOBJ

$$\frac{\Gamma; \Psi \vdash e : \langle c \rangle \qquad \Gamma; \Psi,\ i : c \vdash e' : \tau \qquad i \notin \text{fv}(\tau)}{\Gamma; \Psi \vdash \texttt{letobj}\ i = e\ \texttt{in}\ e' : \tau}$$

STLC-TYPEOF-ANTIQUOTE

$$\frac{i : \hat{t} \in \Psi}{\Psi; \Delta \vdash \text{aq}(i) : \hat{t}}$$

EVAL-LIFTOBJ

$$\frac{}{\langle o \rangle \Downarrow \langle o \rangle}$$

EVAL-LETOBJ

$$\frac{e \Downarrow \langle o \rangle \qquad e'[o/i] \Downarrow v}{\texttt{letobj}\ i = e\ \texttt{in}\ e' \Downarrow v}$$

SUBSTOBJ

$e[o/i] = e'$ is defined by structural recursion, save for:

$\text{aq}(i)[\hat{e}/i] = \hat{e}$

The typing rules should be quite simple to transcribe to Makam:

```
object, class, index : type.
classof : object → class → prop.
classof_index : index → class → prop.
subst_obj : (I_E: index → term) (O: object) (E_OforI: term) → prop.

liftobj : object → term. liftclass : class → typ.
letobj : term → (index → term) → term.

typeof (liftobj O) (liftclass C) :- classof O C.
typeof (letobj E EF') T :-
  typeof E (liftclass C), (i:index → classof_index i C → typeof (EF' i) T).

eval (liftobj O) (liftobj O).
eval (letobj E I_E') V :-
  eval E (liftobj O), subst_obj I_E' O E', eval E' V.
```

ROZA. Great. I'll add the object language in a separate namespace prefix – we can use '`%extend`' for going into a namespace – and I'll just copy-paste our STLC code from earlier on, plus natural numbers. Let me also add our new antiquote as a new STLC term constructor!

```
%extend stlc.
term : type. typ : type. typeof : term → typ → prop.
...
aq : index → term.
%end.
```

HAGOP. Time to add STLC terms as `objects` and their types as `classes`. We can then give the corresponding rule for `classof`. And I think that's it for the typing rules!

```
obj_term : stlc.term → object. cls_typ : stlc.typ → class.
classof (obj_term E) (cls_typ T) :- stlc.typeof E T.
stlc.typeof (stlc.aq I) T :- classof_index I (cls_typ T).
```

(Hagop transcribes the example from before. Writing out the term takes several lines, so he finds himself wishing that Makam supported some way to write terms of object languages in their native syntax.)

```
typeof (letrec (bind (fun power ⇒ body ((* ..long term.. *), power)))) T ?
>> Yes:
>> T := arrow onat (liftclass (cls_typ (stlc.arrow stlc.onat stlc.onat))).
```

ROZA. That's great! The only thing missing to try out an evaluation example too is implementing `subst_obj`. Thanks to `structural_recursion`, though, that is very easy:

```
subst_obj_aux, subst_obj_cases : [Any]
  (Var: index) (Replacement: object) (Where: Any) (Result: Any) → prop.

subst_obj I_Term O Term_OforI :-
  (i:index → subst_obj_aux i O (I_Term i) Term_OforI).
subst_obj_aux Var Replacement Where Result :-
  if (subst_obj_cases Var Replacement Where Result)
  then success
  else (structural_recursion @(subst_obj_aux Var Replacement) Where Result).
subst_obj_cases Var (obj_term Replacement) (stlc.aq Var) Replacement.
```

My definition here is quite subtle, so let me walk you through it. First, we extend the `subst_obj` predicate to work on any type – that's what `subst_obj_aux` is for. We set up the structural recursion, by attempting to see whether the "essential" cases actually apply – those are captured in the `subst_obj_cases` predicate. If they don't, that means we should proceed by structural recursion. I did not mention it before, but the `@` notation that we used to treat a polymorphic constant as a term of type `forall A T` can be used with an arbitrary term as well, to assign it such a type if possible. Finally, the essential case itself is a direct transcription of the pen-and-paper version.

HAGOP. Let me go and reread that a little. (…) I think it makes sense now. Well, is that all? Are we done?

```
eval (letrec (bind (fun power ⇒ body ([ (* .. definition of power *) ],
        (* body of letrec: *) app power (osucc (osucc ozero)))))) V ?
>> Yes!!!
>> V := < obj_term (λx.x * ((λa.a * (λb.1) a) x)) >.
```

ROZA. See, even the Makam REPL is excited[9]! That looks correct, even though there are a lot of administrative redices. We should be able to fix that with the next kind of object in our check-list, though: open STLC terms! That way, instead of having `power` return an object containing a lambda function, it can return an open term. Here's how I would write the same example from before:

[9]We have taken the liberty here to transcribe the result to more meaningful syntax to make it easier to verify.

```
let rec power_aux (n: onat): < [ stlc.onat ] stlc.onat > =
  match n with
    0 ⇒ < [x]. 1 >
  | S n' ⇒ letobj I = power_aux n' in < [x]. stlc.mult ~(I/[x]) x >
```

We have to list out explicitly the variables that an open term depends on, so that's the `[x].` notation I use. Then, we can use contextual types [Nanevski et al. 2008] for the type of those open terms.

HAGOP. Good thing I've already printed the paper out. (...) OK, so it says here that we can use contextual types to record, at the type level, the context that open terms depend on. So let's say an open `stlc.term` of type t that mentions variables of a Δ context would have a contextual type of the form $[\Delta]t$. This is some sort of modal typing, with a precise context.

ROZA. Right. We now get to the tricky part: referring to variables that stand for open terms within other terms! You know what those are, right? Those are Object-level Object-level Meta-variables.

HAGOP. My head hurts; I'm getting OOM errors. Maybe this is easier to implement in Makam than to talk about.

ROZA. Maybe so. Well, let me just say this: those variables will stand for open terms that depend on a specific context Δ, but we might use them at a different context Δ'. We need a *substitution* σ to go from the context of definition into the current context. I think writing down the rules on paper will help:

OB-OB-SYNTAX

$$o ::= \ldots \mid [x_1, \ldots, x_n].\hat{e} \qquad c ::= \ldots \mid [\hat{t}_1, \ldots, \hat{t}_n]\hat{t}$$
$$\hat{e} ::= \ldots \mid \mathsf{aqopen}(i)/\sigma \qquad \sigma ::= [\hat{e}_1, \ldots, \hat{e}_n]$$

CLASSOF-OPENTERM

$$\frac{\Psi; x_1 : \hat{t}_1, \ldots, x_n : \hat{t}_n \vdash \hat{e} : \hat{t}}{\Psi \vdash_o [x_1, \ldots, x_n].\hat{e} : [\hat{t}_1, \ldots, \hat{t}_n]\hat{t}}$$

STLC-TYPEOF-ANTIQUOTEOPEN

$$\frac{i : [\hat{t}_1, \ldots, \hat{t}_n]\hat{t} \in \Psi \qquad \forall i. \Psi; \Delta \vdash \hat{e}_i : \hat{t}_i}{\Psi; \Delta \vdash \mathsf{aqopen}(i)/[\hat{e}_1, \ldots, \hat{e}_n] : \hat{t}}$$

SUBSTOBJ

$$(\mathsf{aqopen}(i)/\sigma)[[x_1, \ldots, x_n].\hat{e}/i] = \hat{e}[\hat{e}_1/x_1, \ldots, \hat{e}_n/x_n] \text{ if } \sigma[[x_1, \ldots, x_n].\hat{e}/i] = [\hat{e}_1, \ldots, \hat{e}_n]$$

HAGOP. I've seen that rule for SUBSTOBJ before, and it is still tricky... We need to replace the open variables in e through the substitution $\sigma = [\hat{e}_1^*, \ldots, \hat{e}_n^*]$. However, the terms $\hat{e}_1^*$ through $\hat{e}_n^*$ might mention the i index themselves, so we first need to apply the top-level substitution for i to σ itself! After that, we do replace the open variables in $\hat{e}$.

ROZA. I feel that we are getting to the point where it's easier to write things down in Makam rather than on paper:

```
obj_openterm : bindmany stlc.term stlc.term → object.
cls_ctxtyp : list stlc.typ → stlc.typ → class.

%extend stlc.
aqopen : index → list term → term.
typeof (aqopen I ES) T :-
  classof_index I (cls_ctxtyp TS T), map typeof ES TS.
%end.

classof (obj_openterm XS_E) (cls_ctxtyp TS T) :-
  openmany XS_E (fun xs e ⇒
    assumemany stlc.typeof xs TS (stlc.typeof e T)).
```

```
subst_obj_cases I (obj_openterm XS_E) (stlc.aqopen I ES) Result :-
  subst_obj_aux I (obj_openterm XS_E) ES ES',
  applymany XS_E ES' Result.
```

HAGOP. I think that's all! This is exciting – let me try it out:

```
(eq _TERM (letrec (bind (fun power ⇒ body ([
   lam onat (fun n ⇒
   case_or_else n (patt_ozero)
     (vbody (liftobj (obj_openterm (bind (fun x ⇒
       body (stlc.osucc stlc.ozero))))))
   (case_or_else n (patt_osucc patt_var)
     (vbind (fun n' ⇒ vbody (
       letobj (app power n') (fun i ⇒
       liftobj (obj_openterm (bind (fun x ⇒
         body (stlc.mult x (stlc.aqopen i [x]))))))))))
   (liftobj (obj_openterm (bind (fun x ⇒ body stlc.ozero))))
   ))], app power (osucc (osucc ozero)))))),
  typeof _TERM T, eval _TERM V) ?
>> Yes:
>> T := liftclass (cls_ctxtyp (cons stlc.onat nil) stlc.onat),
>> V := liftobj (obj_openterm (bind (fun x ⇒ body (
          stlc.mult x (stlc.mult x (stlc.osucc stlc.ozero)))))).
```

It works! That's it! I cannot believe how easy this was!

AUDIENCE. We cannot possibly believe that you are claiming this was easy!

ANONYMOUS AUTHOR. Still, try implementing something like this without a metalanguage… It takes a long time! As a result, it limits our ability to experiment with and iterate on new language-design ideas. That's why we started working on Makam. That took a few years, but now we can at least show a type system like this in 28 pages of a single-column PDF!

ROZA. I wonder where all these voices are coming from?

HAGOP. They somehow sound like the ghosts of people who left academia for industry?

10 IN WHICH OUR HERO ROZA IMPLEMENTS TYPE GENERALIZATION, TYING LOOSE ENDS

> "We mentioned Hindley-Milner / we don't want you to be sad.
> This paper's going to end soon / and it wasn't all that bad.
>
> (Before we get to that, though / it's time to take a break.
> If taksims seem monotonous / then put on some Nick Drake.)
>
> We'll gather unif-variables / with structural recursion
> and if you haven't guessed it yet / we'll get to use reflection."

ROZA. Let me now show you how to implement type generalization for polymorphic `let` in the style of Damas [1984]; Hindley [1969]; Milner [1978]. I've done this before[10], and I need to leave

[10]There is existing work that has considered the problem of ML type generalization in the λProlog setting [Dietzen and Pfenning 1991; Liang 1995]. Our presentation here follows a different approach based on reflective and generic predicates.

for home soon, so bear with me for a bit. The gist will be to reuse the unification support of our metalanguage, capturing the *metalevel unification variables* and generalizing over them. That way we will have a very short implementation, and we won't have to do a deep embedding of unification!

HAGOP. So – you're saying that in λProlog, other than reusing the metalevel function type for implementing object-level substitution, we can also reuse metalevel unification for the object level as well.

ROZA. Exactly! First of all, the typing rule for a generalizing let 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 don't have any side-effectful operations, so there is no need for a value restriction. Transcribing this to Makam is easy, if we assume a predicate for generalizing the type, for now:

```
generalize : (Type: typ) (GeneralizedType: typ) → prop.
let : term → (term → term) → term.

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

Now, for generalization itself, we need the following ingredients based on the typing rule:

- something that picks out free variables from a type, standing for the $\mathrm{fv}(\tau)$ part – or, in our setting, this should really be read as uninstantiated unification variables. Those are the Makam-level unification variables that have not been forced to unify with concrete types because of the rest of the typing rules.
- something that picks out free variables from the local context: the $\mathrm{fv}(\Gamma)$ part. Again, these are the uninstantiated unification variables rather than the free variables. In our case, the context Γ is represented by the local `typeof` assumptions that our typing rules add, so we'll have to look at those somehow.
- a way to turn something that includes unification variables into a $\forall$ type, corresponding to the $\forall \vec{a}.\tau$ part. This essentially abstracts over a number of variables and uses them as the replacement for the unification variables inside τ.

All of those look like things that we should be able to do with our generic recursion and with the reflective predicates we've been using! However, to make the implementation simpler, we will generalize over one variable at a time, instead of all at once – but that should be entirely equivalent to what's described in the typing rule.

First, we will define a `findunif` predicate that returns *one* unification variable *of the right type* from a term, if at least one such variable exists. To implement it, we will make use of a generic operation in the Makam standard library, called `generic_fold`. It is quite similar to `structural_recursion`, but it does a fold through a term, carrying an accumulator through. Pretty standard, really, and its code is similar to what we did already for `structural_recursion`, with no new surprises.

```
findunif_aux : [Any VarType]
  (Var: option VarType) (Current: Any) (Var': option VarType) → prop.
findunif_aux (some Var) _ (some Var).
findunif_aux none (Current : CurrentType) (Result: option VarType) :-
  refl.isunif Current,
```

```
  if (dyn.eq Result (some Current)) then success
  else (eq Result none).
findunif_aux (In: option B) Current Out :-
  generic_fold @findunif_aux In Current Out.
findunif : [Any VarType] (Search: Any) (Found: VarType) → prop.
findunif Search Found :- findunif_aux none Search (some Found).
```

Here, the second rule of `findunif_aux` is the important one – it will only succeed when we encounter a unification variable of the *same type* `VarType` as the one we require. This rule relies on the dynamic `typecase` aspect of the ad-hoc polymorphism in λProlog, making use of the `dyn.eq` standard-library predicate, which has a lax typing:

```
dyn.eq : [A B] A → B → prop.
dyn.eq X X.
```

Though this predicate succeeds for the same case as the standard `eq` does (when its two arguments are unifiable), the difference is that `dyn.eq` only forces the types `A` and `B` to be unified at runtime, rather than statically, too. Otherwise, our rule would only apply when the type `CurrentType` of the current unification variable we are visiting already matches the type that we are searching for, `VarType`.

With `findunif` defined, we should already be able to find *one* (as opposed to all, as described above) uninstantiated unification variable from a type. Here is an example of its use:

```
findunif (arrowmany TS T) (X: typ) ?
>> Yes:
>> X := T.
```

Now we add a predicate `replaceunif` that, given a specific unification variable and a specific term, replaces the variable's occurrences with the term. This will be needed as part of the $\forall\vec{a}.\tau$ operation of the rule. Here I'll need another reflective predicate, `refl.sameunif`, that succeeds when its two arguments are the same exact unification variable; `eq` would just unify them, which is not what we want.

```
replaceunif : [VarType Any]
  (Which: VarType) (ToWhat: VarType) (Where: Any) (Result: Any) → prop.
replaceunif Which ToWhat Where ToWhat :-
  refl.isunif Where, refl.sameunif Which Where.
replaceunif Which ToWhat Where Where :-
  refl.isunif Where, not(refl.sameunif Which Where).
replaceunif Which ToWhat Where Result :- not(refl.isunif Where),
  structural_recursion @(replaceunif Which ToWhat) Where Result.
```

Last, we'll need an auxiliary predicate that tells us whether a unification variable exists within a term. This is easy; it's similar to the above.

```
hasunif_aux : [VarType Any] VarType → bool → Any → bool → prop.
hasunif_aux _ true _ true.
hasunif_aux X false Y true :- refl.sameunif X Y.
hasunif_aux X In Y Out :- generic_fold @(hasunif_aux X) In Y Out.

hasunif : [VarType Any] VarType → Any → prop.
hasunif Var Term :- hasunif_aux Var false Term true.
```

We are now mostly ready to implement `generalize`. We'll do this recursively. The base case is when there are no unification variables within a type left:

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

For the recursive case, we will pick out the first unification variable that we come upon using `findunif`. We will generalize over it using `replaceunif` and then proceed to the rest. Still, there is a last hurdle: we have to skip over the unification variables that are in the Γ environment. For the time being, let's assume a predicate that gives us all the types in the environment, so we can write the recursive case down:

```
get_types_in_environment : [A] A → prop.
generalize T Res :-
  findunif T Var, get_types_in_environment GammaTypes,
  (x:typ → (replaceunif Var x T (T' x), generalize (T' x) (T'' x))),
  if (hasunif Var GammaTypes) then (eq Res (T'' Var))
  else (eq Res (tforall T'')).
```

HAGOP. Oh, clever. But what should `get_types_in_environment` be? Don't we have to go back and thread a list of types through our `typeof` predicate, to which we add a type `T` every time we introduce a new `typeof x T` assumption?

ROZA. Well, we came this far without significantly rewriting our rules, so it's a shame to do that now! Maybe we'll be excused to use yet another reflective predicate that does what we want? There is a way to get a list of all the local assumptions for a predicate, through the reflexive predicate `refl.assume_get`. It turns out that all the rules and connectives we have been using are normal λProlog terms like any other, so there's not really much magic to it. And those assumptions will include all the types in Γ....

```
get_types_in_environment Assumptions :-
  refl.assume_get typeof Assumptions.
```

HAGOP. Wait. It can't be.

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

ROZA. And yet, it can.

11 IN WHICH OUR HEROES SUMMARIZE WHAT THEY'VE DONE AND OUR STORY CONCLUDES BEFORE THE CREDITS START ROLLING

HAGOP. I feel like we've done a lot here. And some of the things we did I don't think I've seen in the literature before, but then again, it's not clear to me what's Makam-specific and what isn't. In any case, I think a lot of people would find that quickly prototyping their PL research ideas using this style of higher-order logic programming is very useful.

ROZA. I agree, though it would be hard for somebody to publish a paper on this. Some of it is novel, some of it is folklore, some of it, we just did in a pleasant way; and we did also use a couple of not-so-pleasant hacks. But let's make a list of what's what.

- We defined HOAS encodings of complicated binding forms, including mutually recursive definitions and patterns, while only having explicit support in our metalanguage for single-variable binding. These encodings seem to have been part of PL folklore, but we believe that a

type like `bindmany` has never been shown as a reusable datatype that λProlog makes possible. We have made use of GADTs to encode some of these binding structures precisely, which we show are supported in λProlog through ad-hoc polymorphism. Defininining GADTs is a novel usage for this λProlog feature. Our binding constructions should be replicable in other λProlog implementations like Teyjus [Gacek et al. 2015; Nadathur and Mitchell 1999] and ELPI [Dunchev et al. 2015].

- We defined a generic predicate to perform structural recursion using a very concise definition. It allows us to define structurally recursive predicates that only explicitly list out the important cases, in what we believe is a novel encoding for the λProlog setting. Any new definitions, such as constructors or datatypes we introduce later, do not need any special provision to be covered by the same predicates. They depend on a number of reflective predicates, most of which are available in other Prolog and λProlog dialects. These predicates are used to reflect on the structure of Makam terms and to get the list of local assumptions; for the most part, their use is limited to predicates that would be part of the standard library, not in user code.
- The above encodings are reusable and have been made part of the Makam standard library. As a result, we were able to develop the type checker for quite an advanced type system, in very few lines of code specific to it, using rules that we believe do not, presentation-wise, stray far from their pen-and-paper versions. Our development includes mutually recursive definitions, polymorphism, datatypes, pattern matching, a conversion rule, Hindley-Milner type generalization, and staging constructs that allow the computation of contextually typed open terms of the simply typed lambda calculus. We are not aware of another metalinguistic framework that allows this level of expressivity and has been used to encode such type-system features with the same level of concision.
- We have also shown that higher-order logic programming allows not just meta-level functions to be reused for encoding object-level binding; there are also cases where meta-level unification can be reused to encode certain object-level features: for example, doing type generalization as in Algorithm W.

HAGOP. Well, that was very interesting; thank you for working with me on this!

ROZA. I enjoyed this, too. Say, if you want to relax, there's a new staging of the classic play by Fischer et al. [2010] downtown – I saw it yesterday, and it is really good!

HAGOP. That's a great idea! You know, I wish there were more plays like it.... Well, good night, and see you on Monday!

12 IN WHICH OUR HEROES ARE NOWHERE TO BE FOUND, LOST IN A SEA OF REFERENCES TO RELATED WORK

The **Racket programming language** was designed to support creation of new programming languages [Felleisen et al. 2015] and has been used to implement a very wide variety of DSLs, including typed languages such as Typed Racket by Tobin-Hochstadt et al. [2011]. We believe that one of the key characteristics of the Racket approach to language implementation is the ability to create towers of abstraction through programmatic manipulation of code. Makam is inspired by this approach to a large extent, and our plan for future work follows similar lines with regards to manipulating Makam code within Makam. Still, there is a lot of potential for cross-fertilization since the presence of first-class binding support in the form of higher-order abstract syntax, together with the built-in support for higher-order unification, makes the λProlog setting significantly different.

The recent development of a method for implementing **type systems as macros** by Chang et al. [2017] is a great validation of the Racket approach and is especially relevant to our use case, as it has

been used to encode type systems similar to ML. The integration that this methodology provides with the rest of the Racket ecosystem offers a number of advantages, as does the Turnstile DSL for writing typing rules close to the pen-and-paper versions. We do believe that the higher-order logic programming setting allows for more expressivity and genericity – for example, we have used the same techniques to define not only typing rules but evaluation rules as well; also, it is not immediately clear to us whether examples such as our MetaML fragment or Hindley-Milner type inference would be as easy to implement in Turnstile, especially since the latter presently lacks support for unification. We are exploring an approach similar to Turnstile to implement a higher-level surface language for writing typing rules using Makam itself.

Evaluation rules can be implemented using another DSL of the Racket ecosystem, namely **PLT Redex** [Felleisen et al. 2009], so a change of framework is required. We believe that staying within the same framework for both typing and evaluation semantics offers advantages, especially for encoding languages where the two aspects are more linked, such as dependently typed languages with the conversion rule. We give one small example of that in the form of the type synonyms example. We also find that the presence of first-class substitution support and the support for structural recursion in Makam offers advantages over PLT Redex.

The **Spoofax language workbench** [Kats and Visser 2010] offers a series of DSLs for implementing different aspects of a language, such as parsing, binding, typing and dynamic semantics. We have found that some of these DSLs have restrictions that would make the implementation of type systems similar to the ones we demonstrate in the present work challenging. Our intention with the design of Makam as a language prototyping tool is for it to be a single expressive core, where all different aspects of a language can be implemented. The **K Framework** [Roşu and Şerbănuţă 2010] is a semantics framework based on rewriting and has been used to implement the dynamic semantics of a wide variety of languages. It has also been shown to be effective for the implementation of type systems [Ellison et al. 2008], treating them as abstract machines that compute types rather than values. The recent addition of a builtin unification procedure has made this approach significantly more effective, allowing the definition of ML type inference; however, the fact that λProlog supports higher-order unification as well renders it applicable in further situations such as dependently typed systems.

Future work. We are exploring the addition of a staging construct to Makam, which allows us to implement extensions to the language within the language itself, by implementing predicates that produce new Makam definitions, rules, etc. Examples of this approach are a library for defining surface syntax for object languages; also, a language for describing the binding structure of an object language, alleviating the non-intuitive aspects of some of our encodings. This approach can also help reduce the reliance on reflective predicates in some of the examples we show (e.g. structural recursion), by restricting the use of reflection within staged code.

ACKNOWLEDGMENTS

We thank Tej Chajed, Stephen Chang, Ben Greenman, Dale Miller, Gopalan Nadathur, and Clément Pit-Claudel, as well as the anonymous reviewers of the present paper and of an earlier version, for their very helpful feedback and suggestions. This project was started while the first author was at MIT. Further work by the first author has been supported by the 20% time program at Originate.

This material is based upon work supported in part by the NSF award CCF-1217501. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. 2014. Abella: A System for Reasoning about Relational Specifications. *Journal of Formalized Reasoning* 7, 2 (2014). https://doi.org/10.6092/issn.1972-5787/4650

Stephen Chang, Alex Knauth, and Ben Greenman. 2017. Type Systems As Macros. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 694–705. https://doi.org/10.1145/3009837.3009886

Luís Damas. 1984. *Type assignment in programming languages.* Ph.D. Dissertation. University of Edinburgh, UK. http://hdl.handle.net/1842/13555

Scott Dietzen and Frank Pfenning. 1991. A Declarative Alternative to "Assert" in Logic Programming. In *Proceedings of the 1991 International Logic Programming Symposium*. MIT Press, 372–386.

Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. 2015. ELPI: fast, Embeddable, λProlog Interpreter. In *Proceedings of LPAR*. Suva, Fiji. https://hal.inria.fr/hal-01176856

Chucky Ellison, Traian Florin Şerbănuţă, and Grigore Roşu. 2008. A rewriting logic approach to type inference. In *International Workshop on Algebraic Development Techniques*. Springer, 135–151. https://doi.org/10.1007/978-3-642-03429-9_10

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex.* MIT Press. http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11885

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 113–128. https://doi.org/10.4230/LIPIcs.SNAPL.2015.113

Sebastian Fischer, Frank Huch, and Thomas Wilke. 2010. A play on regular expressions: functional pearl. In *Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. 357–368. https://doi.org/10.1145/1863543.1863594

Andrew Gacek, Steven Holte, Gopalan Nadathur, Xiaochu Qi, and Zach Snow. 2015. The Teyjus system–version 2.

Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (Jan. 1993), 143–184. https://doi.org/10.1145/138027.138060

Roger Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60.

Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. 444–463. https://doi.org/10.1145/1869459.1869497

Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. 2016. Needle & Knot: Binder Boilerplate Tied Up. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 419–445. https://doi.org/10.1007/978-3-662-49498-1_17

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*. 192–203. https://doi.org/10.1145/1086365.1086390

Chuck C Liang. 1995. *Object-Level Substitution, Unification and Generalization in Meta-Logic.* Ph.D. Dissertation. University of Pennsylvania.

Geoffrey Mainland. 2012. Explicitly heterogeneous metaprogramming with MetaHaskell. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*. 311–322. https://doi.org/10.1145/2364527.2364572

Dale Miller and Gopalan Nadathur. 2012. *Programming with Higher-Order Logic.* Cambridge University Press. https://doi.org/10.1017/CBO9781139021326

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348 – 375. https://doi.org/10.1016/0022-0000(78)90014-4

Gopalan Nadathur and Dale Miller. 1988. An Overview of Lambda-PROLOG. In *Proceedings of the Fifth International Conference on Logic Programming, Seattle, Washington, August 15-19, 1988 (2 Volumes)*. 810–827.

Gopalan Nadathur and Dustin J. Mitchell. 1999. System Description: Teyjus - A Compiler and Abstract Machine Based Implementation of λProlog. In *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*. 287–291. https://doi.org/10.1007/3-540-48660-7_25

Gopalan Nadathur and Xiaochu Qi. 2005. Optimizing the Runtime Processing of Types in Polymorphic Logic Programming Languages. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Geoff Sutcliffe and Andrei Voronkov (Eds.).

Springer Berlin Heidelberg, Berlin, Heidelberg, 110–124.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic* 9, 3 (2008), 23:1–23:49. https://doi.org/10.1145/1352582.1352591

Frank Pfenning and Conal Elliott. 1988. Higher-order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. ACM, New York, NY, USA, 199–208. https://doi.org/10.1145/53990.54010

Brigitte Pientka and Joshua Dunfield. 2010. Beluga: a Framework for Programming and Reasoning with Deductive Systems (System Description). In *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*. 15–21. https://doi.org/10.1007/978-3-642-14203-1_2

Grigore Roşu and Traian-Florin Şerbănuţă. 2010. An overview of the K semantic framework. *J. Log. Algebr. Program.* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

Antonis Stampoulis. 2013. *VeriML: A dependently-typed, user extensible and language-centric approach to proof assistants*. Ph.D. Dissertation. Yale University.

Walid Taha and Tim Sheard. 1997. Multi-stage Programming with Explicit Annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '97)*. ACM, New York, NY, USA, 203–217. https://doi.org/10.1145/258993.259019

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages As Libraries. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 132–141. https://doi.org/10.1145/1993498.1993514

Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96. https://doi.org/10.1017/S1471068411000494

Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2013. Mtac: a monad for typed tactic programming in Coq. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 87–100. https://doi.org/10.1145/2500365.2500579