The Makam Metalanguage

Technical Overview

Antonis Stampoulis and Adam Chlipala
MIT CSAIL
Cambridge, MA, USA

June 30, 2014

1. Motivation

Programming language design is challenging. The language designer is faced with an immense spectrum of choices, whose advantages and disadvantages become apparent only once the alternatives are tried out in practice. Furthermore, producing an actual implementation is hard work, especially when the language supports an advanced type system (e.g. dependent types). Important aspects of the implementation that greatly determine whether it can be put to use in practice, such as performance, error messages presented to the user and interoperability with existing languages, frameworks and libraries, require substantial amounts of additional work. The extensibility of a design is also of utmost importance, yet it is often at odds with producing a realistic implementation of the current design – basic assumptions might change that invalidate the implementation effort already invested. Because of these challenges, a lot of research on programming languages and type systems tends to stay at the prototype level.

The use of various meta-programming techniques is widely understood to be the key solution to this situation. Examples of such techniques are the hygienic macro system of Racket; embedding domain-specific languages in a host language with a rich type system, such as Haskell or Agda; meta-linguistic tools for modeling the semantics of languages, such as PLT Redex and the K Framework; and meta-programming extensions to existing languages for generating and evaluating staged code, as in MetaML and Scala/LMS. We draw inspiration from these techniques, yet our experience with them shows that their expressivity is lacking when the object languages that we want to encode feature rich type systems of their own.

The goal of our research on Makam at MIT is to design and implement a metalanguage that can be used as a practical language experimentation and implementation tool. We are interested in modeling real languages such as OCaml and Haskell and exploring sophisticated extensions to them, such as contextual typing – motivated by my earlier work on VeriML – and type-level computation – motivated by the work of my post-doc advisor, Prof. Adam Chlipala, on Ur/Web. We have arrived at a core formalism by revisiting the design of the λProlog language and refining it for our purposes. This formalism is at once conceptually clear yet surprisingly rich and expressive; even as the designers and implementors of Makam, we have often discovered new possibilities and usage patterns that we did not expect, enabling us to model particular tricky features such as multiple binding and substructural contexts, without any changes to the underlying metalanguage implementation. Makam thus reflects a quality of its namesake: the makam (pl. makamat) of traditional Arabic and Turkish music (similar to the Indian raga and the $\delta\rho\acute{o}\mu o\iota$ of the Greek rembetiko), an improvisational music style comprised of a set of scales, patterns of melodic development and rules for improvisation so rich that musicians trained in them discover new possibilities even after decades of practice.

We take a somewhat extreme stance with respect to meta-programming: the metalanguage implementation should consist of the bare minimum needed to allow object language modeling to be done effectively. We aim to reuse the metalanguage itself for exploring further extensions and optimizations to it, as well as (eventually) for bootstrapping. The end goal of our approach is to produce a bootstrapped compiler compiler tool – that

is, a version of Makam written within Makam, which is given as input a specification of a language, along with specifications of its type system, static analyses, and transformation phases and produces as output an efficient executable implementation of the language.

Though we are still years far from this goal, we trust that the language design is expressive enough to model many interesting ideas from the field of programming language research in a pleasant manner; and also, that our implementation is efficient enough so that it can be used to encode languages of substantial complexity and evaluate their properties through nontrivial examples. We have, for example, modeled a large part of the OCaml language along with its type system and dynamic semantics. Our specification includes data type declarations, polymorphism and pattern-matching in about 800 lines of highly readable code. Our tool further enabled us to explore extensions to the type system, such as Haskell-style type classes and type class instance discovery, by extending the specification with about 100 lines of additional code and without changing the existing specification. We have also modeled a large part of type-directed compilation of System F into Typed Assembly Language, by following closely the classic research paper by Morrisett et al. Last, we have encoded the type system of the VeriML language, which uses a sophisticated contextual type system with dependent types. All of these examples can be coded in a matter of a few days by an expert user given access to the reusable Makam library. We are not aware of any other metalanguage that can handle object languages of comparable complexity with the same level of conciseness and clarity of expression.

Makam thus provides an interesting trade-off between implementation cost and the practicality of the yielded implementation. We believe that it is already at the level that it will prove to be an invaluable tool to researchers in programming languages who want to evaluate new ideas. In the rest of this document, we will present an overview of the main features of Makam and how they can be used to model common features of object languages.

2. Technical overview: The λProlog part of Makam.

The core of the Makam metalanguage is based on logic programming: specifications take the form of declarative and executable Prolog-style rules, which relate – for example – terms with their types, or terms of a higher-level language with their translation to a lower-level language. Our tool can be viewed as an interpreter for those logic programs. We take the atomic terms mentioned in rules to be open terms of the polymorphic λ-calculus. We will explain this choice in detail in the following; for the time being, it suffices to say that it enables us to deal effectively with some key challenges when modeling languages and their type systems[1]. This choice renders our metalanguage as a version of the λProlog language as proposed by Miller and Nadathur [1988].

We will now present the core ideas behind our metalanguage through a number of examples. Note that the features we will present in this section are shared with λProlog, as a reminder or tutorial for our readers; in the next sections we will present the departure points from λProlog and the novel features that are possible in Makam. Let us start with a toy expression language, which consists of arithmetic and Boolean expressions, as well as conditionals. We need to specify in our metalanguage the abstract syntax trees for such expressions. In contrast to normal Prolog dialects, our metalanguage is typed, so we need to declare the constructors of these syntax trees explicitly, along with their types; the type itself must be declared if it does not already exist. We thus start by defining the type `expr` for expressions of the language, along with a number of constructors for it:

```
expr : type.

plus, minus         : expr -> expr -> expr.
lessthan, lessequal : expr -> expr -> expr.
```

[1] Namely, the correct management of binding, the use of unification in presence of open terms and being able to reuse definitions.

```
and, or             : expr -> expr -> expr.
ifthenelse          : expr -> expr -> expr -> expr.
intconst            : int -> expr.
boolconst           : bool -> expr.
```

The types of constructors are curried; in the case of integer and Boolean constants, we use the built-in metalanguage types `int` and `bool` to inject meta-terms of these types into the object language.

Let us now define a type system for this language, in order to disallow expressions that use an integer expression at a place where a Boolean expression is expected. We first model the abstract syntax of the object language types as the meta-type `typ`, similarly as above, in the code that follows. We use different identifiers to avoid confusion with the metalanguage types.

```
typ   : type.

tint  : typ.
tbool : typ.
```

As noted earlier, we model the type system as a logic program: a relation `typeof` that relates an expression with its type. The built-in type `prop` is used as the result type of relations. The meta-type of `typeof` is thus:

```
typeof : expr -> typ -> prop.
```

Examples of the actual typing rules, given as a logic program, follow. Each rule consists of a goal and a number of premises; we use capital letters to denote unification variables. These rules closely resemble specifying the typing judgement of the language through inference rules on paper.

```
typeof (plus A B) tint <- typeof A tint, typeof B tint.
typeof (lessthan A B) tbool <- typeof A tint, typeof B tint.
typeof (ifthenelse A B C) T <- typeof A tbool, typeof B T, typeof C T.
```

Having specified the type system of the language, we can now type check expressions, by posing queries to the `typeof` relation. The execution model then follows normal logic programming: we unify the arguments to `typeof` given in the query with the arguments in each rule, keeping a backtracking point; upon successful unification, we perform queries for the premises; upon failure, we return to the most recent backtracking point and try other alternatives. Issuing the following query:

```
typeof (ifthenelse (lessthan 1 2) (intconst 5) (intconst 6)) T ?
```

our interpreter for the metalanguage will thus reply with `T := tint`, as expected.

2.1 Extensibility.

Due to the open nature of definitions and rules in logic programming, we can easily extend both the expressions of the language, as well as the operations on them – if we view relations such as `typeof` as operations. In this way we partly address the *expression problem*, a common issue in language implementation [Wadler, 1998]. For example, we can easily add strings to the language (again by reusing the meta-level string type):

```
stringconst   : string -> expr.
concat        : expr -> expr -> expr.
isempty       : expr -> expr.

tstring       : typ.
```

```
typeof (stringconst S) tstring.
typeof (concat S1 S2) tstring <- typeof S1 tstring, typeof S2 tstring.
typeof (isempty S) tbool <- typeof S tstring.
```

Towards the other axis of extensibility, we can add a new relation `compile` that will compile an expression into instructions for a stack-based machine. We won't give the full details here, but an (informal) sketch would be as follows:

```
instruction : type.
push : int -> instruction.
pop  : instruction.
add  : instruction.

compile : expr -> list instruction -> prop.
compile (intconst I) (push I).
compile (plus E1 E2) [ E1code , E2code , add ] <- compile E1 E1code, compile E2 E2code.
```

We can now issue queries to our tool not only for type checking, but for compilation of the expression language as well. Yet the use of logic programming means that we can use transformations in both directions, by choosing which argument to leave unspecified as a unification variable. This means that we can decompile programs as well – by issuing a query to `compile` with a concrete instruction list and a unifiable expression. Bidirectionality is often useful: for example, by specifying how to parse concrete syntax into abstract syntax for a language, we are also specifying how to pretty-print abstract syntax of the language, which can be used to print error messages to the user.

2.2 Binding constructs.

The correct handling of variable binding, renaming and substitution is an oft-cited challenge when specifying and implementing programming languages, requiring a lot of tedious and error-prone code (e.g. for implementing variables through de Bruijn indices). Maybe the key motivation and benefit behind the λProlog language design is exactly to address this challenge effectively. λProlog does so through the particularly elegant and powerful technique of higher-order abstract syntax (HOAS). The gist of this technique is to implement correct handling of binding once, in the core of the metalanguage, and reuse it to model binding in the object languages. The dependence on a new variable inside the body of a binding construct is represented through a *meta-level function*; substituting such a variable for a term is represented through *meta-level application*.

It is best to demonstrate this technique through an example: adding a construct for local definitions in the toy expression language we have described so far, with the concrete syntax `let x = e in e'`. The abstract syntax corresponding to it will be `let e e'`; yet based on the above idea, `e'` is understood to be a meta-level function. For example, the expression

```
let x = plus (intconst 1) (intconst 1) in plus x x
```

would be represented as

```
let (plus (intconst 1) (intconst 1)) (fun x => plus x x)
```

Thus the type for the new constructor `let` will be:

```
let : expr -> (expr -> expr) -> expr.
```

This is the main departure of λProlog from normal Prolog-based languages, as it changes the set of atomic terms: rather than being just algebraic expressions (or S-expressions), atomic terms are the terms of the simply typed lambda calculus. Following λProlog our metalanguage also extends the form of allowed premises in the logic programming rules, in order to allow us to work with such terms. We add an introduction form for new

variables, which allows us to inspect the body of meta-level functions by applying new variables to them; and a way to introduce assumptions for the newly introduced variables, in order to locally extend the set of rules. We make use of both of these forms in the following example of the typing rule for `let`:

```
typeof (let E E') T' <- typeof E T, (x:expr -> typeof x T -> typeof (E' x) T').
```

This rule can be read as follows: after determining the type of the expression E as T, determine the type of the body of the declaration E' under the assumption that the newly introduced variable, denoted as x, has type T.

Assuming a predicate `eval` for representing the evaluation semantics of the expression language, the following rule for the semantics of `let` is an example of using meta-level application to represent substitution of a variable for an expression:

```
eval : expr -> expr -> prop.
eval (let E E') V' <- eval E V, eval (E' V) V'.
```

which corresponds to the corresponding rule of big-step semantics:

$$\frac{E \Downarrow V \qquad E'[V/x] \Downarrow V'}{\text{let } x = E \text{ in } E' \Downarrow V'}$$

It is important to realize that the variable `x` in the typing rule above is abstract and that variables and terms are implicitly α-renamed to avoid collisions. For example, when typing the expression `let (intconst 1) (fun x => let (boolconst true) (fun x => x))` there are seemingly two typing assumptions for the variable `x`, namely `typeof x tint` and `typeof x tbool`. Still these two assumptions refer internally to different variables and thus only the latter applies for typing the body of the inner `let`.

The higher-order treatment of variables minimizes the challenge traditionally associated with implementing binding for programming languages. Yet this abstract view is sometimes cumbersome when concrete variable names are important – the most prevalent example being printing variables to the user (what gets printed if the variable got α-renamed?). Frameworks using the higher-order abstract syntax technique, such as λProlog and Twelf, are sometimes problematic in this aspect. Makam takes a hybrid stance against variable naming which allows us to treat variables with concrete names when needed and to control these names based on the naming policies of the object languages, while retaining the benefits of the existing HOAS technique. This is done by associating each abstract variable with a meta-variable standing for its concrete object-level name. Users are then free to explicitly set the concrete name as needed; furthermore, a set of heuristics propagates these concrete names so as to minimize the explicit name manipulation that needs to happen. While this behavior is entirely optional, we have found it useful, as the resulting terms are more readable in practice.

2.3 Unification in the presence of binding.

We remarked above that employing higher-order abstract syntax changes the set of atomic terms over which we define relations, to the terms of the simply typed lambda calculus. We tacitly assumed that the unification procedure is adapted to deal with such terms. Yet this adaptation is anything but trivial, as unification now has to work up to $\beta\eta$-equality so that we can unify, for example, the terms

```
(fun x => plus x x) (intconst 1)
```

and

```
plus (intconst 1) (intconst 1)
```

It is necessary that these terms be unifiable so that the above rule for `eval` actually behaves as expected (setting `E' := fun x => plus x x` and `V := intconst 1`).

In the general case, we have to solve the unification problem between open terms of the λ-calculus up to $\beta\eta$-equality; this problem is referred to as *higher-order unification* and is undecidable in the general case. Following

λProlog and other languages in the HOAS tradition, we restrict the unification problems that we actually solve to a decidable subset called higher-order pattern matching. We defer any unsolved unification goals to be attempted again as soon as more information becomes available (through other unifications). The procedure for higher-order pattern matching is one of the main technical challenges of implementing our metalanguage; we use an explicit substitution approach similar to Dowek et al. [1998]. We are also investigating extensions to the subset that we handle that maintain decidability of unification.

The essence behind including such a powerful unification procedure in the metalanguage is that we can reuse it for specifying aspects of object languages that are related to unification. For example, type inferencing engines for languages such as ML and Haskell can be viewed in large parts as solving particular restrictions of higher-order pattern matching. The same is true for more advanced type inferencing or type elaboration problems, such as those arising in dependently typed languages. We can implement these parts through the meta-level unification procedure, minimizing in this way a traditionally complicated implementation effort. Another example where our unification procedure can effectively be reused is the implementation of (first-order) pattern matching in the evaluation semantics of such languages.

2.4 Polymorphism and higher-order rules.

We allow ML-style prenex polymorphism in our atomic terms, in order to enable reuse of definitions for different types. Polymorphic lists are one such example; we define them as follows:

```
list : type -> type.
nil  : list A.
cons : A -> list A -> list A.
```

Operations over such types are defined using higher-order rules. For example, `map`ping a predicate over a list is specified below.

```
map : (A -> B -> prop) -> list A -> list B -> prop.
map P nil nil.
map P (cons HD TL) (cons HD' TL') <- P HD HD', map P TL TL'.
```

The seasoned functional programmer might be surprised to find that `map` takes a predicate rather than a function of type `A -> B` as its first argument. The reason is that meta-level functions are quite restricted so as to serve the purposes of higher-order abstract syntax; they correspond to *parametric* functions that cannot inspect the structure of their arguments. Otherwise the higher-order abstract syntax technique introduces exotic meta-level terms that do not correspond to valid terms of the object language. Absolutely all computation in Makam happens as part of queries using the `prop` universe. The role that functions of type `A -> B` play in functional languages corresponds here to the predicate type `A -> B -> prop` instead.

The higher-order nature of the rule above suggests that predicates can be viewed as normal atomic terms themselves, as is indeed true in our metalanguage. The constructs used to define the premises of rules (such as "`,`", "`(x:A -> ...)`", etc.) are viewed as syntactic sugar for constructors of the `prop` type:

```
and    : prop -> prop -> prop.
newvar : (A -> prop) -> prop.
assume : prop -> prop -> prop.
```

The built-in constructs that we support allow backtracking choice, logical conditionals and pruning[2]. The connective `ifte P Q R` thus supported is a logical if-then-else construct: it checks to see whether its conditional `P` can succeed; if so, it proceeds with `Q`, preserving all backtracking points for `P` and `Q`. However, if `P` fails, it proceeds with `R`. The pruning connective `once P` is a well-behaved variant of the cut functional-

[2]Following the the LogicT monad of Kiselyov et al. [2005].

ity of normal Prolog implementations: it proceeds with `P` until it succeeds only once, pruning all possible backtracking points within it. Users can define further connectives, such as negation-as-failure:

```
not : prop -> prop.
not P <- ifte (once(P)) failure success.
```

We also implement further built-in predicates to perform operations that are not possible or practical to specify through rules. Examples are operations on built-in types such as integers and strings; and reflective operations on terms to identify and abstract over unification variables (useful for implementing ML-style polymorphism for object languages).

3. Technical overview: The new features of Makam.

As we commented earlier, the features presented so far are directly inherited from the λProlog design. However, Makam has three key refinements that result in a surprising increase in the overall expressive power of the formalism, making it better suited for our purposes. First, Makam is interpreted rather than compiled and as a result does not distinguish between static and dynamic predicates. This allows for increased power in the use of higher-order predicates – for example, we can define predicates that compute other predicates, which can then be used as premises or assumptions. Second, Makam allows polymorphic recursion, which together with dynamic type unification allows us to define powerful new generic structures and auxiliary functions. Last, Makam allows staging and reflection, meaning that Makam programs can generate other Makam programs and inspect part of the logic programming state. This enables users to write new extensions to the metalanguage within the metalanguage itself. These technical decisions have been motivated by concrete needs in the definition of various object languages; it is their combination that gives Makam its expressive power. For the rest of this section, we will introduce these features through the examples that they enable and explain them in more detail.

3.1 Polymorphic binding structures.

Though polymorphic lists are a common feature in many programming languages, the combination of polymorphism with higher-order abstract syntax opens up new possibilities for a number of reusable data structures, which to our knowledge have not been explored as such before. These data structures correspond to different binding structures, such as binding multiple variables at once, mutually recursive bindings, telescopes as used in dependently typed languages and even substructural variables. As we have described above, Makam only provides native support for just the most basic of these – normal, single-binder introduction as in the λ-calculus. The rest can be implemented using Makam itself; we will give a small sample in this section.

Multiple bindings. We will start with a simple and widely used binding structure: binding multiple variables at once. Such a structure could be used for example for uncurried λ-abstraction (as used, for example, in intermediate representations of functional languages past closure conversion), which is the example that we will model here. It could also be used, for example, for modeling patterns in languages with pattern matching like ML and Haskell; and for modeling queries in the variadic π-calculus.

Normal λ-abstraction is encoded in an entirely similar way to the `let` construct we defined earlier:

```
lam : (expr -> expr) -> expr.
arrow : typ -> typ -> typ.
typeof (lam E) (arrow T' T) <-
  (x:expr -> typeof x T' -> typeof (E x) T).
```

Consider now instead a $\vec{\lambda}$ abstraction that binds multiple variables at once, used for example as follows:

$$\vec{\lambda}(f, g, x).f(g\,x)$$

With the typing rule:

$$\frac{\Gamma,\ x_1 : t_1,\ \cdots,\ x_n : t_n \vdash e : t}{\Gamma \vdash \vec{\lambda}(x_1, \cdots, x_n).e : (t_1,\ \cdots,\ t_n) \to t}$$

Our first attempt at modeling this construct, by analogy to the normal λ construct, would perhaps be:

```
(?)  lammany : (list expr -> expr) -> expr.
```

This is not quite what we want: there is a new constructor for lists in the body of the $\vec{\lambda}$ abstraction, but there is no way to refer to its members – remember that the Makam level function type does not allow any computation other than $\beta\eta$-expansion, hence there is no way to refer to the N-th member of a list.

What we want instead is to have N new constructors for the expr type, representing the N new variables, which will be available inside the body of the $\vec{\lambda}$ abstraction. We want something roughly as follows:

```
(?)  lammany : (expr -> (expr -> ... -> expr)) -> expr.
```

How do we write such a constructor? It might seem at first that there needs to be specialized support at the metalanguage level for its type. However, we can in fact code this type using Makam itself, using an auxiliary data type `bindmanyexprs` defined as follows:

```
bindmanyexprs : type.
bindonemore : (expr -> bindmanyexprs) -> bindmanyexprs.
binddone    : expr -> bindmanyexprs.
```

Using this type, we can encode $\vec{\lambda}$ as:

```
lammany : bindmanyexprs -> expr.
```

and use it for the example above as:

```
lammany (bindonemore (fun f => bindonemore (fun g => bindonemore (fun x =>
           binddone (app f (app g x))))))
```

The `bindmanyexprs` type can be generalized based on two type parameters: the type A of the variables that we are binding and the type B of the body that we are binding into. Thus a reusable polymorphic type for multiple binding would be the following:

```
bindmany : type -> type -> type.
bcons : (A -> bindmany A B) -> bindmany A B.
bnil  : B -> bindmany A B.
```

In the particular case of our example these two types happen to coincide, giving the following type for the $\vec{\lambda}$ construct:

```
lammany : bindmany expr expr -> expr.
```

We can imagine, however, cases where the two types are not the same – for example, multiple binding of type parameters for ML-style prenex polymorphism:

```
polylammany : bindmany typ expr -> expr.
```

With these definitions, we have encoded the abstract syntax of the $\vec{\lambda}$ construct with accurate binding behavior, giving us some of the nice properties that we expect from a HOAS representation like having automatic α-conversion as needed. Let us now see how to actually work with this representation – how to perform substitutions in it, how to write predicates over it, etc.

As we presented earlier, Makam offers us some primitive operations for the single-binding case: substitution as meta-level application; a predicate construct for introducing a fresh variable; and a way to assume facts

involving variables. We need equivalent operations for the `bindmany` construct we defined. Fortunately, these operations are definable in Makam using the primitive operations for the single-binding case and thus require no extensions to the base metalanguage.

Before we see how to implement them, let us present their types and usage.

```
apply_many   : bindmany A B -> list A -> B -> prop.
newvars_many : bindmany A B -> (list A -> prop) -> prop.
assume_many  : (A -> B -> prop) -> list A -> list B -> prop.
```

The `apply_many` predicate is used for multiple simultaneous substitution; it uses meta-level application to substitute the variables bound in the `bindmany` construct with the corresponding terms in the list it gets passed as an argument[3]. `newvars_many E P` introduces as many local fresh variables as bound in `E` and executes the predicate `P`; this predicate gets passed a list of the fresh variables. Last, the `assume_many P L L'` construct introduces a number of assumptions concerning the `P` predicate locally. We use two lists `L` and `L'` as input instead of a list of pairs as it results in more readable rules in practice.

We use them as follows for encoding the typing rule of $\vec{\lambda}$:

```
typeof (lammany XSBody) (arrowmany TS T) <-
  newvars_many XSBody (fun xs =>
    assume_many typeof xs TS (
    newmeta (fun Body =>
       apply_many XSBody xs Body,
       typeof Body T))).
```

where we have assumed the following new type:

```
arrowmany : list typ -> typ.
```

Comparing this code with the corresponding code for the normal λ-abstraction the reader will see that most steps are quite similar, barring the fact that we need to explicitly use a predicate for substitution instead of a primitive application. To make the comparison more straightforward, let us desugar the previous specification for typing `lam`, rename variables slightly and use a similar explicit `apply` predicate, as follows:

```
apply : (A -> B) -> A -> B -> prop.
apply F X (F X).
typeof (lam XBody) (arrow T' T) <-
  newvar (fun x =>
   assume (typeof x T') (
    apply XBody x (Body x),
    typeof (Body x) T)).
```

The comparison reveals one complication that is quite subtle – it is evidenced by the `newmeta` predicate in the `lammany` case, which is missing in the `lam` case. We have seen earlier that we need to apply a unification variable explicitly to the local variables that it is allowed to use – as is in fact true for the `Body` unification variable. There is however another way to achieve the same effect, which is what we are using in the `lammany` case: we can declare a new metavariable at a specific point inside a predicate, which will be able to capture all local fresh variables defined at that point. More precisely: by default, all meta-variables are allocated in the very outside scope of a predicate, which means they can capture no local variables at all; but using `newmeta` we can allocate a meta-variable in an inner scope so that local variables of that scope can be captured. We normally use the syntactic sugar `[X]` instead of `newmeta`.

With these in mind, the typing rules for `lam` and `lammany` can be made to look almost identical:

[3] This argument can be viewed as an explicit substitution.

```
typeof (lam XBody) (arrow T' T) <-
     newvar (fun x => assume (typeof x T') ([Body]
     apply XBody x Body, typeof Body T)).
typeof (lammany XSBody) (arrowmany TS T) <-
     newvars_many (fun xs => assume_many typeof xs TS ([Body]
     apply_many XBody xs Body, typeof Body T)).
```

Let us now see how to implement the required operations in Makam. `assume_many` is perhaps the easiest, introducing as many assumptions as there are elements in its input lists:

```
assume_many P [] [] Q <- Q.
assume_many P (A :: AS) (B :: BS) Q <- (P A B -> assume_many P AS BS Q).
```

`apply_many` is just as straightforward, performing meta-level application repeatedly for the variables bound through `bindmany`:

```
apply_many (bnil X)  [] X.
apply_many (bcons F) (HD :: TL) Res <- apply_many (F HD) TL Res.
```

The `newvars_many` predicate is slightly more complicated; it uses an auxiliary predicate to accumulate the list of fresh variables in order to pass it to its continuation predicate:

```
newvars_many_aux : bindmany A B -> (list A -> prop) -> list A -> prop.
newvars_many : bindmany A B -> (list A -> prop) -> prop.
newvars_many XSBody Cont <- newvars_many_aux XSBody Cont [].
newvars_many_aux (bnil _) Cont Vars <- Cont Vars.
newvars_many_aux (bcons Rest) Cont Vars <-
  (x:A -> append Vars [x] Vars', newvars_many_aux (Rest x) Cont Vars').
```

These predicates are not definable as above in the canonical λProlog implementation, Teyjus [Nadathur and Mitchell, 1999]. The culprit is the `assume_many` predicate, which introduces a local assumption for a dynamically determined predicate (through the assumption about P A B). Teyjus has a restriction[4] where such assumptions are not allowed, because they are not supported by its compilation procedure; we use an interpreter-based evaluation procedure instead, rendering statically and dynamically determined predicates entirely identical.

Mutually recursive binding. Having seen the details of multiple binding, a number of additional binding structures are now easy to define. We will briefly mention them here; for details we refer the reader to the `bindutils` library of Makam. One such example are mutually recursive definitions as in the `letrec` construct of ML, as used in the following (diverging) ML term:

`letrec f =` λ`x.g x and g =` λ`x.f x in f 0`

This construct could be defined using the following generic binding structure:

```
bindmutrec : type -> type -> type.
bmutrec : bindmany A (list B) -> bmutrec A B.
```

This will be used to bind a number of A's into the same number of definitions of type B; all definitions can use all the newly bound variables. Using this construct, `letrec` would be defined as:

```
letrec : bindmutrec expr expr -> bindmany expr expr -> expr.
```

writing, for our example above:

```
letrec (bmutrec (bcons (fun f => bcons (fun g => bnil
```

[4] Only strict higher-order hereditary Harrop formulas are allowed as predicates in Teyjus, whereas this is a case of a weak formula.

```
            [lam (fun x => g x), lam (fun x => f x)])))
       (bcons (fun f => bcons (fun g => bnil (app f (intconst 0)))))
```

and for the typing rule:

```
typeof (letrec Defs E') T' <-
   newvars_many Defs (fun xs => assume_many typeof xs TS ([ES Body]
     apply_many Defs xs ES, apply_many E' xs Body,
     map typeof ES TS, typeof Body T')).
```

Telescopes. Our last example of a binding structure will be telescopes, as used for example in the definition of contexts of dependently typed languages. Variable contexts $\Gamma = x_1 : t_1, \cdots, x_n : t_n$ of simply typed languages can be represented as a list of the types t_i together with a `bindmany` structure as seen above. In dependently typed languages, this representation idea breaks down, as the types t_i themselves might mention variables x_j coming before them. Instead, for each element of Γ we need to store the type of variable that it introduces and bind a fresh variable for it at the same time. Towards that effect, we use the following `listbindmany` data structure which mixes together a `list` with a `bindmany` structure:

```
listbindmany : type -> type -> type -> type.
lbcons : A -> (B -> listbindmany A B C) -> listbindmany A B C.
lbnil  : C -> listbindmany A B C.
```

Advanced example: Linear variables. [Readers are advised to skip this section when reading this document for the first time.]

It is possible to define substructural binding structures as well, in order to model variables that are supposed to be used linearly. Example usage cases would be unification variables in patterns. The way such binding structures work is by keeping track whether a variable has been used or not and making sure at the end of a linear scope that all variables have been used. Affine and relevant variables can be modeled similarly.

We have defined a type call `bindlinear` to model binding multiple linear variables at once:

```
bindlinear : type -> type -> type.
bindlinear : bindmany A B -> bindlinear A B.
```

By definition this type is exactly identical to the `bindmany` type; what differs is the `bindlinear_newvars` operation. The other basic operations `many_assume` and `bindlinear_apply` are identical to the `bindmany` case.

The `bindlinear_newvars` operation makes use of an extra predicate called `bindlinear_wf`. This is a predicate that is supposed to be called whenever a variable is encountered while inspecting the body of the binding structure. It will fail if the variable has been already used. After the inspection of the body, `bindlinear_newvars` will check that all bound variables have been used. These two checks together make sure that all bound variables are used exactly once inside the linear scope.

The implementation of `bindlinear_newvars` is as follows:

```
alloc_metas : list A -> list B -> prop.
alloc_metas L L' <- map (fun _ res => [U] eq res U) L L'.

assume_many' : list clause -> prop -> prop.
assume_many' Clauses Body <-
  foldr (fun clause curbody => eq (known clause curbody)) Clauses Body Q,
  Q.

bindlinear_newvars (bindlinear Bind) Q <-
```

```
  newvars_many Bind (fun vars => [Clauses]
   alloc_metas vars (VarIsUsed : list unit),
   map2 (fun v isUsed => eq (bindlinear_wf v <-
                                          isunif isUsed, eq isUsed unit))
           vars VarIsUsed Clauses,
   assume_many' Clauses (Q vars, map0 (fun isUsed => not(isUnif isUsed)) Unifs)).
```

There are two main tricks in this code: first, that we add assumptions which have premises of their own (similarly to top-level rules). Instead of being simple propositions of type `prop`, these are clauses (of type `clause`) – pairs of a goal proposition and a premise proposition. Internally they are constructed through the following constructor, though above we use the same syntax as for toplevel rules:

```
clause : prop -> prop -> clause.
```

Second, we make use of an extra-logical predicate, which inspects the state of the logic programming engine: the predicate `isunif` which is built into Makam. This checks whether its argument is an uninstantiated unification variable. The code above allocates a unification variable `IsUsed` for each linear variable; this unification variable is of course uninstantiated at first. When the first use of a linear variable `x` is encountered and the `bindlinear_wf` predicate is fired, this forces the respective `IsUsed` variable to become instantiated. If the same variable `x` is encountered again, then the call to `bindlinear_wf` will fail. At the end of the scope, we check to see that all `IsUsed` variables have been unified.

3.2 Dynamic type unification and generic predicates.

The polymorphic data structures and predicates that we have seen so far handle all type parameters uniformly – that is, they behave in exactly the same manner irrespectively of the concrete type that type variables are instantiated with. This behavior mirrors the way that polymorphism works in most modern functional languages. However, sometimes it is useful to be able to differentiate between different concrete types and write predicates that depend on their specific structure. Such is the case for predicates that perform convenient operations and also for writing generic traversal predicates; we will present more details in this section. Makam supports this alternate style of polymorphism through a refinement of its unification procedure; the result is that this feature is supported in an elegant way. Technically we say that Makam supports *ad-hoc polymorphism* in addition to the more traditional parametric polymorphism; compared to other λProlog implementations, Makam supports recursive polymorphism and therefore allows the usage patterns we will present.

Definitions by structural recursion. The key practical motivation for this feature has been to make it easy to define predicates that work by structural recursion, save for a few special, essential cases. Such operations are quite common in language definitions: for example, defining syntactic sugar that expands into core abstract syntax; operational semantics based on evaluation contexts; and also substitution operations that go beyond normal first-order syntactic substitution as in the λ-calculus.

Let us present a concrete example here, before we continue with technical details. Suppose we only want to consider the `lam` constructor as part of the core of the term language we have defined, and offer the `lammany` constructor as convenient syntactic sugar. We need to write a predicate that expands that syntactic sugar to yield a core expression:

```
expandsugar : expr -> expr -> prop.
expandsugar (lammany ES) E' <-
  newvars_many ES (fun xs => [Body]
  apply_many ES xs Body,
  foldl (fun x cur body' => [LamBody] eq cur (lam LamBody), eq (LamBody x) body')
        E' xs Body).
```

However, this predicate works only when `lammany` is the top-level node in our abstract syntax tree, and no other `lammany` instances occur within its body. We need to extend the `expandsugar` predicate so that it identifies all occurrences of `lammany` within any subterm and expands those accordingly, with a number of cases for each expression form in our language:

```
expandsugar (ifthenelse E1 E2 E3) (ifthenelse E1' E2' E3') <-
  expandsugar E1 E1', expandsugar E2 E2', expandsugar E3 E3'.
expandsugar (lam E) (lam E') <-
  (x:term -> expandsugar x x -> expandsugar (E x) (E' x)).
...
```

Definitions such as the above are not only quite tedious, especially for large languages, but essentially destroy the modularity that we have come to expect from our language definitions: for every new language construct that we add and its corresponding constructor that we add to the `expr` datatype, we need to add a new clause to the `expandsugar` predicate. Even if that clause is trivial and only performs structural recursion propagating its results, we need to remember to add it, otherwise queries to `expandsugar` will fail. The situation is further complicated when we consider the fact that we make use of auxiliary data structures such as `list`, `bindmany` etc. to define the expression forms of our language: though the auxiliary predicates that we have defined for them certainly allow us to perform structural recursion on their subparts, doing so is not entirely trivial.

Therefore defining operations such as `expandsugar` is another instance of the expression problem. Fortunately, Makam allows us to define such operations through a generic traversal predicate, called `structural`, thus regaining the lost modularity. `structural` can be defined within Makam thanks to dynamic typing unification – the technical feature we present in this section and whose details we will see shortly. Using this predicate we can define `expandsugar` in a modular fashion, roughly as follows[5]:

```
expandsugar_lammany : expr -> expr -> prop.
expandsugar : expr -> expr -> prop.

expandsugar_lammany (lammany ES) E' <-
  newvars_many ES (fun xs => [Body ExpandedBody]
  apply_many ES xs Body,
  expandsugar Body ExpandedBody,
  foldl (fun x cur body' => [LamBody] eq cur (lam LamBody), eq (LamBody x) body')
        E' xs ExpandedBody).

expandsugar E E' <-
  ifte (eq E (lammany ES))
       (expandsugar_lammany E E')
       (structural expandsugar E E').
```

The predicate `expandsugar` is now defined as follows: first, check to see whether we are in the particular case we need to expand, and if so, handle the expansion through a specialized predicate. Otherwise, handle the expression through structural recursion, using the same `expandsugar` predicate for the recursive calls. This definition will now work for all constructors of the `expr` data type and will continue to work even if new constructors are defined. It will also work with any auxiliary structures; it does not take any provisions as to what those structures look like and what kind of auxiliary predicates like `map` and `apply_many` are actually available. This is thanks to the fact that `structural` is defined in a fully generic way.

Let us start with the `dyn` datatype, which allows for dynamically typed terms inside the normally statically typed world of Makam. Its definition is as follows:

[5]Note that the type of `expandsugar` is not precise here; we will slightly refine its implementation later in this section.

```
dyn : type.
dyn : A -> dyn.
```

The dyn datatype has a single constructor named dyn as well (overloading type and constructor names is allowed in Makam). Note that contrary to prior definitions of constructors such as cons and bcons, where all type variables are evident in the target type of the constructor, the type variable A is not present in the result type of the constructor.

What the dyn constructor allows us is to assign the same type to well-typed terms of any type. This is particularly useful when used in combination with other data structures, such as lists. Using the dyn datatype we can create *heterogeneous* lists, where the elements do not have to be of the same type:

```
[ dyn 1 , dyn "hello!" , dyn [ 2, 3, 4 ] ]
```

We can perform normal operations such as map with this list. For example, we might write a predicate to convert a normal list into a heterogeneous one:

```
todynlist : list A -> list dyn -> prop.
todynlist L L' <- map (fun in out => eq out (dyn in)) L L'.
```

We can use this predicate in the inverse direction to get an homogeneous list out of an heterogeneous list where all elements are of the same type:

```
todynlist X [dyn 1, dyn 2, dyn 3] ?
>> X := [1, 2, 3]
```

However, if we use this with a truly heterogeneous list, the query will fail:

```
todynlist X [ dyn 1, dyn "hello!" ] ?
>> Impossible.
```

This reveals that even when using dyn or similar data structures, the hidden types are not lost, and that type safety is preserved.

We can also use map for heterogeneous lists with predicates that differentiate their behavior based on the exact type of each element. For example, consider the following (artificial) double predicate, which doubles integers, concatenates two copies of one string together, or doubles in the same manner all elements of a list:

```
double : dyn -> dyn -> prop.
double (dyn (X : int)) (dyn Y) <- mult X 2 Y.
double (dyn (X : string)) (dyn Y) <- append (X : string) X Y.
double (dyn (X : list A)) (dyn Y) <- todynlist X Z, map double Z Z', todynlist Y Z'.

map double [ dyn 1 , dyn "hello!" , dyn [ 2, 3, 4 ] ] ?
>> [ dyn 2 , dyn "hello!hello!" , dyn [ 4, 6, 8 ] ]
```

Dynamic typing is useful for defining data types other than dyn. For example, the bindmany data type earlier can be refined so that it can bind many terms of any type rather than of a specific one. Such a data type can be useful to define uncurried functions of type and term variables, as is needed in a closure-converted calculus for System F. The definition of the new bindany data structure is quite similar to bindmany:

```
bindany : type -> type.
bacons : (A -> bindany B) -> bindany B.
banil  : B -> bindany B.
```

The auxiliary predicate bindany_newvars makes use of the dyn datatype in order to return a list of the new free variables; the other predicates that are to be used with bindany need similar refinements to their types:

```
bindany_newvars : bindany B -> (list dyn -> prop) -> prop.
```

```
bindany_apply   : bindany B -> list dyn -> B -> prop.
assume_many_dyn : (dyn -> dyn -> prop) -> list dyn -> list dyn -> prop.
```

Predicates using type unification. Predicates that have specialized behavior according to the type of their arguments are especially useful to program various convenience and auxiliary predicates. For example, we might want to offer a way to convert terms of iterated function type (of the form `A -> (A -> ... -> B)`) into the `bindmany` type, so as to allow the cleaner syntax `fun x y z => ...x....y...z..` instead of `bcons (fun x => bcons (fun y ...))` for multiple bindings. This can be defined as follows:

```
fun_to_bindmany : C -> bindmany A B -> prop.
fun_to_bindmany (X : A -> C) (bcons Rest) <-
  (x:A -> fun_to_bindmany (X x) (Rest x)).
fun_to_bindmany (X : B) (bnil X).
```

Another example is the definition of the `headargs` predicate, which deconstructs a term of the form `hd args` into its constituents – its head and its arguments. Though we have not formally described the syntax of Makam terms so far, it suffices to say that they are of four kinds: neutral terms of the form `hd args`, unification variables of the form `X args`, functions of the form `fun a b ... => t'` and constants like strings and integers. The `headargs` predicate is useful in order to deconstruct and reconstruct terms of the first form of an arbitrary, unknown, type. This predicate is crucial in implementing the generic structural recursion traversal predicate. `headargs` is offered as an external predicate by the Makam implementation for efficiency reasons, but a partial implementation within Makam is possible:

```
headargs : A -> B -> list dyn -> prop.

headargs (X : A) HD ARGS <-
  newvmeta (fun (hd : A) =>
       eq X hd, eq HD hd, eq ARGS []).
headargs (X : A) HD ARGS <-
    newvmeta (fun (hd : A1 -> A) =>
       eq X (hd ARG1), eq HD hd, eq ARGS [dyn ARG1]).
headargs (X : A) HD ARGS <-
    newvmeta (fun (hd : A1 -> A2 -> A) =>
       eq X (hd ARG1 ARG2), eq HD hd, eq ARGS [dyn ARG1, dyn ARG2]).
...
```

Note the use of the `newvmeta` predicate which we have not presented before. This predicate introduces a special kind of metavariable that behaves differently during higher-order unification: instead of following the procedure of higher-order pattern matching as we have seen earlier, it is to be unified using the IMITATION phase of higher-order unification solely. As a result it is only unified against free variables. In this way it is similar to the parameter metavariables of Beluga [Pientka and Dunfield, 2008]. Each rule in `headargs` corresponds to a specific number of arguments that the head of a potential form `hd args` expects; in every case, we try to match the imitation metavariable `hd` against the head, and if so, we unify the results of the predicate `HD` and `ARGS` accordingly.

We are now in a position to define the `structural` predicate we presented earlier as the motivation for dynamic typing. What `structural` does is now quite simple to state: it deconstructs terms using `headargs`, applies the recursive call to the arguments, and reconstructs the modified term using the same predicate in the inverse direction. It needs to take special provisions for built-in types `string` and `int` that are populated by constants rather than normal term formers. More importantly, it needs to handle another special type: the function type `A -> B`. The recursive argument it expects needs to work over arbitrary types, so as to be fully generic: for example, we might want to define a predicate by structural recursion that expands syntactic

sugar both at the expression level and at the type level of the object language. A sketch of the definition of `structural` is as follows:

```
structural : (dyn -> dyn -> prop) -> dyn -> dyn -> prop.
structural Rec (dyn (X : string)) (dyn (X : string)).
structural Rec (dyn (X : int))    (dyn (X : int)).
structural Rec (dyn (X : A -> B)) (dyn (Y : A -> B)) <-
  (x:A -> Rec (X x) (Y x)).
structural Rec (dyn (HdArgs : A)) (dyn (HdArgs' : A)) <-
  headargs HdArgs Hd Args,
  map Rec Args Args',
  headargs HdArgs' Hd Args'.
```

The actual definition also handles the case of metavariables, by deferring the structural recursion to when the metavariables are instantiated, and also the inverse direction case where the output of the structural recursion is known and the input is unknown. Still, these details are beyond the scope of this section; we refer the user to the actual implementation of `structural`.

With `structural` in hand, it is now easy to give the precise implementation of the `expandsugar` predicate from the introduction of this section.

```
expandsugar : dyn -> dyn -> prop.
expandsugar E E' <-
  ifte (eq (dyn E) (dyn (lammany ES)))
       (expandsugar_lammany (lammany ES) E'concrete, eq E' (dyn E'concrete))
       (structural expandsugar E E').
```

Making this predicate type-generic and using `structural` allows `expandsugar` to work in the presence of auxiliary data structures like `list` and `bindmany`. Terms of these types can be deconstructed and reconstructed normally through `headargs`, and making `expandsugar` type-generic allows us to perform structural recursion deep within their structure as well.

The above definition reveals that one needs to be quite careful with dynamic typing – the following seemingly equivalent and simpler definition will not work as expected:

```
expandsugar : dyn -> dyn -> prop.
expandsugar (dyn E) (dyn E') <-
  ifte (eq E (lammany ES))
       (expandsugar_lammany (lammany ES) E')
       (structural expandsugar E E').
```

The reason is that typing fixes the type of `E` and `E'` to be of `expr` type, so structural recursion within other types will fail. So annotating terms with `dyn` is no panacea – one needs to be careful with the precise point when unification will happen.

3.3 Staging: Meta-programming meta-programs.

The fact that predicates are themselves terms of the metalanguage has far-reaching consequences. We can define predicates over predicates – predicates that compute new predicates – and use the computed predicates as new goals. Similarly, top-level commands such as defining a new term and adding a clause to a predicate can be encoded as terms. These commands themselves can be computed through normal Makam predicates and their result can be evaluated so as to have the same effect as if the commands were written by the user in the program text.

Essentially Makam allows us to *compute and evaluate metalanguage programs within the metalanguage itself*, without requiring any complicated staging or meta-programming extensions. This feature comes thanks to to encoding predicates and commands as normal Makam terms and also due to the fact that Makam is interpreted, so statically present and dynamically computed terms are not distinguished. This feature of our metalanguage is unique compared to existing implementations of λProlog, which rely on a compilation model and disallow queries based on dynamically computed predicates.

The staging feature of Makam is relatively new, so we have not explored its consequences as much as the features we have presented so far. Still, we believe it will be of utmost importance in allowing users to develop extensions to the metalanguage within the metalanguage itself – for example, a way to add mode declarations to predicates, a feature not supported by the core Makam language. Also, staging is very useful for optimization purposes. We have used it, for example, to create a parser generator for object languages: given a grammar specification, a parser generation predicate produces a number of Makam clauses that implement the grammar, therefore avoiding any additional interpretation overheads. In this way, staging allows us in many cases to reduce the impact of any possible shortcomings of the metalanguage or unanticipated features and programming patterns that become useful.

Because the staging feature is still preliminary and under exploration we will not present examples to the same level of detail as we have done so far; we will only give a sketch of our motivation behind them and of the way they work.

Object-level definitions. Consider the case of encoding top-level definitions of our object language, of the form:

```
let factorial = letrec f n =
                  if n = 0 then 1 else n * (f (n-1))
                in f
```

One way to add them to our model would be to explicitly add a new clause to the `typeof` predicate and to an (assumed) `eval` predicate that implements an interpreter for the language based on its operational semantics:

```
factorial : expr.
typeof factorial (arrow tint tint).
eval factorial (letrec (bcons (fun f => ...)) (bcons (fun f => bnil f))).
```

This is a suboptimal solution. First of all, it is risky: the user might give the wrong type for the new `factorial` constant, thus destroying type safety. Second, it does not offer an explicit Makam-level term to correspond to the `let` definition of the object language.

Staging allows us to do better. First, we can define a type for top-level definitions of the object language and a constructor for the `let` definition, as follows:

```
def : type.
letdef : expr -> expr -> def.
```

We can now create a predicate that will check definitions for well-formedness and generate the required assumptions – either to be added to the context, or as top-level clauses of the metalanguage:

```
wfdef : def -> list clause -> prop.
wfdef (letdef Const E) [ typeof Const T,
                         eval   Const E ] <-
  typeof E T.
```

Now the query `wfdef (letdef factorial (letrec (bcons ...))) X` will give us the right assumptions after consulting the type-checker we have already implemented, thus eliminating the risk we noted above:

```
wfdef (letdef factorial (letrec (bcons ...))) X ?
>> X := [ typeof factorial (arrow tint tint) ,
          eval factorial (letrec (bcons ...)) ]
```

We can turn these assumptions into top-level Makam *commands* of the built-in `cmd` data type. This type is defined through a number of constructors, which are just taken into special account by the Makam implementation, similarly to the `prop` and `clause` types:

```
cmd : type.
cmd_newclause : clause -> cmd.
cmd_newterm   : string -> A -> cmd.
cmd_many      : list cmd -> cmd.
```

This commands work as follows: `cmd_newclause` adds a new clause to the rule database of Makam; `cmd_newterm S T` defines a new term named `S`, whose type is determined through the type of the second argument `T` – the exact structure of the `T` argument is unimportant. The `cmd_many` command allows us to sequence a number of commands.

With these in mind, we can implement a predicate that turns assumptions into top-level commands, as well as a predicate that combines it with definition well-formedness, as follows:

```
clauses_to_cmd : list clause -> cmd -> prop.
clauses_to_cmd Clauses (cmd_many Cmds) <-
  map (fun clause cmd => eq cmd (cmd_newclause clause)) Clauses Cmds.

def_cmd : def -> cmd -> prop.
def_cmd Def Cmd <- wfdef Def Assumptions, clauses_to_cmd Assumptions Cmd.
```

Makam has a special top-level command, so as to be able to execute the commands that are generated through predicates such as `def_cmd`: the staging command `cmd_stage`, of type:

```
cmd_stage : (cmd -> prop) -> prop.
```

It can be invoked through the following syntax:

```
‘( Predicate ).
```

where `Predicate` must be a single-argument predicate that produces a command. For example, our definition of `factorial` can now be done as follows:

```
factorial : expr.
‘( def_cmd (letdef factorial (letrec (bcons ...))) ).
```

This will have exactly the same effect as the user-provided commands in the beginning of this section. The difference is that the definition of `factorial` is type-checked, thanks to the implementation of `wfdef`.

There still is one shortcoming: we declare the constant separately from its definition; and arbitrary expressions can be “defined” through `let`, leading to unexpected behavior. These can be fixed by declaring and defining the new constant in one step:

```
letdef : string -> expr -> def.
def_cmd : def -> cmd -> prop.
def_cmd (letdef Const E) (cmd_many [ cmd_newterm Const (X : expr) ,
                                     cmd_stage (fun cmd =>
                                      lookup Const ConstTerm,
                                      eq cmd (cmd_many [
                                       cmd_newclause (typeof ConstTerm T),
```

```
                                        cmd_newclause (eval   ConstTerm E) ])) ]).

‘( do_def "factorial" (letrec (bcons (fun f ...))) ).
```

Using an extra indirection step through `cmd_stage` is necessary for the above code to type-check. The Makam term corresponding to the `factorial` constant is not available before the `cmd_newterm` command is actually run; we therefore have no static way to refer to it. However, once it is defined, we can get the actually defined term based on its name through the external `lookup` predicate, and use that to form the rest of the necessary commands.

Language extension: Otherwise clauses. Sometimes we want to add a clause to a predicate which is only applicable if all other clauses fail. This is useful for example for implementing the conversion rule without switching to an algorithmic formulation of the typing rules. We can manually force this behavior by splitting up the predicate into two and having a top-level predicate that chooses between the two accordingly:

```
typeof : expr -> typ -> prop.
typeof_main : expr -> typ -> prop.
typeof_otherwise : expr -> typ -> prop.

typeof E T <- ifte (typeof_main E T) success (typeof_otherwise E T).

typeof_main (intconst _) tint.
typeof_main (lam E) (arrow T1 T2) <-
  (x:expr -> typeof_main x T1 -> typeof (E x) T2).

typeof_otherwise E T <- typeof_main E T', conversion T T'.
```

However, there are a number of caveats: the user needs to remember to add local clauses (as needed, for example, for the `lam` case) to the `typeof_main` predicate; also, if the user has not taken a special provision to define the `typeof` predicate in this style, they will need to backpatch all their code to support it when an otherwise clause becomes necessary.

What we could do instead is process the definitions and clauses that the user makes so as to transform them accordingly based on the above pattern. In this way we can generally support otherwise clauses even though the core Makam language does not have a special provision for them. We ideally want to write code as follows:

```
typeof : expr -> typ -> prop.

typeof (intconst _) tint.
typeof (lam E) (arrow T1 T2) <-
  (x:expr -> typeof x T1 -> typeof (E x) T2).
otherwise (typeof E T) <-
  typeof_main E T', conversion T T'.
```

Though we do not give the details here, it is possible to implement a predicate `otherwise_transform` of type:

```
otherwise_transform : cmd -> cmd -> prop.
```

This predicate inspects the input command and makes any necessary additions – for example, if the command corresponds to the definition of a new predicate `P`, it defines the two auxiliary predicates `P_main` and `P_otherwise` as well and generates the single clause for the `P` predicate as above. Also, if the command corresponds to a new clause added to `P`, it does the necessary changes so that it refers to the `P_main` clause

instead; if it corresponds to a clause for the placeholder predicate `otherwise P`, it changes it to `P_otherwise`, etc.

We are experimenting with a feature in Makam called command preprocessors: every command that would normally be issued is first processed through a predicate like `otherwise_transform` which potentially transforms the command before staging the result. Essentially, enabling a command preprocessor like `otherwise_transform` makes all the following top-level commands `Cmd1, Cmd2, ...` correspond to:

```
‘( otherwise_transform Cmd1 ).
‘( otherwise_transform Cmd2 ).
...
```

However, this feature is still under development. We trust that it will be very useful in the future for implementing new extensions to the metalanguage.

Generating parsers and pretty-printers. So far our most important implementation that relies on staging is a parsing and pretty-printing methodology for object languages, based on Parsing Expression Grammars (PEGs) as presented by Ford [2004]. A description of this implementation is far beyond the scope of this section; we will present it at a very high level here.

Specifying a grammar relies on PEGs of type `peg A`: an expression that consumes part of a string and produces a term of type `A`. This type is inhabited by the base PEG combinators, like concatenation, deterministic choice and lookahead; further combinators can be defined in terms of the base ones. It also includes a semantic action combinator typed as `peg A -> (A -> B -> prop) -> peg B`. We implement a compiler for PEGs in Makam as a predicate of the type:

```
compile_parser : peg A -> (string -> option (tuple string A) -> prop) -> prop.
```

This compiler yields a predicate which, given a string, either returns the parsed term and the unconsumed suffix of the string, or fails. Therefore, this predicate functions as a *parser generator* in the style of bison and yacc – with the important difference that it is written within the same metalanguage and preserves typing. The presence of recursion in the grammar is handled through indirection, by adding the propositions yielded by the compiler as top-level clauses. As of now, we do not support left recursion, but we are investigating ways to support it in the future.

Due to the bidirectionality of logic programming, one would expect that the parser yielded by `compile_parser` could be used in the inverse direction as a pretty-printer. This option, unfortunately, is not effectively possible: the reason is that the amount of backtracking in the inverse direction is prohibitive for any practical purposes. However, we can define a pretty-printer generator instead, which yields predicates that have better computational behavior for that purpose:

```
compile_printer : peg A -> (tuple string A -> option string -> prop) -> prop.
```

There are two key challenges in our methodology: First, making sure that the semantic actions that we use are invertible, so that they can be used both for parsing and for pretty-printing. We currently handle this through another common meta-programming techniques: embedding a domain-specific language within our metalanguage (having our metalanguage function as the host language). We have defined a domain-specific language for writing the semantic actions in a functional style. This language is translated within Makam into rules that either function in the forward direction, when the input of functions is known, or in the backwards direction, when only their output is available.

The other main challenge is how to go from concrete syntax into higher-order abstract syntax, especially for non-trivial binding structures like mutual recursion. In that case, if we are to produce higher-order abstract syntax directly through our parsing rules, we need to know all the names bound in a mutual recursion before we can parse any definition; the naïve way to handle this would need a very expensive lookahead operation.

What we do instead is parse concrete syntax into an intermediate first-order representation with concrete names; when parsing is complete, we force variable definition and lookup on this representation, yielding normal higher-order abstract syntax terms. These are done without any per-constructor definitions by the user, preserving typing and guaranteeing that only well-bound terms will be yielded.

Our parsing and pretty-printing methodology still requires some refinement and performance tune-up before it can be used practically for large terms; once these are done as part of our future work on Makam, we will present the full details of our methodology.

3.4 Reflective predicates.

Reflection, the ability to view into the structure and evaluation state of running Makam programs, could be viewed as the dual to staging: whereas staging allows us to generate code programmatically that will be run in the future, reflection allows us to gain information about the code that has already been executed. Makam supports partial reflection through a number of external predicates; we have in fact seen two of them already:

`isunif : A -> prop`, which tells us whether its argument is an uninstantiated meta-variable or not.

`lookup : string -> A -> prop`, which looks up a constant by name and returns the constant itself.

These predicates are *extra-logical*: if we read the propositions and rules of Makam as corresponding to logical formulas in a suitable higher-order logic, and the Makam interpreter as performing proof search for those formulas, predicates like these destroy the correspondence as they reveal implementation details of the proof search process. However, we do not intend to view Makam programs in this light, contrary to most logic programming languages, taking a more 'operational' stance towards the nature of the language. If correspondence to propositions in a specific logic is needed – e.g. if we want to prove metatheoretic results about an object language $\mathcal{O}$ that we have encoded – then that logic itself should be encoded within Makam as another object language $\mathcal{L}$; the Makam definitions of a type system etc. for $\mathcal{O}$ should be then translated into $\mathcal{L}$. This translation itself can be done through an appropriate meta-level predicate `makam_to_L` that consumes a list of Makam clauses and produces a list of logical definitions.

An example where extra-logical predicates are important in achieving conciseness is Hindley-Milner generalization as in ML. We will need two new external predicates:

`getunif : A -> B -> prop` which returns the first uninstantiated meta-variable in the term of type `A` that is of type `B`, and

`absunif : A -> B -> (B -> A) -> prop`, for example `absunif E X F` which abstracts over all occurrences of the uninstantiated meta-variable `X` within `E`, yielding a function `F`. It guarantees that for the returned function `F`, the substitution `F X` will be equal to `E`.

Hindley-Milner generalization for let bindings works as follows. For a definition `let x = e in e'`, we perform type checking in the definition part `e`, performing type unification as needed; any undetermined types are quantified over to yield the polymorphic type of `x` within `e'`. Being based on unification as well, the typing procedure for our object language as encoded within Makam will proceed equivalently, yielding the same type with undetermined metavariables at the same places. For example, querying for the type of the identity function yields:

```
typeof (lam (fun x => x)) T ?
>> T := tarrow T1 T1
```

To complete the process, we just need to quantify over the undetermined type metavariables. Using `getunif` and `absunif` we can encode this in Makam as follows:

```
typgen : typ -> typ -> prop.
typgen T T <- not(getunif T (X : typ)).
typgen T (tpi F') <-
```

```
  getunif T (X : typ), absunif T X F,
  (a:typ -> typgen (F a) (F' a)).

typeof (let E E') T <-
  typeof E T, typgen T Tpoly,
  (x:expr -> typeof x Tpoly -> typeof (E' x) T).
```

In this way, the following query returns the right type:

```
typeof (let (lam (fun x => x)) (fun id => id)) T ?
>> T := tpi (fun a => tarrow a a)
```

Many more reflective predicates might prove useful in the future; for example, it might be useful to have a way to look up all clauses of a given predicate, or all constructors of a given data type. We have not yet attempted to provide a complete set of reflective predicates (for any notion of "complete") but we are adding them as needed.

4. Summary

We have presented the design of Makam, a metalanguage that enables the prototyping of programming languages, type systems and transformations between languages, through concise declarative rules. This design removes various significant challenges traditionally associated with language prototyping. It is based on a small conceptual core that refines the λProlog language, resulting in a surprisingly rich and expressive formalism. We have a beta-stage implementation of our metalanguage; our experiments with it indicate that it is able to handle realistic languages with sophisticated type systems. The object programs that it can efficiently handle are relatively small, but of sufficient size to do the kinds of experiments that are important during language and type system design research. We will soon be releasing Makam as an open-source OPAM package. Our continued development effort will focus on practical features like better debugging methodologies for object-level predicates; a complete parsing and pretty-printing methodology; and on further exploration of the formalism guided by examples from the past few decades of programming languages research. We are confident that Makam will prove useful for programming language experimentation and prototyping and urge PL researchers to try it out and use it for their explorations, as well as to help us along in further developing Makam itself.

References

G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In *Proceedings OF JICSLP'96*. The MIT press, 1998.

B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, volume 39, pages 111–122. ACM, 2004.

O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 192–203, New York, NY, USA, 2005. ACM.

D. Miller and G. Nadathur. An overview of λprolog. In *Proc. of the 5th Int. Conf. on Logic Programming*, 1988.

G. Nadathur and D. J. Mitchell. System description: Teyjusa compiler and abstract machine based implementation of λprolog. In *Automated DeductionCADE-16*, pages 287–291. Springer, 1999.

B. Pientka and J. Dunfield. Programming with proofs and explicit contexts. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and Practice of Declarative Programming*, pages 163–173. ACM New York, NY, USA, 2008.

P. Wadler. The expression problem. *Java-genericity mailing list*, 1998.