# Functional programming with $\lambda$-tree syntax: Draft

Ulysse Gérard and Dale Miller
Inria Saclay & LIX, Ecole Polytechnique

June 7, 2018

**Abstract**

We present the design of a new functional programming language, MLTS, that uses the $\lambda$-*tree* syntax approach to encoding bindings that appear within data structures. In this setting, bindings never become free nor escape their scope: instead, binders in data structures are permitted to *move* into binders within programs. The design of MLTS—whose concrete syntax is based on that of OCaml—includes additional sites within programs that directly support the movement of bindings. We illustrate the features of MLTS by presenting several collections of examples. We also present a typing discipline that naturally extends the typing of OCaml programs. In order to formally define the language's semantics, we present an abstract syntax for MLTS and a natural semantics for its evaluation. We shall view such natural semantics as a logical theory with a rich logic that includes both nominal abstraction and the $\nabla$-quantifier: as a result, the actual natural semantic specification can be given a succinct and elegant presentation.

## 1 Introduction

Even from the earliest days of high-level programming, functional programming languages were used to build systems that manipulated the syntax of various programming languages and logics. For example, Lisp was a common language for building theorem provers, interpreters, compilers, parsers, and the ML programming language was designed as a "meta-language" for a proof checker [?]. While these various tasks involved the manipulation of syntax, none of these earliest functional programming languages provided support for a key feature of almost all programming languages and logics: variable binding. Of course, bindings in syntactic expressions have been given a range of different treatments within the functional programming setting.

Common approaches are to implement bindings by explicitly using variable names or, in a more abstract way, by using de Bruin's nameless dummies [?]. Since such techniques are quite complex to get right and since bindings are so pervasive, a great deal of energy has gone into making libraries of procedures that can help dealing with binders: for example, there is the *locally nameless* approach [?, ?, ?] and the *parametric higher-order abstract syntax* approach [?].

Extending a functional programming language with new features has also been considered: for example, there have been the FreshML [?, ?] and C$\alpha$ML [?] extensions to ML-style functional programming languages. Also, entirely new functional programming languages, such as the dependently typed Beluga [?] language, have been designed and implemented with the goal to support bindings in syntax. In the domain of logic programming and theorem prover design, several designs and implemented systems exist that incorporate approaches to binding: such systems include Isabelle's generic reasoning core [?], $\lambda$Prolog [?, ?], Qu-Prolog [?], Twelf [?], $\alpha$Prolog [?], the Minlog prover [?], and the Abella theorem prover [?].

In this paper we present MLTS, a new language extending (the core of) the OCaml programming language that incorporates the $\lambda$-*tree syntax* approach to encoding the abstract syntax of data structures containing binders. Briefly, we can define the $\lambda$-tree syntax approach to syntax as following the three tenets: (1) Syntax is encoded as simply typed $\lambda$-terms in which the primitive types are identified with syntactic categories. (2) Equality of syntax must include $\alpha$, $\beta_0$, and $\eta$ conversion. (These conversion steps are describe more in Section ??.) (3) Bound variables never become free: instead, their binding scope can move. This latter tenet introduces the most characteristic aspect of $\lambda$-tree syntax which is often called *binder mobility.* MLTS is, in fact, an acronym for *mobility* and $\lambda$-*tree syntax.*

## 2 The new features of MLTS

We shall assume that the reader is familiar with basic conventions of OCaml [?] (many are shared with most ML-like programming languages). MLTS contains the following four new language features.

1. Datatypes can be extended to contain new *nominal* constants and the `(new X in body)` program phrase provides a binding that declares that the nominal `X` is new within the lexical scope given by `body`.

2. The *backslash* (`\` as an infix symbol that associates to the right) is used to form an abstraction of a nominal over its scope. For example, `(X\body)` is a syntactic expression that hides the nominal `X` in the scope `body`. Thus the backslash *introduces* an abstraction. The `@`, conversely, *eliminates* an abstraction: for example, the expression `((X\body) @ Y)` denotes the result of substituting the abstracted nominal `X` with the nominal `Y` in `body`. Expressions involving `@` are restricted to be of the form `(m @ X1 ... Xj)` where `m` is a term (including a pattern matching variable) and `X1`, . . ., `Xj` are nominals bound within the scope of the pattern binding on `m`.

3. A new typing constructor `=>` is used to type bindings within term structures. This constructor is an addition to the already familiar constructor `->` used for typing functional expressions.

4. Rules within match-expression can also contain the `(nab X in rule)` binder: in the scope of this binder, the symbol `X` can match existing nominals introduced by the `new` binder and the `\` operator. Note that `X` is bound over the entire rule (including both the left and right-side of the rule).

All three bindings expressions—`(X\body)`, `(new X in body)` and `(nab X in rule)`—are subject to alphabetic renaming of bound variables, just as the names of variables bound in `let` declarations and function definitions. Since nominals are best thought of as constructors, we follow the OCaml convention of capitalizing their names. We are assuming that in all parts of MLTS, the names of nominals (of bound variables in general) are not available to programs since $\alpha$-conversion (the alphabetic change of bound variables) is always applicable. Thus, compilers are free to implement nominals in any number of ways, even ways in which they do not have, say, print names.

The restriction on the structure of expressions of the form `(m @ X1 ... Xj)` are essentially the same as those required by *higher-order pattern unification* [?]: as a result, pattern matching in this setting is a simple generalization of usual first-order pattern matching. We note that the expression `(X\ r @ X)` is interchangeable with the simple expression `r`: that is, when `r` is of `=>` type, the $\eta$-rule is available in this way.

We now present several sets of examples of MLTS programs in the next sections. We shall hope that the informal semantics given above plus the simplicity of the examples will give a working understanding of the semantics of MLTS. We delay the formal definition of the operational semantics of MLTS until Section ??.

## 3 MLTS examples: the untyped $\lambda$-calculus

The untyped $\lambda$-terms can be defined in MLTS as the following datatype:

```
type tm =
     | App of tm * tm
     | Abs of tm => tm ;;
```

The use of the `=>` type constructor here indicates that the argument of `Abs` is an *abstraction* of a `tm` over a `tm`. Just as the type `tm` denotes a syntactic category of untyped $\lambda$-terms, the type `tm => tm` denotes the syntactic category of terms abstracted over such terms.

The following MLTS program computes the size of an untyped $\lambda$-term.

```
let rec size term =
        match term with
        | App(n, m)   -> 1 + (size n) + (size m)
        | Abs(r)      -> 1 + (new X in size (r @ X))
        | nab X in X -> 1;;
```

```
let rec vacp1 t = match t with
    | Abs(X\X)                      -> false
    | nab Y in Abs(X\Y)             -> true
    | Abs(X\ App(m @ X, n @ X)) -> (vacp1 (Abs m)) && (vacp1 (Abs n))
    | Abs(X\ Abs(Y\ r @ X Y))    -> new Y in vacp1(Abs(X\ r @ X Y))
    | t                             -> false ;;

let vacp2 t = match t with
    | Abs(r) -> new X in
                  let rec aux term = match term with
                         | X          -> false
                         | nab Y in Y -> true
                         | App(m, n)  -> (aux m) && (aux n)
                         | Abs(u)     -> new Y in aux (u @ Y)
                   in aux (r @ X)
    | t      -> false;;

let vacp3 t = match t with
    | Abs(X\s)   -> true
    | t          -> false ;;
```

Figure 1: Three implementations of the function that determines if its argument is a vacuous $\lambda$-term.

For example, `(size (App(Abs(X\X),Abs(X\X))))` evaluates to 5. In the second match rule, the match-variable `r` will be bound to an expression built using the backslash. On the right of that rule, `r` is applied to a single argument which is a newly provided constructor of type `tm`. The third match rule contains the `nab` binder that allows the token `X` to match any nominal. (Note that the three match rules used to define `size` could have been listed in any order and that the third match rule could have been written more simply as `| x -> 1`.) Subsequent calls to the `size` function can be seen as evolving as follows.

```
size (Abs (X\ (Abs (Y\ (App(X,Y))))));;
1 + new X in (size (Abs (Y\ (App(X,Y)))));;
1 + new X in 1 + new Y in (size (App(X,Y)));;
1 + new X in 1 + new Y in 1 + (size X) + (size Y);;
1 + new X in 1 + new Y in 1 + 1 + 1;;
```

The first call to `size` will binds the pattern variable `r` to `(X\ (Abs (Y\ (App(X,Y)))))`. It is important to note that the names of bound variables within MLTS programs and data structures are fictions. Here, we chose convenient names for the reading of these structures: internally, such names can be compiled into, say, nameless dummies.

As a second example, consider a program that returns the boolean `true` if and only if its argument is a $\lambda$-abstraction for which the bound variable is vacuous in its scope; otherwise, it returns `false`. Figure **??** contains three implementations of this boolean-valued function. Note that, in the implementation of `vacp2`, once the outermost match rule determines that the argument is a $\lambda$-abstraction, a new nominal is created and used to play the role of the $\lambda$-abstracted variable. The internal `aux` function is then defined to search the body of that $\lambda$-abstraction in search of that new nominal. The third implementation, `vacp3`, is not (overtly) recursive since the entire effort of checking for the vacuous binding can be done during pattern matching. The first match rule of this third implementation is essentially asking the question: is there an instantiation for the (pattern) variable $s$ so that the equation $Abs(\lambda x.s)$ equals $t$? This question can be posed as asking if the logical formula $\exists s.(Abs(\lambda x.s)) = t$ can be proved. In this latter form, it should be clear that since substitution is intended as a logical operation, the result of substituting for $s$ never allows for variable capture. Hence, every instance of the existential quantifier yields an equation with a left-hand side that is a vacuous abstraction. Of course, for this kind of pattern matching to work as described, determining this match requires a recursive analysis of the term $t$.

Figure **??** presents a datatype for the untyped $\lambda$-calculus in De Bruijn's style nameless dummies [?] as well as the functions that can convert between that syntax and the one with explicit bindings. The auxiliary functions `nth` and `index` take a list of nominals as their second argument: `nth` takes also an integer `n` and returns the $n^{th}$ nominal in that list while `index` takes a nominal and returns its ordinal

```
type deb =
     | Dapp of deb * deb
     | Dabs of deb
     | Dvar of int;;

let rec nth n l = match (n, l) with
  | (0, x::k) -> x
  | (c, x::k) -> nth (c - 1) k;;

let index x l =
    let rec aux c x k = match (x, k) with
          | nab X   in (X, X::(l @ X))   -> c
          | nab X Y in (X, Y::(l @ X Y)) -> aux (c + 1) x (l @ X Y)
     in aux 0 x l;;

let rec trans prefix term = match term with
  | App(m, n)  -> Dapp(trans prefix m, trans prefix n)
  | Abs r      -> new X in Dabs(trans (X::prefix) (r @ X))
  | nab Y in Y -> Dvar (index Y prefix);;

let rec dtrans prefix term = match term with
  | Dapp(m, n) -> App(dtrans prefix m, dtrans prefix n)
  | Dabs r     -> Abs(X\ dtrans (X::prefix) r)
  | Dvar c     -> nth c prefix;;
```

Figure 2: De Bruijn's nameless dummy syntax and its conversions with type `tm`.

position in that list. For example, the value of

```
trans [] (Abs(X\ Abs(Y\ Abs(Z\ App(X, Abs(W\Z))))));;
```

is the term `DAbs(DAbs(DAbs(DApp(Dvar 2, DAbs(Dvar 1)))))` of type `deb`. If `dtrans []` is applied to this second term, the former term is returned (modulo $\alpha$-convergence, of course).

Figure ?? defines the function `(subst t x u)` that computes the result of substituting $u$ for $x$ in $t$ (often written as $[u/x]t$). This function is then used by the second function for computing the $\beta$-normal form of a given term of type `tm`. This figure also contains the Church numeral for 2 and operations for addition and multiplication on Church numerals. In the resulting evaluation context, the values computed by both `(beta (App(App(plus,two),two)))` and `(beta (App(App(times,two),two)))` both are the Church numeral for 4.

Note the first two rules in the `subst` function in Figure ??. The first rule, namely,

```
 | nab X   in (X,X) -> u
```

can match an argument that is pair of *identical* nominals while the second rule, namely,

```
 | nab X Y in (X,Y) -> Y
```

can match an argument that is pair of *different* nominals. The order of these two rules in this match expression could be reversed without affecting computation. Expressions whose concrete syntax have nested binders using the same name are disambiguated by the parser by linking the named variable with the closest binder. Thus, the concrete syntax `(Abs(X\ Abs(X\ X)))` is parsed as a term $\alpha$-equivalent to `(Abs(Y\ Abs(X\ X)))`. Similarly, the expression `(let n = 2 in let n = 3 in n)` is parsed as an expression $\alpha$-equivalent to `(let m = 2 in let n = 3 in n)`: this expression has value `3`.

For a final, simple example of computing on the untyped $\lambda$-calculus, consider introducing a mirror version of `tm`, as is done in the top of Figure ??, and writing the function that constructs the mirror term in `tm'` from an input term `tm`. While this computation is straightforward, it is achieved by adding a context (an association list) as an extra argument: that context maintains the association of bound variables of type `tm` and those of type `tm'`. The value of `id [] (Abs(X\ Abs(Y\ App(X,Y))))` is `(Abs'(X\ Abs'(Y\ App'(X,Y))))` (the types of `X` and `Y` in these two expressions are, of course, different).

```
let rec subst t x u = match (x,t) with
  | nab X   in (X,X) -> u
  | nab X Y in (X,Y) -> Y
  | (x, Abs r)       -> Abs(Y\ subst (r @ Y) x u)
  | (x, App(m,n))    -> App(subst m x u, subst n x u);;

let rec beta t = match t with
  | Abs r       -> Abs(Y\ beta (r @ Y))
  | nab X in X -> X
  | App(m, n)   ->
    let m = beta m in let n = beta n in
    begin
        match m with
        | Abs r -> new X in beta (subst (r @ X) X n)
        | w -> App(m, n)
    end ;;

let two   = Abs(F\ Abs(X\ App(F, App(F, X))));;
let plus  = Abs(M\ Abs(N\ Abs(F\ Abs(X\
                            App(App(M, F), App(App(N, F),X))))));;
let times = Abs(M\ Abs(N\ Abs(F\ Abs(X\
                            App(App(M, App(N, F)), X)))));;
```

Figure 3: The function for computing the substitution $[t/x]u$ and the (partial) function that returns the $\beta$-normal form of its argument.

```
let rec assoc x alist = match alist with
  | ((u,y)::alst) -> if (u = x) then y else (assoc x alst);;

type tm' =
     | App' of tm' * tm'
     | Abs' of tm' => tm';;

let rec id gamma term = match term with
  | App(m,n)    -> App'(id gamma m,id gamma n)
  | Abs(r)      -> new X in Abs'(Y\ (id ((X,Y)::gamma) (r @ X)))
  | nab X in X -> assoc X gamma;;
```

Figure 4: Translating from `tm` to its mirror version `tm'`.

## 4 MLTS example: the $\pi$-calculus

The $\pi$-calculus [?, ?] is a language for modeling processes in which interactions are name-based. In particular, this calculus permits communication via named channels, including the communication of the names of the channels themselves. The basic calculus has two syntactic categories: names and processes.

Process expressions are defined by the following syntax rule.

$$P := 0 \quad | \quad P \,|\, P \quad | \quad P + P \quad | \quad x(y).P \quad | \quad \bar{x}y.P \quad | \quad [x = y].P \quad | \quad \tau.P \quad | \quad (y)P \quad | \quad !\,P.$$

Here, $x$ and $y$ range over names. The process $0$ cannot perform any actions. The expressions $P \mid P$ and $P + P$ denote, respectively, the parallel composition and the choice of two processes. The next four expressions are *prefixed* processes:

- $x(y).P$ represents a process that can accept a name on the channel $x$ and will then become $P$ with $y$ bound to the input name;
- $\bar{x}y.P$ is a process that can output the name $y$ on the channel $x$;
- $[x = y].P$ is a process that can become $P$ provided that the names $x$ and $y$ are equal;
- $\tau.P$ is a process that can evolve through a silent action.

The expression $(y)P$ represents the restriction of the name $y$ to $P$: interactions can take place internally to $P$ through this name but the process cannot communicate externally along the channels $\bar{y}$ or $y$. Finally, $!\,P$ denotes the parallel composition of any number of copies of $P$.

To represent expressions of the $\pi$-calculus in MLTS, we define the two datatypes `name` and `proc` for names and processes that are given in Figure ??. Note that the two process expressions $x(y).P$ and $(y)P$ embody a binding notion. The $\lambda$-tree syntax for these expressions will accordingly include an explicit abstraction. For example, the two $\pi$-calculus expressions

$$(y)\bar{a}y.((y(w).0) \mid (\bar{b}b.0)) \qquad \text{and} \qquad (y)\bar{a}y.((y(w).\bar{b}b.0) + (\bar{b}b.y(w).0))$$

are encoded in MLTS with the terms, respectively.

```
Nu(Y\ Out(A,Y,Par(In(Y, W\ Null),Out(B,B,Null))))
Nu(Y\ Out(A,Y,Plus(In(Y, W\ Out(B,B,Null)),Out(B,B, In(Y, W\ Null)))))
```

One way to demonstrate the expressiveness of the $\pi$-calculus is to encode within it the call-by-name evaluation in the untyped $\lambda$-calculus. Such a translation function was given by Milner in [?] and it can be written as follows.

$$\begin{array}{rcl} [\![x]\!](u) & = & \bar{x}u.0 \\ [\![\lambda x\ M]\!](u) & = & u(x).u(v).[\![M]\!](v) \\ [\![(M\ N)]\!](u) & = & (v).([\![M]\!](v) \mid (x).(\bar{v}x.\bar{v}u.!x(w).[\![N]\!](w))) \end{array}$$

Here, the translation function $[\![\cdot]\!]()$ takes an untyped $\lambda$-term and a name and returns a process function that expects to receive its arguments. We encoded such a located process expression simply by using the `Loc` constructor applied to the abstraction of the location over a process: this is provided by the `located` type in Figure ??. For example, the value of `(transf [] Abs(X\X))` is

```
Loc(U\ In(U,X\ In(U,(Y\ Out(X,Y, Null)))))
```

Note that this translation uses the technique described in Figure ?? of using an association list of nominals.

## 5 Higher-order programming examples

Recall the familiar "fold-right" higher-order function.

```
let rec foldr f a lst = match lst with
  | [] -> a
  | x :: xs -> f x (foldr f a xs);;
```

```
type name =
  | A
  | B
  | C;;

type proc =
  | Null
  | Plus of proc * proc
  | Par  of proc * proc
  | In   of name * (name => proc)
  | Out  of name * name * proc
  | Eqn  of name * name * proc
  | Taup of proc
  | Bang of proc
  | Nu   of name => proc;;
```

Figure 5: Two data types for encoding the $\pi$-calculus.

```
type located =
     | Loc of name => proc;;

let rec trans gamma term = match term with
  | App(m, n) ->
    begin match (trans gamma m, trans gamma n) with
          | (Loc p, Loc q) ->
                Loc(U\ Nu(V\ Par(p @ V,
                    Nu(X\ Out(V, X, Out(V, U, Bang(In(X, q))))))))
    end
  | Abs(m) -> new X in Loc(U\ In(U, Y\
    begin match (trans ((X,Y)::gamma) (m @ X)) with
          | Loc p -> In(U, V\ p @ V)
     end))
  | nab X in X -> Loc(U\ Out(assoc X gamma, U, Null));;
```

Figure 6: Encoding of the call-by-name evaluation of untyped $\lambda$-terms into the $\pi$-calculus.

```
let mapvar fvar term =
    maptm (fun m -> fun n -> App(m, n)) (fun r -> Abs(X\ r X))
          fvar term;;

let lookup sub var = match var with
  | nab X in X ->
    let rec aux s = match s with
                    | []         -> X
                    | (X,t)::sub -> t
                    | (y,t)::sub -> aux sub
     in aux sub;;

let subst_tm sub = mapvar (lookup sub);;
```

Figure 7: Substitution as mapping lookup over a term's structure.

```
let fv term = maptm union (fun r -> new X in remove X (r X))
                          (fun x -> x::[]) term;;

let size term =
    maptm (fun x -> fun y -> 1 + x + y)
          (fun r -> new X in 1 + (r X)) (fun x -> 1) term;;

let terminals term =
    maptm (fun x -> fun y -> x + y)
          (fun r -> new X in (r X)) (fun x -> 1) term;;
```

Figure 8: Additional examples of computing with the higher-order function `maptm`.

This function can be viewed as replacing all occurrences of `::` with the binary function `f` and all occurrences of `[]` with `a`. The following higher-order program does something similar for the datatype of untyped $\lambda$-terms `tm`.

```
let rec maptm fapp fabs fvar term = match term with
    | App(m,n) -> fapp (maptm fapp fabs fvar m) (maptm fapp fabs fvar n)
    | Abs(r)   -> fabs (fun x -> maptm fapp fabs fvar (r @ x))
    | nab X in X -> fvar X;;
```

In particular, the constructors `App` and `Abs` are replaced by functions `fapp` and `fabs` respectively. In addition, the function `fvar` is applied to all nominals encountered in the term. This higher-order function can be used to define a number of other useful and familiar functions.

In Figure ??, the `lookup` function takes a substitution and a nominal and returns the value associated to that nominal in the substitution. Here, the substitution is a term of type `(tm * tm) list`. The application of a substitution to an untyped $\lambda$-term can then be seen as the result of applying the `lookup` function to every variable in the term. The `mapvar` function is a specialization of the `maptm` function to just that purpose. Using the functions in Figure ??, the expressions

```
Abs(X\ (mapvar  (fun x -> X) (Abs(U\ Abs(V\ App(U,V))))));;
new X in new Y in lookup ((X, Abs(U\U))::(Y, Abs(U\ App(U,U)))::[]) X;;
new X in new Y in lookup ((X, Abs(U\U))::(Y, Abs(U\ App(U,U)))::[]) Y;;
```

evaluate to the following three $\lambda$-terms.

```
Abs(X\ Abs(Y\ Abs(Z\ App(X, X))))
Abs(X\X)
Abs(X\ App(X,X))
```

Figure ?? describes the following three functions: `fv` constructs the list of free variables in a term; `size` is a re-implementation of the `size` function presented in Section ??.

## 6 Typing

Given that MLTS is a rather mild extension to OCaml at the syntax level, a typing system for MLTS is simple to present and follows standard practices. Figure **??** contains the rules for typing the new features of MLTS: additional rules for encoding `let` and `let rec` constructions (as well as for built-in types such as integers) must also be added, but these follow the usual pattern. The inference rules in this figure involve the following typing judgments.

$$\Gamma \vdash \texttt{M} : \texttt{A} \qquad \Gamma \vdash \texttt{A} : \texttt{R} : \texttt{B} \qquad \Gamma \vdash \texttt{M} : \texttt{A} \vdash \Delta \qquad \text{open } \texttt{A}$$

In all of these rules, $\Gamma$ is the usual association between bound variables and a type: in our situation, $\Gamma$ will associate both variables and nominals to type expressions. (We also assume that the order of pairs in $\Gamma$ is not important: formally, such contexts are multisets of pairs.) The first of these judgments is the usual typing judgment between a program expression `M` and `A`. The second of these judgments is used to type a rewriting rule `R` that has a left-hand side of type `A` and a right-hand side of type `B`. For example, the following typing judgment should be provable

$$\Gamma \vdash \texttt{tm} : \texttt{Abs(r) -> 1 + (new X in size (r @ X))} : \texttt{int}$$

Since this rule expression is intended to be closed (that is, the variable `r` is quantified implicitly around this rule), the actual value of $\Gamma$ will not impact this particular typing judgment. The third typing judgment above is used to analyze the left-hand-side of a match rule: in particular, $\Gamma \vdash \texttt{M} : \texttt{A} \vdash \Delta$ holds if during the process of analyzing the pattern `M`, pattern variables are produced (since these are implicitly quantified) and placed into the typing context $\Delta$. For example, the following should be provable.

$$\Gamma \vdash \texttt{Abs(r)} : \texttt{tm} : \{\texttt{r} : \texttt{tm => tm}\}$$

Some of the inference rules in Figure **??** contain premises of the form (open `A`) where `A` is a primitive type. Types for which this judgment holds are called *open types* and are the types of bindings in the `new` and backslash expressions: equivalently, open types can contain nominals. For our purposes here, we can assume that every type that is defined in a program (using the `type` command) is assumed to be open. For example, the judgment (open `tm`) needs to be true so that the type `tm => tm` can be formed in the various typing rules. On the other hand, the built-in type for integers `int` should not be considered open in this sense. In the $\pi$-calculus example (Figure **??**), the type `name` must be considered open in this sense, while the type `proc` could be considered either open or not. Clearly a keyword must be added to datatype declarations to indicate if a type is intended as open in this sense.

In the inference rules in Figure **??**, whenever we extend the typing context $\Gamma$ to, say, $\Gamma, \texttt{X} : \texttt{A}$, we always assume that `X` is not declared a type in $\Gamma$ already. Since $\alpha$-conversion is always possible within terms, this assumption can always be satisfied. Note that since pattern variables will be restricted so that they have at most one occurrence in a given pattern, the union of contexts, in the form $\Delta_1, \ldots, \Delta_n$ never attributes more than one type to the same variable.

The prototype implementation [**?**] of MLTS contains a type inference engine that runs on top of $\lambda$Prolog: given the hypothetical judgments available in $\lambda$Prolog, the implemented typing system is structured differently (but equivalently) to the one given in Figure **??**.

## 7 Abstract syntax: the untyped $\lambda$-calculus and arity typing

Although MLTS is designed as a strongly typed functional programming language, evaluation for this language is fundamentally untyped. The *abstract syntax* for MLTS is essentially the untyped $\lambda$-calculus with a few extensions to this core calculus.

Recall the semantic description of the untyped $\lambda$-calculus given by Scott in [**?**]. Scott was able to present a semantic domain $D$ that was isomorphic to its own function space: that is, $D \equiv [D \to D]$. This equivalence is witnessed by the two continuous mappings $\Phi\colon D \to (D \to D)$ (encoding application) and $\Psi\colon (D \to D) \to D$ (encoding abstraction). For example, the untyped $\lambda$-term $\lambda x \lambda y((xy)y)$ is encoded as a value in domain $D$ using the expression $(\Psi(\lambda x(\Psi(\lambda y(\Phi(\Phi\ X\ Y)\ X))))))$.

Note that syntactically, application in the untyped $\lambda$-calculus is captured by two domain-level features: function application and the mapping $\Phi$. Similarly, abstraction is captured by two domain-level features:

$$\frac{}{\Gamma, x : C \vdash \texttt{x} : \texttt{C}} \qquad \frac{\Gamma \vdash \texttt{M} : \texttt{A -> B} \qquad \Gamma \vdash \texttt{N} : \texttt{A}}{\Gamma \vdash \texttt{(M N)} : \texttt{B}} \qquad \frac{\Gamma, \texttt{x} : \texttt{A} \vdash \texttt{M} : \texttt{B}}{\Gamma \vdash \texttt{(fun x -> M)} : \texttt{A -> B}}$$

$$\frac{\Gamma, \texttt{X} : \texttt{A} \vdash \texttt{M} : \texttt{B} \qquad \text{open } \texttt{A}}{\Gamma \vdash \texttt{(new X in M)} : \texttt{B}} \qquad \frac{\Gamma, \texttt{X} : \texttt{A} \vdash \texttt{M} : \texttt{B} \qquad \text{open } \texttt{A}}{\Gamma \vdash \texttt{(X \textbackslash{} M)} : \texttt{A => B}}$$

$$\frac{\Gamma \vdash \texttt{r} : \texttt{A1 => ... => An => A} \quad \Gamma \vdash \texttt{t1} : \texttt{A1} \quad \ldots \quad \Gamma \vdash \texttt{tn} : \texttt{An}}{\Gamma \vdash \texttt{(r @ t1 ... tn)} : \texttt{A}}$$

$$\frac{\Gamma \vdash \texttt{term} : \texttt{B} \qquad \Gamma \vdash \texttt{B} : \texttt{R1} : \texttt{A} \qquad \ldots \qquad \Gamma \vdash \texttt{B} : \texttt{Rn} : \texttt{A}}{\Gamma \vdash \texttt{match term with R1 | ... | Rn} : \texttt{A}}$$

$$\frac{\Gamma, \texttt{X} : \texttt{C} \vdash \texttt{A} : \texttt{R} : \texttt{B} \qquad \text{open } \texttt{C}}{\Gamma \vdash \texttt{A} : \texttt{nab X in R} : \texttt{B}} \qquad \frac{\Gamma \vdash \texttt{L} : \texttt{A} \vdash \Delta \qquad \Gamma, \Delta \vdash \texttt{R} : \texttt{B}}{\Gamma \vdash \texttt{A} : \texttt{L -> R} : \texttt{B}}$$

$$\frac{}{\Gamma, \texttt{x} : \texttt{A} \vdash \texttt{x} : \texttt{A} \vdash \cdot} \qquad \frac{\Gamma \vdash \texttt{X1} : \texttt{A1} \ \ldots\ \Gamma \vdash \texttt{Xn} : \texttt{An} \qquad \text{open } \texttt{A1} \ldots \text{open } \texttt{An}}{\Gamma \vdash \texttt{(r @ X1 ... Xn)} : \texttt{A} \vdash \texttt{r} : \texttt{A1 => ... => An => A}}$$

$$\frac{\Gamma \vdash \texttt{p} : \texttt{A} \vdash \Delta_1 \qquad \Gamma \vdash \texttt{q} : \texttt{B} \vdash \Delta_2}{\Gamma \vdash \texttt{(p,q)} : \texttt{A * B} \vdash \Delta_1, \Delta_2}$$

$$\frac{\Gamma \vdash \texttt{t1} : \texttt{A1} \vdash \Delta_1 \quad \ldots \quad \Gamma \vdash \texttt{tn} : \texttt{An} \vdash \Delta_n}{\Gamma \vdash \texttt{C(t1,...,tn)} : \texttt{A} \vdash \Delta_1, \ldots, \Delta_n} \ \text{provided } C \text{ is a constructor of type } \texttt{A1*...*An -> A}$$

Figure 9: Typing rules based on the concrete syntax for the new features of MLTS.

function abstraction (the creation of an element of $[D \to D]$) and the mapping $\Psi$. We can thus identify two different *syntactic categories* in this encoding: those denoted by the domain $D$ and those identified by the domain of (continuous) functions $D \to D$. In what follows, we need to make a similar distinction between $(\lambda x.T)$ of type $D \to D$ and $(\Psi(\lambda x.T))$ of type $D$.

To capture this distinction in a more general setting, we employ the notion of *arity typing* that has been used by Martin-Löf [?]. In particular, we inductively define arity types as follows.

- There is one primitive arity type, written as $\mathbf{0}$.
- If $\rho_1$ and $\rho_2$ are arity types then so is $(\rho_1 \to \rho_2)$.
- If $\rho_1, \ldots, \rho_n$ $(n \geq 2)$ are arity types then so is $\rho_1 \otimes \cdots \otimes \rho_n$.

Here, $\mathbf{0}$ formally plays a role in the syntax of expressions that is played by domain $D$ in denotational semantics. As is common practice, the infix arrow $\to$ associates to the right. In the encoding of the untyped $\lambda$-calculus, the operator $\Phi$ takes two arguments of arity type $\mathbf{0}$ while the operator $\Psi$ takes one argument of arity $\mathbf{0} \to \mathbf{0}$. The arity type constructor $\otimes$ actually presents no central role here: instead, it is available to form the *uncurried* form of constructors and functions. In particular, we follow the usual OCaml convention that constructors must have arity $(\rho_1 \otimes \ldots \otimes \rho_n) \to \rho_0$ where $\rho_0$ is a primitive arity. The abstract syntax of such constructors could well have the curried arity type $\rho_1 \to \ldots \to \rho_n \to \rho_0$.

In most formalizations of ML-style programming languages, expressions of non-zero arity are generally only involved with formally specifying the application of a function to its argument: all other features of the language only take arguments of arity type $\mathbf{0}$. In MLTS, expressions of non-zero arity play extended roles: for example, in MLTS, pattern matching variables can have non-primitive arity while in most ML-languages, pattern variables are always of primitive arity. Similarly, the constructor `In` used in the $\pi$-calculus encoding (Figure ??) has arity type $(\mathbf{0} \otimes (\mathbf{0} \to \mathbf{0})) \to \mathbf{0}$. It is important to keep arity typing and ML-style typing separated. For example, the type of `maptm` in Section ?? can be inferred to be $(A \to A \to A) \to ((tm \to A) \to A) \to (tm \to A) \to tm \to A$ (for all types $A$). The arity typing of `maptm` is, however, the simple expression $(\mathbf{0} \otimes \mathbf{0} \otimes \mathbf{0} \otimes \mathbf{0}) \to \mathbf{0}$. As we shall see in the following sections, the arity typing is used in the specification of the operational semantics of MLTS.

## 8 Formalizing the design of MLTS

Bindings are such an intimate part of the nature of syntax that we should expect that our high-level programming languages to account for them directly in, for example, any built-in notion of equality or matching. Another reason to include binders as a primitive within a functional programming languages is that their semantics has a well understood declarative and operational treatment. For example, Church's integration of $\lambda$-binding in the higher-order logic STT [?] is an elegant integration in which term and formula equality if given by $\alpha\beta\eta$-conversion on simply typed $\lambda$-terms. Church's integration is also a popular one in theorem proving—being the core logic of the Isabelle [?], HOL [?, ?], and Abella [?] theorem provers—as well as the logic programming language $\lambda$Prolog [?]. Given the existence of these provers, a good literature now exists that describes how to effectively implement STT and closely related logics. Below, we describe what that literature can tell us about the meaning and implementation of the novel features of MLTS.

### 8.1 Equality modulo $\alpha$, $\beta$, $\eta$ conversion

The abstract syntax behind MLTS is essentially a simply typed $\lambda$-term where the types are identified with arity types over the primitive arity $\mathbf{0}$ and the binary arity constructor $\to$. Furthermore, the equality theory of such terms is given by the familiar $\alpha$, $\beta$, $\eta$ conversion rules. As a result, a programming language that adopts this notion of equality cannot take an abstraction and return, say, the name of its bound variable: since that name can be changed via the $\alpha$-conversion, such an operation would not be a proper function. Thus, it is not possible to decompose the untyped $\lambda$-term $\lambda x.t$ into the two components $x$ and $t$. Not being able to retrieve a bound variables name might appear as a serious deficiency but, in fact, it can be a valuable feature of the language: for example, a compiler does not need to maintain such names and can choose any number of different, low-level representations of bindings to exploit during execution. Since the names of bindings seldom have semantically meaningful value, dropping them entirely is an interesting design target. Functional programming languages have many other notions of such abstractions: for example, ML-style languages, the location in memory of a reference cell is not a proper value.

## 8.2 Pattern unification and matching

Since we are not able to decompose bindings into their bound variable and body, we need to find alternative means for analyzing the structure of terms containing bindings. As our earlier examples illustrated, matching within patterns can be used to probe terms and their bindings. If we do not place restrictions on the use of pattern variables, then patterns can have complex behaviors. Consider, for example, the following type definition.

```
type i =
  | U of i => i
  | V of i => i;;

type word =
  | W of i => i;;
```

Expressions of type `word` can be used to encodes words over a two letter alphabet {u, v}: for example, `(W (X\ X))` encodes the empty word, `(W (X\ U (V (V (U X)))))` encodes "uvvu", etc. The pattern

```
match W(X\ U (V (V (U X)))) with
  | W(X\ f @ (g @ X))  ->  (W(X\ f @ X),  W(X\ g @ X))
```

could be used to decompose the given word into a pair of two words. Of course, there are several ways to do such a decomposition (in this particular case with a four-letter word there are five such decompositions). While such multiple decompositions might work well in a logic programming setting where backtracking is always a possibility, we do not wish to have such a possibility within our functional programming language.

Fortunately, there is a natural restriction on pattern variables that guarantees that a match either fails or succeeds with at most one solution. That restriction is the following: every occurrence of an expression of the form `(r @ X1 ... Xn)` in the left-hand side of a match rule must be such that the pattern variable `r` is applied to $n \geq 0$ *distinct* nominals `X1 ... Xn` and those nominals are bound *within* the scope of the binding for `r`. For example, the following expression is not well formed

```
Abs(X\ (match Abs(Y\ App(X,Y)) with
             | Abs(Z\ r @ Z X) -> Abs(Z\ r @ X Z)));;
```

since the scope of the nominal `X` contains the (implicit) scope of the pattern variable `r`, which is around the rule `(Abs(Z\ r @ Z X) -> Abs(Z\ r @ X Z))`. One the other hand, following expression is well formed

```
Abs(X\ (match Abs(Y\ App(X,Y)) with
             | nab W in Abs(Z\ r @ Z W) -> Abs(Z\ r @ W Z)));;
```

since the scopes of both nominals to which `r` is applied in the left-hand-side of the pattern rule are in the scope of the scope for `r`. The value of this latter expression is `Abs(X\ Abs(Y\ App(Y,X)))`. This restriction can also be illustrated within a purely logical setting as follows. Let $j$ be a primitive type and let $f : j \to j \to j$ be a simply typed constant. The formula $\exists G : j \to j\ \forall x : j\ [G\ x = (f\ x\ x)]$ has a unique proof in which $G$ is instantiated by the term $\lambda w.(f\ w\ w)$. Note that the binding scope of the variable $x$ is inside the binding scope of the variable $G$. If, however, one switches the order of the quantifiers, yielding $\forall x : j\ \exists G : j \to j\ [G\ x = (f\ x\ x)]$, then there are four different proofs of this equation: if one replaces the outermost universal quantifier with the eigenvariable, say $a$, then there are four different solutions for $G$, namely, $\lambda w.(f\ a\ a)$, $\lambda w.(f\ a\ w)$, $\lambda w.(f\ w\ a)$, and $\lambda w.(f\ w\ w)$.

The subset of higher-order unification in which unification variables (a.k.a., logic variables, meta-variables, pattern variables) are applied to distinct bound variables restricted as described above, is called *higher-order pattern unification* or $L_\lambda$ *unification* [?]. (See [?] for a functional programming implementation of such unification.) This particular subset of higher-order unification is commonly implemented in theorem provers such as Abella [?], Minlog [?], and Twelf [?] as well as recent implementations of λProlog [?, ?].

The following results regarding higher-order pattern unification can be found in [?].

1. It is decidable. The paper [?] claims that it is, in fact, solvable in linear time.

2. It is unitary, meaning that if there is a unifier then there exists a most general unifier.
3. It does not depend on typing (or on arity). As a result, it is possible to add it to the evaluator for MLTS based on untyped terms.
4. The only form of $\beta$-conversion that is needed to solve such unification problems is what is called $\beta_0$-conversion which is a form of the $\beta$ rule that equates $(\lambda x.t)x$ with $t$.

An equivalent way to write the $\beta_0$-conversion rule (assuming the presence of $\alpha$-conversion) is that $(\lambda x.t)y$ converts to $t[y/x]$ *provided* that $y$ is not free in $\lambda x.t$. Notice that applying $\beta_0$ reduction actually makes a term get smaller: as a result it is not a surprise that such unification (and, hence, matching) has low computational complexity.

## 8.3 $\beta_0$ versus $\beta$

In order to ensure that a matching a rule either fails or has a unique, most general solution, we will insist that in the left side of a match rule, all subexpressions of the form `(r @ X1 ... Xn)` are such that the scope of the binding for `r` contains the scope of the bindings for the distinct variables in `X1`, ..., `Xn`. On the right-hand side of a match rule, however, it seems that one has an interesting choice. If on the right, we have an expression of the form `(r @ t1 ... tn)` then clearly, the terms `t1`, ..., `tn` are intended to be substituted into the abstract that is instantiated for the pattern variable `r`: that is, we need to use $\beta$-conversion on this redex. One choice is that we restrict the terms `t1`, ..., `tn` to be distinct nominals just as on the left-hand-side: in this case, $\beta$-reduction of the expression `(r @ t1 ... tn)` requires only $\beta_0$ reductions. A second choice is that we allow the terms `t1`, ..., `tn` to be unrestricted: in this case, $\beta$-reduction of the expression `(r @ t1 ... tn)` requires more general (and costly) $\beta$ reductions.

In our current design of MLTS, we prefer the first design choice: if we really wish to substitute into pattern variable using more general terms, we can always make calls to explicit substitution functions, such as the one displayed in Figure **??**. The second choice is also sensible since the code for substitution can be automated by a compiler of MLTS and automatically invoked to reduce such `@` expressions on the right-hand side of a match rule.

A similar trade-off between allowing $\beta$-conversion or just $\beta_0$ conversion has also been studied within the theory and design of the $\pi$-calculus. In particular, the full $\pi$-calculus allows the substitution of arbitrary names into input prefixes (modeled by $\beta$-conversion) while the $\pi_I$-calculus ($\pi$-calculus with internal mobility [?]) is restricted in such a way that the only instances of $\beta$-conversion are, in fact, $\beta_0$ conversion.

## 8.4 Match rule quantification

Match rules in MLTS contain two kinds of quantification. The familiar quantification of pattern variables can be interpreted as being universal quantifiers. For example, the first rule defining the `size` function in Section **??**, namely,

```
| App(n, m)  -> 1 + (size n) + (size m)
```

can be encoded as the logical statement

$$\forall \mathtt{m} \forall \mathtt{n}[\mathtt{(size\ (App(n,\ m)))} = \mathtt{1\ +\ (size\ n)\ +\ (size\ m)}].$$

While the third rule contains the binder `nab`

```
| nab X in X -> 1;;
```

which corresponds approximately to the $\nabla$-quantifier (pronounced nabla) that is found in various efforts to formalize the metatheory of computational systems [?, ?]. That is, this rule can be encoded as the quantified equation $\nabla \mathtt{x}.(\mathtt{size\ x} = \mathtt{1})$.

Although there are two kinds of quantifiers around such match rules, the ones corresponding to the universal quantifiers are implicit while the one corresponding to the $\nabla$-quantifier is explicit. Our design for MLTS places the implicit quantifiers at outermost scope: that is, the quantification over a match rule is of the form $\forall\nabla$. Another choice might be to allow some (all) universal quantifiers to be explicitly written and placed among any `nab` bindings. While this is a sensible choice, the $\forall\nabla$-prefixes is, in fact, a reduction class in the sense that if one has a $\forall$ quantifier inside a $\nabla$-quantifier, it is possible to rotate that $\nabla$-quantifier inside using a technique called *raising* [?, ?]. That is, the formula $\nabla x : \gamma \forall y : \tau(Bxy)$ is logically equivalent to the formula $\forall h : (\gamma \to \tau)\nabla x : \gamma(Bx(hx))$: note that as the $\nabla$-quantifier of type $\gamma$ is moved to the right over a universal quantifier, the type of that quantifier is raise from $\tau$ to $\gamma \to \tau$. Thus, it is possible for an arbitrary mixing of $\forall$ and $\nabla$ quantifiers to be simplified to be of the form $\forall\nabla$.

### 8.5 Nominal abstraction

Before we can present the formal operational semantics of MLTS, we need to introduce one final logical concept. The notion of *nominal abstraction* [?] which allows implicit bindings represented by nominal to be moved into explicit abstractions over terms. The following notation is useful for defining this relationship.

Let $t$ be a term, let $c_1, \ldots, c_n$ be distinct nominals that possibly occur in $t$, and let $y_1, \ldots, y_n$ be distinct variables not occurring in $t$ and such that, for $1 \leq i \leq n$, $y_i$ and $c_i$ have the same type. Then we write $\lambda c_1 \ldots \lambda c_n.t$ to denote the term $\lambda y_1 \ldots \lambda y_n.t'$ where $t'$ is the term obtained from $t$ by replacing $c_i$ by $y_i$ for $1 \leq i \leq n$.

There is an ambiguity in the notation introduced above in that the choice of variables $y_1, \ldots, y_n$ is not fixed. However, this ambiguity is harmless: the terms that are produced by acceptable choices are all equivalent under a renaming of bound variables.

Let $n \geq 0$ and let $s$ and $t$ be terms of type $\tau_1 \to \cdots \to \tau_n \to \tau$ and $\tau$, respectively; notice, in particular, that $s$ takes $n$ arguments to yield a term of the same type as $t$. The formula $s \trianglerighteq t$ is a *nominal abstraction of degree* $n$ (or, simply, a *nominal abstraction*). The symbol $\trianglerighteq$ is used here in an overloaded way in that the degree of the nominal abstraction it participates in can vary. The nominal abstraction $s \trianglerighteq t$ of degree $n$ is said to hold just in the case that $s$ $\lambda$-converts to $\lambda c_1 \ldots c_n.t$ for some nominal $c_1, \ldots, c_n$.

Clearly, nominal abstraction of degree 0 is the same as equality between terms based on $\lambda$-conversion, and we will use $=$ to denote this relation in that case. In the more general case, the term on the left of the operator serves as a pattern for isolating occurrences of nominal. For example, if $p$ is a binary constructor and $c_1$ and $c_2$ are nominals, then the nominal abstractions of the first row below hold while those in the second row do not.

$$\begin{array}{lll}
\lambda x.x \trianglerighteq c_1 & \lambda x.p\ x\ c_2 \trianglerighteq p\ c_1\ c_2 & \lambda x.\lambda y.p\ x\ y \trianglerighteq p\ c_1\ c_2 \\
\lambda x.x \not\trianglerighteq p\ c_1\ c_2 & \lambda x.p\ x\ c_2 \not\trianglerighteq p\ c_2\ c_1 & \lambda x.\lambda y.p\ x\ y \not\trianglerighteq p\ c_1\ c_1
\end{array}$$

A logic with equality generalized to nominal abstraction has been studied in [?, ?] where a logic, named $\mathcal{G}$, that contains fixed points, induction, coinduction, $\nabla$-quantification, and nominal abstraction is given a sequent calculus presentation. Cut-elimination for $\mathcal{G}$ is proved in [?] and algorithms and implementations for nominal abstraction are presented in [?, ?]. An important feature of the Abella prover—$\nabla$ in the head of a definition—can be explained and implemented using nominal abstraction [?].

## 9 Natural semantic specification of MLTS

We can now define the operational semantics of MLTS by giving inference rules in the style of natural semantic (a.k.a. big-step semantic) following Kahn [?]. The semantic definition for the core of MLTS is defined in Figure **??**. Since those inference rules are written using an abstract syntax for MLTS, we need to describe briefly how that abstract syntax is derived from the concrete syntax we have presented for our several examples.

Instead of detailing the translation from concrete to abstract syntax, we illustrate this translation with an example. (A parser for MLTS and a transpiler into $\lambda$Prolog code is available for online use and for downloading at [?].) For example, the $\lambda$Prolog code in Figure **??** is the result of parsing and transpiling the MLTS program for `size` given in Section **??**.

The backslash (as infix notation) is also used in $\lambda$Prolog to denote binders; the `@` denotes the untyped $\lambda$-calculus application; `lam` denotes the untyped $\lambda$-calculus abstraction; recursive function definitions are encoded using the `fixpt` operator; the infix symbol `==>` denotes a match rule; `nab` denotes $\nabla$-quantification; and `all` and `all'` are explicit universal quantification bindings for variables of arity **0** and $\mathbf{0} \to \mathbf{0}$, respectively. Finally, the concrete syntax `(let x = t in s)` is translated to the abstract syntax `(let (x\ s) t)`.

$$\frac{\vdash \mathit{val}\ V}{\vdash V \Downarrow V} \qquad \frac{\vdash M \Downarrow F \quad \vdash N \Downarrow U \quad \vdash \mathit{apply}\ F\ U\ V}{\vdash M@N \Downarrow V} \qquad \frac{\vdash (R\ (\mathit{fixpt}\ R)) \Downarrow V}{\vdash (\mathit{fixpt}\ R) \Downarrow V}$$

$$\frac{\vdash C \Downarrow \mathit{tt} \quad \vdash L \Downarrow V}{\vdash \mathit{cond}\ C\ L\ M \Downarrow V} \qquad \frac{\vdash C \Downarrow \mathit{ff} \quad \vdash M \Downarrow V}{\vdash \mathit{cond}\ C\ L\ M \Downarrow V}$$

$$\frac{\vdash M \Downarrow U \quad \vdash (R\ U) \Downarrow V}{\vdash (\mathit{let}\ M\ R) \Downarrow V} \qquad \frac{\vdash (R\ U) \Downarrow V}{\vdash \mathit{apply}\ (\mathrm{lam}\ R)\ U\ V}$$

$$\frac{\vdash \nabla x.(E\ x) \Downarrow (V\ x)}{\vdash \lambda x.Ex \Downarrow \lambda x.Vx} \qquad \frac{\vdash \nabla x.(E\ x) \Downarrow V}{\vdash \mathit{new}\ E \Downarrow V}$$

$$\frac{\vdash \text{pattern}\ T\ \mathit{Rule}\ U \quad \vdash U \Downarrow V}{\vdash (\text{match}\ T\ (\mathit{Rule} :: \mathit{Rules})) \Downarrow V} \qquad \frac{\vdash (\text{match}\ T\ \mathit{Rules}) \Downarrow V}{\vdash (\text{match}\ T\ (\mathit{Rule} :: \mathit{Rules})) \Downarrow V}$$

$$\frac{\vdash \exists x.\text{pattern}\ T\ (P\ x)\ U}{\vdash \text{pattern}\ T\ (\mathit{all}\ \lambda x.P\ x)\ U} \qquad \frac{\vdash (\lambda z_1 \ldots \lambda z_m.(t \Longrightarrow s)) \trianglerighteq (T \Longrightarrow U)}{\vdash \text{pattern}\ T\ (\mathit{nab}\ z_1 \ldots \mathit{nab}\ z_m.(t \Longrightarrow s))\ U}$$

Figure 10: A natural semantic specification of evaluation.

It is intended that the inference rules given in Figure **??** are, in fact, notions for formulas in the logic $\mathcal{G}$. For example, schema variables of the inference figure are universally quantified around the intended formula; the horizontal line is an implication; the list of premises is a conjunction; and $\Downarrow$ is a binary (infix) predicate, etc. Some of the features of $\mathcal{G}$ are exploited by some of those inference rules. Some of the features from $\mathcal{G}$ are enumerated here.

In the rules for `fixpt`, `let`, and `apply`, a variable of arity type $\mathbf{0} \to \mathbf{0}$ (namely, $R$) is applied to a term of arity type $\mathbf{0}$. These rules make use of the underlying equality theory of simply typed $\lambda$-terms in $\mathcal{G}$ to perform a substitution. In the rule for apply, for example, if $R$ is instantiated by the term $\lambda w.t$ and $U$ is instantiated by the term $s$, then the expression written as $(R\ U)$ is equal (in $\mathcal{G}$) to the result of substituting $s$ for the free occurrences of $w$ in $t$: that is, to the result of a $\beta$-reduction on the expression $((\lambda w.t)\ s)$.

Existential quantification is written explicitly into the first rule for patterns. It is possible (as is done in other rules) to drop the explicit existential quantifier and instead have the quantification be implicitly universally quantified around the entire rule. We write it explicitly here to highlight the fact that solving the problem of finding instances of pattern variables in matching rules is lifted to the general problem of finding substitution terms in $\mathcal{G}$. Also, the arity of the existentially quantified variable in that inference rule can range over $\mathbf{0}$, $\mathbf{0} \to \mathbf{0}$, $\mathbf{0} \to \mathbf{0} \to \mathbf{0}$, ….

The proof rules for natural semantics are nondeterministic in principle. Consider attempting to prove that $t$, a term of arity type $\mathbf{0}$, has a value: that is, $\exists V, t \Downarrow V$. It can be the case that no proof exists or that there might be several proofs with different values for $V$. No proofs are possible if, for example, the condition in a conditional phrase does not evaluate to a boolean or if there are insufficient match rules provided to cover all the possible values given to a match expression. Similarly, if there are overlapping

```
prog "size" (fixpt size\ lam term\
  match term
   [(all n\ all m\ ((app n m) ==>
                     (sum @ (i 1) @ (sum @ (size @ n) @ (size @ m))))),
    (all' r\        ((abs r)  ==>
                     (sum @ (i 1) @ (new x\ size @ (r x))))),
    (nab x\         (x        ==> (i 1)))]).
```

Figure 11: The abstract syntax of the `size` program.

matching rules, it is possible to have multiple values computed. Ultimately, we will want to provide a static check that could issue a warning if the rules listed in a match expression are not exhaustive. Making sure that only the first successful match in a list of matching rules is selected is easily achieved in the implementation of natural semantics: for example, we needed to add a single Prolog-cut operator into our λProlog implementation of the natural semantics of MLTS to satisfy that restriction.

The *nominal abstraction* of $\mathcal{G}$ is directly invoked to solve pattern matching in which nominals are explicitly abstracted using the `nab` binding construction. When attempting to prove the judgment $\vdash$ pattern $T$ *Rule* $U$, the inference rules in Figure **??** eventually lead to an attempt to prove in $\mathcal{G}$ an existentially quantified nominal abstraction of the form

$$\exists x_1 \ldots \exists x_n [(\lambda z_1 \ldots \lambda z_m.(t \Longrightarrow s)) \trianglerighteq (T \Longrightarrow U)].$$

Here, the arrow $\Longrightarrow$ is simply a formal (syntactic) pairing operator. The schema variables $x_1, \ldots, x_n$ can be of arity $\mathbf{0}$, $\mathbf{0} \to \mathbf{0}$, $\mathbf{0} \to \mathbf{0} \to \mathbf{0}$, ... and can appear free only in $t$ and $s$: furthermore, if any of these variables are free in $s$ they must be free in $t$. Also, if any of the variables $z_1, \ldots, z_m$ (all of which are assumed to have arity $\mathbf{0}$) are free in $s$ they are also free in $t$. While the variables $x_1, \ldots, x_n$ cannot appear more than once in $t$, the variables $z_1, \ldots, z_m$ are not restricted in this fashion. In order to prove the formula $\exists \bar{x}(\lambda \bar{z}.t) \trianglerighteq s$, one must find a collection of distinct nominals $\bar{c}$ and witness terms $\bar{t}$ that do not contain any of the elements of $\bar{c}$ such that $[\bar{t}/\bar{x}, \bar{c}/\bar{z}]t = s$ [**?**].

It is worth pointing out that given the way we have defined the operational semantics of MLTS, it is immediate that "nominals cannot escape their scopes". For example, the expression `(new X in X)` does not have a value (in abstract syntax, this expression translates to `(new X\ X)`). More precisely, there is no proof of $\vdash \exists v.(new \lambda x.x) \Downarrow v$ using the inference rules in Figure **??**. To understand why this is an immediate consequence of the specification of evaluation, consider the formula

$$\forall E \forall V[(\nabla x.(E\ x) \Downarrow V) \supset (new\ E \Downarrow V)].$$

Given that the scope of the $\nabla x$ is inside the scope of $\forall V$, it is not possible for any instance of this formula to allow the $x$ binder to appear as the second argument of the $\Downarrow$ predicate. While such escaping is easily ruled out using this logical specification, a direct implementation of this logic must incur a cost, however, to constantly ensure that no escaping is permitted. (See Section **??** for more discussion on this point.)

## 10 Binder mobility

We started this programming language project with the desire to treat binders in syntax as directly and naturally as possible. We approached this project by designing the MLTS language which contains more binders features than, say, OCaml: it has not only the usual binders for building functions and for refactoring computation (via, say, the `let` construction) but also new binders that are directly linked to binders in data (via the `new X in`, `nab X in`, and `X\` operators). Finally, the natural semantics of MLTS in $\mathcal{G}$ and its implementation in λProlog are all based on using logics that contain rich binding operators that go beyond the usual universal and existential quantifiers. It is worth noting that if one were to write MLTS programs that do not need to manipulate data structures containing bindings, then the new binding features of MLTS would not be needed and neither would the novel features of both $\mathcal{G}$ and λProlog. Thus, in a sense, binders have not been formally implemented in this story: instead, binders of one kind have been implemented and specified using binders in another system. We were able to complete a prototype implementation of MLTS since we know how to implement the high-level logics, and those techniques can be applied directly to the natural semantic specification.

One way to view the processing a binder is that one needs to first *open* the abstraction, process the result (by "freshening" the newly freed names), and then *close* abstraction [**?**]. In the setting of MLTS, it is better to view such processing as the *movement* of a binders: that is, the binder in a data structure actually gets re-identified with an actual binder in the programming language. As we illustrated in Section **??** with the following step-by-step evaluation

```
size (Abs (X\ (Abs (Y\ (App(X,Y))))));;
new X in 1 + (size (Abs (Y\ (App(X,Y)))));;
new X in 1 + new Y in 1 + (size (App(X,Y)));;
new X in 1 + new Y in 1 + 1 + (size X) + (size Y);;
new X in 1 + new Y in 1 + 1 + 1 + 1;;
```

the bound variable occurrences for X and Y simply move. It is never the case that a bound variable actually becomes free: instead, it just becomes bound elsewhere. Thus, our strategy for strengthening the expressiveness of MLTS over other ML-style languages has been to add to the language more binding sites to which bindings can move.

## 11 Interpreters for MLTS

We have a prototype implementation of MLTS. A parser from our extended OCaml syntax and a transpiler that generates λProlog code are implemented in OCaml. A simple evaluator and type checker written in λProlog can then be used to type check and execute MLTS code. The implementation of the evaluator in λProlog is rather compact (about 140 lines of code) but not completely trivial since neither ∇-quantification nor nominal abstraction are native to λProlog: they needed to be implemented. Both the Teyjus [?] and the Elpi [?] implementations of λProlog can be used to execute the MLTS interpreter.

The `TryMLTS` web site [?] provides a means for anyone with a recent web browser to create and execute MLTS programs online without needing to install any software. Since Elpi, the parser, and the transpiler are written in OCaml, web-based execution was made possible by compiling the OCaml bytecode to a Javascript client library with `js_of_ocaml` [?].

There is little about this prototype implementation that is focused on providing an effective implementation of MLTS. Instead, the prototype is a useful device for exploring the exact meaning and possible uses of the new program features. Never-the-less, we can comment here briefly on some costs of the underlying system that will likely appear in any implementation of MLTS.

### 11.1 Nominal-escape checking

As we have mentioned in Section **??**, nominals are not allowed to escape their scope during evaluation and quantifier alternation can be used to enforce this restriction at the logic level. When one implements the logic, one needs to implement (parts of) the unification of simply typed λ-terms [?] and such unification is constantly checking that bound variable scopes are properly restricted. There are times, however, when the expensive check for escaping nominals are not, in fact, needed. For example, it is possible to rewrite the inference rule in Figure **??** for the `new` binding operator as the following rule.

$$\frac{\vdash \nabla x.(E\ x) \Downarrow (U\ x) \qquad U = \lambda x.V}{\vdash \mathit{new}\ E \Downarrow V}$$

Here, both $U$ and $V$ are quantified universally around the inference rule. Attempting a proof of the first premise can result in the construction of some (possibly large) value, say $t$ such that $\vdash (E\ x) \Downarrow t$ holds. We can immediately form the binding of $U \mapsto \lambda x.t$ without checking the structure of $t$. The second premise is where the examination of $t$ may need to take place: if $x$ is free in $t$, then there is no substitution for $V$ that makes $\lambda x.t$ equal to $\lambda x.V$. This check can be expensive, of course, since one might in principle need to examine the entire structure of $t$ to solve this second premise. There are many situations, however, where such an examination is not needed and they can be revealed by the typing system. For example, if the type of $U$ is, say, `tm => int`, there should not be any possible way for an untyped λ-term to have an occurrence inside an integer. Furthermore, there are static methods for examining type declarations in order to describe if a type $\tau_1 \to \tau_2$ (for primitive types $\tau_1$ and $\tau_2$) can be inhabited by only vacuous λ-terms (see, for example, [?, Section 11]). Of course, if the types of $\tau_1$ and $\tau_2$ are the same (say, `tm`), then type information is not useful here and a check of the entire structure $t$ might be necessary. Other static checks and program analysis might be possible as a way to reduce the costs of checking for escaping nominals: the paper [?] includes such static checks albeit for a technically different functional programming language, namely FreshML [?].

### 11.2 Costs of moving binders

As we have mentioned before, binders are able to move from, say, a term-level binding to a program-level binding by the use of $\beta_0$. In particular, if $y$ is a binder that does not appear free in the abstraction $\lambda x.B$ then the $\beta_0$ reduction of $(\lambda x.B)y$ causes the $x$ binding in $B$ to move and to be identified with the $y$ binder in $B[y/x]$. If one must actually do the substitution of $y$ for $x$ in $B$, a possibly large term (at least its spine) must be copied. However, there are some situations where this movement of a binding can be inexpensive. For example, consider again the following match rule for `size`.

```
          | Abs(r) -> 1 + (new X in size (r @ X))
```

If we assume that the underlying implementation of terms use De Bruijn's nameless dummies, it is possible to understand the rewriting needed in applying this match clause to be a constant time operation. In particular, if `r` is instantiated with an abstraction then it's top-level constructor would indicate where a binder of value 0 points. If we were to compile the syntax `(r @ X)` as simply meaning that that top-level constant is stripped away, then a binder of value 0 in the resulting term would automatically point (move) to being bound by the `new X` binder. While such a treatment of binder mobility without doing substitution is possible in many of our examples, it does not cover all cases. In general, a more involved scheme for implementing binder mobility must be considered. This kind of analysis and implementation of binder mobility is used in the ELPI implementation of λProlog [?].

## 12 Future work

There is clearly much more work to do. While the examples presented in this paper illustrate that the new features in MLTS can provide elegant and direct support for computing with binding structures, we plan to develop many more examples. The general area of theorem proving implementation and compiler construction is an early target for us. A more effective implementation is also something we wish to target soon. It seems likely that we will need to consider extensions to the usual abstract machine models for functional programming in order to get such a direct implementation.

The cost of basic operations in MLTS must also be understood better. As we noted in Section **??**, we could design pattern matching in clauses in such a way that they might require the recursive descent of entire terms to in order to know if a match was successful. Of course, the language could be designed so that such a costly check is never performed during pattern matching: for example, one could insist that every pattern variable is `@`-applied to a list of *all* nominal abstractions that are in the scope of the binding for that pattern variable. In that case, a recursive descent of terms is not needed.

Given the additional expressivity of MLTS, the usual static checks used to produce warnings for non-exhaustive matchings are missing cases that we should add. As mentioned in Section **??**, still other static checks are needed to help a future compiler avoid making costly checks and to guide the users into writing correct programs.

It would also be interesting to see to what extent binders might interact with a range of non-functional features like references found in languages such as OCaml. A natural starting point to explore the possible meaningful interaction of these kinds of features would be to use a natural semantic treatment of these non-functional feature using linear logic [?]: the logical features of $\mathcal{G}$ should also work well in a linear logic setting.

Finally, the treatment of syntax with bindings almost always leads to the need to manipulate contexts and association lists that relate bindings to either other bindings, to types, or to bits of code. We have already seen association lists used in Figures **??** and **??**. It seems likely that more sophisticated MLTS examples will require singling out contexts for special treatment. Although the current design of MLTS does not commit to any special treatment of context, we are interested to see what kind of treatment will actually prove useful in a range of applications.

## 13 Related work

The term *higher-order abstract syntax (HOAS)* was originally introduced in [?] to describe an encoding technique available in λProlog. A subsequent paper identified HOAS as a technique "whereby variables of an object language are mapped to variables in the metalanguage" [?]. When applied to functional programming, this latter description of HOAS describes the mapping of bindings in syntax to the bindings which create functions. Unfortunately, this encoding technique often lacks adequacy (since "exotic terms" can appear [?]), and structural recursion can slip away [?]. The term λ-tree syntax was introduced in [?] to describe the different approach that we have used here.

The $ML_\lambda$ [?] extension to ML allowed for bindings to appear in data structures. Similar to MLTS, the $ML_\lambda$ language contained two different arrow type constructors (`->` and `=>`) and pattern matching was extended to allow for pattern variables to be applied to a list of distinct bound variables. The `new` operator of MLTS could be emulated by using the backslash operator and a "discharge" function. Critically missing from that language was anything similar to the `nab` binding of MLTS. Also, no formal specification and no implementation were ever offered.

Nominals and nominal abstraction, in the sense used in this paper, was first conceived, studied, and implemented as part of the Abella theorem prover [?]. Although the design of Abella does not use the $\trianglerighteq$ relation directly, the notion of "$\nabla$ in the head" of definitions is essentially equivalent to having the $\trianglerighteq$ relation in the logic.

The Delphin [?] and Beluga [?] computer systems provide functional programming support for object-level terms that are taken from the dependently typed $\lambda$-calculus LF. These systems are rather ambitious and make many extensions to the core of ML programming languages. For example, these languages have well established notions of contexts that are part of their programming language's design and evaluation. Our approach here has been much more minimal and incremental.

The FreshML [?] and C$\alpha$ML [?] functional programming languages provide an approach to names based on nominal logic [?]. In a sense, these language provide for an abstract treatment of names and naming. Once naming is available, binding structures can also be implemented. In a sense, the design of these two ML-variants are also more ambitious than the design goal intended for MLTS: in the later, we were not focused on naming but just bindings.

The recent paper [?] introduces a syntactic framework that treats bindings as primitives. That framework is then integrated with various tools and with the framework of contextual types (similar to that found in Beluga) in order to provide a programmer of, say, OCaml with sophisticate tools for the manipulation of syntax and binders. A possible future target for MLTS could be to provide some aspects of such tools more directly in the language itself.

## 14 Conclusion

While the $\lambda$-tree syntax approach to computing with syntax containing bindings has been successfully developed within the logic programming setting (in particular, in $\lambda$Prolog and Twelf), we have illustrated in this paper that this approach can also be captured in a functional programming language if we add to the language new binding operators. Most of the expressiveness of MLTS arises from its increased use of program-level binding. The sophistication needed to correctly exploit binders and quantifiers in MLTS is easily transferred from familiarity with quantification in, for example, predicate logic.

We have also presented a number of MLTS programs and we note that they are both natural and unencumbered by concerns about managing bound variable names. We have also presented a typing discipline for MLTS as well as a formal specification of its natural semantics: this latter task was aided by being able to directly exploit a rich logic, called $\mathcal{G}$, that incorporates $\lambda$-tree syntax principles within quantificational logic. Finally, this natural semantic specification was directly implementable in $\lambda$Prolog. As a consequence, a prototype implementation is available for helping to judge the expressiveness of MLTS programs.

**Acknowledgments.** We thank Kaustuv Chaudhuri, François Pottier, Enrico Tassi, Gabriel Scherer, and the reviewers on an earlier draft of this paper for their helpful comments and observations.