

Functional programming with

λ

-tree syntax

Ulysse Gérard, Dale Miller, Gabriel Scherer

▶ **To cite this version:**

Ulysse Gérard, Dale Miller, Gabriel Scherer. Functional programming with

λ

-tree syntax. PPDP 2019 - 21st International Symposium on Principles and Practice of Programming Languages, Oct 2019, Porto, Portugal. pp.1-16, 10.1145/3354166.3354177 . hal-02368906

HAL Id: hal-02368906
https://inria.hal.science/hal-02368906

Submitted on 19 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Functional programming with λ-tree syntax

Ulysse Gérard, Dale Miller, and Gabriel Scherer
Inria & LIX, Ecole Polytechnique
Palaiseau, France

ABSTRACT

We present the design of a new functional programming language, MLTS, that uses the *λ-tree* syntax approach to encoding bindings appearing within data structures. In this approach, bindings never become free nor escape their scope: instead, binders in data structures are permitted to *move* to binders within programs. The design of MLTS includes additional sites within programs that directly support this movement of bindings. In order to formally define the language's operational 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 within a rich logic that includes both *nominal abstraction* and the *∇-quantifier*: as a result, the natural semantics specification of MLTS can be given a succinct and elegant presentation. We present a typing discipline that naturally extends the typing of core ML programs and we illustrate the features of MLTS by presenting several examples. An on-line interpreter for MLTS is briefly described.

CCS CONCEPTS

• **Software and its engineering** → *Functional languages*; *Data types and structures.*

1 INTRODUCTION

Even from the earliest days of high-level programming, functional programming languages were used to build systems that manipulate the syntax of various programming languages and logics. For example, Lisp was a common language for building theorem provers, interpreters, compilers, and parsers, and the ML programming language was designed as a "meta-language" for a proof checker [22]. While these various tasks involve 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.

Bindings in syntactic expressions have been given, of course, a range of different treatments within the functional programming setting. Common approaches are to implement bindings by using variable names or, in a more abstract way, by using de Bruijn indexes [10]. Since such techniques are quite complex to get right and since bindings are so pervasive, a great deal of energy has gone into making tools and libraries that can help deal with binders: for example, there is the *locally nameless* approach [4, 21, 30] and the *parametric higher-order abstract syntax* approach [8].

Extending a functional programming language with features that support bindings in data has been considered before: for example, there have been the FreshML [53, 59] and CαML [52] extensions to ML-style functional programming languages. Also, entirely new functional programming languages, such as the dependently typed Beluga [47] language, have been designed and implemented with the goal of supporting bindings in syntax. In the domains of logic programming and theorem proving, several designs and implemented systems exist that incorporate approaches to binding: such systems include Isabelle's generic reasoning core [43], λProlog [37, 40], Qu-Prolog [6], Twelf [46], αProlog [5], the Minlog prover [57], and the Abella theorem prover [3].

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

This paper contains the following contributions.

- We present the design of MLTS, a new functional language prototype for dealing with bindings, aiming at expressivity and generality.
- We show how the treatment of bindings that has been successful in the logic programming and theorem proving systems λProlog, Twelf, and Abella, can be incorporated into a functional programming language.
  At the same time, MLTS remains a ML-family language; nominals are treated similarly to constructors of algebraic datatypes (in expressions and patterns), distinguishing our design from existing proposals, such as Delphin and Beluga.
- We present some of the metatheory of MLTS.
- We have a full prototype implementation that is accessible online.

This paper is organized as follows. Section 2 introduces the language MLTS and aims to give a working understanding to the reader of its new constructs and current implementation. Section 3 presents some of the foundational aspects of MLTS' design, which comes from the proof-search (logic programming) paradigm, along with its natural semantics. Section 4 contains a formal description of the typing system for MLTS as well as some static restrictions we impose on the language to obtain good reasoning principles. We also state here some meta-theorems about MLTS. In Section 5 we elaborate on several issues that surround the insertion of binder mobility into this functional programming language. Finally, in Sections 6, 7, and 8 we present future work, related work, and conclude.

2 A TOUR OF MLTS

We chose the concrete syntax of MLTS to be an extension of that of the OCaml programming language (a program in MLTS not using the new language features should be accepted by the `ocamlc` compiler). We assume that the reader is familiar with basic syntactic

conventions of OCaml [42], many of which are shared with most ML-like programming languages.

This section presents the new constructs of MLTS along with a set of examples. We also provide a web application, TryMLTS [20], that can serve as a companion during the reading of this introduction to the language.

2.1 The binding features of MLTS

MLTS contains the following five new language features.

(1) Datatypes can be extended to contain new *nominal* constants and the `(new X in M)` program phrase introduces a fresh nominal `X` in the scope of the evaluation of the term `M`. The value of this expression is the value of `M`, provided that this value does not contain any remaining occurrence of `X` – this would be a nominal escape failure. For example, the term `(new X in X)` fails during evaluation.
(2) A new typing constructor `=>` is used to type bindings within term structures. This constructor is an addition to the already familiar constructor `->` used for function types.
(3) 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.
(4) The infix symbol `@` *eliminates* an abstraction: for example, the expression `((X\body) @ t)` denotes the result of the capture-avoiding substitution of the abstracted nominal `X` by the term `t` in `body`. The notation `(t @ u v)` stands for `(t @ u @ v)` (`@` associates to the left).
(5) Clauses within match-expressions can also contain the `(nab X in p -> t)` binding form. Here, `X` is a nominal local to the clause `p -> t`. At runtime it will be substituted by a nominal `Y` from the ambient context that appears in the scrutiny of the match at the same position than `X` does in `p` but does not already appear in `p -> t`.

These new term operators have the following precedence from highest to lowest: `@`, `new` and `\`. Other operators have the same precedences and associativity than in OCaml. Thus the expression `fun r -> X\ new Y in r @ X` reads as: `fun r -> (X\ (new Y in (r @ X)))`. All three binding expressions—`(X\body)`, `(new X in body)` and `(nab X in rule)`—are subject to α-renaming of bound variables, just as the names of variables bound in `let` declarations and function definitions. As we shall see, nominals are best thought of as constructors: as a consequence, 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 α-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.

We enforce a few restrictions (discussed in Section 4.2) on match expressions: Every `nab`-bounded nominals must occur rigidly (defined in Section 4.2.3) in the pattern and expressions of the form `(m @ X1 ... Xj)` in patterns are restricted so that `m` is a pattern variable and `X1`, ..., `Xj` are distinct nominals bound within the scope of the pattern binding on `m` (which, as a pattern variable, is scoped outside the scopes of `nab`-bound nominals and over the whole rule). This restriction is essentially the same as the one required by *higher-order pattern unification* [32]: 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, η-equality holds.

We now present two series of examples of MLTS programs. We hope that the informal presentation 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 3.4.

2.2 Examples: the untyped λ-calculus

The untyped λ-terms can be defined in MLTS as the 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 a *binding abstraction* of a `tm` over a `tm`. Notice the absence of clause for variables. In MLTS, such a type, called an *open* type, can be extended with a collection of nominal constructors of type `tm`. Just as the type `tm` denotes a syntactic category of untyped λ-terms, the type `tm => tm` denotes the syntactic category of terms abstracted over such terms.

Following usual conventions, 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 α-equivalent to `(Abs(Y\ Abs(X\ X)))`. Similarly, the expression `(let n = 2 in let n = 3 in n)` is parsed as an expression α-equivalent to `(let m = 2 in let n = 3 in n)`: this expression has value `3`.

The MLTS program in Figure 1 computes the size of an untyped λ-term `t`. For example, `(size (App(Abs(X\X), Abs(X\X))))` evaluates to `5`. In the second match rule, the match-variable `r` is 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 nominal constructor of type `tm`. The third match rule contains the `nab` binder that allows the token `X` to match any nominal: alternatively, that last clause could have matched any non-`App` and non-`Abs` term by using the clause `| _ -> 1`. (Note that as written, the three match rules used to define `size` could have been listed in any order.) The following sequence of expressions shows the evolution of a computation involving the `size` function.

```
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` 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: in the expressions above, binding names are chosen for readability.

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

Figure 1: A program for computing the size of a λ-term.

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

let rec beta t = match t with
  | nab X in X -> X
  | Abs r      -> Abs(Y\ beta (r @ Y))
  | App(m, n)  ->
    let m = beta m in let n = beta n in
    begin
        match m with
        | Abs r -> beta (subst r n)
        | _ -> 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 2: The function that computes the substitution $[t/x]u$ and the (partial) function that computes the β-normal form of its argument.

Figure 2 defines the function `(subst t u)` that takes an abstraction over terms `t` and a term `u` and returns the result of substituting the (top-level) bound variable of `t` with `u`. This function works by first introducing a new nominal `X` and then defining an auxiliary function that replaces that nominal in a term with the term `u`. Finally, that auxiliary function is called on the expression `(t @ X)` which is the result of "moving" the top-level bound variable in `t` to the binding occurrence of the expression `new X in`. (As we note in Section 5.3, such binder movement can sometimes be implemented in constant time.) This substitution function has the type `(tm => tm) -> (tm -> tm)`: that is, it is used to inject the abstraction type `=>` into the function type `->`. Substitution is then used by the second function of Figure 2, `beta`, to compute the β-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 `(beta (App(App(plus, two), two)))` and `(beta (App(App(times, two), two)))` are both the Church numeral for 4.

For another example, consider a program that returns `true` if and only if its argument, of type `tm => tm`, is such that its top-level bound variable is a vacuous binding (i.e., the bound variable is not free in its scope). Figure 3 contains three implementations of this boolean-valued function. The first implementation proceeds by

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

let rec vacp2 t =
    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 (t @ X);;

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

Figure 3: Three implementations for determining if an abstraction is vacuous.

```
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'`.

matching patterns with the prefix `X\`, thereby, matching expressions of type `tm => tm`. The second implementation uses a different style: it creates a new nominal `X` and proceeds to work on the term `t @ X`, in the same fashion as the `size` example. The internal `aux` function is then defined to search for occurrences of `X` in that term. The third implementation, `vacp3`, is not (overtly) recursive since the entire effort of checking for the vacuous binding is 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 $\lambda X.s$ equals t? This question can be posed as asking if the logical formula $\exists s.(\lambda X.s) = t$ 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, this kind of pattern matching requires a recursive analysis of the term t and that can make pattern matching costly. To address that cost, pattern matching can be restricted so that such patterns do not occur (see Section 6) or static checks can be added that often make such recursive descents unnecessary (see Section 5.3).

```
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 5: De Bruijn's style syntax and its conversions with type `tm`.

For another simple example of computing on the untyped λ-calculus, consider introducing a mirror version of `tm`, as is done in Figure 4, and writing the function that constructs the mirror term in `tm'` from an input term `tm`. This computation is achieved by adding a context (an association list) as an extra argument that 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).

Figure 5 presents a datatype for the untyped λ-calculus in De Bruijn's style [10] 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 position in that list. For example, the value of

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

is the term `Dabs(Dabs(Dabs(Dapp(Dvar 2, Dvar 0))))` of type `deb`. If `dtrans []` is applied to this second term, the former term is returned (modulo α-renaming, of course).

2.3 Examples: Higher-order programming

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

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

```
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 ->
      match x with
            | nab X in X ->
              maptm fapp fabs fvar (r @ X))
   | nab X in X -> fvar X;;

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 rec remove x l = match l with
    | []    -> []
    | h::tl -> if h = x then remove x tl
                        else h::(remove x tl);;

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 6: Various computations on untyped λ-terms using higher-order programs. Note that there are several occurrences of `(r X)` above that should not be written as `(r @ X)`.

This function can be viewed as replacing all occurrences of `::` with the binary function `f` and all occurrences of `[]` with `a`. The higher-order program `maptm` in Figure 6 does the analogous operation on the datatype of untyped λ-terms `tm`. 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. For example, `mapvar` function is a specialization of the `maptm` function that just applies a given function to all nominals in an untyped λ-term. The application of a substitution (an expression of type `(tm * tm) list`) to a term of type `tm` can then be seen as the result of applying the `lookup`

function to every variable in the term (using `mapvar`). Using the functions in Figure 6, the three 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 λ-terms.

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

Three additional functions are defined in Figure 6: `fv` constructs the list of free variables in a term; `size` is a re-implementation of the `size` function presented in Section 2.2; and `terminals` counts the number of variable occurrences (terminal nodes) in its argument.

2.4 Current prototype implementation

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 are used to type-check and execute the generated MLTS code. The implementation of the evaluator in λProlog is rather compact but not completely trivial since the natural semantics of MLTS (presented in Section 3.4) contains features (namely, ∇-quantification and nominal abstraction) that are not native to λProlog: they needed to be implemented. Both the Teyjus [54] and the Elpi [12] implementations of λProlog can be used to execute the MLTS interpreter. 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` [27].

There is little about this prototype implementation that is focused on providing an efficient implementation of MLTS. Instead, the prototype is useful for exploring the exact meaning and possible uses of the new program features.

3 THE LOGICAL FOUNDATIONS OF A SEMANTIC DEFINITION OF MLTS

Bindings are such an intimate part of the nature of syntax that we should expect that our high-level programming languages account for them directly: for example, any built-in notion of equality or matching should respect at least α-conversion. (The paper [36] contains an extended argument of this point in the setting of logic programming and proof assistants.) Another reason to include binders as a primitive within a functional programming languages is that their semantics have a well understood declarative and operational treatment. For example, Church's higher-order logic STT [9] contains an elegant integration of bindings in both terms and formulas. His logic also identifies equality for both terms and formulas with $\alpha\beta\eta$-conversion. Church's integration is also a popular one in theorem proving—being the core logic of the Isabelle [44], HOL [23, 25], and Abella [3] theorem provers—as well as the logic programming language λProlog [37]. Given the existence of these provers, a good literature now exists that describes how to effectively implement STT and closely related logics. Since the formal specifications of evaluation and typing will be given using inference rules and since such rules can be viewed as quantified formulas, this literature provides means for implementing MLTS.

3.1 Equality modulo α, β, η conversion

The abstract syntax behind MLTS is essentially a simply typed λ-term that encodes untyped λ-calculus, as described in Section 3.4. Furthermore, the equality theory of such terms is given by the familiar α, β, η 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 α-conversion, such an operation would not be a proper function. Thus, it is not possible to decompose the untyped λ-term $\lambda x.t$ into the two components x and t. Not being able to retrieve a bound variable's 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 choice. That choice is similar to one taken in ML-style languages in which the location in memory of a reference cell is not maintained as a value in the language.

The relation of λ-conversion is invoked when evaluating the expression `(t @ s1 ... sn)`. As we shall see, MLTS is a typed language so we can assume that the expressions `s1`, ..., `sn` have types $\gamma_1, \ldots, \gamma_n$, respectively, and that `t` must have type $\gamma_1 \Rightarrow \cdots \Rightarrow \gamma_n \Rightarrow \gamma_0$. Thus, `t` is η-equivalent to a term with n abstractions, for example, `X1\...Xn\ t'` and the value of the expression `(t @ s1 ... sn)` is the result of performing λ-normalization of `((X1\...Xn\ t') s1 ... sn)`.

3.2 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 2.2, namely,

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

can be encoded as the logical statement

```
∀m∀n[(size (App(n, m))) =  1 + size n + size m].
```

The third match rule for `size` contains the binder `nab`

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

which corresponds approximately to the *generic* ∇-quantifier (pronounced nabla) that is found in various efforts to formalize the metatheory of computational systems (see [3, 39] and Section 3.4). That is, this rule can be encoded as ∇`x.(size x = 1)`: that is, the size of a nominal constant is 1.

Although there are two kinds of quantifiers around such match rules, the ones corresponding to the universal quantifiers are implicit in the concrete syntax while the ones corresponding to the ∇-quantifiers are 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 ∇-quantifier, it is possible to rotate that ∇-quantifier inside using a technique called *raising* [32, 39]. 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 ∇-quantifier of type γ is moved to the right over a universal quantifier, the type of that quantifier is raised from τ to $\gamma \to \tau$. Thus, it is possible for an arbitrary mixing of $\forall$ and ∇ quantifiers to be simplified to be of the form $\forall\nabla$.

3.3 Nominal abstraction

Before we can present the formal operational semantics of MLTS, we need to introduce one final logical concept, *nominal abstraction*, which allows implicit bindings represented by nominals to be moved into explicit abstractions over terms [18]. 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 $1 \le i \le n$, 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 $1 \le i \le n$. There is an ambiguity in this notation in that the choice of variables $y_1, \ldots, y_n$ is not fixed. This ambiguity is, however, harmless since the terms that are produced by acceptable choices are all equivalent under α-conversion.

Let $n \ge 0$ and let s and t be terms of type $\tau_1 \to \cdots \to \tau_n \to \tau$ and τ, 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 overloaded since it can be use at different degrees (generally, the degree can be determined from context). The nominal abstraction $s \trianglerighteq t$ of degree n is said to hold just in the case that s is λ-convertible to $\lambda c_1 \ldots c_n . t$ for some distinct nominals $c_1, \ldots, c_n$.

Clearly, nominal abstraction of degree 0 is the same as equality between terms based on λ-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 nominals. 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.

$$\lambda x.x \trianglerighteq c_1 \qquad \lambda x.p\ x\ c_2 \trianglerighteq p\ c_1\ c_2 \qquad \lambda x.\lambda y.p\ x\ y \trianglerighteq p\ c_1\ c_2$$
$$\lambda x.x \not\trianglerighteq p\ c_1\ c_2 \qquad \lambda x.p\ x\ c_2 \not\trianglerighteq p\ c_2\ c_1 \qquad \lambda x.\lambda y.p\ x\ y \not\trianglerighteq p\ c_1\ c_1$$

A logic with equality generalized to nominal abstraction has been studied in [16, 18] where a logic, named $\mathcal{G}$, that contains fixed points, induction, coinduction, ∇-quantification, and nominal abstraction is given a sequent calculus presentation. Cut-elimination for $\mathcal{G}$ is proved in [16, 18] and algorithms and implementations for nominal abstraction are presented in [16, 60]. An important feature of the Abella prover—∇ in the head of a definition—can be explained and encoded using nominal abstraction [17].

3.4 Natural semantics specification of MLTS

We can now define the operational semantics of MLTS by giving inference rules in the style of natural semantics (a.k.a. big-step semantics) following Kahn [28]. The semantic definition for the core of MLTS is defined in Figure 7. Since those inference rules are written using a higher-order abstract syntax for MLTS, directly inspired by λProlog term representations; we briefly describe how that abstract syntax is derived from the concrete syntax.

Instead of detailing the translation from concrete to abstract syntax, we illustrate this translation with an example. There is an implementation of MLTS that includes a parser and a transpiler into λProlog code: this system is available for online use and for download at https://trymlts.github.io [20]. For example, the λProlog code in Figure 8 is the abstract syntax for the MLTS program for `size` given in Section 2.2.

The backslash (as infix notation) is also used in λProlog to denote binders and it is the only λProlog primitive in Figure 8. The other constructors are introduced to encode MLTS abstract syntax trees.

This encoding of MLTS syntax is a generalization of the familiar semantic encoding of the untyped λ-calculus given by Scott in 1970 [58], in which a semantic domain D and two continuous mappings (retracts) $\Phi\colon D \to (D \to D)$ (encoding application) and $\Psi\colon (D \to D) \to D$ (encoding abstraction) are used to encode the untyped λ-calculus. For example, the untyped λ-calculus $\lambda x \lambda y((xy)y)$ is encoded as a value in domain D using the expression $(\Psi(\lambda x(\Psi(\lambda y(\Phi(\Phi\ x\ y)\ y))))))$. In Figure 8, the constructors `c_App` and `c_Abs` represents the Φ and Ψ functions, respectively. The λProlog abstraction operator (backslash) is used to build expressions that correspond to inhabitants of $D \to D$.

The constant `fix` represents anonymous fixpoints, to which recursive functions are translated (we also have an n-ary fixpoint for mutually-recursive functions). Note that `fix x \ t` is idiomatic λProlog syntax for the application `fix (x \ t)`, omitting parentheses to use `fix` in the style of a binder.

The expression `lam x \ ...` represents the MLTS expression `fun x -> ...`; in our abstract syntax we write lam($\lambda X. \ldots$). (We do not make a syntactic distinction between X and x which are just variables, but we use uppercase variables in the abstract syntax for variables that represent nominals in the language.) Similarly, the expression `new X \ ...` encodes `new X in ...`; in our abstract syntax we write new ($\lambda X. \ldots$). The expression-former `match` represents pattern-matching, it expects a scrutinee and a list of clauses. Clauses are built from the infix operator `==>`, taking a pattern on the left and a term on the right, and from quantifiers `all`, to introduce universally-quantified variables (implicit in MLTS programs), and `nab` to introduce nominals. `all`-bound variables and `nab`-bound nominals have the type of expressions; they are injected in patterns by `pvar` and `pnom`. `pvariant` (in patterns) and `variant` (in expressions) denote datatype constructor applications, they expect a datatype constructor and a list of arguments. `special` expects the name of a run-time primitive (arithmetic operations, polymorphic equality...) and a list of arguments. `int` represents integer literals. Finally, we use explicit AST expression-formers `backslash` and `arobase` (a French name for `@`) and pattern-formers `pbackslash` and `parobase` to represent the constructions `\` and `@` of MLTS. Only `arobase` is present in this example.

It is intended that the inference rules given in Figure 7 are, in fact, notations for formulas in the logic $\mathcal{G}$. For example, schema variables of the inference rules are universally quantified around the intended formula; the horizontal line is an implication; the list

$$
\begin{array}{rcll}
 & & \text{values} & \\
V & ::= & X & \\
 & | & \mathrm{lam}(\lambda x.M\ x) & \\
 & | & \mathrm{backslash}\ (\lambda X.V\ X) & \\
 & | & \mathrm{variant}\ c\ [V_1, \ldots, V_n] &
\end{array}
$$

$$\frac{}{\vdash \mathrm{lam}\ R \Downarrow \mathrm{lam}\ R} \qquad \frac{\vdash \forall i \in [1;n],\ T_i \Downarrow V_i}{\vdash \mathrm{variant}\ c\ [T_1, \ldots, T_n] \Downarrow \mathrm{variant}\ c\ [V_1, \ldots, V_n]} \qquad \frac{\vdash \nabla X.(E\ X) \Downarrow V}{\vdash \mathrm{new}\ (\lambda X.E\ X) \Downarrow V}$$

$$\frac{\vdash M \Downarrow \mathrm{lam}\ R \quad \vdash N \Downarrow U \quad \vdash (R\ U) \Downarrow V}{\vdash \mathrm{app}\ M\ N \Downarrow V} \qquad \frac{\vdash M \Downarrow U \quad \vdash (R\ U) \Downarrow V}{\vdash (\mathrm{let}\ M\ R) \Downarrow V} \qquad \frac{\vdash R\ (\mathrm{fix}\ R) \Downarrow V}{\vdash \mathrm{fix}\ R \Downarrow V} \qquad \frac{\vdash M \Downarrow \mathrm{backslash}\ R \quad \vdash (R\ X) \Downarrow V}{\vdash \mathrm{arobase}\ M\ X \Downarrow V}$$

$$\frac{\vdash \nabla X.(E\ X) \Downarrow (V\ X)}{\vdash \mathrm{backslash}\ (\lambda X.E\ X) \Downarrow \mathrm{backslash}\ (\lambda X.V\ X)} \qquad \frac{\vdash \mathrm{clause}\ T\ \mathit{Rule}\ U \quad \vdash U \Downarrow V}{\vdash (\mathrm{match}\ T\ (\mathit{Rule}::\mathit{Rules})) \Downarrow V} \qquad \frac{\vdash \neg(\exists u,\ \mathrm{clause}\ T\ \mathit{Rule}\ u) \quad \vdash (\mathrm{match}\ T\ \mathit{Rules}) \Downarrow V}{\vdash (\mathrm{match}\ T\ (\mathit{Rule}::\mathit{Rules})) \Downarrow V}$$

$$\frac{\vdash \exists x.\mathrm{clause}\ T\ (P\ x)\ U}{\vdash \mathrm{clause}\ T\ (\mathrm{all}\ (\lambda x.P\ x))\ U} \qquad \frac{\vdash \mathrm{matches}\ T\ P \quad \vdash (\lambda Z_1 \ldots \lambda Z_m.(p \Longrightarrow u)) \trianglerighteq (P \Longrightarrow U)}{\vdash \mathrm{clause}\ T\ (\mathrm{nab}\ Z_1 \ldots \mathrm{nab}\ Z_m.(p \Longrightarrow u))\ U}$$

$$\frac{\vdash \forall i \in [1;n],\ \mathrm{matches}\ t_i\ p_i}{\vdash \mathrm{matches}\ (\mathrm{variant}\ c\ [t_1, \ldots, t_n])\ (\mathrm{pvariant}\ c\ [p_1, \ldots, p_n])} \qquad \frac{\mathrm{nominal}(c)}{\vdash \mathrm{matches}\ c\ (\mathrm{pnom}\ c)} \qquad \frac{}{\vdash \mathrm{matches}\ x\ (\mathrm{pvar}\ x)}$$

Figure 7: A natural semantics specification of evaluation.

```
(fix size \ lam term \
  match term
  [(all m \ all n \
     (pvariant c_App [(pvar n), (pvar m)]) ==>
     (special add [(special add [(int 1),
                    (app size n)]),
                    (app size m)])),
   (all r \ (pvariant c_Abs [pvar r]) ==>
      (special add
               [(int 1),
                (new X \ app size
                             (arobase r X))])),
   (nab X \ (pnom X) ==> (int 1))])
```

Figure 8: The abstract syntax of the `size` program.

of premises is a conjunction; and $\Downarrow$ is a binary (infix) predicate, etc. Some features of $\mathcal{G}$ are exploited by some of those inference rules: those features are enumerated below.

Figure 7 starts with a grammar for values. In addition to lambda-abstractions, backslash -expressions (with a value as the body) and variant values, (open) values also include nominals. Evaluating a closed term can never produce a nominal, but evaluation rules under binders may return nominals.

In the rules for app, `let` and `fix`, 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 λ-terms in $\mathcal{G}$ to perform a substitution. In the rule for apply, for example, if R is instantiated to 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 β-reduction on the expression $((\lambda w.t)\ s)$. (While matching and applying patterns is limited to β_0-reduction, full β-reduction is used for the natural semantic specification.)

Existential quantification is written explicitly into the first rule for patterns. 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 $\mathcal{G}$.

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. 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. Conversely, the variables introduced by `all` and nab in patterns may have several satisfying values, if they are not used in the pattern itself, or only in flexible occurrences (see Section 4.2.3).

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 $\vdash \mathrm{clause}\ T\ \mathit{Rule}\ U$, the inference rules in Figure 7 eventually lead to an attempt to prove in $\mathcal{G}$ an existentially quantified nominal abstraction of the form

$$\exists x_1 \ldots \exists x_n [(\lambda Z_1 \ldots \lambda Z_m.(p \Longrightarrow u)) \trianglerighteq (P \Longrightarrow U)].$$

Here, the arrow $\Longrightarrow$ is simply a formal (syntactic) pairing operator, expecting a pattern on the left and a term on the right. The schema variables $x_1, \ldots, x_n$ can appear free only in p and u.

The last ingredient of our pattern-matching rule is the judgment ($\vdash \mathrm{matches}\ T\ P$) that checks that a term or value T is indeed matched by a pattern P. Since patterns and terms are encoded using two distinct syntactic categories, this judgment relates pattern-formers to the corresponding term-formers. Nominals are embedded in patterns by the pnom(c) pattern-former, which matches a corresponding nominal—the condition nominal(c) can be expressed in terms of nominal abstraction $(\lambda X.\ X) \trianglerighteq c$. Term variables introduced by all are embedded in patterns by the pvar pattern-former, and they can match any term x—note that in this rule, x denotes an

arbitrary term, substituted for a term variable by the all-handling rule.

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 $(\lambda X.X)$). More precisely, there is no proof of $\vdash \exists v.(\text{new } (\lambda X.X)) \Downarrow v$ using the rules in Figure 7. To see why this is an immediate consequence of the specification of evaluation, consider the formula (which encodes the rule in Figure 7 for new)

$$\forall E \forall V[(\nabla X.(E\ X) \Downarrow V) \supset (\text{new } E \Downarrow V)].$$

Given that the scope of the ∇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 may incur a cost, however, to constantly ensure that no escaping is permitted. (See Section 5.2 for more discussion on this point.)

4 TYPING RULES AND RESTRICTIONS, SMALL-STEP SEMANTICS, META-THEOREMS

In this section we present a typing discipline for MLTS, followed by a few restrictions on pattern matching necessary for it to remain well behaved and the establishment of standard formal results.

4.1 Typing

Given that MLTS is a rather mild extension of OCaml at the syntax level, a typing system for MLTS is simple to present and follows standard practices. Figure 9 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.

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

In all of these rules, Γ is the usual association between bound variables and a type: in our situation, Γ will associate both variables and nominals to type expressions. (We also assume that the order of pairs in Γ is not important.) 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 clause `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.

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

Since this rule expression is intended to be closed (that is, the variable `r` is quantified implicitly around this rule), the actual value of Γ 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, $\Gamma \vdash \mathtt{M} : \mathtt{A} \dashv \Delta$ 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 Δ. For example, the following should be provable.

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

Some of the inference rules in Figure 9 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 presumed 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. 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 9, whenever we extend the typing context Γ to, say, $\Gamma, \mathtt{X} : \mathtt{A}$, we assume that `X` is not declared in Γ already. Since α-conversion is always possible within terms, this assumption can always be satisfied. Note that since pattern variables are restricted (as is usual) 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 TryMLTS [20] of MLTS contains a type inference engine that runs on top of λProlog: given the hypothetical judgments available in λProlog, the implemented typing system is structured differently (but equivalently) to the one given in Figure 9. By using λProlog, we were able to turn this typing system into one that does type inference: this type inference engine does not infer polymorphic typing, however.

4.2 Restriction on 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 that we may wish to avoid during evaluation.

4.2.1 Unique occurrence of pattern variables. We impose a familiar restriction on the match rules: a pattern variable must have exactly one occurrence within a match pattern. Asking for at least one occurrence avoids under-specified pattern variables, that could be bound to anything. As is typical in ML-style languages, asking for at most one occurrence relieves pattern matching from the need to check equality of terms. Since terms can be large, pattern matching could involve a costly recursive descent of terms; we forbid repeated occurrences of pattern variables and force the programmer to insert equality checking outside the pattern matching operation. Thus, instead of defining `memb : tm -> tm list -> bool` with the following code using a repeated match variable

```
let rec memb x l = match (x,l) with
    | (x,[])     -> false
    | (x,(x::l)) -> true
    | (y,(x::l)) -> memb x l;;
```

we can require the programmer to write an equality predicate for type `tm` and then rewrite the program above as follows.

```
let rec eqtm t s = match (t,s) with
    | (App(m1,m2), App(n1,n2)) -> eqtm m1 n1 &&
                                  eqtm m2 n2
```

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

$$\frac{\Gamma \vdash \mathtt{M} : \mathtt{A} \qquad \Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{M} : \mathtt{B}}{\Gamma \vdash \mathtt{let\ x\ =\ M\ in\ N} : \mathtt{B}} \qquad \frac{\Gamma, \mathtt{X} : \mathtt{A} \vdash \mathtt{M} : \mathtt{B} \qquad \mathtt{open\ A}}{\Gamma \vdash \mathtt{(X\ \backslash\ M)} : \mathtt{A} \texttt{ => } \mathtt{B}}$$

$$\frac{\Gamma \vdash \mathtt{r} : \mathtt{A1} \texttt{ => } \ldots \texttt{ => } \mathtt{An} \texttt{ => } \mathtt{A} \quad \Gamma \vdash \mathtt{t1} : \mathtt{A1} \quad \ldots \quad \Gamma \vdash \mathtt{tn} : \mathtt{An}}{\Gamma \vdash \mathtt{(r\ @\ t1\ \ldots\ tn)} : \mathtt{A}} \qquad \frac{\mathtt{C} : \mathtt{A1}, \ldots, \mathtt{An} \to \mathtt{B} \qquad \Gamma \vdash \mathtt{t1} : \mathtt{A1} \quad \ldots \quad \Gamma \vdash \mathtt{tn} : \mathtt{An}}{\Gamma \vdash \mathtt{C(t1}, \ldots, \mathtt{tn)} : \mathtt{B}}$$

$$\frac{\Gamma \vdash \mathtt{M} : \mathtt{A} \quad \Gamma \vdash \mathtt{N} : \mathtt{B}}{\Gamma \vdash \mathtt{(M,\ N)} : \mathtt{A} * \mathtt{B}} \qquad \frac{\Gamma, \mathtt{X} : \mathtt{A} \vdash \mathtt{M} : \mathtt{B} \qquad \mathtt{open\ A}}{\Gamma \vdash \mathtt{(new\ X\ in\ M)} : \mathtt{B}} \qquad \frac{\Gamma \vdash \mathtt{term} : \mathtt{B} \qquad \Gamma \vdash \mathtt{B} : \mathtt{R1} : \mathtt{A} \qquad \ldots \qquad \Gamma \vdash \mathtt{B} : \mathtt{Rn} : \mathtt{A}}{\Gamma \vdash \mathtt{match\ term\ with\ R1\ |\ \ldots\ |\ Rn} : \mathtt{A}}$$

$$\frac{\Gamma, \mathtt{X} : \mathtt{C} \vdash \mathtt{A} : \mathtt{R} : \mathtt{B} \qquad \mathtt{open\ C}}{\Gamma \vdash \mathtt{A} : \mathtt{nab\ X\ in\ R} : \mathtt{B}} \qquad \frac{\Gamma \vdash \mathtt{L} : \mathtt{A} \dashv \Delta \qquad \Gamma, \Delta \vdash \mathtt{R} : \mathtt{B}}{\Gamma \vdash \mathtt{A} : \mathtt{L\ \texttt{->}\ R} : \mathtt{B}} \qquad \frac{\Gamma \vdash \mathtt{X1} : \mathtt{A1} \ \ldots\ \Gamma \vdash \mathtt{Xn} : \mathtt{An} \quad \mathtt{open\ A1} \ldots \mathtt{open\ An}}{\Gamma \vdash \mathtt{(r\ @\ X1\ \ldots\ Xn)} : \mathtt{A} \dashv \mathtt{r} : \mathtt{A1} \texttt{ => } \ldots \texttt{ => } \mathtt{An} \texttt{ => } \mathtt{A}}$$

$$\frac{}{\Gamma \vdash \mathtt{x} : \mathtt{A} \dashv \mathtt{x} : \mathtt{A}} \qquad \frac{\Gamma \vdash \mathtt{p} : \mathtt{A} \dashv \Delta_1 \qquad \Gamma \vdash \mathtt{q} : \mathtt{B} \dashv \Delta_2}{\Gamma \vdash \mathtt{(p,q)} : \mathtt{A} * \mathtt{B} \dashv \Delta_1, \Delta_2} \qquad \frac{\mathtt{C} : \mathtt{A1}, \ldots, \mathtt{An} \to \mathtt{B} \qquad \Gamma \vdash \mathtt{p1} : \mathtt{A1} \dashv \Delta_1 \quad \ldots \quad \Gamma \vdash \mathtt{pn} : \mathtt{An} \dashv \Delta_n}{\Gamma \vdash \mathtt{C(p1}, \ldots, \mathtt{pn)} : \mathtt{B} \dashv \Delta_1, \ldots, \Delta_n}$$

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

```
    | (Abs r, Abs s)  -> new X in eqtm (r @ X)
                                       (s @ X)
    | nab X in (X, X) -> true
    | _               -> false;;

let rec memb x l = match (x,l) with
    | (x,[])     -> false
    | (x,(y::l)) -> if (eqtm x y)
                    then true else (memb x l);;
```

Given the definition of the `tm` datatype, it is clear that a compiler for MLTS could define its own equality predicate for this type. In that case, repeated variable occurrences in patterns could be allowed since resolving such patterns could be done using these equality predicates.

4.2.2 Restricted use of higher-order pattern variables. Since pattern variables within match rules can have higher-order types, occurrences of those variables within patterns need to be restricted: otherwise, undesirable features of higher-order matching could appear. Fortunately, there is a natural restriction on occurrences of 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))`.

This restriction can be motivated 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 an eigenvariable (or nominal), 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_λ *unification* [32]. (We assume here the usual convention that unification problems and matching problems only involve terms that are in β-normal form.) This particular subset of higher-order unification is commonly implemented in theorem provers such as Abella [3], Minlog [57], and Twelf [46] as well as recent implementations of λProlog [12, 54]. A functional programming implementation of such unification is given in [41].

The following results about higher-order pattern unification are proved in [32].

(1) It is decidable and unitary, meaning that if there is a unifier then there exists a most general unifier.
(2) It does not depend on typing. As a result, it is possible to add it to the evaluator for MLTS based on untyped terms.
(3) The only form of β-conversion that is needed to solve such unification problems is what is called β_0-conversion which is a form of the β rule that equates $(\lambda x.t)x$ with t.

An equivalent way to write the β_0-conversion rule (assuming the presence of α-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 β_0 reduction actually makes a term smaller and does not introduce new β redexes: as a result it is not a surprise that such unification (and, hence, matching) has low computational complexity.

4.2.3 All `nab` *bound variables must have a rigid occurrence.* There is an additional restriction on match rules that is associated to the `nab` binder that appear in such rules. We say that an occurrence of a `nab`-quantified nominal is *flexible* if it is in the scope of an

@. For example, in the code in Figure 10, the nominal binding W has two occurrences that are flexible: one each within (r @ Z W) and (r @ W Z). All other occurrences of a nab-bound nominal are *rigid*. For example, in the match rule | nab X in X -> 1, X has a binding occurrence and a rigid occurrence. In the auxiliary function used by the index function in Figure 5, namely,

```
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)
```

the nominals X and Y have both rigid and flexible occurrences within their scope.

The one additional restriction that we need is the following: every nab-bound variable must have at least one rigid occurrence in the left part of the match rule (the pattern) that falls within the scope of its binder. For example, the code in Figure 10 does not satisfy this restriction since every occurrence of W in the pattern is flexible (there is just one such occurrence).

This restriction ensures that each nab-bound nominal in a matching clause is mapped to a uniquely-determined nominal of the ambient context. As interesting counter-examples, consider

```
match Z with
| nab X Y in  (r @ X Y) -> term
```

where Z is a nominal, and

```
match 1 with
| nab X in 1 -> t
```

which are both ruled out by this restriction. In the first example, there are two instantiations for r that make this match succeed, namely, using the terms X\Y\X and X\Y\Y. This breaks the determinacy property – Theorem 4.3. In the second example, the nominal X is completely unconstrained by the pattern. If this program was allowed, our natural semantics dictates that it should behave as new X in t; the restriction guarantees that new is the only language construct that may introduce dynamic nominal-escape failures.

4.3 Small-step operational semantics

As a complement to the natural (big-step) semantics of Figure 7, we developed a small-step operational semantics of MLTS. Its two salient features are as follows: (1) the small-step treatment of evaluation contexts clarifies the moments during reduction where escape-checking must be performed (this is often left implicit in the natural semantics), and (2) its treatment of pattern-matching does *not* use nominal-abstraction – it implements an equivalent but lower-level mechanism. This lower-level expression of the handling of nabla-bound nominals in pattern-matching gives a more operational intuition of the language, and it also guides practical implementations in languages without native support for nominal abstraction. In fact, we co-evolved this operational semantics with the λProlog implementation of the language, the former guiding the latter, with

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

Figure 10: Code that does not satisfy the restriction on occurrences of nab bound variables.

evaluation contexts

$$\begin{array}{rcl} E[\Box] & ::= & \Box \\ & | & \text{app } M\ E \mid \text{app } E\ N \\ & | & \text{backslash } (\lambda X.E) \mid \text{arobase } E\ X \\ & | & \text{new } (\lambda x.E) \\ & | & \text{variant } c\ [M_1 \dots M_k, E, M_{k+2} \dots M_n] \\ & | & \text{match } E\ [R_1, \dots, R_n] \end{array}$$

$$\frac{}{\text{app (lam } R)\ V \rightsquigarrow^{\text{hd}} R\ V} \qquad \frac{}{\text{arobase (backslash } R)\ X \rightsquigarrow^{\text{hd}} R\ X}$$

$$\frac{}{\text{fix } R \rightsquigarrow^{\text{hd}} R\ (\text{fix } R)} \qquad \frac{M \rightsquigarrow^{\text{hd}} M'}{E[M] \rightsquigarrow E[M']}$$

$$\frac{X \notin V}{E[\text{new } (\lambda X.V)] \rightsquigarrow E[V]}$$

Figure 11: Small step reduction: core fragment

rigid paths

$$\begin{array}{rcl} \pi & ::= & \Box \\ & | & \text{variant } C\ i\ \pi \\ & | & \text{backslash } (\lambda X.\pi) \\ & | & \text{arobase } \pi\ X \end{array}$$

Rigid occurrence in a value: $V' \in_\pi V$

$$\frac{}{V' \in_\Box V'} \qquad \frac{V' \in_\pi V_k}{V' \in_{(\text{variant } C\ k\ \pi)} \text{variant } c\ [V_1, \dots, V_n]}$$

$$\frac{\nabla X.\ V' \in_\pi V}{V' \in_{(\text{backslash } (\lambda X.\pi))} (\text{backslash } (\lambda X.V))}$$

$$\frac{V' \in_\pi \text{backslash } (\lambda X.V)}{V' \in_{(\text{arobase } \pi\ X)} V}$$

Rigid occurrence in a pattern: $p' \in_\pi p$

$$\frac{}{p' \in_\Box p'} \qquad \frac{p' \in_\pi p_k}{p' \in_{(\text{variant } C\ k\ \pi)} \text{variant } c\ [p_1, \dots, p_n]}$$

$$\frac{\nabla X.\ p' \in_\pi p}{p' \in_{(\text{backslash } (\lambda X.\pi))} (\text{backslash } (\lambda X.p))}$$

$$\frac{p' \in_\pi p}{p' \in_{(\text{arobase } \pi\ X)} (\text{arobase } p\ X)}$$

Rigid occurrence in a clause: $p' \in_\pi R$

$$\frac{\nabla Z.\ p' \in_\pi R}{p' \in_\pi \text{nab } (\lambda Z.R)} \qquad \frac{\nabla x.\ p' \in_\pi R\ x}{p' \in_\pi \text{all } R} \qquad \frac{p' \in_\pi p}{p' \in_\pi p \to M}$$

Figure 12: Rigid paths in values and patterns

the bugs found playing with the latter informing changes to the

$$\frac{V \text{ with } R \rightsquigarrow \emptyset, N}{\text{match } V\ (R{::}Rs) \rightsquigarrow^{\text{hd}} N} \qquad \frac{\nexists N.\ (V \text{ with } R \rightsquigarrow \emptyset, N)}{\text{match } V\ (R{::}Rs) \rightsquigarrow^{\text{hd}} \text{match } V\ Rs}$$

Matching a value against a clause: V with $R \rightsquigarrow \sigma, N$

$$\frac{\nabla X.\ \exists \pi, Y.\quad \begin{array}{ll} X \in_\pi R\ X & Y \notin R\ X \qquad Y \notin \sigma \\ Y \in_\pi V & V \text{ with } R\ Y \rightsquigarrow \sigma, N \end{array}}{V \text{ with nab } R \rightsquigarrow \sigma, N}$$

$$\frac{\nabla x.\ V \text{ with } R \rightsquigarrow \sigma[x \mapsto V_x], N\ x}{V \text{ with (all } R) \rightsquigarrow \sigma, N\ V_x} \qquad \frac{V \text{ matches } p \text{ as } \sigma}{V \text{ with } (p \to N) \rightsquigarrow \sigma, N}$$

Matching a value against a pattern: V matches p as σ

$$\frac{\nabla X.\ V\ X \text{ matches } p\ X \text{ as } \sigma}{\text{backslash } V \text{ matches backslash } p \text{ as } \sigma}$$

$$\frac{\text{backslash } V \text{ matches } p \text{ as } \sigma}{V\ X \text{ matches arobase } p\ X \text{ as } \sigma}$$

$$\frac{\forall i \in 1..n, \quad V_i \text{ matches } p_i \text{ as } \sigma_i}{C(V_1 \ldots V_n) \text{ matches } C(p_1 \ldots p_n) \text{ as } \biguplus_i \sigma_i}$$

$$\frac{}{V \text{ matches } x \text{ as } (x \mapsto V)} \qquad \frac{}{V \text{ matches } _ \text{ as } \emptyset} \qquad \frac{}{X \text{ matches } X \text{ as } \emptyset}$$

Figure 13: Small step reduction: pattern-matching

former – using the natural semantics as a reference specification for what the behavior should be.

Due to space restrictions, we will not give a fully detailed explanation of this operational semantics. For the details, the figures will have to speak for themselves, we will below give a high-level presentation of the rules.

Core language (without pattern-matching). Figure 11 gives a small-step operational semantics for the fragment of the language without pattern-matching. We use the standard approach of decomposing reduction into a head reduction and evaluation contexts.

Our evaluation contexts allow reduction under the nominal abstraction (backslash ($\lambda X.E$) is an evaluation context): it does not delay computation like the standard λ-abstraction does.

The other non-standard aspect of this fragment is the treatment of the name-creation construct new ($\lambda X.M$). Instead of trying to "generate a fresh nominal" in the small-step semantics, we simply allow reduction under new binders – the stack of new in the current evaluation context is the set of "ambient nominals" available at this point of the program execution. In addition to the standard rule allowing reduction under context, we have an extra contextual rule to allow popping a new binder off the context: when the term inside that binder has been fully evaluated to a value, so we have a term of the form E[new ($\lambda X.V$)], we can remove the binder after performing an escape check ($X \notin V$), continuing evaluation with $E[V]$. If the escape check fails, the term is stuck – this is the presentation in our semantics of nominal escape as a dynamic failure.

Paths of rigid occurrences. As we explained in Section 4.2.3, a clause of the form nab ($\lambda X.p \to M$) is only accepted if the nominal X has at least one rigid occurrence in the pattern p. The operational semantics uses this criterion. In Figure 12, we define a grammar of *rigid paths* π, which represent evidence that a given occurrence of a sub-pattern (sub-value) in a pattern (value) is in rigid position, as defined by the judgments $p' \in_\pi p$ and $v' \in_\pi v$.

Looking at the path (arobase π X) in a pattern (arobase p X) selects a sub-value by looking at π in p. In terms, (arobase v X) is not a value, but any value V X can be eta-expanded to the (non-value) form (arobase (backslash $\lambda X.V\ X$) X), so we look for the sub-value at path π in (backslash $\lambda X.V\ X$).

Operational semantics of pattern matching. The treatment of pattern-matching in this operational semantics, given in Figure 13 is not particularly small-step: matching a value against a clause is a single step, so it is more big-step in nature. The key interest of these rules is that they do not use nominal abstraction, and instead "implement" the same behavior in a more computational style.

The judgment (v matches p as σ) holds when the value v can be matched against the value p, by performing the substitution σ – from pattern variables in p into sub-values of v. The inputs of the judgment are v and p, and the substitution σ is an output of the inference process.

The judgment (v with $R \rightsquigarrow \emptyset, N$) holds when the value v can be matched against the clause R, returning a right-hand-side N to evaluate. In N, the pattern variables bound in R (by the clause-former all ($\lambda x.R$)) have already been substituted with the corresponding sub-values of v. In the general case, we want to define the meaning of matching a value v against a clause R *after* having traversed some all-quantifications, that is with extra pattern variables in the ambient context; the general form of the judgment is v with $R \rightsquigarrow \sigma, N$, where σ is a substitution from those ambient pattern variables, which still occur free in N.

The correspondence with the natural semantics is as follows: v with $R \rightsquigarrow \sigma, N$ in the operational semantics holds if and only if clause $v\ R[\sigma]\ N[\sigma]$ holds in the natural semantics.

4.4 Formal properties of MLTS

Given the restrictions of Section 4.2, we can list the following three formal properties about MLTS.

Theorem 4.1 (Nominals do not escape). *Let E be the abstract syntax of an MLTS program that does not contain any free nominal. If $\vdash E \Downarrow V$ is provable then V does not contain any free nominals.*

The proof of this follows from a simple induction on the structure of proofs in the logic $\mathcal{G}$: the precise nature of the semantic specification given in Figure 7 is not relevant. The systematic use of the ∇-quantifier guarantees this conclusion.

Theorem 4.2 (Type preservation). *If the typing judgment $\vdash E : A$ and the evaluation judgment $\vdash E \Downarrow V$ holds, then so does $\vdash V : A$.*

The proof is mostly standard, but it must handle the pattern-matching rule defined by nominal abstraction. This is done using our rigid paths π. We can easily prove that if the judgment clause V nab ($\lambda Z_1 \ldots$ nab ($\lambda Z_n.p \to N$)) holds, then the path π_i of Z_i in p is also the path of some nominal Y_i in v. Then one needs an intermediate lemma to say that the type A of a value or pattern and a path π within that value or pattern uniquely determine the

type B of the sub-value or the sub-pattern; because p and v have the same type, the nominals Z_i and Y_i must also have the same type, which is key to the type-preservation argument.

Theorem 4.3 (Determinacy of evaluation). *If* $\vdash E \Downarrow V$ *and* $\vdash E \Downarrow U$ *then* $V = U$.

The proof of this theorem follows the usual outline. Again, rigid paths are used in the pattern-matching rule to justify that the nominals bound by nabla-abstraction are uniquely determined.

Detailed proofs of these theorems can be found in the forthcoming Ph.D. dissertation of the first author [19].

5 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 with more binders than, say, OCaml: it has not only the usual binders for building functions and for refactoring computation (via 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 the implementers of λProlog provide a low-level implementation of bindings that we are able to use in our static and dynamic semantics specifications.

One way to view the processing of a binder is that one first *opens* the abstraction, processes the result (by "freshening" the freed names), and then *closes* the abstraction [52]. In the setting of MLTS, it is better to view such processing as the *movement* of a binder: 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 2.2 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 becomes free: instead, it just becomes bound elsewhere.

5.1 β_0 versus β

As we describe in Section 4.2.2, we 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 scopes 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 abstraction that is instantiated for the pattern variable `r`: that is, we need to use β-conversion on this redex. One design choice is that we restrict the terms `t1`, . . ., `tn` to be distinct nominals just as on the left-hand-side: in this case, β-reduction of the expression `(r @ t1 ... tn)` requires only β_0 reductions. A second choice is that we allow the terms `t1`, . . ., `tn` to be unrestricted: in this case, β-reduction of the expression `(r @ t1 ... tn)` requires more general (and costly) β-reductions. Our current implementation allows for these richer forms of `@` expressions.

A similar trade-off between allowing β-conversion or just β_0 conversion has also been studied within the theory and design of the π-calculus. In particular, the full π-calculus allows the substitution of arbitrary names into input prefixes (modeled by β-conversion) while the π_I-calculus (π-calculus with internal mobility [55]) is restricted in such a way that the only instances of β-conversions are, in fact, β_0-conversions (see Chapter 11 in [37]).

Another reason to identify the β_0 fragment of β-conversion is that β_0 reduction provides support for binder mobility and it can be given effective implementations, sometimes involving only constant time operations (see Section 5.3).

5.2 Nominal-escape checking

As we have mentioned in Section 3.4, 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 [26] 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. In particular, it is possible to rewrite the inference rule in Figure 7 for the `new` binding operator as the following rule.

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

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 type 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 τ_1 and τ_2) can be inhabited by at most vacuous λ-terms (see, for example, [33]). Of course, if the types of τ_1 and τ_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 analyses might be possible as a way to reduce the costs of

checking for escaping nominals: the paper [53] includes such static checks albeit for a technically different functional programming language, namely FreshML [59].

5.3 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 β_0. In particular, if y is a binder that does not appear free in the abstraction $\lambda x.B$ then the β_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 indexes, 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 its 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].

6 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 centered on the general area of implementing theorem provers and compiler construction. 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. A first step in this direction would be to first design a small-step (SOS) semantics equivalent of our natural semantics.

The cost of basic operations in MLTS must also be understood better. As we noted in Section 2.2, we could design pattern matching in clauses in such a way that they might require the recursive descent of entire terms in order to know if a match was successful. The language could also 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 5, still other static checks are needed to help a future compiler avoid making costly checks. Finally, adding polymorphic typing should be possible following the pattern already established by OCaml.

It is also interesting to see to what extent binders interact with a range of non-functional features, such as references. A natural starting point to explore the possible interaction of effectful features would be to use a natural semantics treatment based on linear logic (see, for example, [7, 34]): the logical features of $\mathcal{G}$ should also work well in a linear logic setting.

Finally, the treatment of syntax with bindings generally leads to the need to manipulate contexts and association lists that relate bindings to other bindings, to types, or to bits of code. We have already seen association lists used in Figure 4. 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.

7 RELATED WORK

The term *higher-order abstract syntax (HOAS)* was introduced in [45] 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" [46]. When applied to functional programming, this description implies the mapping of bindings in syntax to the bindings that create functions. Unfortunately, such encoding technique often lacks adequacy (since "exotic terms" can appear [11]), and structural recursion can slip away [15]. The terms *λ-tree syntax* [36, 38] and *binder mobility* [35] were later introduced to describe the different and more syntactic approach that we have used here.

7.1 Systems with two arrow type constructors

The ML_λ [31] extension to ML is similar to MLTS in that it also contains 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 the "discharge" function of ML_λ. Critically missing from that language was anything similar to the `nab` binding of MLTS. Also, no formal specification and no implementation were ever offered. Licata & Harper [29] have used the universe feature of Agda 2 to provide an implementation of bindings in data structures that also relies on supporting two different implications-as-types.

Nominals and nominal abstraction, in the sense used in this paper, were first conceived, studied, and implemented as part of the Abella theorem prover [3]. While Abella only has one arrow type constructor, that arrow type maps to the `=>` of MLTS: this is possible in Abella since computation is performed at the level of relations and not functions. As a result, the function type arrow `->` of MLTS and OCaml is not needed. Thus the distinction mentioned in [29] between an arrow for computation and an arrow for binding is, in fact, also present in Abella, although computations are not represented functionally.

7.2 Systems with one arrow type constructor

The Delphin design is probably the closest to MLTS, in particular [56] introduced a programming-language version of the ∇ quantifier from [39], whose usage is related to the ∇ of MLTS. In Delphin, ∇ introduces normal term variables (there is no separate class of nominal constants), while MLTS presents nominals as closer to datatype constructors, with a natural usage in pattern-matching.

Delphin makes nominal-escape errors impossible at runtime by imposing a static discipline to prevent them, while MLTS allows runtime failure in order to allow for more experimentation. The original proposal in [56] uses a type modality that imposes a strict FIFO discipline on free variables. This discipline was found too constraining; [50] completely eschews a `new` construct (its $vx.\,e$ binder actually corresponds to nominal abstraction `X\e` in MLTS), and [51] uses a type-based restriction (type subordination), only allowing to introduce a fresh nominal in expressions whose return types only contains values that cannot contain this nominal. This discipline accepts some examples from our paper, for example `size` in Figure 1 and `id` in Figure 4, but rejects other (safe) programs, such as the second and third one-liner examples of Section 2.3. Richer static disciplines have been proposed for FreshML [49, 53], but they add complexity, and interact poorly with the introduction of mutable state; MLTS is an experimental design aiming for expressivity, so we decided to allow dynamic escape failures instead.

Beluga [47] allows the programmer to use both dependent types and recursive definitions as well as an integrated notion of context (along with a method to describe certain invariants using *context schema*). Static checks of Beluga programs can be used to prove the formal correctness of Beluga programs (commonly by proving that a given piece of program code is, in fact, a total function). As a result, a checked Beluga program is often a formal proof. Since a wide range of formal systems can be encoded naturally using dependently typed λ-terms [1, 24], Beluga programs can be used for both programming with and reasoning about the metatheory of those formal systems. Since bindings and contexts are part of the vocabulary of Beluga, these formal proofs can capture the metatheory of logical and computational systems (such as natural deduction proof systems and the operational semantics of rich programming languages). The goal of MLTS is intended only to support programming and not directly reasoning: the intent of the new features of MLTS is only to support the manipulation of syntax containing bindings. A possibly interesting comparison between MLTS and Beluga might be explored by using typing and contexts in the latter in a mostly trivial way. It is likely that Beluga could code most MLTS programs although using different primitives.

7.3 Systems using nominal logic

The FreshML [59] and CαML [52] functional programming languages provide an approach to names based on nominal logic [48]. These two programming languages 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 latter, we were not focused on naming but just bindings.

The recent paper [14] 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 sophisticated tools for the manipulation of syntax and binders. A possible future target for MLTS could be to provide such tools more directly in the language itself.

7.4 Challenge problems and benchmarks

Genuine comparisons between different programming languages are generally hard to achieve. For example, in the area of logical frameworks and related theorem provers, there are also a number of formal systems and computer implementations. In order to understand the relative merits of these different systems, challenge problems and benchmarks [2, 13] have been proposed to help people sort out specific merits and challenges of one system relative to another. In depth comparisons of the programming languages described above will probably require similar in-depth comparisons on representative programming tasks.

8 CONCLUSION

While the λ-tree syntax approach to computing with syntax containing bindings has been successfully developed within the logic programming setting (in particular, in λProlog and Twelf), we provide in this paper another example of how binding can be captured in a functional programming language. 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 a skill most people have learned from using quantification in, for example, predicate logic.

We have 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 allows capturing both λ-tree syntax and binder mobility. Finally, the natural semantics specification and the typing system were directly implementable in λProlog. A prototype implementation is available for helping to judge the expressiveness of MLTS programs.

Acknowledgments. We thank Kaustuv Chaudhuri, François Pottier, Enrico Tassi, the HOPE Workshop 2018 audience, and the anonymous reviewers for their helpful comments and observations.

REFERENCES

[1] Arnon Avron, Furio Honsell, Ian A. Mason, and Robert Pollack. 1992. Using Typed Lambda Calculus to Implement Formal Systems on a Machine. *Journal of Automated Reasoning* 9 (1992), 309–354. https://doi.org/10.1007/BF00245294

[2] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The POPLmark Challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference (LNCS)*. Springer, 50–65. https://doi.org/10.1007/11541868_4

[3] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. 2014. Abella: A System for Reasoning about Relational Specifications. *Journal of Formalized Reasoning* 7, 2 (2014), 1–89. https://doi.org/10.6092/issn.1972-5787/4650

[4] Arthur Charguéraud. 2011. The Locally Nameless Representation. *Journal of Automated Reasoning* (May 2011), 1–46. https://doi.org/10.1007/s10817-011-9225-2

[5] James Cheney and Christian Urban. 2004. Alpha-Prolog: A Logic Programming Language with Names, Binding, and Alpha-Equivalence. In *Logic Programming, 20th International Conference (LNCS)*, Bart Demoen and Vladimir Lifschitz (Eds.), Vol. 3132. Springer, 269–283. https://doi.org/10.1007/978-3-540-27775-0_19

[6] Anthony S. K. Cheng, Peter J. Robinson, and John Staples. 1991. Higher Level Meta Programming in Qu-Prolog 3: 0. In *Logic Programming, Proceedings of the Eigth International Conference, Paris, France, June 24-28, 1991*, Koichi Furukawa (Ed.). MIT Press, 285–298.

[7] Jawahar Chirimar. 1995. *Proof Theoretic Approach to Specification Languages*. Ph.D. Dissertation. University of Pennsylvania. http://www.lix.polytechnique.fr/Labo/Dale.Miller/chirimar/phd.ps

[8] Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 143–156. https://doi.org/10.1145/1411204.1411226

[9] Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *J. of Symbolic Logic* 5 (1940), 56–68. https://doi.org/10.2307/2266170

[10] Nicolaas Govert de Bruijn. 1972. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with an Application to the Church-Rosser Theorem. *Indagationes Mathematicae* 34, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

[11] Joëlle Despeyroux, Amy Felty, and Andre Hirschowitz. 1995. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*. 124–138. https://doi.org/10.1007/BFb0014049

[12] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. 2015. ELPI: Fast, Embeddable, λProlog Interpreter. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings (LNCS)*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.), Vol. 9450. Springer, 460–468. https://doi.org/10.1007/978-3-662-48899-7_32

[13] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. 2015. The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations: Part 2–A Survey. *J. of Automated Reasoning* 55, 4 (2015), 307–372. https://doi.org/10.1007/s10817-015-9327-3

[14] Francisco Ferreira and Brigitte Pientka. 2017. Programs Using Syntax with First-Class Binders. In *Proceedings of the 26th European Symposium on Programming, ESOP 2017, Uppsala, Sweden (Lecture Notes in Computer Science)*, Hongseok Yang (Ed.), Vol. 10201. Springer, 504–529.

[15] M. J. Gabbay and A. M. Pitts. 1999. A new approach to abstract syntax involving binders. In *14th Symp. on Logic in Computer Science*. IEEE Computer Society Press, 214–224.

[16] Andrew Gacek. 2009. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. Ph.D. Dissertation. University of Minnesota.

[17] Andrew Gacek, Dale Miller, and Gopalan Nadathur. 2008. Combining generic judgments with recursive definitions. In *23th Symp. on Logic in Computer Science*, F. Pfenning (Ed.). IEEE Computer Society Press, 33–44. https://doi.org/10.1109/LICS.2008.33

[18] Andrew Gacek, Dale Miller, and Gopalan Nadathur. 2011. Nominal abstraction. *Information and Computation* 209, 1 (2011), 48–73. https://doi.org/10.1016/j.ic.2010.09.004

[19] Ulysse Gérard. 2019. *Computing with relations, functions, and bindings*. Ph.D. Dissertation. University of Paris Saclay. http://www.lix.polytechnique.fr/Labo/Dale.Miller/gerard19phd.pdf

[20] Ulysse Gérard, Dale Miller, and Gabriel Scherer. 2018. Try MLTS Online. https://trymlts.github.io/.

[21] A. Gordon. 1994. A Mechanisation of Name-Carrying Syntax up to Alpha-Conversion. In *International Workshop on Higher Order Logic Theorem Proving and its Applications (Lecture Notes in Computer Science)*, Vol. 780. 414–426.

[22] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. 1979. *Edinburgh LCF: A Mechanised Logic of Computation*. LNCS, Vol. 78. Springer.

[23] Michael J. C. Gordon. 1991. Introduction to the HOL System. In *Proceedings of the International Workshop on the HOL Theorem Proving System and its Applications*, Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley (Eds.). IEEE Computer Society, 2–3.

[24] Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (1993), 143–184.

[25] John Harrison. 2009. HOL Light: an overview. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 60–66.

[26] Gérard Huet. 1975. A Unification Algorithm for Typed λ-Calculus. *Theoretical Computer Science* 1 (1975), 27–57. https://doi.org/10.1016/0304-3975(75)90011-0

[27] js-of-ocaml 2018. Js_of_ocaml. http://ocsigen.org/js_of_ocaml/.

[28] Gilles Kahn. 1987. Natural Semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (LNCS)*, Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing (Eds.), Vol. 247. Springer, 22–39.

[29] Daniel R. Licata and Robert Harper. 2009. A Universe of Binding and Computation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 123–134. https://doi.org/10.1145/1596550.1596571

[30] Conor McBride and James McKinna. 2004. Functional pearl: I am not a number - I am a free variable. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*, Henrik Nilsson (Ed.). ACM, 1–9. http://doi.acm.org/10.1145/1017472.1017477

[31] Dale Miller. 1990. An Extension to ML to Handle Bound Variables in Data Structures: Preliminary Report. In *Proceedings of the Logical Frameworks BRA Workshop*. Antibes, France, 323–335. http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/mll.pdf Available as UPenn CIS technical report MS-CIS-90-59.

[32] Dale Miller. 1991. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *J. of Logic and Computation* 1, 4 (1991), 497–536. https://doi.org/10.1093/logcom/1.4.497

[33] Dale Miller. 1992. Unification under a mixed prefix. *Journal of Symbolic Computation* 14, 4 (1992), 321–358. https://doi.org/10.1016/0747-7171(92)90011-R

[34] Dale Miller. 1996. Forum: A Multiple-Conclusion Specification Logic. *Theoretical Computer Science* 165, 1 (Sept. 1996), 201–232. https://doi.org/10.1016/0304-3975(96)00045-X

[35] Dale Miller. 2004. Bindings, mobility of bindings, and the ∇-quantifier. In *18th International Conference on Computer Science Logic (CSL) 2004 (LNCS)*, Jerzy Marcinkowski and Andrzej Tarlecki (Eds.), Vol. 3210. 24. https://doi.org/10.1007/978-3-540-30124-0_4

[36] Dale Miller. 2018. Mechanized Metatheory Revisited. *Journal of Automated Reasoning* (04 Oct. 2018). https://doi.org/10.1007/s10817-018-9483-3

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

[38] Dale Miller and Catuscia Palamidessi. 1999. Foundational Aspects of Syntax. *Comput. Surveys* 31 (Sept. 1999).

[39] Dale Miller and Alwen Tiu. 2005. A proof theory for generic judgments. *ACM Trans. on Computational Logic* 6, 4 (Oct. 2005), 749–783. https://doi.org/10.1145/1094622.1094628

[40] Gopalan Nadathur and Dale Miller. 1988. An Overview of λProlog. In *Fifth International Logic Programming Conference*. MIT Press, Seattle, 810–827. http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/iclp88.pdf

[41] Tobias Nipkow. 1993. Functional Unification of Higher-Order Patterns. In *8th Symp. on Logic in Computer Science*, M. Vardi (Ed.). IEEE, 64–74.

[42] OCaml. 2018. http://ocaml.org/.

[43] Lawrence C. Paulson. 1989. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning* 5 (Sept. 1989), 363–397.

[44] Lawrence C. Paulson. 1994. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer Verlag.

[45] Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 199–208.

[46] Frank Pfenning and Carsten Schürmann. 1999. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th Conf. on Automated Deduction (CADE) (LNAI)*, H. Ganzinger (Ed.). Springer, Trento, 202–206. https://doi.org/10.1007/3-540-48660-7_14

[47] Brigitte Pientka and Joshua Dunfield. 2010. Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description). In *Fifth International Joint Conference on Automated Reasoning (LNCS)*, J. Giesl and R. Hähnle (Eds.). 15–21.

[48] Andrew M. Pitts. 2003. Nominal Logic, A First Order Theory of Names and Binding. *Information and Computation* 186, 2 (2003), 165–193.

[49] A. M. Pitts and M. J. Gabbay. 2000. A Metalanguage for Programming with Bound Names Modulo Renaming. In *Mathematics of Program Construction. 5th International Conference, MPC2000, Ponte de Lima, Portugal, July 2000. Proceedings (LNCS)*, R. Backhouse and J. N. Oliveira (Eds.), Vol. 1837. Springer, Heidelberg, 230–255.

[50] Adam Poswolsky and Carsten Schürmann. 2008. Practical programming with higher-order encodings and dependent types. In *Proceedings of the European Symposium on Programming (ESOP 2008)*.

[51] Adam Poswolsky and Carsten Schürmann. 2008. System Description: Delphin - A Functional Programming Language for Deductive Systems, In International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008), A. Abel and C. Urban (Eds.). *Electr. Notes Theor. Comput. Sci.* 228, 113–120.

[52] François Pottier. 2006. An Overview of Cαml. In *Proceedings of the ACM-SIGPLAN Workshop on ML (ML 2005) (Electr. Notes Theor. Comput. Sci.)*, Vol. 148. 27–52. https://doi.org/10.1016/j.entcs.2005.11.039

[53] François Pottier. 2007. Static name control for FreshML. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. IEEE, 356–365.

[54] Xiaochu Qi, Andrew Gacek, Steven Holte, Gopalan Nadathur, and Zach Snow. 2015. The Teyjus System – Version 2. http://teyjus.cs.umn.edu/ http://teyjus.cs.umn.edu/.

[55] Davide Sangiorgi. 1996. π-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science* 167, 2 (1996), 235–274.

[56] Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. 2005. The nabla-calculus. Functional Programming with Higher-order Encodings. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA'05)*. https://doi.org/10.1007/11417170_25

[57] Helmut Schwichtenberg. 2006. Minlog. In *The Seventeen Provers of the World (LNCS)*, Freek Wiedijk (Ed.), Vol. 3600. Springer, 151–157. https://doi.org/10.1007/11542384_19

[58] Dana Scott. 1970. Outline of a Mathematical Theory of Computation. In *Proceedings, Fourth Annual Princeton Conference on Information Sciences and Systems*. Princeton University, 169–176. Also, Programming Research Group Technical Monograph PRG–2, Oxford University.

[59] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. 2003. FreshML: Programming with Binders Made Simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*. ACM Press, 263–274.

[60] Yuting Wang, Kaustuv Chaudhuri, Andrew Gacek, and Gopalan Nadathur. 2013. Reasoning about Higher-Order Relational Specifications. In *Proceedings of the 15th International Symposium on Princples and Practice of Declarative Programming (PPDP)*, Tom Schrijvers (Ed.). Madrid, Spain, 157–168. https://doi.org/10.1145/2505879.2505889