Functional programming with λ-tree syntax: a progress report

Ulysse Gérard Dale Miller
Inria-Saclay & LIX, Ecole Polytechnique
Palaiseau, France

In this progress report, we highlight the design of the functional programming language MLTS which we have recently proposed elsewhere. This language uses the λ-*tree* syntax approach to encoding data structures that contain bindings. In this setting, bound variables never become free nor escape their scope: instead, binders in data structures are permitted to *move* into binding sites within programs. The concrete syntax of MLTS is based on the one for OCaml but includes additional binders within programs that directly support the mobility of bindings. The natural semantics of MLTS can be viewed as a logical theory within the logic $\mathcal{G}$, which forms the basis of the Abella proof system and which includes nominal abstractions and the ∇-quantifier. Here, we provide several examples of MLTS programs. We also illustrate how many Abella relational specifications that are known to specify functions can be rewritten as functions in MLTS.

1 Introduction

The λ-tree syntax approach to encoding data structures that contain binders (which includes a wide range of syntactic expressions) has been successfully used in a wide range of specifications, programs, and reasoning systems. Most of these systems, however, are based on logic programming or, more generally, the proof search paradigms. One exception to that trend is Beluga [19], a functional programming language that is based on contextual modal logic [16]. Beluga provides first-class treatments for not only bindings but also contexts: the latter play important roles in mechanizing reasoning involving, for example, type systems.

In this progress report, we report on a recently designed functional programming language we call MLTS. With this language, we attempt a modest goal: can we extend an ML-like functional programming language in order to allow a first-class treatment to data structures containing bindings? We are focused on the specification of computation in a traditional sense: the concerns of supporting deduction (mechanized metatheory) are not an immediate goal. As a result, we have not attempted to build into this language some specific notion of context. For us, one criterion of a successful design would be that programs written on data structures without binding should be seen as programs in the original, non-extended programming language.

Accompanying this progress report is the `TryMLTS` website [9] which contains the following: a full length draft report [8] that describes MLTS in more detail, the full sources (λProlog, OCaml, JavaScript) as a git repository, and a web-browser-based environment in which one can create and execute MLTS programs online without needing to install any software. In our overview of MLTS, we repeat some of the material from [8] in sections 2 and 3

Submitted to:
LFMTP 2018

2 Design principles and new features

The programming language MLTS extends (the core of) the OCaml programming language with a treatment of the λ*-tree syntax* approach to encoding data structures containing binders [15]. Briefly, the λ-tree syntax approach to bindings involves the following three tenets: (1) Syntax is encoded as *untyped* λ-terms (although simple types can be used, for example, to distinguish different syntactic categories). (2) Equality of syntax must include α, β_0, and η conversion.[1] (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* [14]. MLTS is, in fact, an acronym for *mobility* and λ*-tree syntax*. (We do not use the term *higher-order abstract syntax* since the literature of functional programming and constructive type theory has used that term to also refer to the mapping of binding structures into function spaces [3, 5, 10], which is a completely different approach to what we do here.) Note that the intent of the first tenet mentioned above is to *de-emphasize* typing in the core notion of λ-tree syntax: doing so allows it to be adapted to many different typing disciplines.

Our strategy for strengthening the expressiveness of ML-style languages has been to add to the language more binding sites to which bindings can move: in fact, MLTS contains three new binders to support the mobility needed for λ-tree syntax. In particular, MLTS has a syntax similar to that of OCaml [18] except that we add to OCaml the following four new features.

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

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

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

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

All three bindings expressions—`(X\body)`, `(new X in body)` and `(nab X in rule)`—are subject to alphabetic renaming of bound variables, just as the names of variables bound in `let` declarations and function definitions. Since nominals are best thought of as constructors, we follow the OCaml convention of capitalizing their names. We are assuming that in all parts of MLTS, the names of nominals (of bound variables in general) are not available to programs since 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.

[1] By β_0 conversion we mean the subset of β-conversion where β-redexes $((\lambda x.B)t)$ are restricted so that t is a bound variable that is not free in $\lambda x.B$ [13].

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

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;;
```

Figure 1: An MLTS program for computing the size of untyped λ-terms along with a series of expressions that presents the naive computation that the size of a particular λ-term is 5.

The restriction on the structure of expressions of the form `(m @ X1 ... Xj)` are essentially the same as those required by *pattern unification* (a.k.a. L_λ-unification) [13]: as a result, pattern matching in this setting is a simple generalization of usual first-order matching. Given the η-rule, if `r` is of `=>` type, the expressions `r` and `(X\ r @ X)` are interchangeable.

3 Some MLTS example programs

We now present several examples of MLTS programs in this section. All of these examples are available via the website [9].

Untyped λ-terms can be defined in MLTS as the following datatype:

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

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

The MLTS program in Figure 1 computes the size of an untyped λ-term and shows a sequence of rewritings that provides a naive model of how that function computes. For example, the expression `(size (Abs(X\ (Abs(Y\ App(X, Y))))))` evaluates to 5 (see Figure 1). In the second match rule in the definition of `size`, 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 constructor of type `tm`. The first call to `size` will bind the pattern variable `r` to `(X\ (Abs (Y\ (App(X,Y)))))`. The third match rule contains the `nab` binder that allows the token `X` to match any nominal. Note that in the evaluation (rewriting) steps denoting a `size` computation, no bound variable actually becomes free: instead, the binding within `Abs`-terms *move* to the binding in `new` expressions.

As a second example, consider a program that returns the boolean `true` if and only if its argument is a λ-abstraction for which the bound variable is vacuous in its scope; otherwise, it returns `false`. Figure 2 contains three implementations of this boolean-valued function. Note that, in the implementation of `vacp2`, once the outermost match rule determines that the argument is a λ-abstraction, a new nominal is created and used to play the role of the λ-abstracted variable. The internal `aux` function is then defined

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

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

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

Figure 2: Three implementations of the function that determines if its argument is a vacuous λ-term.

to search the body of that λ-abstraction for that new nominal. The third implementation, `vacp3`, is not (overtly) recursive since the entire effort of checking for the vacuous binding can be done during pattern matching. The first match rule of this third implementation is essentially asking the question: is there an instantiation for the (pattern) variable s so that the equation $Abs(\lambda x.s)$ equals t? This question can be posed as asking if the logical formula $\exists s.(Abs(\lambda x.s)) = t$ can be proved. In this latter form, it should be clear that since substitution is intended as a logical operation, the result of substituting for s never allows for variable capture. Hence, every instance of the existential quantifier yields an equation with a left-hand side that is a vacuous abstraction. For this kind of pattern matching to work as described, determining this match requires a recursive analysis of the term t. Of course, if one feels that pattern matching must execute in time independent of the terms being matched, then this use of a pattern variable could easily be made illegal (just as repeated pattern variables are made illegal in most pattern matching mechanisms).

For the third and final example of this section, consider the simple and direct implementation of

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

Figure 3: An implementation of (capture-avoiding) substitution: `subst t x u` returns the result of replacing the nominal `x` in `t` with `u`.

substitution given in Figure 3. There are at least two features of this definition that are worth noting. First, the order of the clauses can be changed without affecting the function's computation. For example, a successful match using the second clause necessarily implies that `X` and `Y` are different nominals: thus, there is no overlap between the first and second clause. Second, this substitution is *capture-avoiding* because the MLTS rewriting mechanism is capture-avoiding: in particular, the binding on `Y` cannot capture any nominals that may appear in the argument `u`. Such rewriting is always possible since α-conversion is available and the choice of `Y` can be picked to avoid all nominals in `u`.

4 Formal specifications of typing and natural semantic evaluation

MLTS programs can be given types in much the same way as other ML-like programming languages are given types. As one expects, λProlog provides a simple implementation of type inference and type checking for MLTS programs. An actual λProlog implementation for type inference is about 50 lines of code (although it does not currently capture full let-polymorphism).

Typing is not used during evaluation but typing is an invariant of evaluation (as is usual for ML-style programming languages). A different kind of typing is, however, important to evaluation. In particular, we need to be able to distinguish between a syntactic category (say, denoting program expressions) and an abstraction of one such category over another. For this we adopt the notion of arity types from Martin-Löf [17]. Essentially, the backslash binder allows to raise the arity of an object by one. Conversely, the `Abs` constructor from Section 3 expects to be applied to an argument that has arity 1 and it returns an expression that is arity 0. If an MLTS program does not manipulate any data structures containing bindings, then all pattern variables in such programs will all have arity 0. Observe that this remains true even for higher-order programs such as `map`, `foldr`, etc: thus, expressions that have higher-order types (using `->`) can still be of arity 0.

The natural semantic specification of MLTS evaluation [8] can be given as a small set of clauses within the $\mathscr{G}$ logic [6]. The `nab` and the `new` bindings correspond naturally to nominal abstraction and to the ∇-quantifier, both of which are available in $\mathscr{G}$. Since λProlog does not contain any feature that corresponds to `nab`, our prototype implementation of natural semantics in λProlog was a bit more involved and is about 250 lines of code. The λProlog implementations of type inference and evaluation are both available from the website [9].

5 Similarities with Abella

It is often the case that a relation specified in, say, Prolog or λProlog, encodes a function. In the paper [7], the authors used focusing to describe how functional computations can be performed using relational specification when those relational specifications are known to compute functions. A different question to consider in this setting, however, is: can we transform the syntax of a relational specification of a function and produce a functional program of that encoded function? For example, one can imagine transforming the usual Prolog [4] `append` relation into a, say, OCaml function for computing the appending of two input lists. A possible additional ingredient to such a transformation might also be a formal proof that the `append` relational specification actually determines a function when we view the first two arguments as inputs and the third argument as output. We shall not attempt to formalize such a transformation here. Instead we consider how a naive approach to transforming, say, Prolog to OCaml might be extended in order to transform some Abella [1, 2] specification to MLTS. For the examples below, we make the arbitrary assumption that if a relation is actually a function, it is the last argument of the relation that depends on the other arguments as inputs.

Figure 4 presents Abella relational specifications that correspond closely to functional specifications we have presented in Section 3. Whatever informal understanding one might have of transforming Prolog to OCaml, it seems that the following two observations can be added. If the Abella specification contains `nabla`-in-the-head, then the corresponding clause in the MLTS program contains a `nab` binder. If the Abella specification contains a `nabla` in the body of a clause, then the corresponding clause in the MLTS program contains either a `new` binder or an explicit λ-binder (the backslash).

Although details of pattern matching and the treatment of auxiliary definitions are different between Abella and functional programming, there are a number of similarities. We offer these similarities in part as a way to help motivate the particular choices used in the design of MLTS.

We have left one MLTS program in Section 3 without a corresponding specification in Abella, namely, the `vacp3` function. The most natural Abella specification corresponding to that function would be the following.

```
Define vacp3 : tm -> bool -> prop by
  vacp3 (abs x\T) tt ;
  vacp3 (abs T)   ff ;
  vacp3 (app M N) ff ;
  nabla x, vacp3 x ff.
```

The relation defined here does not, in fact, encode a function since the first two clauses overlap. For example, it is possible to prove that this `vacp3` relation is multi-valued.

```
Theorem vacp3-multivalued :
  forall B, vacp3 (abs x\ abs y\y) B -> (B = tt \/ B = ff).
  intros. case H1. left. search. right. search.
```

It seems the only way to specify the boolean valued function for determining vacuousness is via a recursion over syntax.

6 Conclusion and future work

In 1990, the second author proposed an extension to ML, called ML_λ [12]. Although there is some overlap between that early proposal and the one for MLTS, ML_λ did not have two critical features: nominals and `nab` binder in match patterns. The lack of these features greatly weakened that earlier proposal. Recent advances in proof-theoretic treatments of binding have allowed us to provide a greatly improved design.

Our immediate goals with MLTS are two fold. First, we wish to develop more examples in the metaprogramming areas of automating logic and building compilers. Second, we wish to explore whether or not an abstract machine such as the SECD machine [11] can be built to provide an effective model of computation for MLTS.

One of the things we will most certainly encounter with applying MLTS to metaprogramming tasks is the need to encode and manipulate contexts. Currently, contexts are treated like any other data structures: list of nominals, list of pairs of nominals and type expression, etc. Almost certainly, we will find that many different kinds of contexts will share a number of features that we might wish to incorporate into MLTS: we plan to consider such extensions to the language as we gain familiarity with such applications.

```
Define size : tm -> nat -> prop by
  size (app M N) (s S) :=
            exists S1 S2, size M S1 /\ size N S2 /\ plus S1 S2 S;
  size (abs R) (s S) := nabla x, size (R x) S;
  nabla x, size x (s z).

Define vacp1 : tm -> bool -> prop by
  vacp1 (abs x\x) ff ;
  nabla y, vacp1 (abs x\y) tt ;
  vacp1 (abs x\ app (M x) (N x)) B := exists B1 B2,
        vacp1 (abs M) B1 /\ vacp1 (abs N) B2 /\ and B1 B2 B;
  vacp1 (abs x\ abs y\ R x y) B :=
        nabla y, vacp1 (abs x\ R x y) B ;
  vacp1 (app M N) ff ;
  nabla x, vacp1 x ff.

Define vacp2aux : tm -> tm -> bool -> prop by
  nabla x, vacp2aux x x ff ;
  nabla x y, vacp2aux x y tt ;
  nabla x, vacp2aux x (app (M x) (N x)) B := exists B1 B2,
     nabla x, (vacp2aux x (M x) B1 /\ vacp2aux x (N x) B2) /\
     and B1 B2 B;
  nabla x, vacp2aux x (abs y\ R x y) B :=
     nabla x y, vacp2aux x (R x y) B.

Define vacp2 : tm -> bool -> prop by
  vacp2 (abs T) B := nabla x, vacp2aux x (T x) B ;
  vacp2 (app M N) ff ;
  nabla x, vacp2 x ff.

Define subst : tm -> tm -> tm -> tm -> prop by
nabla x,   subst x x U U;
nabla x y, subst y x (U y) y;
nabla x,   subst (app (M x) (N x)) x U (app SM SN) :=
    nabla x, subst (M x) x U SM /\ nabla x, subst (N x) x U SN;
nabla x,   subst (abs (R x)) x U (abs SR) :=
    nabla x y, subst (R x y) x (U y) (SR y).
```

Figure 4: The Abella specification of four relations, which when viewed as MLTS could match the specifications in Section 3.

References

[1] (2012): *The Abella Prover*. Available at `http://abella-prover.org/`.

[2] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu & Yuting Wang (2014): *Abella: A System for Reasoning about Relational Specifications*. *Journal of Formalized Reasoning* 7(2), doi:10.6092/issn.1972-5787/4650. Available at `http://jfr.unibo.it/article/download/4650/4137`.

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

[4] W. F. Clocksin & Chris Mellish (1987): *Programming in Prolog, 3rd Edition*. Springer.

[5] Joëlle Despeyroux, Amy Felty & Andre Hirschowitz (1995): *Higher-order abstract syntax in Coq*. In: *Second International Conference on Typed Lambda Calculi and Applications*, pp. 124–138.

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

[7] Ulysse Gérard & Dale Miller (2017): *Separating Functional Computation from Relations*. In Valentin Goranko & Mads Dam, editors: *26th EACSL Annual Conference on Computer Science Logic (CSL 2017)*, *LIPIcs* 82, pp. 23:1–23:17, doi:10.4230/LIPIcs.CSL.2017.23.

[8] Ulysse Gérard & Dale Miller (2018): *Functional programming with λ-tree syntax: Draft*. Available from `https://trymlts.github.io/`.

[9] Ulysse Gérard & Dale Miller (2018): *TryMLTS*. `https://trymlts.github.io/`.

[10] Martin Hofmann (1999): *Semantical analysis of higher-order abstract syntax*. In: *14th Symp. on Logic in Computer Science*, IEEE Computer Society Press, pp. 204–213.

[11] P. J. Landin (1964): *The Mechanical Evaluation of Expressions*. *Computer Journal* 6(5), pp. 308–320.

[12] 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, pp. 323–335. Available at `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/mll.pdf`. Available as UPenn CIS technical report MS-CIS-90-59.

[13] Dale Miller (1991): *A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification*. *J. of Logic and Computation* 1(4), pp. 497–536.

[14] Dale Miller (2004): *Bindings, mobility of bindings, and the ∇-quantifier*. In Jerzy Marcinkowski & Andrzej Tarlecki, editors: *18th International Conference on Computer Science Logic (CSL) 2004*, *LNCS* 3210, p. 24.

[15] Dale Miller & Gopalan Nadathur (2012): *Programming with Higher-Order Logic*. Cambridge University Press, doi:10.1017/CBO9781139021326.

[16] Aleksandar Nanevski, Frank Pfenning & Brigitte Pientka (2008): *Contextual Model Type Theory*. *ACM Trans. on Computational Logic* 9(3), pp. 1–49.

[17] Bengt Nordstrom, Kent Petersson & Jan M. Smith (1990): *Programming in Martin-Löf's type theory : an introduction*. International Series of Monographs on Computer Science, Oxford: Clarendon.

[18] OCaml (2018): `http://ocaml.org/`.

[19] Brigitte Pientka & Joshua Dunfield (2010): *Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)*. In J. Giesl & R. Hähnle, editors: *Fifth International Joint Conference on Automated Reasoning*, *LNCS* 6173, pp. 15–21.