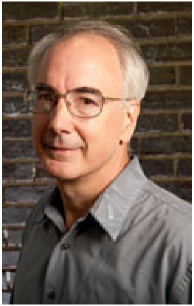
Functional programming with λ−tree syntax

Ulysse Gérard, Dale Miller, **Gabriel Scherer**

Parsifal, Inria Saclay, France

HOPE, September 23rd, 2018


Introduction

MLTS is an ongoing language design experiment. WIP!
Extend ML with binder handling constructs from λProlog and Abella.

Theory: in logic programming, computation from *proof search*.
Binders: a new *quantifier* in the logic: ∇x, "for a fresh x".

Implementation: online, compiles to λProlog.
`https://voodoos.github.io/mlts/`

Look and feel: a funny mix of FreshML and HOAS.
Mobility and λ-Tree Syntax.

MLTS: datatypes with binders

MLTS extends ML with binders.

Normal ML datatypes are *closed*.

Example of *open* datatype:

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

(notice: no constructor for variables)

MLTS: datatypes with binders

MLTS extends ML with binders.

Normal ML datatypes are *closed*.

Example of *open* datatype:

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

(notice: no constructor for variables)
Inhabitants:

$\lambda x.\ x$	`Abs(X \ X)`
$\lambda x.\ (x\ x)$	`Abs(X \ App(X, X))`
$(\lambda x.\ x)\ (\lambda x.\ x)$	`App(Abs(X \ X), Abs(X \ X))`

MLTS crash course

`subst : lam -> lam -> lam`

`subst t x u` is $t[x\backslash u]$.

```
let rec subst t x u = match (x, t) with
```

MLTS crash course

`subst : lam -> lam -> lam`

`subst t x u` is $t[x\backslash u]$.

`nab X in (X, X)` will only match if `x = t = X` is a nominal.

```
let rec subst t x u = match (x, t) with
| nab X in (X, X) -> u
```

MLTS crash course

`subst : lam -> lam -> lam`

`subst t x u` is $t[x \backslash u]$.

`nab X Y in (X, Y)` will only match two distinct nominals.

```
let rec subst t x u = match (x, t) with
| nab X in (X, X) -> u
| nab X Y in (X, Y) -> Y
```

MLTS crash course

`subst : lam -> lam -> lam`

`subst t x u` is $t[x\backslash u]$.

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

MLTS crash course

`subst : lam -> lam -> lam`

`subst t x u` is $t[x\backslash u]$.

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

MLTS crash course

`subst : lam -> lam -> lam`

`subst t x u` is $t[x\backslash u]$.

`r : lam => lam`

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

MLTS crash course

`subst : lam -> lam -> lam`

`subst t x u` is $t[x \backslash u]$.

`r : lam => lam`

`r @ Y : lam`

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

MLTS crash course

```
subst : lam -> lam -> lam
```

`subst t x u` is $t[x\backslash u]$.

```
r : lam => lam
(Y\ r @ Y) : lam => lam
```

```
r @ Y : lam
```

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

MLTS crash course

`subst : lam -> lam -> lam`

`subst t x u` is $t[x\backslash u]$.

`r : lam => lam`

`(Y\ r @ Y) : lam => lam`

`r @ Y : lam`

`Abs(Y\ r @ Y) : lam`

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

MLTS crash course

`subst : lam -> lam -> lam`

`subst t x u` is $t[x\backslash u]$.

In `Abs(Y \ subst (r @ Y) x u)`, no variable is ever free.
Binders move.

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

Binder type

`(a => b)`: "open" values of type `b` under a binder of type `a`.

introduction `X \ t`, elimination `t @ X`.

Binder type

(a => b): "open" values of type b under a binder of type a.
introduction X \ t, elimination t @ X.

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

(X \ X)                 : lam => lam
Abs(X \ App(X, X)) : lam
(fun r -> Abs(Y \ App(r @ Y, r @ Y))
                        : (lam => lam) -> lam
```

Binder type

(a => b): "open" values of type b under a binder of type a.
introduction X \ t, elimination t @ X.

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

(X \ X)                : lam => lam
Abs(X \ App(X, X))     : lam
(fun r -> Abs(Y \ App(r @ Y, r @ Y))
                       : (lam => lam) -> lam
```

$$\frac{\Gamma, X : A \vdash t : B}{\Gamma \vdash X \backslash t : A \Rightarrow B} \qquad \frac{\Gamma \vdash t : A \Rightarrow B \quad (X : A) \in \Gamma}{\Gamma \vdash t \mathbin{@} X : B}$$

Binder type

(a => b): "open" values of type b under a binder of type a.
introduction X \ t, elimination t @ X.

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

(X \ X)                 : lam => lam
Abs(X \ App(X, X))      : lam
(fun r -> Abs(Y \ App(r @ Y, r @ Y))
                        : (lam => lam) -> lam
```

$$\frac{\Gamma, X : A \vdash t : B}{\Gamma \vdash X \backslash t : A \Rightarrow B} \qquad \frac{\Gamma \vdash t : A \Rightarrow B \quad (X : A) \in \Gamma}{\Gamma \vdash t \mathbin{@} X : B}$$

(and in patterns)

`new` binder?

How to perform that substitution : $(\lambda y.\ y\ x)[x \backslash \lambda z.\ z]$?

`new` binder?

How to perform that substitution : $(\lambda y.\ y\ x)[x\backslash\lambda z.\ z]$?

```
subst (Abs(Y\ App(Y, ?))) ? (Abs(Z\ Z));;
```

`new` binder?

How to perform that substitution : $(\lambda y.\ y\ x)[x \backslash \lambda z.\ z]$?

```
subst (Abs(Y\ App(Y, ?))) ? (Abs(Z\ Z));;
```

We need a way to introduce a nominal to call `subst`.

new binder?

How to perform that substitution : $(\lambda y.\ y\ x)[x \backslash \lambda z.\ z]$?

```
subst (Abs(Y\ App(Y, ?))) ? (Abs(Z\ Z));;
```

We need a way to introduce a nominal to call `subst`.

```
new X in subst (Abs(Y\ App(Y, X))) X (Abs(Z\ Z));;
```

`new` binder?

How to perform that substitution : $(\lambda y.\ y\ x)[x \backslash \lambda z.\ z]$?

```
subst (Abs(Y\ App(Y, ?))) ? (Abs(Z\ Z));;
```

We need a way to introduce a nominal to call `subst`.

```
new X in subst (Abs(Y\ App(Y, X))) X (Abs(Z\ Z));;
  ⟶   Abs(Y\ App(Y, Abs(Z\ Z)))
```

`new` binder?

How to perform that substitution : $(\lambda y.\ y\ x)[x \backslash \lambda z.\ z]$?

```
subst (Abs(Y\ App(Y, ?))) ? (Abs(Z\ Z));;
```

We need a way to introduce a nominal to call `subst`.

```
new X in subst (Abs(Y\ App(Y, X))) X (Abs(Z\ Z));;
  ⟶   Abs(Y\ App(Y, Abs(Z\ Z)))
```

`new X in`: a scope in which a new nominal `X` is available.

`new` binder?

How to perform that substitution : $(\lambda y.\ y\ x)[x \backslash \lambda z.\ z]$?

```
subst (Abs(Y\ App(Y, ?))) ? (Abs(Z\ Z));;
```

We need a way to introduce a nominal to call `subst`.

```
new X in subst (Abs(Y\ App(Y, X))) X (Abs(Z\ Z));;
  ⟶   Abs(Y\ App(Y, Abs(Z\ Z)))
```

`new X in`: a scope in which a new nominal `X` is available.

Effect: escape checking / occurs check.
(Safer when returning a closed type.)

Pure substitution

$$\frac{\Gamma, x \vdash t \qquad \Gamma \vdash u}{\Gamma \vdash t[u/x]}$$

```
let rec subst (t : lam => lam) (u : lam) : lam =
  match t with
```

Pure substitution

$$\frac{\Gamma, x \vdash t \qquad \Gamma \vdash u}{\Gamma \vdash t[u/x]}$$

```
let rec subst (t : lam => lam) (u : lam) : lam =
  match t with
  | X \ X ->
      u
```

Pure substitution

$$\frac{\Gamma, x \vdash t \qquad \Gamma \vdash u}{\Gamma \vdash t[u/x]}$$

```
let rec subst (t : lam => lam) (u : lam) : lam =
  match t with
  | X \ X ->
      u
  | nab Y in (X \ Y) ->
      Y
```

Pure substitution

$$\frac{\Gamma, x \vdash t \qquad \Gamma \vdash u}{\Gamma \vdash t[u/x]}$$

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

Pure substitution

$$\frac{\Gamma, x \vdash t \qquad \Gamma \vdash u}{\Gamma \vdash t[u/x]}$$

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

Beta reduction

```
let rec beta t =
  match t with
  | nab X in X -> X
```

Beta reduction

```
let rec beta t =
  match t with
  | nab X in X -> X
  | Abs r -> Abs (Y\ beta (r @ Y))
```

Beta reduction

```
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
```

Beta reduction

```
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)
```

Beta reduction

```
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)
```

Beta reduction

```
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
;;
```

Pattern matching

Unification modulo α, β_0 and η.
β_0: $(\lambda x.B)y = B[y/x]$ provided y is not free in $\lambda x.B$

Pattern matching

Unification modulo α, β_0 and η.
β_0: $(\lambda x.B)y = B[y/x]$ provided y is not free in $\lambda x.B$

Implied restrictions:

- Applications lists are distinct nominals.
  (`nab X1 X2 in C(r @ X1 X2) -> ...`).
- In `r @ X`, the nominal `X` is not free in `r`.

Pattern matching

Unification modulo α, β_0 and η.
β_0: $(\lambda x.B)y = B[y/x]$ provided y is not free in $\lambda x.B$

Implied restrictions:

- Applications lists are distinct nominals.
  (`nab X1 X2 in C(r @ X1 X2) -> ...`).
- In `r @ X`, the nominal `X` is not free in `r`.

```
| X \ App (m @ X, n @ X) ->
    App (subst m u, subst n u)
```

Pattern matching

Unification modulo α, β_0 and η.
β_0: $(\lambda x.B)y = B[y/x]$ provided y is not free in $\lambda x.B$

Implied restrictions:

- Applications lists are distinct nominals. (`nab X1 X2 in C(r @ X1 X2) -> ...`).
- In `r @ X`, the nominal `X` is not free in `r`.

```
| X \ App (m @ X, n @ X) ->
    App (subst m u, subst n u)
```

This is called higher-order pattern unification.
Decidable, most general unifiers.

Interpreter in λProlog: just ML

ML admits type-erasure:

can define an operational semantics on untyped terms.

$\Longrightarrow$ untyped interpreter in λ-prolog, all ML types map to tm.

```
kind tm         type.
type app        tm -> tm -> tm.
type lam        (tm -> tm) -> tm.
type let        tm -> (tm -> tm) -> tm.
type match      tm -> clauses -> tm.

type K          tm -> ... -> tm -> tm.

type cp         tm -> tm -> prop.
type eval       tm -> tm -> prop.

eval (let Def Body) VB :-
  eval Def VD,
  eval (Body VD) VB.
```

Interpreter in λProlog: MLTS

To extend to MLTS,

$$\texttt{transl(a => b)} = \texttt{tm -> transl(b)} \qquad \texttt{transl(\_)} = \texttt{tm}$$

Interpreter in λProlog: MLTS

To extend to MLTS,

transl(`a => b`) = `tm -> ` transl(`b`)   transl(`_`) = `tm`

transl(`X \ t`) = `x \ ` transl(`t`)   transl(`t @ x`) = transl(`t`) `x`

Interpreter in λProlog: MLTS

To extend to MLTS,

$$\text{transl}(\texttt{a => b}) = \texttt{tm -> } \text{transl}(\texttt{b}) \qquad \text{transl}(\_) = \texttt{tm}$$

$$\text{transl}(\texttt{X \textbackslash{} t}) = \texttt{x \textbackslash{} } \text{transl}(\texttt{t}) \qquad \text{transl}(\texttt{t @ x}) = \text{transl}(\texttt{t})\ \texttt{x}$$

```
type new_τ    (tm -> τ) -> τ.
```

Interpreter in λProlog: MLTS

To extend to MLTS,

$$\text{transl}(\texttt{a => b}) = \texttt{tm -> } \text{transl}(\texttt{b}) \qquad \text{transl}(\_) = \texttt{tm}$$

$$\text{transl}(\texttt{X \textbackslash\ t}) = \texttt{x \textbackslash\ } \text{transl}(\texttt{t}) \qquad \text{transl}(\texttt{t @ x}) = \text{transl}(\texttt{t})\ \texttt{x}$$

```
type new_τ    (tm -> τ) -> τ.
```

```
eval (new_τ T) V :- pi x \ eval (T x) V.
```

Demo time?

Implementation by Ulysse Gérard.

Technology: Menhir + his code + Elpi + `js_of_ocaml` + Nice web stuff.

Conclusion & Future work

- This treatment of bindings has a clean semantic inspired by Abella.
- The interpreter was quite simple to write : ≈140 lines of code

Future work:

- More examples in the meta-programming area (a compiler ?)
- Provide an operational semantics (small-step?) without primitive binding constructs.
- Statics checks such as pattern matching exhaustivity, use of distinct pattern variables in pattern application, nominals escaping their scope, etc.
- Design a "real" implementation. A compiler ? An extension to OCaml ? An abstract machine ?

https://trymlts.github.io

Thank you

Concrete syntax typing rules (1/2)

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

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

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

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

Concrete syntax typing rules (2/2)

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

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

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

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

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

Natural semantics for the abstract syntax ($\mathcal{G}$-logic [Gacek, 2009, Gacek et al., 2011]) (1/2)

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

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

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

Natural semantics for the abstract syntax (2/2)

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

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

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

$$\frac{\dfrac{\vdash \lambda X.(X \Longrightarrow s) \trianglerighteq (Y \Longrightarrow U)}{\vdash \text{pattern}\ \ Y\ \ (\text{nab}\ X\ \text{in}\ (X \Longrightarrow s))\ \ U} \qquad \vdash U \Downarrow V}{\vdash \text{match}\ \ Y\ \ \text{with}\ \ \ (\text{nab}\ X\ \text{in}\ (X \Longrightarrow s)) \Downarrow V}$$

Gacek, A. (2009).
A Framework for Specifying, Prototyping, and Reasoning about Computational Systems.
PhD thesis, University of Minnesota.

Gacek, A., Miller, D., and Nadathur, G. (2011).
Nominal abstraction.
Information and Computation, 209(1):48–73.

Miller, D. and Nadathur, G. (2012).
Programming with Higher-Order Logic.
Cambridge University Press.

Miller, D. and Palamidessi, C. (1999).
Foundational aspects of syntax.
ACM Computing Surveys, 31.

Nordstrom, B., Petersson, K., and Smith, J. M. (1990).
Programming in Martin-Löf's type theory : an introduction.
International Series of Monographs on Computer Science. Oxford: Clarendon.