Functional programming with $\lambda$–tree syntax

Ulysse Gérard and Dale Miller
Gallium Seminar, March 12, 2018

Inria Saclay
Palaiseau France
Introduction: the context

Functional programming (FP) languages are popular tools to build systems (parsers, compilers, theorem provers...) that manipulate the syntax of various programming languages and logics.
Functional programming (FP) languages are popular tools to build systems (parsers, compilers, theorem provers...) that manipulate the syntax of various programming languages and logics.

Variable binding is a common denominator of these objects.

But only few FP languages natively provide constructs to handle them. However a number of libraries exists along with first class extensions.

Libs: AlphaLib, Cαml

Languages: FreshML, Beluga...
Successful efforts in the logic programming world, using an elegant mixing of λ-terms and higher-order logic: \(\lambda\)-tree syntax.
Successful efforts in the logic programming world, using an elegant mixing of λ-terms and higher-order logic: \(\lambda\)-tree syntax.

We describe a new FP language, MLTS, based on these techniques.

Work in progress / Preliminary work
The substitution case

Our sample example: substitution

\[
\text{val subst : term } -> \text{ var } -> \text{ term } -> \text{ term}
\]

Such that “\text{subst t x u}” is \( t[x/u] \).
A simple way to handle bindings in vanilla OCaml is to use strings to represent variables:

```
type tm =
    | Var of string
    | App of term * term
    | Abs of string * term
```
Handmade: The "naive" way...

A simple way to handle bindings in vanilla OCaml is to use strings to represent variables:

```ocaml
type tm =
  | Var of string
  | App of term * term
  | Abs of string * term
```

And then proceed recursively:

```ocaml
let rec subst t x u = match t with
  | Var y -> if x = y then u else Var y
  | App(m, n) -> App(subst m x u, subst n x u)
  | Abs(y, body) -> ?
```
Handmade: ...the painful way

\[
\mid \text{Abs}(y, \text{body}) \rightarrow \\
\text{if } (x \neq y) \text{ then} \\
\qquad \text{Abs}(y, \text{body}) \\
\text{else } \text{Abs}(y, \text{subst body } x u)
\]
Handmade: ...the painful way

| Abs(y, body) ->
  | if (x = y) then
  |   Abs(y, body)
  | else Abs(y, subst body x u)

And what if t contains y? y instances in t would be captured.

We need to check for free variables in t and rename them if necessary...
There are several approaches to handle bindings:

- Var as strings
- De Bruijn’s nameless dummies [de Bruijn, 1979]

But they all need to be carefully implemented.
There are several approaches to handle bindings:

- Var as strings
- De Bruijn’s nameless dummies [de Bruijn, 1979]

But they all need to be carefully implemented.

Can we automate this tedious and pervasive task?
There are several approaches to handle bindings:

- Var as strings
- De Bruijn’s nameless dummies [de Bruijn, 1979]

But they all need to be carefully implemented.

Can we automate this tedious and pervasive task?

Caml [Pottier, 2006]
Cαml is a tool that generates an OCaml module to manipulate datatypes with binders. (example from the Little Calculist blog)

```ocaml

sort var

type tm =
  | Var of atom var
  | App of tm * tm
  | Abs of < lambda >

type lambda binds var = atom var * inner tm
```

let rec subst t x u =
    match t with
    | Var y -> if Var.Atom.equal x y then u else Var y
    | App(m, n) -> App (subst m x u, subst n x u)
    | Abs abs ->
        let x', body = open_lambda abs in
        Abs (create_lambda (x', subst body x u))
MLTS version of subst

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

;;
```
MLTS version of subst

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

;;

Some inhabitants:

\[ \lambda x. x \]
\[ \lambda x. (x x) \]
\[ (\lambda x. x) (\lambda x. x) \]
\[ \text{Abs}(X \backslash X) \]
\[ \text{Abs}(X \backslash \text{App}(X, X)) \]
\[ \text{App}(\text{Abs}(X \backslash X), \text{Abs}(X \backslash X)) \]
```
MLTS version of subst

... 

let rec subst t x u =
    match (x, t) with
MLTS version of subst

... 

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

nab X Y in (X, Y) will only match for 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
  | (x, App(m, n)) ->
    App(subst m x u, subst n x u)
In `Abs(Y \ subst (r @ Y) x u)`, the abstraction is opened, modified and rebuilt without ever freeing any bound variable.
How to perform that substitution: \((\lambda y. y x)[x \backslash \lambda z. z]\)?
MLTS version of subst

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

\[
\text{subst (Abs(Y \ App(Y, \ ?))) ? (Abs(Z \ Z))} ; ;
\]
How to perform that substitution: \((\lambda y. y x)[x\backslash \lambda z. z]\)?

\[\text{subst (Abs(Y\ App(Y, ?))) ? (Abs(Z\ Z))};\]

We need a way to introduce a nominal to call subst.
How to perform that substitution: \((\lambda y. y x)[x\lambda z. z]\)?

\[
\text{subst } (\text{Abs}(Y\ App(Y, ?))) ? (\text{Abs}(Z\ Z));
\]

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

\[
\text{new } X \text{ in subst } (\text{Abs}(Y\ (\text{App}(Y, X)))) X (\text{Abs}(Z\ Z));
\]
How to perform that substitution: \((\lambda y. y \, x)[x \backslash \lambda z. \, z]\)?

\[
\text{subst } (\text{Abs}(Y \backslash \text{App}(Y, \, ?))) \, ? \, (\text{Abs}(Z \backslash \, Z));;
\]

We need a way to introduce a nominal to call \text{subst}.

\[
\text{new } X \, \text{in } \text{subst } (\text{Abs}(Y \backslash (\text{App}(Y, \, X)))) \, X \, (\text{Abs}(Z \backslash \, Z));;
\rightarrow \quad \text{Abs}(Y \backslash \text{App}(Y, \, \text{Abs}(Z \backslash \, Z)))
\]
In order to formalize MLTS, we need to introduce a very simple type system called **Arity typing** due to Martin-Löf [Nordstrom et al., 1990]. Arity types for MLTS are either:

- The primitive arity 0
- An expression of the form $0 \rightarrow \cdots \rightarrow 0$
In order to formalize MLTS, we need to introduce a very simple type system called Arity typing due to Martin-Löf [Nordstrom et al., 1990]. Arity types for MLTS are either:

- The primitive arity 0
- An expression of the form $0 \rightarrow \cdots \rightarrow 0$

The primitive type is used to denote most programming language expressions and phrases. The type $0 \rightarrow \cdots \rightarrow 0$, with $n + 1$ occurrences of 0, is the type used to denote the “syntactic category of an $n$-ary abstraction”.

MLTS features: =>, backslash and at

The type constructor => is used to declare bindings (of non-zero arity) in datatypes.

Example: ((X\ body) @ Y) denotes the result of instantiating the abstracted nominal X with the nominal Y in body.
The type constructor => is used to declare bindings (of non-zero arity) in datatypes.

The infix operator \ introduces an abstraction of a nominal over its scope. Such an expression is applied to it arguments using @, thus eliminating the abstraction.

\[
\begin{align*}
\Gamma \vdash t : B & \quad \text{and} \quad \Gamma, X : A \vdash t : B \\
\Gamma \vdash X \backslash t : A \Rightarrow B & \quad \text{and} \quad \Gamma \vdash t : A \Rightarrow B \quad (X : A) \in \Gamma \\
\Gamma \vdash t @ X : B &
\end{align*}
\]
MLTS features: $\Rightarrow$, backslash and at

The type constructor $\Rightarrow$ is used to declare bindings (of non-zero arity) in datatypes.

The infix operator $\backslash$ introduces an abstraction of a nominal over its scope. Such an expression is applied to it arguments using $\@$, thus eliminating the abstraction.

\[
\frac{\Gamma, X : A \vdash t : B}{\Gamma \vdash X \backslash t : A \Rightarrow B} \quad \frac{\Gamma \vdash t : A \Rightarrow B \quad (X : A) \in \Gamma}{\Gamma \vdash t @ X : B}
\]

Example

\(((X \backslash \text{body}) @ Y)\) denotes the result of instantiating the abstracted nominal $X$ with the nominal $Y$ in body.
MLTS features: new and nab

The new X in binding operator provides a scope within expressions in which a new nominal X is available.
The `new X in` binding operator provides a scope within expressions in which a new nominal X is available.

Patterns can contain the `nab X in` binder: in its scope the symbol X can match constructors introduced by `new` and `\`. 

Pattern variables can have non-zero arity and they can be applied (using `@`) to an argument list that consists of distinct variables that are bound in the scope of pattern variables:

Abs(X r @ X)
The *new X in* binding operator provides a scope within expressions in which a new nominal $X$ is available.

Patterns can contain the *nab X in* binder: in its scope the symbol $X$ can match constructors introduced by *new* and $\backslash$.

Pattern variables can have non-zero arity and they can be applied (using @) to an argument list that consists of distinct variables that are bound in the scope of pattern variables:

$$\text{Abs}(X\backslash r \ @ X)$$
The **new** X **in** binding operator provides a scope within expressions in which a new nominal X is available.

Patterns can contain the **nab** X **in** binder: in its scope the symbol X can match constructors introduced by **new** and \.

Pattern variables can have non-zero arity and they can be applied (using @) to an argument list that consists of distinct variables that are bound in the scope of pattern variables:

\[
\text{Abs}(X\, r \, @ \, X) \\
\exists r. \text{Abs}(X\, r \, @ \, X)
\]
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 ->
    new X in beta (subst (r @ X) X n)
  | _ -> App(m, n)
  end
;;
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)))
| _                                -> false ;;
One more example: vacuosity

```plaintext
let vacuous t = match t with
| Abs(X\s)  -> true
| _        -> false ;;
```
let vacuous $t = \text{match } t \text{ with}$

<table>
<thead>
<tr>
<th>$\text{Abs}(X \backslash s)$</th>
<th>$\rightarrow$</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
<td>$\rightarrow$</td>
<td>false</td>
</tr>
</tbody>
</table>

$\text{match } t \text{ with } \text{Abs}(X \backslash s) \equiv \exists s. (\lambda x.s) = t$

(Recursion is hidden in the matching procedure)
We perform unification modulo $\alpha$, $\beta_0$ and $\eta$.

$\beta_0$: $(\lambda x. B)y = B[y/x]$ provided $y$ is not free in $\lambda x. B$ (or alternatively $(\lambda x. B)x = B$)
We perform unification modulo $\alpha$, $\beta_0$ and $\eta$.

$\beta_0$: $(\lambda x. B)y = B[y/x]$ provided $y$ is not free in $\lambda x. B$ (or alternatively $(\lambda x. B)x = B$

We give ourselves the following restrictions:

- Pattern variables are applied to at most a list of distinct variables.
- These variables are bound in the scope of pattern variables. $(\text{In } (x \ @ \ X \ Y) \text{ The scope of } X \text{ and } Y \text{ must be inside the scope of } r.)$
Pattern matching

We perform unification modulo $\alpha$, $\beta_0$ and $\eta$.

$\beta_0$: $(\lambda x. B)y = B[y/x]$ provided $y$ is not free in $\lambda x. B$ (or alternatively $(\lambda x. B)x = B$

We give ourself the following restrictions:

- Pattern variables are applied to at most a list of distinct variables.
- These variables are bound in the scope of pattern variables. (In $(r \circ X Y)$ The scope of $X$ and $Y$ must be inside the scope of $r$.)

This is called higher-order pattern unification or $L_\lambda$-unification [Miller and Nadathur, 2012].

Such higher-order unification is decidable, unitary, and can be done without typing.
Some matching examples

\[ a : i \quad f : i \rightarrow i \quad g : i \rightarrow i \rightarrow i \]

(1) \[ \lambda x \lambda y (f \ (H \ x)) \quad \lambda u \lambda v (f \ (f \ u)) \]
(2) \[ \lambda x \lambda y (f \ (H \ x)) \quad \lambda u \lambda v (f \ (f \ v)) \]
(3) \[ \lambda x \lambda y (g \ (H \ y \ x) \ (f \ (L \ x))) \quad \lambda u \lambda v (g \ u \ (f \ u)) \]
(4) \[ \lambda x \lambda y (g \ (H \ x) \ (L \ x)) \quad \lambda u \lambda v (g \ (g \ a \ u) \ (g \ u \ u)) \]

(1) \[ H \mapsto \lambda w (f \ w) \]
Some matching examples

\[ a : i \quad f : i \rightarrow i \quad g : i \rightarrow i \rightarrow i \]

(1) \( \lambda x \lambda y (f (H x)) \) \quad \( \lambda u \lambda v (f (f u)) \)

(2) \( \lambda x \lambda y (f (H x)) \) \quad \( \lambda u \lambda v (f (f v)) \)

(3) \( \lambda x \lambda y (g (H y x) (f (L x))) \) \quad \( \lambda u \lambda v (g u (f u)) \)

(4) \( \lambda x \lambda y (g (H x) (L x)) \) \quad \( \lambda u \lambda v (g (g a u) (g u u)) \)

(1) \( H \mapsto \lambda w (f \ w) \)

(2) match failure
Some matching examples

\[a : i \quad f : i \to i \quad g : i \to i \to i\]

(1) \(\lambda x \lambda y(f(H x))\)

(2) \(\lambda x \lambda y(f(H x))\)

(3) \(\lambda x \lambda y(g(H y x) (f (L x)))\)

(4) \(\lambda x \lambda y(g(H x) (L x))\)

(1) \(H \mapsto \lambda w(f w)\)

(2) match failure

(3) \(H \mapsto \lambda y \lambda x.x\) \quad \(L \mapsto \lambda x.x\)

(4) \(H \mapsto \lambda x.(g a x)\) \quad \(L \mapsto \lambda x.(g x x)\)
Our prototype interpreter is written in λProlog. The ocaml-style concrete syntax is translated to a λProlog program which is then evaluated by the interpreter.
Our prototype interpreter is written in λProlog. The ocaml-style concrete syntax is translated to a λProlog program which is then evaluated by the interpreter.

```ml
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)
;;
```
Translation

\begin{verbatim}
prog "subst" (fixpt subst \ lam t \ lam x \ lam u \\
match pair $ x $ t \\
[ nab x \ (pr x x \imp u), \\
nab x \ nab y \ (pr x y \imp y), \\
all x \ all m \ all n \ (pr x (App (pr m n)) \\
\imp (App (pair $ (subst $ m $ x $ u) \\
\quad $ (subst $ n $ x $ u)))), \\
all x \ all 'r \ (pr x (Abs r) \\
\imp (Abs (y \ (subst $ (r y) $ x $ u)))) \\
])
\end{verbatim}
Natural semantics for MLTS

\[\vdash \text{val } V \quad \vdash M \downarrow F \quad \vdash N \downarrow U \quad \vdash \text{apply } F U V \quad \vdash (R(\text{fixpt } R)) \downarrow V\]

\[\vdash V \downarrow V \quad \vdash M \downarrow V \quad \vdash M \downarrow U \quad \vdash (R U) \downarrow V\]

\[\vdash (fixpt R) \downarrow V\]

\[\vdash C \downarrow tt \quad \vdash L \downarrow V \quad C \downarrow ff \quad \vdash C \downarrow ff \quad \vdash M \downarrow V \quad \vdash M \downarrow U \quad (R U) \downarrow V\]

\[\vdash \text{cond } C L M \downarrow V \quad \vdash \text{cond } C L M \downarrow V \quad \vdash \text{cond } C L M \downarrow V \quad \vdash \text{let } M R \downarrow V\]

\[\vdash \nabla x. (E x) \downarrow V \quad \vdash \text{new } E \downarrow V \quad \vdash \text{apply } (\text{lam } R) U V\]

\[\vdash \text{pattern } T \text{ Rule } U \quad \vdash \text{pattern } T \text{ Rule } U \quad \vdash \text{pattern } T \text{ Rule } U \quad \vdash \text{pattern } T \text{ Rule } U \quad \vdash \text{pattern } T \text{ Rule } U \quad \vdash \text{pattern } T \text{ Rule } U \]

\[\vdash \exists x. \text{pattern } T (P x) U \quad \vdash \exists x. \text{pattern } T (P x) U \quad \vdash \exists x. \text{pattern } T (P x) U \quad \vdash \exists x. \text{pattern } T (P x) U \quad \vdash \exists x. \text{pattern } T (P x) U \quad \vdash \exists x. \text{pattern } T (P x) U\]

\[\vdash [\lambda z_1 \ldots \lambda z_m. (t \mapsto s)] \mapsto (T \mapsto U)\]

\[\vdash [\lambda z_1 \ldots \lambda z_m. (t \mapsto s)] \mapsto (T \mapsto U)\]

\[\vdash [\lambda z_1 \ldots \lambda z_m. (t \mapsto s)] \mapsto (T \mapsto U)\]

\[\vdash [\lambda z_1 \ldots \lambda z_m. (t \mapsto s)] \mapsto (T \mapsto U)\]

\[\vdash [\lambda z_1 \ldots \lambda z_m. (t \mapsto s)] \mapsto (T \mapsto U)\]

\[\vdash [\lambda z_1 \ldots \lambda z_m. (t \mapsto s)] \mapsto (T \mapsto U)\]

\[\vdash [\lambda z_1 \ldots \lambda z_m. (t \mapsto s)] \mapsto (T \mapsto U)\]

\[\vdash [\lambda z_1 \ldots \lambda z_m. (t \mapsto s)] \mapsto (T \mapsto U)\]
Nominal abstraction [Gacek et al., 2011]

Let:

- $t$ be a term
- $c_1, \ldots, c_n$ be distinct nominal constants that may occur in $t$
- $y_1, \ldots, y_n$ be distinct variables not occurring in $t$

Such that $y_i$ and $c_i$ have the same type.

Then $\lambda c_1 \ldots \lambda c_n.t$ denotes the term $\lambda y_1 \ldots \lambda y_n.t'$ where $t'$ is the term obtained from $t$ by replacing all $c_i$ by $y_i$. 
Nominal abstraction [Gacek et al., 2011]

Let:

- $t$ be a term
- $c_1, \ldots, c_n$ be distinct nominal constants that may occur in $t$
- $y_1, \ldots, y_n$ be distinct variables not occurring in $t$

Such that $y_i$ and $c_i$ have the same type.

Then $\lambda c_1 \ldots \lambda c_n.t$ denotes the term $\lambda y_1 \ldots \lambda y_n.t'$ where $t'$ is the term obtained from $t$ by replacing all $c_i$ by $y_i$.

**Definition**

Let $s$ and $t$ be terms of types $\tau_1 \to \cdots \to \tau_n \to \tau$ and $\tau$ for $n \geq 0$.

The expression $s \triangleright t$, a nominal abstraction of degree $n$, holds just in the case that $s$ $\lambda$-converts to $\lambda c_1 \ldots c_n.t$ for some nominal constants $c_1, \ldots, c_n$. 
The term on the left of the $\triangleright$ operator serves as a pattern for isolating occurrences of nominal constants.
Examples

The term on the left of the $\triangleright$ operator serves as a pattern for isolating occurrences of nominal constants.

**Example**

For example, if $p$ is a binary constructor and $c_1$ and $c_2$ are nominal constants:

<table>
<thead>
<tr>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x.x \triangleright c_1$</td>
</tr>
<tr>
<td>$\lambda x.x \triangleright p \ c_1 \ c_2$</td>
</tr>
</tbody>
</table>

Nominal abstraction of degree $(n) \ 0$ is the same as equality between terms based on $\lambda$-conversion.
Illustrating the last rule

\[ \vdash \lambda X. (X \rightarrow s) \triangleright (Y \rightarrow U) \]

\[ \vdash \text{pattern } Y \ (\text{nab } X \text{ in } (X \rightarrow s))\ U \vdash U \Downarrow V \]

\[ \vdash \text{match } Y \text{ with } (\text{nab } X \text{ in } (X \rightarrow s)) \Downarrow V \]
Nominals do not escape their scopes

Given the richness of the logic behind the natural semantics, we can prove that nominals do not escape their scope.

\[ \vdash \nabla x. (E \ x) \downarrow V \]

\[ \vdash \text{new } E \downarrow V \]

The universal quantifier \( \forall V \) is outside the scope of \( \nabla x \).
Nominals do not escape their scopes

Given the richness of the logic behind the natural semantics, we can prove that nominals do not escape their scope.

\[ \vdash \nabla x. (E \ x) \Downarrow V \]

\[ \vdash new \ E \Downarrow V \]

The universal quantifier \( \forall V \) is outside the scope of \( \nabla x \).

The \( \lambda \text{Prolog} \) implementation has a cost to make that guarantee: every unification problem, in principle, needs to check for escaping nominals.

Static checks will certainly need to be developed in order to ensure that such checks are not always needed.
The natural semantics is implemented in $\lambda$Prolog by extending an interpreter from the 2012 book by Miller and Nadathur. Type inference was easy to implement in $\lambda$Prolog.
The natural semantics is implemented in $\lambda$Prolog by extending an interpreter from the 2012 book by Miller and Nadathur. Type inference was easy to implement in $\lambda$Prolog.

The parser and transpiler from the concrete syntax to the $\lambda$Prolog code is written in OCaml.

We provide a website for experimenting with MLTS using the Elpi $\lambda$Prolog interpreter compiled to JavaScript thanks to js of OCaml: [https://voodoos.github.io/mlts](https://voodoos.github.io/mlts)
Current implementation

The natural semantics is implemented in $\lambda$Prolog by extending an interpreter from the 2012 book by Miller and Nadathur. Type inference was easy to implement in $\lambda$Prolog.

The parser and transpiler from the concrete syntax to the $\lambda$Prolog code in written in OCaml.

We provide a website for experimenting with MLTS using the Elpi $\lambda$Prolog interpreter compiled to javascript thanks to js_of_ocaml:

[https://voodooos.github.io/mlts](https://voodooos.github.io/mlts)
Future work

- More complex examples
- Subject reduction, progress, etc.
- Statics checks such as pattern matching exhaustivity, use of distinct pattern variables in pattern application etc.
- Make definitive choices about every remaining aspects of this prototype (should we restrict @ to $\beta_0$ reductions? Should constructors introduced by \ always be of zero arity?)
- Design a real implementation. A compiler? An extension to OCaml?
Thank you
let vacp 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(r) -> new Y in aux (r @ X)
in aux (r @ X)
| _ -> false
;;
- The syntax is encoded as simply typed $\lambda$-terms. Syntactic categories are mapped to simple types.
- Equality of syntax is equated to $\alpha$, $\beta_0$, $\eta$ conversion. Often restrictions are in place so that beta-zero will be complete for beta.
- Bound variables never become free, instead, their binding scope can move.

