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.

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.

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...

Introduction: our approach

Successful efforts in the logic programming world, using an elegant mixing of λ-terms and higher-order logic: λ-tree syntax.

Introduction: our approach

Successful efforts in the logic programming world, using an elegant mixing of λ-terms and higher-order logic: λ-tree syntax.

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

Work in progress / Premilinary work

The substitution case

Our sample example: substitution

```
val subst : term -> var -> term -> term
```

Such that "`subst t x u`" is `t[x\u]`.

Handmade: The "naive" way...

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:

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

And then proceed recursively:

```
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

```
| Abs(y, body) ->
  if (x = y) then
    Abs(y, body)
  else Abs(y, 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...

Handmade

There are several approches to handle bindings:

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

But they all need to be carefully implemented.

Handmade

There are several approches 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 ?

Handmade

There are several approches 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 ?

Cαml [Pottier, 2006]

Cαml

Cαml is a tool that generates an OCaml module to manipulate datatypes with binders. (example from the Little Calculist blog)

```
sort var

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

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

Cαml

```
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

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

MLTS version of subst

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

Some 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 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.

MLTS version of subst

...

```
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.

MLTS version of subst

...

```
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 version of subst

...

```
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)
```

In `Abs(Y\ subst (r @ Y) x u)`, the abstraction is opened, modified and rebuilt without ever freeing any bound variable.

MLTS version of subst

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

MLTS version of subst

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

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

MLTS version of subst

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`.

MLTS version of subst

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));;
```

MLTS version of subst

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));;
```

$\longrightarrow$ `Abs(Y\ App(Y, Abs(Z\ Z)))`

Arity typing

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 $0 \to \cdots \to 0$

Arity typing

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 $0 \to \cdots \to 0$

The primitive type is used to denote most programming language expressions and phrases. The type $0 \to \cdots \to 0$, 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.

MLTS features: =>, backslash and at

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.

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

MLTS features: =>, backslash and at

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.

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

Example

`((X\ 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.

MLTS features: new and nab

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 `\`.

MLTS features: new and nab

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)
```

MLTS features: new and nab

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)
```

$\exists r.$ `Abs(X\ r @ X)`

One more example: 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 ->
          new X in beta (subst (r @ X) X n)
      | _ -> App(m, n)
    end
;;
```

One more example: vacuosity more

```
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

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

One more example: vacuosity

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

`match t with Abs(X\s)` $\equiv$ $\exists s.(\lambda x.s) = t$

(Recursion is hidden in the matching procedure)

Pattern matching

We perform unification modulo α, β_0 and η.

β_0: $(\lambda x.B)y = B[y/x]$ provided y is not free in $\lambda x.B$ (or alternatively $(\lambda x.B)x = B$

Pattern matching

We perform unification modulo α, β_0 and η.

β_0: $(\lambda x.B)y = B[y/x]$ provided y is not free in $\lambda x.B$ (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 @ X Y)` The scope of X and Y must be inside the scope of r.)

Pattern matching

We perform unification modulo α, β_0 and η.

β_0: $(\lambda x.B)y = B[y/x]$ provided y is not free in $\lambda x.B$ (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 @ X Y)` The scope of X and Y must be inside the scope of r.)

This is called higher-order pattern unification or L_λ-unification [Miller and Nadathur, 2012].

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

Some matching examples

$$a : i \quad f : i \to i \quad g : i \to i \to i$$

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

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

Some matching examples

$$a : i \quad f : i \to i \quad g : i \to i \to i$$

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

(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)	$\lambda x \lambda y (f\ (H\ x))$	$\lambda u \lambda v (f\ (f\ u))$
(2)	$\lambda x \lambda y (f\ (H\ x))$	$\lambda u \lambda v (f\ (f\ v))$
(3)	$\lambda x \lambda y (g\ (H\ y\ x)\ (f\ (L\ x)))$	$\lambda u \lambda v (g\ u\ (f\ u))$
(4)	$\lambda x \lambda y (g\ (H\ x)\ (L\ x))$	$\lambda u \lambda v (g\ (g\ a\ u)\ (g\ u\ u))$

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

(2) match failure

(3) $H \mapsto \lambda y \lambda x . x \qquad L \mapsto \lambda x . x$

(4) $H \mapsto \lambda x . (g\ a\ x) \qquad L \mapsto \lambda x . (g\ x\ x)$

Translation

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.

Translation

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.

```
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

```
prog "subst" (fixpt subst\ lam t\ lam x\ lam u\
  match pair $ x $ t
  [nab x\ (pr x x ==> u),
   nab x\ nab y\ (pr x y ==> y),
   all x\ all m\ all n\ (pr x (App (pr m n))
    ==> App $ (pair $ (subst $ m $ x $ u)
                    $ (subst $ n $ x $ u))),
   all x\ all' r\ (pr x (Abs r)
    ==> Abs (y\ (subst $ (r y) $ x $ u)))
  ]).
```

Natural semantics for MLTS

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

$$\frac{\vdash C \Downarrow tt \quad \vdash L \Downarrow V}{\vdash cond\ \ C\ \ L\ \ M \Downarrow V} \qquad \frac{\vdash C \Downarrow ff \quad \vdash M \Downarrow V}{\vdash cond\ \ C\ \ L\ \ M \Downarrow V} \qquad \frac{\vdash M \Downarrow U \quad \vdash (R\ \ U) \Downarrow V}{\vdash (let\ \ M\ \ R) \Downarrow V}$$

$$\frac{\vdash \nabla x.(E\ x) \Downarrow V}{\vdash new\ \ E \Downarrow V} \qquad \frac{\vdash (R\ \ U) \Downarrow V}{\vdash apply\ \ (\text{lam}\ \ R)\ \ U\ \ V}$$

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

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

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 τ for $n \geq 0$.

The expression $s \trianglerighteq t$, a nominal abstraction of degree n, holds just in the case that s λ-converts to $\lambda c_1 \ldots c_n.t$ for some nominal constants $c_1, \ldots, c_n$.

Examples

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

Examples

The term on the left of the $\trianglerighteq$ 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:

$$\begin{array}{lll} \lambda x.x \trianglerighteq c_1 & \lambda x.p\ x\ c_2 \trianglerighteq p\ c_1\ c_2 & \lambda x.\lambda y.p\ x\ y \trianglerighteq p\ c_1\ c_2 \\ \lambda x.x \ntrianglerighteq p\ c_1\ c_2 & \lambda x.p\ x\ c_2 \ntrianglerighteq p\ c_2\ c_1 & \lambda x.\lambda y.p\ x\ y \ntrianglerighteq p\ c_1\ c_1 \end{array}$$

Nominal abstraction of degree (n) 0 is the same as equality between terms based on λ-conversion.

Illustrating the last rule

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

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.

$$\frac{\vdash \nabla x.(E\ x) \Downarrow V}{\vdash \textit{new}\ E \Downarrow V}$$

The universal quantifier $\forall V$ is outside the scope of ∇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.

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

The universal quantifier $\forall V$ is outside the scope of ∇x.

The λProlog 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.

Current implementation

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

Current implementation

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

The parser and transpiler from the concrete syntax to the λProlog code in written in OCaml.

Current implementation

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

The parser and transpiler from the concrete syntax to the λProlog code in written in OCaml.

We provide a website for experimenting with MLTS using the Elpi λProlog interpreter compiled to javascript thanks to js_of_ocaml:

`https://voodoos.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 β_0 reductions ? Should constructors introduced by `\` always be of zero arity ?)
- Design a real implementation. A compiler ? An extension to OCaml ?

Thank you

Other vacuous

```
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
;;
```

back

λ-tree syntax

- The syntax is encoded as simply typed λ-terms. Syntactic categories are mapped to simple types.
- Equality of syntax is equated to α, β_0, η 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.

de Bruijn, N. G. (1979).
Lambda calculus notation with namefree formulas involving symbols that represent reference transforming mappings.
Indag. Math., 40(3):348–356.

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.

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.

Pottier, F. (2006).
An overview of Cαml.
In *Proceedings of the ACM-SIGPLAN Workshop on ML (ML 2005)*, volume 148 of *Electr. Notes Theor. Comput. Sci.*, pages 27–52.