Functional programming with $\lambda$–tree syntax

Ulysse Gérard and Dale Miller

LFMTP, July 7, 2018

Inria Saclay
Palaiseau France
Functional programming (FP) languages are popular tools to build systems that manipulate the syntax of programming languages and logics.

Variable binding is a common denominator of these objects.

A number of libraries exists along with first class extensions, but only few FP languages natively provide constructs to handle bindings.

Libs: AlphaLib, Caml... and Bindlib!

Languages: Beluga, FreshML...
The logic programming community also worked on **first-class binding structures** : $\lambda$Prolog, Abella...

Computation is expressed as proof search.

- Bindings are encoded using $\lambda$-abstractions and equality is up to $\alpha$, $\beta$, $\eta$ conversion (\$\lambda$-tree syntax [Miller and Palamidessi, 1999])
- A new binding quantifier, $\forall$ can be added to the underlying logic to work with **nominals**

This allows bindings in data structures to **move** to the formula level and to the proof level.
Our goal: enrich ML with bindings support in the style of Abella. We describe a new functional programming language, MLTS, whose concrete syntax is based on that of OCaml.

Work in progress...
The substitution case-study

Term substitution:

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

Such that “subst t x u” is t[x/u].
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

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) -> ?
```

Ca\text{ml}, given a type with binders, \textit{generates} an OCaml module to manipulate inhabitants of this type.

```
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
  | ... 
  | Abs abs ->

  let x', body = (open_lambda abs) in

  Abs(create_lambda (x', subst body x u))
type tm =
    | App of tm * tm
    | Abs of tm => tm

;;

Some inhabitants:

\( \lambda x. x \)
\( \lambda x. (x x) \)
\((\lambda x. x) (\lambda x. x)\)
... 

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

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

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

\[
\rightarrow\ \text{Abs}(Y\backslash \text{App}(Y, \text{Abs}(Z\backslash Z)))
\]
Two type systems

• MLTS is designed as a strongly typed functional programming language and type checking is performed before evaluation.
• But evaluation itself only need a simpler type system: 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 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 its arguments using @, thus eliminating the abstraction.

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

**Example**

\( Y \backslash ((X \backslash \text{body}) @ Y) \) denotes the result of instantiating the abstracted nominal \( X \) with the nominal \( Y \) in body.
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 nominals introduced by **new** and \. 
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
  end
;
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 @ Y)
        in aux (r @ X)
| _ -> false
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 can be applied to at most a list of distinct nominals. ($\text{nab } X_1 \ X_2 \ \text{in } C(r \ @ \ X_1 \ X_2) \rightarrow \ldots$)
- These nominals must be bound in the scope of pattern variables. (In $\forall r \ \text{nab } X_1 \ X_2 \ \text{in } C(r \ @ \ X_1 \ X_2)$ the scopes of $X_1$ and $X_2$ are 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 and unitary.
Natural semantics and implementation

Natural semantics for MLTS is fully declarative inside the logic $G$. This fragment of the $G$-logic is implemented in $\lambda$Prolog. We translate the ocaml-style concrete syntax into the abstract syntax in $\lambda$Prolog before evaluation.

Given the richness of the $G$-logic on which is based the natural semantics, we can prove that nominals do not escape their scope:

$$ \forall \exists V. \text{eval}(\text{new } X \text{ in } X) \ V $$
Conclusion & Future work

- This treatment of bindings has a clean semantic inspired by Abella.
- The interpreter was quite simple to write: \( \approx 140 \) lines of code.
- More examples in the meta-programming area (a compiler?)
- 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
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)
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:

$$\lambda x. x \triangleright c_1 \quad \lambda x. p \ x \ c_2 \triangleright p \ c_1 \ c_2 \quad \lambda x. \lambda y. p \ \ x \ \ y \triangleright p \ c_1 \ c_2$$

$$\lambda x. x \ntriangleright p \ c_1 \ c_2 \quad \lambda x. p \ \ x \ \ c_2 \ntriangleright p \ c_2 \ c_1 \quad \lambda x. \lambda y. p \ \ x \ \ y \ntriangleright p \ c_1 \ c_1$$

Nominal abstraction of degree $(n)$ 0 is the same as equality between terms based on $\lambda$-conversion.
Concrete syntax typing rules (1/2)

\[
\frac{\Gamma, x : C \vdash x : C}{\Gamma, x : C \vdash x : C}
\]

\[
\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash (M \ N) : B}
\]

\[
\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\text{fun } x \rightarrow M) : A \rightarrow B}
\]

\[
\frac{\Gamma, X : A \vdash M : B \quad \text{open } A}{\Gamma \vdash (\text{new } X \ in \ M) : B}
\]

\[
\frac{\Gamma, X : A \vdash M : B \quad \text{open } A}{\Gamma \vdash (X \ \backslash \ M) : A \Rightarrow B}
\]

\[
\frac{\Gamma \vdash r : A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow A \quad \Gamma \vdash t_1 : A_1 \quad \ldots \quad \Gamma \vdash t_n : A_n}{\Gamma \vdash (r \ @ \ t_1 \ \ldots \ \ t_n) : A}
\]
Concrete syntax typing rules (2/2)

\[\Gamma \vdash \text{term} : B \quad \Gamma \vdash B : R_1 : A \quad \ldots \quad \Gamma \vdash B : R_n : A \]
\[\Gamma \vdash \text{match term with} \ R_1 \mid \ldots \mid R_n : A\]

\[\Gamma, X : C \vdash A : R : B \quad \text{open} \ C \]
\[\Gamma \vdash A : \text{nab} \ X \ \text{in} \ R : B\]
\[\Gamma \vdash L : A \vdash \Delta \quad \Gamma, \Delta \vdash R : B \]
\[\Gamma \vdash A : L \rightarrow R : B\]

\[\Gamma \vdash t_1 : A_1 \vdash \Delta_1 \quad \ldots \quad \Gamma \vdash t_n : A_n \vdash \Delta_n \]
\[\Gamma \vdash C(t_1, \ldots, t_n) : A \vdash \Delta_1, \ldots, \Delta_n \quad \text{C of type} \ A_1 \ast \ldots \ast A_n \rightarrow A\]

\[\Gamma \vdash X_1 : A_1 \quad \ldots \quad \Gamma \vdash X_n : A_n \quad \text{open} \ A_1 \ldots \text{open} \ A_n \]
\[\Gamma \vdash (r @ X_1 \ldots X_n) : A \vdash r : A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow A\]

\[\Gamma \vdash \text{p : A} \vdash \Delta_1 \quad \Gamma \vdash \text{q : B} \vdash \Delta_2 \]
\[\Gamma \vdash (p, q) : A \ast B \vdash \Delta_1, \Delta_2\]
### Natural semantics for the abstract syntax

(*G*-logic [Gacek, 2009, Gacek et al., 2011])

\[\vdash val \ V \quad \vdash M \Downarrow F \quad \vdash N \Downarrow U \quad \vdash apply \ F \ U \ V\]

\[\vdash V \Downarrow V \quad \vdash M \Downarrow F \quad \vdash N \Downarrow U \quad \vdash M \circ N \Downarrow V\]

\[\vdash (R \ U) \Downarrow V \quad \vdash (R \ (fixpt \ R)) \Downarrow V\]

\[\vdash apply \ (\lambda R \ U \ V \quad \vdash apply \ (\lambda R \ U \ V) \quad \vdash (fixpt \ R) \Downarrow V\]

\[\vdash C \Downarrow \text{tt} \quad \vdash L \Downarrow V \quad \vdash C \Downarrow \text{ff} \quad \vdash M \Downarrow V\]

\[\vdash cond \ C \ L \ M \Downarrow V \quad \vdash cond \ C \ L \ M \Downarrow V\]
Natural semantics for the abstract syntax (2/2)

\[ \vdash \nabla x. (E \ x) \Downarrow (V \ x) \]
\[ \vdash x \ \downarrow \ E \ x \Downarrow x \ \downarrow \ V \ x \]
\[ \vdash \nabla x. (E \ x) \Downarrow V \]
\[ \vdash \text{new} \ E \Downarrow V \]

\[ \vdash \text{pattern} \ T \ \text{Rule} \ U \quad \vdash U \Downarrow V \]
\[ \vdash (\text{match} \ T \ (\text{Rule :: Rules})) \Downarrow V \]
\[ \vdash (\text{match} \ T \ (\text{Rule :: Rules})) \Downarrow V \]

\[ \vdash \exists x. \text{pattern} \ T \ (P \ x) \ U \quad \vdash (\lambda z_1 \ldots \lambda z_m. (t \Rightarrow s)) \supseteq (T \Rightarrow U) \]
\[ \vdash \text{pattern} \ T \ (\text{all} \ (x \ \downarrow \ P \ x)) \ U \quad \vdash \text{pattern} \ T \ (\text{nab} \ z_1 \ldots \text{nab} \ z_m. (t \Rightarrow s)) \ U \]

\[ \vdash \lambda X. (X \Rightarrow s) \supseteq (Y \Rightarrow U) \]
\[ \vdash \text{pattern} \ Y \ (\text{nab} \ X \ \text{in} \ (X \Rightarrow s)) \ U \quad \vdash U \Downarrow V \]
\[ \vdash \text{match} \ Y \ \text{with} \ (\text{nab} \ X \ \text{in} \ (X \Rightarrow s)) \Downarrow V \]
**A Framework for Specifying, Prototyping, and Reasoning about Computational Systems.**

**Nominal abstraction.**

**Programming with Higher-Order Logic.**
Cambridge University Press.

**Foundational aspects of syntax.**