Functional programming with $\lambda-$tree syntax

Ulysse Gérard and Dale Miller

LFMTP, July 7, 2018

Inria Saclay
Palaiseau France

Introduction

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, Cαml... and Bindlib !

Languages: Beluga, FreshML...

Introduction: the logical approach

The logic programming community also worked on first-class binding structures : λProlog, Abella...

Computation is expressed as proof search.

- Bindings are encoded using λ-abstractions and equality is up to α, β, η conversion (λ-tree syntax [Miller and Palamidessi, 1999])
- A new binding quantifier, ∇ 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.

Introduction: MLTS

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]`.

Handmade

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)  -> ?
```

Cαml (example from the Little Calculist blog)

Cαml, given a type with binders, 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
```

Cαml

```
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))
```

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 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)
```

`r : tm => tm` `r @ Y : tm`

`(Y\ r @ Y) : tm => tm` `Abs(Y\ r @ Y): tm`

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

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)))`

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 $0 \rightarrow \cdots \rightarrow 0$

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 its arguments using `@`, thus eliminating the abstraction.

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

Example

`Y\ ((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.

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

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: vacuity

```
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 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 can be applied to at most a list of distinct nominals. (`nab X1 X2 in C(r @ X1 X2) -> ...`)
- These nominals must be bound in the scope of pattern variables. (In ∀r `nab X1 X2 in C(r @ X1 X2)` the scopes of X1 and X2 are 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 and unitary.

Natural semantics and implementation

Natural semantics for MLTS is fully declarative inside the logic $\mathcal{G}$.

This fragment of the $\mathcal{G}$-logic is implemented in λProlog. We translate the ocaml-style concrete syntax into the abstract syntax in λProlog before evaluation.

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

$$\nvdash \exists\, V.\ \mathit{eval}(\texttt{new X in X})\ V$$

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

- 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

Other vacuous

```
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)

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:

$$\lambda x.x \trianglerighteq c_1 \qquad \lambda x.p\ x\ c_2 \trianglerighteq p\ c_1\ c_2 \qquad \lambda x.\lambda y.p\ x\ y \trianglerighteq p\ c_1\ c_2$$

$$\lambda x.x \ntrianglerighteq p\ c_1\ c_2 \qquad \lambda x.p\ x\ c_2 \ntrianglerighteq p\ c_2\ c_1 \qquad \lambda x.\lambda y.p\ x\ y \ntrianglerighteq p\ c_1\ c_1$$

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

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)

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

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

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

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

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

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.