Functional programming with $\lambda-$tree syntax

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

Inria Saclay and LIX, École Polytechnique
Palaiseau France

ANU, 8 May 2019

Introduction

Functional programming 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 feature of most syntactic structures.

In the area of theorem provers, the POPLMark Challenge (2005) singled out the lack of binder support in provers as a serious impediment to formalizing meta-theory.

Introduction

Functional programming 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 feature of most syntactic structures.

In the area of theorem provers, the POPLMark Challenge (2005) singled out the lack of binder support in provers as a serious impediment to formalizing meta-theory.

Support for binders in functional languages is experimental and fractured.

- Some libraries exists: AlphaLib, Cαml, etc.
- New languages: FreshML, Delphin, Beluga, etc.

We introduce MLTS (as an extension to the core of OCaml) to treat binding structures in a primitive fashion.

Two language paradigms, two approaches to bindings

The term higher-order abstract syntax—the use of programming-level binding to support syntax-level binding—is badly ambiguous.

In logic programming, for example, λProlog and Twelf, this approach leads to an elegant, compact, and declarative treatment of bindings. The Abella theorem prover formalizes that approach.

In functional programming, it has lead to using function spaces to encode binding structures. This approach is wildly different and problematic.

Our approach: λ-tree syntax and binder mobility

With MLTS, we are attempting to move the lessons learned from the logic programming world into the functional programming world. We emphasizes two key concepts.

- Functional programs need more binding sites so term-level bindings can move to programming-level bindings.
  Alan Perlis: "There is no such things as a free variables."
- All operations on syntax must respect α-conversion and (at least some of) β-conversion.

Together, we have the λ-tree syntax approach to bindings.

MLTS stands for

- mobility and lambda-tree syntax

Our approach: λ-tree syntax and binder mobility

With MLTS, we are attempting to move the lessons learned from the logic programming world into the functional programming world. We emphasizes two key concepts.

- Functional programs need more binding sites so term-level bindings can move to programming-level bindings.
  Alan Perlis: "There is no such things as a free variables."
- All operations on syntax must respect α-conversion and (at least some of) β-conversion.

Together, we have the λ-tree syntax approach to bindings.

MLTS stands for

- mobility and lambda-tree syntax
- ... or most likely to succeed
- ... or most long term solution

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)
```

But occurrences of `y` in `u` can be "captured."

We need to check for free variables in t and rename them if necessary...

Handmade

There are several approaches to handle bindings:

- Var as strings
- Var as fresh names
- De Bruijn's nameless dummies

But they all need to be carefully implemented.

Can we automate this tedious and pervasive task ?

Cαml [Pottier 2006]

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))
```

But bindings and substitutions are logic

Bindings, substitutions, α-conversion, etc, are all features of well understood and popular logics.

- Church's 1940 Simple Theory of Types underlies HOL, Isabelle, λProlog, etc.

They are not "yet another data structure to get implemented anyway that works...".

MLTS version of subst

```
type tm =
  | App of tm * tm
  | Abs of tm => tm;; (* Note new arrow *)
```

Some inhabitants (all of type `tm`):

$\lambda x.\ x$	`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))`

λ-abstraction is written as infix backslash (following λProlog).

Initial capital letters denote constructors (following OCaml).

Since nominals are essentially scoped constructors, they are capitalized also.

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 do we perform the substitution:

$$(\lambda y.\ y\ x)[x \backslash \lambda z.\ z]?$$

Something like

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

MLTS version of subst

How do we perform the substitution:

$$(\lambda y.\ y\ x)[x \backslash \lambda z.\ z]?$$

Something like

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

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

MLTS version of subst

How do we perform the substitution:

$$(\lambda y.\ y\ x)[x \backslash \lambda z.\ z]?$$

Something like

```
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)))`

Computing the size of an untyped λ-term

```
let rec size term =
    match term with
| App(n, m)   -> 1 + (size n) + (size m)
| Abs(r)      -> 1 + (new X in size (r @ X))
| nab X in X -> 1;;
```

A sample computation:

```
size (Abs (X\ (Abs (Y\ (App(X,Y))))))
new X in 1 + (size (Abs (Y\ (App(X,Y)))))
new X in 1 + new Y in 1 + (size (App(X,Y)))
new X in 1 + new Y in 1 + 1 + (size X)+(size Y)
new X in 1 + new Y in 1 + 1 + 1 + 1
5
```

MLTS features: =>, backslash and @

The type constructor `=>` is used to declare bindings in datatypes.

MLTS features: $=>$, backslash and @

The type constructor `=>` is used to declare bindings 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.

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

The backslash introduces `=>` and the `@` eliminates it.

MLTS features: $=>$, backslash and @

The type constructor `=>` is used to declare bindings 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.

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

The backslash introduces `=>` and the `@` eliminates it.

Example

`((X\ body) @ Y)` denotes a β-redex: replace 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 the nominals 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 the nominals introduced by `new` and `\`.

Pattern variables can have `=>` type 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 the nominals introduced by `new` and `\`.

Pattern variables can have `=>` type 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)`

Three new promised binding sites: backslash `\`, new, and nab.

An 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
;;
```

An example: vacuous 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 ;;
```

An example: vacuous

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

An example: vacuous

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

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

Variable capture is not allowed in substitutions.

Recursion over term structures is hidden in matching.

Pattern matching

We perform matching modulo α, β_0 and η.

β_0: $(\lambda x.B)x = B$

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

Pattern matching

We perform matching modulo α, β_0 and η.

β_0: $(\lambda x.B)x = B$

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

Patterns are further restricted:

- Pattern variables are applied to a (possibly empty) list of distinct variables: e.g., `(r @ X Y)`.
- The variables in the list are bound within the scope of the pattern variables.

Pattern matching

We perform matching modulo α, β_0 and η.

β_0: $(\lambda x.B)x = B$

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

Patterns are further restricted:

- Pattern variables are applied to a (possibly empty) list of distinct variables: e.g., `(r @ X Y)`.
- The variables in the list are bound within the scope of the pattern variables.

Such pattern matching is a subset of higher-order pattern unification (a.k.a. L_λ-unification).

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

Some matching examples

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

$$\begin{array}{lll}
(1) & \lambda x \lambda y(f\ (H\ x)) & \lambda u \lambda v(f\ (f\ u)) \\
(2) & \lambda x \lambda y(f\ (H\ x)) & \lambda u \lambda v(f\ (f\ v)) \\
(3) & \lambda x \lambda y(g\ (H\ y\ x)\ (f\ (L\ x))) & \lambda u \lambda v(g\ u\ (f\ u)) \\
(4) & \lambda x \lambda y(g\ (H\ x)\ (L\ x)) & \lambda u \lambda v(g\ (g\ a\ u)\ (g\ u\ u))
\end{array}$$

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

Some matching examples

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

$$\begin{array}{lll}
(1) & \lambda x \lambda y(f\ (H\ x)) & \lambda u \lambda v(f\ (f\ u)) \\
(2) & \lambda x \lambda y(f\ (H\ x)) & \lambda u \lambda v(f\ (f\ v)) \\
(3) & \lambda x \lambda y(g\ (H\ y\ x)\ (f\ (L\ x))) & \lambda u \lambda v(g\ u\ (f\ u)) \\
(4) & \lambda x \lambda y(g\ (H\ x)\ (L\ x)) & \lambda u \lambda v(g\ (g\ a\ u)\ (g\ u\ u))
\end{array}$$

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

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) $H \mapsto \lambda y \lambda x . x \quad L \mapsto \lambda x . x$

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

Translation

Our prototype interpreter translates the OCaml-style concrete syntax into a λProlog term that is then evaluated by the interpreter written in λProlog.

Translation

Our prototype interpreter translates the OCaml-style concrete syntax into a λProlog term that is then evaluated by the interpreter written in λProlog.

```
let subst t u = new X in
    let rec aux t = match t with
      | X -> u
      | nab Y in Y -> Y
      | App(u, v) -> App(aux u, aux v)
      | Abs r -> Abs(Y\ aux (r @ Y))
    in aux (t @ X);;

(*  subst : (tm => tm) -> tm -> tm  *)
```

Translation

```
prog "subst" (lam t0 \ lam u \ new X \ let
   (fix aux \ lam t1 \
        match t1
        [(arr (pnom X) u), (nab Y \ arr
           (pnom Y) Y),
         (all 0 \ all u1 \
            arr (pvariant App [(pvar u1),
               (pvar 0)])
            (variant App [(app aux u1),
               (app aux 0)])),
         (all r \
            arr (pvariant Abs [pvar r])
            (variant Abs
             [backslash Y \ app aux
               (arobase r Y)])))]) aux \
    app aux (arobase t0 X)).
```

Natural semantics for MLTS: evaluation ($\Downarrow$)

$$\frac{}{\vdash \mathsf{lam}\ R \Downarrow \mathsf{lam}\ R} \qquad \frac{\vdash \forall i \in [1;n],\ \ T_i \Downarrow V_i}{\vdash \mathsf{variant}\ c\ [T_1, \ldots, T_n] \Downarrow \mathsf{variant}\ c\ [V_1, \ldots, V_n]}$$

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

$$\frac{\vdash M \Downarrow \mathsf{lam}\ R \qquad \vdash N \Downarrow U \qquad \vdash (R\ \ U) \Downarrow V}{\vdash \mathsf{app}\ \ M\ \ N \Downarrow V} \qquad \frac{\vdash M \Downarrow U \qquad \vdash (R\ \ U) \Downarrow V}{\vdash (\mathsf{let}\ \ M\ \ R) \Downarrow V}$$

$$\frac{\vdash R\ (\mathsf{fix}\ \ R) \Downarrow V}{\vdash \mathsf{fix}\ \ R \Downarrow V} \qquad \frac{\vdash \nabla x.(E\ \ x) \Downarrow V}{\vdash new(\lambda x.E\, x) \Downarrow V}$$

$$\frac{\vdash M \Downarrow \mathsf{backslash}\ \ R \qquad \vdash (R\ \ X) \Downarrow V}{\vdash \mathsf{arobase}\ \ M\ X \Downarrow V} \qquad \frac{\vdash \nabla x.(E\ \ x) \Downarrow (V\ \ x)}{\vdash \mathsf{backslash}\ (\lambda x.E\, x) \Downarrow \mathsf{backslash}\ (\lambda x.V\, x)}$$

Natural semantics for MLTS: match and clause

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

$$\frac{\vdash \neg(\exists u, \text{clause } T \ Rule \ u) \qquad \vdash (\text{match } T \ Rules) \Downarrow V}{\vdash (\text{match } T \ (Rule :: Rules)) \Downarrow V}$$

$$\frac{\vdash \exists x.\text{clause } T \ (P \ x) \ U}{\vdash \text{clause } T \ (\text{all } (\lambda x.P \ x)) \ U}$$

$$\frac{\vdash \text{matches } T \ P \qquad \vdash (\lambda z_1 \ldots \lambda z_m.(p \Longrightarrow u)) \trianglerighteq (P \Longrightarrow U)}{\vdash \text{clause } T \ (\text{nab } z_1 \ldots \text{nab } z_m.(p \Longrightarrow u)) \ U}$$

$$\frac{\vdash \forall i \in [1; n], \text{matches } t_i \ p_i}{\vdash \text{matches } (\text{variant } c \ [t_1, \ldots, t_n]) \ (\text{pvariant } c \ [p_1, \ldots, p_n])}$$

$$\frac{\text{nominal}(c)}{\vdash \text{matches } c \ (\text{pnom } c)} \qquad \frac{}{\vdash \text{matches } x \ (\text{pvar } x)}$$

Nominal abstraction: $\trianglerighteq$

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 $c_1, \ldots, c_n$.

Equality if nominal abstraction of degree 0 .

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.

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

$$
\begin{array}{lll}
\lambda x.x \trianglerighteq c_1 & \lambda x.p\ x\ c_2 \trianglerighteq p\ c_1\ c_2 & \lambda x.\lambda y.p\ x\ y \trianglerighteq p\ c_1\ c_2 \\
\lambda x.x \not\trianglerighteq p\ c_1\ c_2 & \lambda x.p\ x\ c_2 \not\trianglerighteq p\ c_2\ c_1 & \lambda x.\lambda y.p\ x\ y \not\trianglerighteq p\ c_1\ c_1
\end{array}
$$

Illustrating the match/pattern rule

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

Nominals do not escape their scopes

The logic behind the natural semantics ensures 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.

Nominals do not escape their scopes

The logic behind the natural semantics ensures 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.

λProlog ensures that no binding escapes its scope since such checks are built into unification.

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

Static checks will need to be developed in order to ensure that such checks are not always needed.

Current implementation

Type inference was easy to implement in λProlog.

The natural semantics are usually easy to implement in λProlog: however, the ∇ and $\trianglerighteq$ are not part of λProlog so they needed to be implemented.

Current implementation

Type inference was easy to implement in λProlog.

The natural semantics are usually easy to implement in λProlog: however, the ∇ and $\trianglerighteq$ are not part of λProlog so they needed to be implemented.

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

Current implementation

Type inference was easy to implement in λProlog.

The natural semantics are usually easy to implement in λProlog: however, the ∇ and $\trianglerighteq$ are not part of λProlog so they needed to be implemented.

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

Since the Elpi implementation of λProlog (by Enrico Tassi) is written in OCaml and since js_of_ocaml compiles OCaml bytecode to javascript, we could provide a website for experimenting with MLTS.

https://trymlts.github.io/

Future work

- More complex examples
- More formal proofs: small step semantics, subject reduction, progress, etc.
- Statics checks such as pattern matching exhaustivity, etc.
- Make definitive choices about remaining aspects of this prototype (should we restrict `@` to β_0 reductions? Should constructors introduced by `\` always be of primitive type?)
- Design a real implementation and an abstract machine.
- A compiler? An extension to OCaml?

References

- Online demo and full source code: `https://trymlts.github.io/`
- "Functional programming with λ-tree syntax" by Ulysse Gérard, M, and Gabriel Scherer. Draft paper available at `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/mlts-draft-may-2019.pdf` (In preparation.)

Thank you

Another implementation of vacuous checking

```
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 β_0 is complete for β.
- Bound variables never become free, instead, their binding scope can move.