Visitors Unchained

Using **visitors** to traverse **abstract syntax with binding**

François Pottier

WG 2.8, Edinburgh
June 15, 2017

The problem

Manipulating abstract syntax with binding requires many standard operations:

- **“opening”** and **“closing”** binders (in the locally nameless representation);
- **shifting** (in de Bruijn’s representation);
- deciding α**-equivalence**;
- testing whether a name **occurs** free in a term;
- performing capture-avoiding **substitution**;
- **converting** user-supplied terms to the desired internal representation;
- etc., etc.

This requires a lot of **boilerplate**, a.k.a. **nameplate** (Cheney).

Isn't this a solved problem?

It may well be, depending on your programming language of choice.

Haskell has

- FreshLib (Cheney, 2005),
- Unbound (Weirich, Yorgey, Sheard, 2011),
- Bound (Kmett, 2013?),
- maybe more?

These libraries may have bad performance, though (?).

OCaml has... not much.

- Cαml (F.P., 2005) is monolithic, inflexible, and has performance issues, too.

(The problem also arises in theorem provers. Not studied here.)

Goal

I wish to **scrap my nameplate**, in **OCaml**, in a manner that is

- as **modular**, open-ended, **customizable** as possible,
- while relying on **as little code generation** as possible,
- while (simultaneously!) supporting **multiple representations** of names,
- and supporting **multiple binding constructs**, possibly user-defined.

It turns out that this can be done by exploiting the **“visitor”** design pattern.

Visitors

Installation & configuration

Installation:

```
opam update
opam install visitors
```

To configure ocamlbuild, add this in `_tags`:

```
true: \
  package(visitors.ppx), \
  package(visitors.runtime)
```

To configure Merlin, add this in `.merlin`:

```
PKG visitors.ppx
PKG visitors.runtime
```

An “iter” visitor

Annotating a type definition with `[@@deriving visitors { ... }]`...

```
type expr =
  | EConst of int
  | EAdd of expr * expr
  [@@deriving visitors { variety = "iter" }]
```

An "iter" visitor

Annotating a type definition with `[@@deriving visitors { ... }]`...

```
type expr =
  | EConst of int
  | EAdd of expr * expr
  [@@deriving visitors { variety = "iter" }]
```

```
class virtual ['self] iter = object (self : 'self)
  inherit [_] VisitorsRuntime.iter
  method visit_EConst env c0 =
    let r0 = self#visit_int env c0 in
    ()
  method visit_EAdd env c0 c1 =
    let r0 = self#visit_expr env c0 in
    let r1 = self#visit_expr env c1 in
    ()
  method visit_expr env this =
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd (c0, c1) ->
        self#visit_EAdd env c0 c1
end
```

... causes a **visitor class** to be auto-generated.

An “iter” visitor

Annotating a type definition with `[@@deriving visitors { ... }]`...

```
type expr =
  | EConst of int
  | EAdd of expr * expr
  [@@deriving visitors { variety = "iter" }]
```

```
class virtual ['self] iter = object (self : 'self)
  inherit [_] VisitorsRuntime.iter
  method visit_EConst env c0 =
    let r0 = self#visit_int env c0 in
    ()
  method visit_EAdd env c0 c1 =
    let r0 = self#visit_expr env c0 in
    let r1 = self#visit_expr env c1 in
    ()
  method visit_expr env this =
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd (c0, c1) ->
        self#visit_EAdd env c0 c1
end
```

one method per
data constructor

... causes a **visitor class** to be auto-generated.

An “iter” visitor

Annotating a type definition with `[@@deriving visitors { ... }]`...

```
type expr =
  | EConst of int
  | EAdd of expr * expr
  [@@deriving visitors { variety = "iter" }]
```

```
class virtual ['self] iter = object (self : 'self)
  inherit [_] VisitorsRuntime.iter
  method visit_EConst env c0 =
    let r0 = self#visit_int env c0 in
    ()
  method visit_EAdd env c0 c1 =
    let r0 = self#visit_expr env c0 in
    let r1 = self#visit_expr env c1 in
    ()
  method visit_expr env this =
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd (c0, c1) ->
        self#visit_EAdd env c0 c1
end
```

one method per data type

... causes a **visitor class** to be auto-generated.

An “iter” visitor

Annotating a type definition with `[@@deriving visitors { ... }]`...

```
type expr =
  | EConst of int
  | EAdd of expr * expr
  [@@deriving visitors { variety = "iter" }]
```

```
class virtual ['self] iter = object (self : 'self)
  inherit [_] VisitorsRuntime.iter
  method visit_EConst env c0 =
    let r0 = self#visit_int env c0 in
    ()
  method visit_EAdd env c0 c1 =
    let r0 = self#visit_expr env c0 in
    let r1 = self#visit_expr env c1 in
    ()
  method visit_expr env this =
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd (c0, c1) ->
        self#visit_EAdd env c0 c1
end
```

an environment
is pushed down

... causes a **visitor class** to be auto-generated.

An “iter” visitor

Annotating a type definition with `[@@deriving visitors { ... }]`...

```
type expr =
  | EConst of int
  | EAdd of expr * expr
  [@@deriving visitors { variety = "iter" }]
```

```
class virtual ['self] iter = object (self : 'self)
  inherit [_] VisitorsRuntime.iter
  method visit_EConst env c0 =
    let r0 = self#visit_int env c0 in
    ()
  method visit_EAdd env c0 c1 =
    let r0 = self#visit_expr env c0 in
    let r1 = self#visit_expr env c1 in
    ()
  method visit_expr env this =
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd (c0, c1) ->
        self#visit_EAdd env c0 c1
end
```

default behavior
is to do nothing

... causes a **visitor class** to be auto-generated.

An “iter” visitor

Annotating a type definition with `[@@deriving visitors { ... }]`...

```
type expr =
  | EConst of int
  | EAdd of expr * expr
  [@@deriving visitors { variety = "iter" }]
```

```
class virtual ['self] iter = object (self : 'self)
  inherit [_] VisitorsRuntime.iter
  method visit_EConst env c0 =
    let r0 = self#visit_int env c0 in
    ()
  method visit_EAdd env c0 c1 =
    let r0 = self#visit_expr env c0 in
    let r1 = self#visit_expr env c1 in
    ()
  method visit_expr env this =
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd (c0, c1) ->
        self#visit_EAdd env c0 c1
end
```

behavior at type `int` is inherited

... causes a **visitor class** to be auto-generated.

A “map” visitor

There are several varieties of visitors:

```
type expr =
  | EConst of int
  | EAdd of expr * expr
  [@@deriving visitors { variety = "map" }]
```

a “map” visitor
is requested

```
class virtual ['self] map = object (self : 'self)
  inherit [_] VisitorsRuntime.map
  method visit_EConst env c0 =
    let r0 = self#visit_int env c0 in
    EConst r0
  method visit_EAdd env c0 c1 =
    let r0 = self#visit_expr env c0 in
    let r1 = self#visit_expr env c1 in
    EAdd (r0, r1)
  method visit_expr env this =
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd (c0, c1) ->
        self#visit_EAdd env c0 c1
end
```

default behavior
is to rebuild a tree

Using a “map” visitor

Inherit a visitor class and **override** one or more methods:

```
let add e1 e2 =     (* A smart constructor. *)
  match e1, e2 with
  | EConst 0, e
  | e, EConst 0 -> e
  | _, _ ->          EAdd (e1, e2)

let optimize : expr -> expr =
  let v = object (self)
    inherit [_] map
    method! visit_EAdd env e1 e2 =
      add
        (self#visit_expr env e1)
        (self#visit_expr env e2)
  end in
  v # visit_expr ()
```

This addition-optimization pass is **unchanged** if more expression forms are added.

What we have seen so far

- **Several built-in varieties**: `iter`, `map`, ...
- **Arity two**, too: `iter2`, `map2`, ...
- Generated visitor methods are **monomorphic** (in this talk),
- and their types are **inferred**.
- Visitor classes are nevertheless **polymorphic**.
- **Polymorphic** visitor methods can be hand-written and inherited.

Support for parameterized data types

Visitors can traverse parameterized data types, too.

- But: how does one traverse a subtree of type `'a`?

Two approaches are supported:

- declare a **virtual visitor method** `visit_'a`
- pass a **function** `visit_'a` to every visitor method.
  - allows / requires methods to be polymorphic in `'a`
  - more compositional

In this talk: a bit of both (details omitted...).

Visiting preexisting types

Lists can be visited, too.

a preexisting parameterized type

```
type expr =
  | EConst of int
  | EAdd of expr list
  [@@deriving visitors { variety = "iter" }]
```

```
class virtual ['self] iter = object (self : 'self)
  inherit [_] VisitorsRuntime.iter
  method visit_EConst env c0 =
    let r0 = self#visit_int env c0 in
    ()
  method visit_EAdd env c0 =
    let r0 = self#visit_list self#visit_expr env c0 in
    ()
  method visit_expr env this =
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd c0 ->
        self#visit_EAdd env c0
end
```

Visiting preexisting types

visitor method is passed
a visitor function

Lists can be visited, too.

```
type expr =
  | EConst of int
  | EAdd of expr list
  [@@deriving visitors { variety = "iter" }]
```

```
class virtual ['self] iter = object (self : 'self)
  inherit [_] VisitorsRuntime.iter
  method visit_EConst env c0 =
    let r0 = self#visit_int env c0 in
    ()
  method visit_EAdd env c0 =
    let r0 = self#visit_list self#visit_expr env c0 in
    ()
  method visit_expr env this =
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd c0 ->
        self#visit_EAdd env c0
end
```

Visiting preexisting types

Lists can be visited, too.

visitor method
is inherited

```
type expr =
  | EConst of int
  | EAdd of expr list
  [@@deriving visitors { variety = "iter" }]
```

```
class virtual ['self] iter = object (self : 'self)
  inherit [_] VisitorsRuntime.iter
  method visit_EConst env c0 =
    let r0 = self#visit_int env c0 in
    ()
  method visit_EAdd env c0 =
    let r0 = self#visit_list self#visit_expr env c0 in
    ()
  method visit_expr env this =
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd c0 ->
        self#visit_EAdd env c0
end
```

Predefined visitor methods

The class `VisitorsRuntime.map` offers this method:

```
class ['self] map = object (self)
  (* One of many predefined methods: *)
  method private visit_list: 'env 'a 'b .
    ('env -> 'a -> 'b) -> 'env -> 'a list -> 'b list
  = fun f env xs ->
      match xs with
      | [] ->
          []
      | x :: xs ->
          let x = f env x in
          x :: self # visit_list f env xs
end
```

This method is **polymorphic**, so multiple instances of `list` are not a problem.

Visitors – a summary

Although they follow fixed patterns, visitors are quite **versatile**.

They are like higher-order functions, only more **customizable** and **composable**.

More fun with visitors:

- visitors for **open data types** and their fixed points (link);
- visitors for **hash-consed data structures** (link);
- **iterators** out of visitors (link).

In the remainder of this talk:

- Can we traverse **abstract syntax with binding**?

Visitors Unchained

Dealing with binding

Can a visitor traverse **abstract syntax with binding** constructs?

Dealing with binding

Can a visitor traverse **abstract syntax with binding** constructs?

Can this be done in a **modular** way?

Dealing with binding

Can a visitor traverse **abstract syntax with binding** constructs?

Can this be done in a **modular** way?

Exactly which **separation of concerns** should one enforce?

Dealing with binding

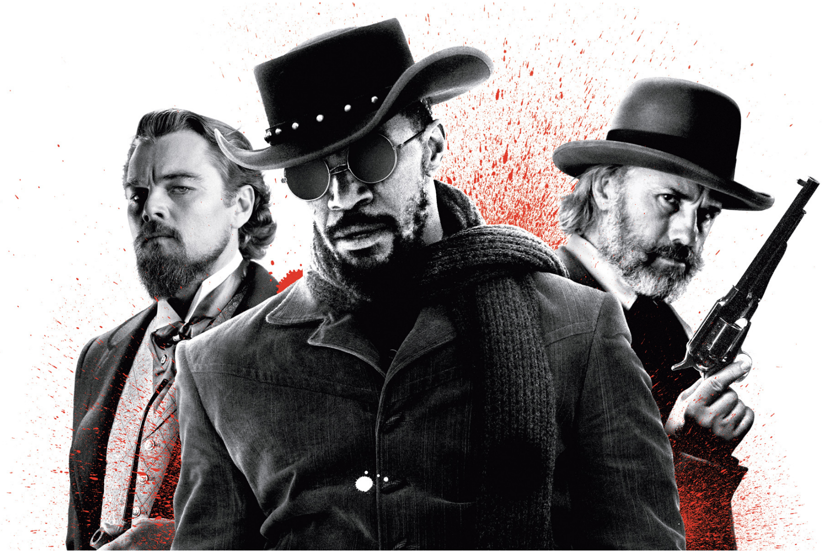
Can a visitor traverse **abstract syntax with binding** constructs?

Can this be done in a **modular** way?

Exactly which **separation of concerns** should one enforce?

- There are many **binding constructs**,
  - there are even **combinator languages** for describing binding structure!
- and many common **operations** on terms,
  - often specific of one **representation** of names and binders,
  - sometimes specific of **two** such representations, e.g., **conversions**.
- Can we insulate the **end user** from this complexity?

We suggest distinguishing **three** principals...

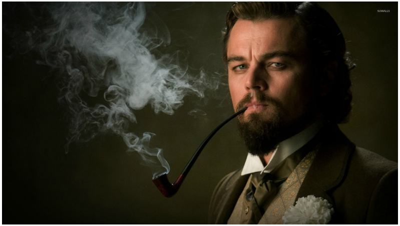



end
user


end user
operations specialist


end user
binder maestro
operations specialist

The end user



Desiderata

The end user wishes:

- to describe the structure of ASTs in a concise and **declarative** style,
- not to be bothered with implementation details,
- possibly to have access to **several representations** of names,
- to get access to a toolkit of **ready-made operations** on terms.

Example: abstract syntax of the λ-calculus

Let the type `('bn, 'term) abs` be a synonym for `'bn * 'term`.

The end user defines his syntax as follows:

```
type ('bn, 'fn) term =
  | TVar of 'fn
  | TLambda of ('bn, ('bn, 'fn) term) abs
  | TApp of ('bn, 'fn) term * ('bn, 'fn) term
[@@deriving visitors { variety = "map";
                       ancestors = ["BindingForms.map"] }]
type raw_term      = (string, string) term
type nominal_term  = (Atom.t, Atom.t) term
type debruijn_term = (unit,      int) term
```

He gets **multiple representations** of names.

- At least two are used in any single application. (Parsing. Printing.)

He gets **visitors** for free. The method `visit_abs` is used at abstractions.

- `iter`, `map`, `iter2` needed in practice. Focusing on `map` in this talk.

Example: abstract syntax of the λ-calculus

Let the type `('bn, 'term) abs` be a synonym for `'bn * 'term`.

provided by the binder maestro

The end user defines his syntax as follows:

```
type ('bn, 'fn) term =
  | TVar of 'fn
  | TLambda of ('bn, ('bn, 'fn) term) abs
  | TApp of ('bn, 'fn) term * ('bn, 'fn) term
[@@deriving visitors { variety = "map";
                       ancestors = ["BindingForms.map"] }]
type raw_term      = (string, string) term
type nominal_term  = (Atom.t, Atom.t) term
type debruijn_term = (unit,      int) term
```

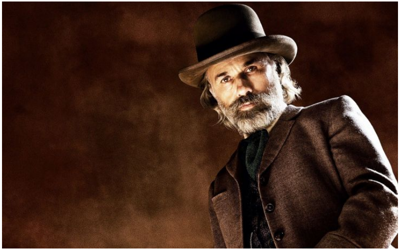
He gets **multiple representations** of names.

- At least two are used in any single application. (Parsing. Printing.)

He gets **visitors** for free. The method `visit_abs` is used at abstractions.

- `iter`, `map`, `iter2` needed in practice. Focusing on `map` in this talk.

The binder maestro



An easy job?

Implementing `visit_abs` is the task of our sophisticated binder maestro.

The key is to **extend the environment** when entering the scope of a binder.

Easy?

An easy job?

Implementing `visit_abs` is the task of our sophisticated binder maestro.

The key is to **extend the environment** when entering the scope of a binder.

Easy? Maybe — yet, the binder maestro:

- does not know **what operation** is being performed,
- does not know **what representation(s)** of names are in use,
- therefore does not know the types of names and environments,
- let alone **how** to extend the environment.

What he knows is **where** and **with what names** to extend the environment.

A convention

The binder maestro agrees on a **deal** with the operations specialist.

> *“I tell you when to extend the environment; you do the dirty work.”*

The binder maestro **calls** a method which the operations specialist **provides**:

```
(* A hook that defines how to extend the environment. *)
method private virtual extend: 'env -> 'bn1 -> 'env * 'bn2
```

This is a bare-bones **API** for describing binding constructs.

Visiting an abstraction

The class `BindingForms.map` offers the method `visit_abs`:

```
class virtual ['self] map = object (self : 'self)
  (* A visitor method for the type abs. *)
  method private visit_abs: 'term1 'term2 . _ ->
    ('env -> 'term1 -> 'term2) ->
    'env -> ('bn1, 'term1) abs -> ('bn2, 'term2) abs
  = fun _ visit_'term env (x1, t1) ->
      let env, x2 = self#extend env x1 in
      let t2 = visit_'term env t1 in
      x2, t2
  (* A hook that defines how to extend the environment. *)
  method private virtual extend: 'env -> 'bn1 -> 'env * 'bn2
end
```

This method:

Visiting an abstraction

The class `BindingForms.map` offers the method `visit_abs`:

```
class virtual ['self] map = object (self : 'self)
  (* A visitor method for the type abs. *)
  method private visit_abs: 'term1 'term2 . _ ->
    ('env -> 'term1 -> 'term2) ->
    'env -> ('bn1, 'term1) abs -> ('bn2, 'term2) abs
  = fun _ visit_'term env (x1, t1) ->
      let env, x2 = self#extend env x1 in
      let t2 = visit_'term env t1 in
      x2, t2
  (* A hook that defines how to extend the environment. *)
  method private virtual extend: 'env -> 'bn1 -> 'env * 'bn2
end
```

This method:

- takes a **visitor function** for terms, an **environment**,

Visiting an abstraction

The class `BindingForms.map` offers the method `visit_abs`:

```
class virtual ['self] map = object (self : 'self)
  (* A visitor method for the type abs. *)
  method private visit_abs: 'term1 'term2 . _ ->
    ('env -> 'term1 -> 'term2) ->
    'env -> ('bn1, 'term1) abs -> ('bn2, 'term2) abs
  = fun _ visit_'term env (x1, t1) ->
      let env, x2 = self#extend env x1 in
      let t2 = visit_'term env t1 in
      x2, t2
  (* A hook that defines how to extend the environment. *)
  method private virtual extend: 'env -> 'bn1 -> 'env * 'bn2
end
```

This method:

- takes a **visitor function** for terms, an **environment**,

Visiting an abstraction

The class `BindingForms.map` offers the method `visit_abs`:

```
class virtual ['self] map = object (self : 'self)
  (* A visitor method for the type abs. *)
  method private visit_abs: 'term1 'term2 . _ ->
    ('env -> 'term1 -> 'term2) ->
    'env -> ('bn1, 'term1) abs -> ('bn2, 'term2) abs
  = fun _ visit_'term env (x1, t1) ->
      let env, x2 = self#extend env x1 in
      let t2 = visit_'term env t1 in
      x2, t2
  (* A hook that defines how to extend the environment. *)
  method private virtual extend: 'env -> 'bn1 -> 'env * 'bn2
end
```

This method:

- takes a **visitor function** for terms, an **environment**,
- an abstraction, i.e., a **pair** of a name and a term, and

Visiting an abstraction

The class `BindingForms.map` offers the method `visit_abs`:

```
class virtual ['self] map = object (self : 'self)
  (* A visitor method for the type abs. *)
  method private visit_abs: 'term1 'term2 . _ ->
    ('env -> 'term1 -> 'term2) ->
    'env -> ('bn1, 'term1) abs -> ('bn2, 'term2) abs
  = fun _ visit_'term env (x1, t1) ->
      let env, x2 = self#extend env x1 in
      let t2 = visit_'term env t1 in
      x2, t2
  (* A hook that defines how to extend the environment. *)
  method private virtual extend: 'env -> 'bn1 -> 'env * 'bn2
end
```

This method:

- takes a **visitor function** for terms, an **environment**,
- an abstraction, i.e., a **pair** of a name and a term, and
- returns a pair of a **transformed name** and a **transformed term**.

Visiting an abstraction

That’s **all there is** to single-name abstractions.

More binding constructs later on...

For now, let’s turn to the final participant.

The operations specialist

A toolbox of operations

There are many operations on terms that the end user might wish for:

- testing terms for **equality** up to α-equivalence,
- finding out which names are **free** or **bound** in a term,
- applying a **renaming** or a **substitution** to a term,
- **converting** a term from one representation to another,
- (plus application-specific operations.)

Implementing an operation

To implement one operation, the specialist decides:

- the **types** of names and environments,
- **what to do** at a **free name** occurrence,
- **how to extend** the environment when entering the scope of a **bound name**.

Implementing `import`

As an example, let's implement `import`, which converts raw terms to nominal terms.

1. An import environment maps strings to atoms:

```
module StringMap = Map.Make(String)
type env = Atom.t StringMap.t
let empty : env = StringMap.empty
```

Implementing `import`

2. When the scope of *x* is entered,
the environment is extended with a mapping of the string *x* to a fresh atom *a*.

```
let extend (env : env) (x : string) : env * Atom.t =
  let a = Atom.fresh x in
  let env = StringMap.add x a env in
  env, a
```

(An **atom** carries a unique integer identity.)

This is true regardless of which binding constructs are traversed.

Implementing `import`

3. When an occurrence of the string *x* is found,
the environment is looked up so as to find the corresponding atom.

```
exception Unbound of string
let lookup (env : env) (x : string) : Atom.t =
  try StringMap.find x env
  with Not_found -> raise (Unbound x)
```

Implementing `import`

The previous instructions are grouped in a little class — a “kit”:

```
class ['self] map = object (_ : 'self)
  method private extend     = extend
  method private visit_'fn = lookup
end
```

This is `KitImport.map`.

That’s all there is to it... **but...**

Gluey business

The end user must **work** a little bit to **glue** everything together...

For each operation, the end user must write 5 lines of glue code:

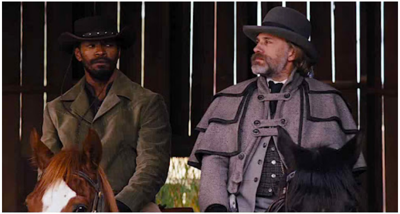
```
let import_term env t =
  (object
    inherit [_] map              (* generated by visitors *)
    inherit [_] KitImport.map (* provided by AlphaLib   *)
  end) # visit_term env t
```

For 15 operations, this hurts.

Functors can help in simple cases, but are not flexible enough.

C-like **macros** help, but are ugly. Is there a better way?

Towards advanced binding constructs



Defining new binding constructs

There are **many binding constructs** out there.

- “`let`”, “`let rec`”, patterns, telescopes, ...

We have seen how to **programmatically** define a binding construct.

Can it be done in a more **declarative** manner?

A domain-specific language

Here is a little language of **binding combinators**:

t	$::=$	$\ldots$	sums, products, free occurrences of names, etc.
	$\mid$	abstraction(p)	a pattern, with embedded subterms
p	$::=$	$\ldots$	sums, products, etc.
	$\mid$	binder(x)	a binding occurrence of a name
	$\mid$	outer(t)	an embedded term
	$\mid$	rebind(p)	a pattern in the scope of any bound names on the left

Inspired by Cαml (F.P., 2005) and Unbound (Weirich et al., 2011).

A domain-specific language

Here is a little language of **binding combinators**:

$$
\begin{array}{lcll}
t & ::= & \ldots & \text{sums, products, free occurrences of names, etc.} \\
  & | & \text{abstraction}(p) & \text{a pattern, with embedded subterms} \\
  &   &   & \\
p & ::= & \ldots & \text{sums, products, etc.} \\
  & | & \text{binder}(x) & \text{a binding occurrence of a name} \\
  & | & \text{outer}(t) & \text{an embedded term} \\
  & | & \text{rebind}(p) & \text{a pattern in the scope of any bound names on the left} \\
  & | & \text{inner}(t) & \text{— sugar for rebind(outer}(t)\text{))}
\end{array}
$$

Inspired by Cαml (F.P., 2005) and Unbound (Weirich et al., 2011).

A domain-specific language

Here is a little language of **binding combinators**:

t	$::=$	$\ldots$	sums, products, free occurrences of names, etc.
	$\mid$	abstraction(p)	a pattern, with embedded subterms
	$\mid$	bind(p, t)	— sugar for abstraction($p \times$ inner(t))
p	$::=$	$\ldots$	sums, products, etc.
	$\mid$	binder(x)	a binding occurrence of a name
	$\mid$	outer(t)	an embedded term
	$\mid$	rebind(p)	a pattern in the scope of any bound names on the left
	$\mid$	inner(t)	— sugar for rebind(outer(t))

Inspired by Cαml (F.P., 2005) and Unbound (Weirich et al., 2011).

Example use: telescopes

A dependently-typed λ-calculus whose Π and λ forms involve a telescope:

```
#define tele     ('bn, 'fn) tele
#define term     ('bn, 'fn) term
(* The types that follow are parametric in 'bn and 'fn: *)

type tele =
  | TeleNil
  | TeleCons of 'bn binder * term outer * tele rebind

and term =
  | TVar of 'fn
  | TPi  of (tele, term) bind
  | TLam of (tele, term) bind
  | TApp of term * term list

[@@deriving visitors {
  variety = "map";
  ancestors = ["BindingCombinators.map"]
}]
```

Implementation

These primitive constructs are just annotations:

```
type 'p abstraction = 'p
type 'bn binder = 'bn
type 't outer = 't
type 'p rebind = 'p
```

Their presence triggers calls to appropriate (hand-written) `visit_` methods.

Implementation

While visiting a pattern, we keep track of:

- the **outer environment**, which existed outside this pattern;
- the **current environment**, extended with the bound names encountered so far.

Thus, while visiting a pattern, we use a richer type of **contexts**:

```
type 'env context = { outer: 'env; current: 'env ref }
```

— Not every visitor method need have the same type of environments!

With this in mind, the implementation of the `visit_` methods is straightforward...

Implementation

This code takes place in a `map` visitor:

```
class virtual ['self] map = object (self : 'self)
  method private virtual extend: 'env -> 'bn1 -> 'env * 'bn2
  (* The four visitor methods are inserted here... *)
end
```

1. At the root of an abstraction, **a fresh context** is allocated:

```
  method private visit_abstraction: 'env 'p1 'p2 .
    ('env context -> 'p1 -> 'p2) ->
    'env -> 'p1 abstraction -> 'p2 abstraction
  = fun visit_p env p1 ->
      visit_p { outer = env; current = ref env } p1
```

Implementation

2. When a bound name is met, the **current** environment is **extended**:

```
method private visit_binder: _ ->
  'env context -> 'bn1 binder -> 'bn2 binder
= fun visit_'bn ctx x1 ->
    let env = !(ctx.current) in
    let env, x2 = self#extend env x1 in
    ctx.current := env;
    x2
```

Implementation

3. When a term that is **not in the scope** of the abstraction is found,
it is visited in the **outer** environment.

```
method private visit_outer: 'env 't1 't2 .
  ('env -> 't1 -> 't2) ->
  'env context -> 't1 outer -> 't2 outer
= fun visit_t ctx t1 ->
    visit_t ctx.outer t1
```

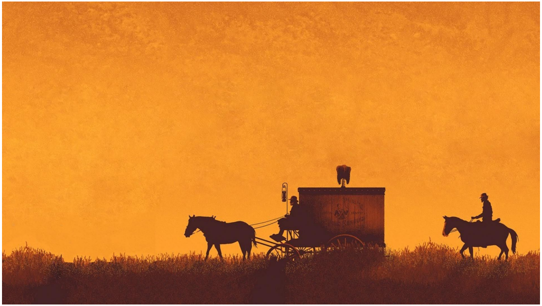
Implementation

4. When a subpattern marked `rebind` is found,
the **current** environment is installed as the **outer** environment.

```
method private visit_rebind: 'env 'p1 'p2 .
  ('env context -> 'p1 -> 'p2) ->
  'env context -> 'p1 rebind -> 'p2 rebind
= fun visit_p ctx p1 ->
    visit_p { ctx with outer = !(ctx.current) } p1
```

This affects the meaning of `outer` inside `rebind`.

Conclusion



Conclusion

Visitors are **powerful**.

Visitor classes are **partial**, **composable** descriptions of operations.

Visitors **can** traverse **abstract syntax with binding**.

- Syntax, binding forms, operations can be **separately** described.
- Syntax and even binding forms can be described in a **declarative** style.
- Open-ended, customizable approach.

Limitations:

- C-like macros are currently required.
- No proofs.
- Some operations may not fit the visitor framework;
- Some binding forms do not easily fit in the low-level framework or in the higher-level DSL, e.g., Unbound's *Rec*.