Visitors Unchained

Using visitors
to traverse abstract syntax with binding

François Pottier

ICFP 2017, Oxford
September 5, 2017

Visitors Unchained

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

OBJECTS AT ICFP!?

François Pottier

ICFP 2017, Oxford
September 5, 2017

Visitors Unchained

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

OBJECTS AT ICFP!?

François Pottier

BINDERS, AGAIN!?

ICFP 2017, Oxford
September 5, 2017

Boilerplate alert!

Manipulating abstract syntax with binding can be a **chore**.

Whenever one creates a new language, one must typically implement:

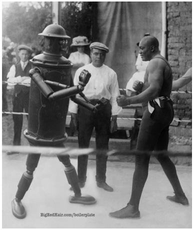
- **substitution**, α**-equivalence**, **free names**, – (all representations)
- **opening** / **closing**, **shifting**, – (some representations)
- **converting** between representations...

This boilerplate code is known as **nameplate** (Cheney).

It is **large**, **boring**, **error-prone**.

It does not seem easily reusable, as it is **datatype-specific**.

Fighting boilerplate!



In this talk & paper:

A way of **getting rid of nameplate**, in **OCaml**, which

- supports **multiple representations** of names and conversions between them,
- supports **complex binding constructs**,
- is **modular** and **open-ended**, that is, user-extensible,
- relies on **as little code generation** as possible.

Based on a combination of auto-generated **visitors** and library code.

- `visitors`, an OCaml syntax extension (released);
- `AlphaLib`, an OCaml library (at a preliminary stage).

Wait! Isn't this a solved problem already?

Several **Haskell** libraries address this problem: FreshLib, Unbound, Bound...
They exploit Haskell's support for **generic programming** (SYB, RepLib, ...)

Cool stuff can be done in **Coq** (Schäfer *et al.*, 2015) and **Agda** (Allais *et al.*, 2017).

In the **OCaml** world:

- Cαml (F.P., 2005), an ad hoc code generator; monolithic, inflexible.
- Yallop ports SYB to MetaOCaml+implicits: **next talk!** The way of the future?

– this talk: making do with **vanilla OCaml**.

Visitors

Generating a “map” visitor, in a nutshell

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

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

Generating a “map” visitor, in a nutshell

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

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

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

```
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
```

Generating a “map” visitor, in a nutshell

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

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

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

```
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
```

one method per data type

Generating a “map” visitor, in a nutshell

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

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

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

```
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
```

one method per
data constructor

Generating a “map” visitor, in a nutshell

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

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

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

```
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

Generating a “map” visitor, in a nutshell

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

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

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

```
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
```

an environment
is pushed down

Using a “map” visitor, in a nutshell

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

```
let optimize : expr -> expr =
  let v = object(self)
    inherit [_] map
    method! visit_EAdd env e1 e2 =
      match self#visit_expr env e1,self#visit_expr env e2 with
      | EConst 0, e                   (* 0 + e = e *)
      | e, EConst 0 -> e              (* e + 0 = e *)
      | e1, e2      -> EAdd (e1, e2)
  end in
  v # visit_expr ()
```

No changes to this code are needed when more expression forms are added.

Visiting preexisting / parameterized types

Integers and lists can be visited, too.

a preexisting type

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

```
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 =
    let r0 = self#visit_list (self#visit_expr) env c0 in
    EAdd r0
  method visit_expr env this = ...
end
```

A visitor can be **“taught”** to traverse a data structure!

Visiting preexisting / parameterized types

Integers and lists can be visited, too.

visitor method
is inherited

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

```
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 =
    let r0 = self#visit_list (self#visit_expr) env c0 in
    EAdd r0
  method visit_expr env this = ...
end
```

A visitor can be **"taught"** to traverse a data structure!

Visiting preexisting / parameterized types

Integers and lists can be visited, too.

a preexisting parameterized type

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

```
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 =
    let r0 = self#visit_list (self#visit_expr) env c0 in
    EAdd r0
  method visit_expr env this = ...
end
```

A visitor can be **“taught”** to traverse a data structure!

Visiting preexisting / parameterized types

Integers and lists can be visited, too.

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

inherited visitor method is passed a visitor function

```
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 =
    let r0 = self#visit_list (self#visit_expr) env c0 in
    EAdd r0
  method visit_expr env this = . . .
end
```

A visitor can be **“taught”** to traverse a data structure!

Visitors – summary

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

They are **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:

- Traversing **abstract syntax with binding**.

Visitors Unchained

Traversing syntax with binding

For **modularity**, it seems desirable to distinguish **three** concerns:

1. Describing a **binding construct**.
2. Describing an **operation** on terms.
   - usually specific of one **representation** of names and binders,
   - sometimes specific of **two** such representations, e.g., **conversions**.
3. The **end user** should be insulated from this complexity.

– 1 & 2 are part of `AlphaLib`.


end user
binder maestro
operations specialist

The end user

The end user

The end user describes the structure of ASTs in a concise, **declarative** style.

For example, this is the syntax of the λ-calculus:

```
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"] }]
```

The type `('bn, 'term) abs` represents an **abs**traction of one name in one term.

The end user

The end user describes the structure of ASTs in a

provided by the binder maestro

For example, this is the syntax of the λ-calculus:

```
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"] }]
```

The type `('bn, 'term) abs` represents an **abs**traction of one name in one term.

The end user

The end user describes the structure of ASTs in a concise, **declarative** style.

For example, this is the syntax of the λ-calculus:

```
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"] }]
```

The type `('bn, 'term) abs` represents an **abs**traction of one name in one term.

The end user gets a **visitor** for free.

The end user

The end user describes the structure of ASTs in a concise, **declarative** style.

For example, this is the syntax of the λ-calculus:

```
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"] }]
```

The type `('bn, 'term) abs` represents an **abs**traction of one name in one term.

The end user gets a **visitor** for free.

He gets **multiple representations** of names:

```
type raw_term      = (string, string) term
type nominal_term  = (Atom.t, Atom.t) term
type debruijn_term = (unit,      int) term
```

The binder maestro

The binder maestro

The maestro writes the module `BindingForms`. (Part of `AlphaLib`.)

He **defines** binding constructs and **teaches** visitors how to traverse them.

In memory, an abstraction of one name in one term is just a **pair**:

```
type ('bn, 'term) abs = 'bn * 'term
```

Traversing it requires **extending the environment** — roughly like this:

```
class virtual ['self] map = object (self : 'self)
 (* A visitor method for the type abs. *)
 method visit_abs _ visit_'term env (x1, t1) =
  let env, x2 = self#extend env x1 in (* extend env with x1 *)
  let t2 = visit_'term env t1 in      (* then visit t1 *)
  (x2, t2)
end
```

There is a catch, though – what on earth should the method `extend` do?

The catch

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 deal

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.

The operations specialist

Implementing an operation

To implement one operation on terms, the specialist decides:

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

Example: converting raw terms to nominal terms

The specialist writes the module `KitImport`. (Also part of `AlphaLib`.)

```
type env = Atom.t StringMap.t (* a map of strings to atoms *)
let empty : env = StringMap.empty
exception Unbound of string
class ['self] map = object (_ : 'self)
  (* At a binder, generate a fresh atom and extend the env. *)
  method extend (env : env) (x : string) : env * Atom.t =
    let a = Atom.fresh x in
    let env = StringMap.add x a env in
    env, a
  (* At a name occurrence, look up the environment. *)
  method visit_'fn (env : env) (x : string) : Atom.t =
    try StringMap.find x env
    with Not_found -> raise (Unbound x)
end
```

Done? Almost.

Gluey business

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

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

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

As there are many operations, this is unpleasant.

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

I use C-like **macros**, but this is ugly. Is there a better way?

Conclusion

Takeaway thoughts

Generated visitors allow a limited form of **generic programming**.

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

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

- Syntax, binding forms, operations are described **separately**.
- Syntax is described in a **declarative** style.
- In the paper: towards **a DSL for binding constructs**.

Limitations

Not everything is perfect:

- **Three** visitor classes are needed: `map`, `iter`, `iter2`.
- The end user must use a C-like **macro** that I provide.
- Some binding constructs **cannot be implemented** at all.
  - e.g., nonlinear patterns – (not representation-independent)
- Some binding constructs **are not easily supported** in the high-level DSL.
  - e.g., Unbound's *Rec* – (seems to require multiple subtraversals)
- More **practical experience** is needed. (~~Guinea pigs~~ Users welcome!)

Backup

Features of the visitors package

- **Several built-in varieties** of visitors: `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...).

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.

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:
    (* The method's type: *)
    'term1 'term2 . _ ->
    ('env -> 'term1 -> 'term2) ->
    'env -> ('bn1, 'term1) abs -> ('bn2, 'term2) abs
    (* The method's code: *)
  = 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:
    (* The method's type: *)
    'term1 'term2 . _ ->
    ('env -> 'term1 -> 'term2) ->
    'env -> ('bn1, 'term1) abs -> ('bn2, 'term2) abs
    (* The method's code: *)
  = 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:
    (* The method's type: *)
    'term1 'term2 . _ ->
    ('env -> 'term1 -> 'term2) ->
    'env -> ('bn1, 'term1) abs -> ('bn2, 'term2) abs
    (* The method's code: *)
  = 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:
    (* The method's type: *)
    'term1 'term2 . _ ->
    ('env -> 'term1 -> 'term2) ->
    'env -> ('bn1, 'term1) abs -> ('bn2, 'term2) abs
    (* The method's code: *)
  = 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:
    (* The method's type: *)
    'term1 'term2 . _ ->
    ('env -> 'term1 -> 'term2) ->
    'env -> ('bn1, 'term1) abs -> ('bn2, 'term2) abs
    (* The method's code: *)
  = 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**.

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**:

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
	$\mid$	inner(t)	— sugar for rebind(outer(t))

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`.