Visitors

version 20260520

François Pottier
Inria Paris
francois.pottier@inria.fr

Contents

1 Introduction **3**
- 1.1 What is a visitor? . . . 3
- 1.2 What does this package offer? . . . 3

2 Walkthrough **3**
- 2.1 Setup . . . 3
- 2.2 Defining an `iter` visitor . . . 4
- 2.3 Using an `iter` visitor . . . 5
- 2.4 What is the type of a visitor? . . . 5
- 2.5 `map` visitors . . . 7
- 2.6 `endo` visitors . . . 7
- 2.7 `reduce` visitors . . . 9
- 2.8 `mapreduce` visitors . . . 9
- 2.9 `fold` visitors . . . 11
- 2.10 Visitors of arity two . . . 11
- 2.11 Visitors for a family of types . . . 15
- 2.12 Visitors for parameterized types . . . 15
  - 2.12.1 Monomorphic visitor methods for parameterized types . . . 17
  - 2.12.2 Polymorphic visitor methods for parameterized types . . . 17
- 2.13 Dealing with references to preexisting types . . . 19
- 2.14 Generating visitors for preexisting types . . . 21

3 Advanced examples **22**
- 3.1 Visitors for open and closed data types . . . 22
- 3.2 Visitors for hash-consed abstract syntax trees . . . 25
- 3.3 From visitors to iterators . . . 25

4 Little-known aspects of OCaml objects **25**
- 4.1 Type inference for concrete and virtual methods . . . 25
- 4.2 Virtues of self-parameterized classes . . . 26
- 4.3 Simplifying the type of a self-parameterized class . . . 29
- 4.4 Monomorphic methods, polymorphic classes . . . 29
- 4.5 Where the expressiveness of OCaml's type system falls short . . . 29

5 Reference **34**
- 5.1 Parameters . . . 34
- 5.2 How to examine the generated code . . . 34
- 5.3 Structure of the generated code . . . 34
- 5.4 Supported forms of types . . . 36
- 5.5 Opaque components . . . 37
- 5.6 Alternate method names . . . 38
- 5.7 Alternate construction code in `map` visitors . . . 38

Les visites font toujours plaisir, si ce n'est en arrivant, du moins en partant.
— Jean de La Bruyère

1 Introduction

1.1 What is a visitor?

A visitor class for a data structure is a class whose methods implement a traversal of this data structure. By default, the visitor methods do not have interesting behavior: they just cause control to go down into the data structure and come back up, without performing any actual computation. Nevertheless, by defining subclasses where one method or a few methods are overridden, nontrivial behavior can be obtained. Therefore, visitors allow many operations on this data structure to be defined with little effort.

Visitor classes come in several varieties. An `iter` visitor traverses a data structure and returns no result. It can nevertheless have side effects, including updating a piece of mutable state, raising an exception, and performing input/output. A `map` visitor traverses a data structure and returns another data structure: typically, a copy of its argument that has been transformed in some way. An `endo` visitor is a variant of a `map` visitor that preserves physical sharing when possible. A `reduce` visitor traverses a data structure and returns a value that somehow summarizes it: computing the size of a data structure is a typical example. A `mapreduce` visitor performs the tasks of a `map` visitor and a `reduce` visitor at the same time, possibly allowing symbiosis between them. All of these can be viewed as special cases of the `fold` visitor, which performs a bottom-up computation over a data structure. The class `fold` is equipped with virtual methods that can be overridden (in a subclass) so as to specify what computation is desired.

Visitors also come in several arities. The visitors mentioned above have arity one: they traverse one data structure. However, it is sometimes necessary to simultaneously traverse two data structures of identical shape. For this purpose, there are visitors of arity two: here, they are known as `iter2`, `map2`, `reduce2`, `mapreduce2`, and `fold2` visitors.

1.2 What does this package offer?

Visitors have extremely regular structure. As a result, whereas implementing them by hand is boring and error-prone, generating them automatically is often possible. The `visitors` package extends the syntax of OCaml[1] so as to make it easy for the programmer to request the automatic generation of visitor classes. Visitor classes for many forms of user-defined data types can be generated and, if necessary, combined (via multiple inheritance) with hand-written visitor classes, making the framework quite powerful.

2 Walkthrough

2.1 Setup

In order to install the `visitors` package, an `opam` user should issue the following commands:

```
opam update
opam install visitors
```

To use the package, a user of Dune should add the following incantation to her project's `dune` file, inside a `library` or `executable` stanza:

```
(preprocess (pps visitors.ppx))
```

To use the `visitors` package in OCaml's interactive "toplevel" environment, launch `ocaml` and type the following commands:

[1] Technically, `visitors` is a plugin for `ppx_deriving`, which itself is a preprocessor extension for the OCaml compiler.

```
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 : _ -> int -> unit =
    fun env -> fun c0 -> let r0 = self#visit_int env c0 in
      ()
  method visit_EAdd : _ -> expr -> expr -> unit =
    fun env c0 c1 ->
    let r0 = self#visit_expr env c0 in
    let r1 = self#visit_expr env c1 in
    ()
  method visit_expr : _ -> expr -> unit =
    fun env this ->
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd (c0, c1) ->
        self#visit_EAdd env c0 c1
end
```

Figure 1: A visitor of the `iter` variety

```
#use "topfind";;
#require "visitors.ppx";;
```

2.2 Defining an `iter` visitor

Suppose we need to manipulate arithmetic expressions built out of integer literals and binary additions. The abstract syntax of these expressions can be described by an algebraic data type `expr`, shown in the first part of Figure 1. By annotating this type definition with `[@@deriving visitors { ... }]`, we request the automated generation of a visitor for expressions. The annotation `[@@deriving visitors { ... }]` must carry at least one parameter, `variety`, which indicates what variety of visitor is desired.

The code of the visitor class, which is automatically generated and in normal use remains invisible, is shown in the second part of Figure 1. The name of this class is by default the value of the `variety` parameter. It can be changed, if desired, by explicitly supplying a `name` parameter.

A visitor takes the form of an OCaml class, whose methods are named after the types and data constructors that appear in the type definition. In Figure 1, for instance, the method `visit_expr` is named after the type `expr`, while the methods `visit_EConst` and `visit_EAdd` are named after the data constructors `EConst` and `EAdd`.

Different varieties of visitors differ in the computation that is performed "on the way up", after the recursive calls have finished, therefore differ in the return types of the visitor methods. `iter` is the simplest variety. An `iter` visitor performs no computation on the way up, so its methods have return type `unit`.

In an `iter` visitor, the generated visitor methods do nothing. In Figure 1, for instance, the method `visit_expr` inspects its argument `this` and recursively invokes either `visit_EConst` or `visit_EAdd`, as appropriate. The method `visit_EConst` does nothing.[2] The method `visit_EAdd` performs two recursive calls to `visit_expr`, which does nothing, so `visit_EAdd` itself does nothing.

[2]More precisely, it calls the method `visit_int`, which is inherited from the class `VisitorsRuntime.iter`, and does nothing. This call to `visit_int` can be avoided, if desired, by using `(int[@opaque])` instead of `int`; see §5.5.

```
let count (e : expr) : int =
  let v = object
    val mutable count = 0
    method count = count
    inherit [_] iter as super
    method! visit_EAdd env e0 e1 =
      count <- count + 1;
      super#visit_EAdd env e0 e1
  end in
  v#visit_expr () e;
  v#count
```

This follows Figure 1.

Figure 2: Counting the number of addition nodes in an expression

Every method is parameterized with an environment `env`, which is carried down into every recursive call and is otherwise unused. The type of this environment is undetermined: it is up to the (user-defined) subclasses of the visitor class to decide what the type of `env` should be and (possibly) where and how this environment should be enriched.

The fields of a data constructor or record are traversed left to right, in the order they are declared. In a list-like data structure, the field that holds a pointer to the list tail should be declared last, so as to ensure that the traversal requires constant stack space.

2.3 Using an `iter` visitor

Naturally, traversing a data structure without actually computing anything is not a very sensible thing to do. Things become interesting when at least one visitor method is overridden so as to give rise to nontrivial behavior. Suppose, for instance, that we wish to count the number of addition nodes in an expression. This can be done as shown in Figure 2. We create an object `v` that is both a counter (that is, an object equipped with a mutable field `count`) and a visitor, and we override its method `visit_EAdd` so that the counter is incremented every time this method is invoked. There remains to run the visitor, by invoking its `visit_expr` method, and to return the final value of the counter. The environment, in this example, is unused; we let it have type `unit`.

This may seem a rather complicated way of counting the addition nodes in an expression. Of course, one could give a direct recursive definition of the function `count`, in a few lines of code, without using a visitor at all. The point of employing a visitor, as done in Figures 1 and 2, is that no changes to the code are required when the type of expressions is extended with new cases.

2.4 What is the type of a visitor?

As can be seen in this document, in a generated visitor, each method carries a type annotation. This annotation is partial: the type of the parameter `env` is unspecified. This is intentional: the type of this parameter can be influenced by the classes that are inherited via the `ancestors` parameter (§5.1). Because a ppx syntax extension transforms untyped syntax trees to untyped syntax trees, the `visitors` syntax extension does not have access to type information and cannot predict the type of the parameter `env`.

For this reason, the `visitors` syntax extension cannot generate the type of a visitors class. In other words, the annotation `[@@deriving visitors { ... }]` can be used only in an `%.ml` file, not in an `%.mli` file. When it is used in an `%.ml` file, the corresponding `%.mli` file should either be omitted altogether or be written by hand.

Figure 3 shows the type that is inferred via such a command for the `iter` visitor of Figure 1. This type is rather verbose, for two reasons. First, an explicit type equation, introduced by the `constraint` keyword, relates the type parameter `'self` with an OCaml object type that lists the public methods with their types. Second, the class `iter` has many more methods than one might think. This is because it

```
class virtual ['self] iter : object ('self)
  constraint 'self =
    < visit_EAdd : 'env -> expr -> expr -> unit;
      visit_EConst : 'env -> int -> unit;
      visit_expr : 'env -> expr -> unit;
      .. >
  method visit_EAdd : 'env -> expr -> expr -> unit
  method visit_EConst : 'env -> int -> unit
  method visit_expr : 'env -> expr -> unit
  (* These methods are inherited from [VisitorsRuntime.iter]: *)
  method private visit_array :
    'env 'a. ('env -> 'a -> unit) -> 'env -> 'a array -> unit
  method private visit_bool : 'env. 'env -> bool -> unit
  method private visit_bytes : 'env. 'env -> bytes -> unit
  (* ... and many more ... *)
end
```

Figure 3: An inferred type for the `iter` visitor of Figure 1

```
class virtual ['self] iter : object ('self)
  method visit_EAdd   : 'monomorphic. 'env -> expr -> expr -> unit
  method visit_EConst : 'monomorphic. 'env -> int -> unit
  method visit_expr   : 'monomorphic. 'env -> expr -> unit
end
```

Figure 4: A simplified type for the `iter` visitor of Figure 1

inherits a large number of private methods from the class `VisitorsRuntime.iter`. In the present case, all of these methods except `visit_int` are in fact unused.

Fortunately, this complicated type can be manually simplified. This is done in Figure 4. Two main simplifications have been performed. First, we have omitted all private methods. Indeed, the most important property of private methods in OCaml is precisely the fact that it is permitted to hide their existence. Second, we have omitted the type constraint that bears on the type variable `'self`, as it is in fact redundant: it is implicit in OCaml that the type of "self" must be an object type that lists the public methods. The bizarre-looking "`'monomorphic.`" annotations indicate that the methods have monomorphic types. (This notational trick is explained in §4.3.) This means that the type variable `'env` is not quantified at the level of each method[3], but at the level of the class. This means that the three methods must agree on the type of the environment, and that this type is presently undetermined, but can be determined in a subclass.

The class type shown in Figure 4 cannot be further simplified. The methods `visit_EConst` and `visit_EAdd` cannot be hidden, as they are public. That said, if one wished to hide them, one could add the parameter `public = ["visit_expr"]` to the annotation `[@@deriving visitors { ... }]` in Figure 1. These two methods would then be declared private in the generated code, so it would be permitted to hide their existence.

Although we have claimed earlier that one cannot in the general case predict the type of a generated visitor method, or even predict whether a generated method will be well-typed, it is possible to define a convention which in many cases can be adhered to. This convention is presented later on (§2.12.2, §2.13, Figure 22).

[3]That would be undesirable, as it would force each method to treat the environment as an argument of unknown type!

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

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

Figure 5: A visitor of the `map` variety

2.5 `map` visitors

An `iter` visitor returns no result. Although, as illustrated previously (§2.3), it can use private mutable state to accumulate information, there are applications for which such a visitor is not suitable. One class of such applications is tree transformations. To transform an expression into an expression, one should use a visitor of another variety, namely `map`.

A `map` visitor is shown in Figure 5. In comparison with the `iter` visitor of Figure 1, the generated code is identical, except that, instead of returning the unit value `()`, the method `visit_EConst` reconstructs an `EConst` expression, while the method `visit_EAdd` reconstructs an `EAdd` expression.

A `map` visitor behaves (by default) as an identity function: it constructs a copy of the data structure that it visits. If the data structure is immutable, this is rather pointless: in order to obtain nontrivial behavior, at least one method should be overridden. If the data structure is mutable, though, even the default behavior is potentially of interest: it constructs a deep copy of its argument.

2.6 `endo` visitors

`endo` visitors are a slight variation of `map` visitors. Whereas a `map` visitor systematically allocates a copy of the memory block that it receives as an argument, an `endo` visitor (Figure 6) first tests if the newly allocated block would have exactly the same contents as the original block, and if so, re-uses the original block instead. This trick allows saving memory: for instance, when a performing a substitution operation on a term, the subterms that are unaffected by the substitution are not copied.

One potential disadvantage of `endo` visitors, in comparison with `map` visitors, is that these runtime tests have a runtime cost. A more serious disadvantage is that `endo` visitors have less general types: in an `endo` visitor, the argument type and return type of every method must coincide, whence the name "`endo`". (An endomorphism is a function of a set into itself.) `map` visitors are not subject to this restriction: for an illustration, see §3.2 and Figure 28.

`endo` visitors should be created only for immutable data structures. Although the tool can produce an `endo` visitor for a mutable data structure, this is discouraged, as it may lead to unexpected behavior.

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

```
class virtual ['self] endo = object (self : 'self)
  inherit [_] VisitorsRuntime.endo
  method visit_EConst : _ -> expr -> int -> expr =
    fun env this c0 ->
    let r0 = self#visit_int env c0 in
    if c0 == r0 then this else EConst r0
  method visit_EAdd : _ -> expr -> expr -> expr -> expr =
    fun env this c0 c1 ->
    let r0 = self#visit_expr env c0 in
    let r1 = self#visit_expr env c1 in
    if (c0 == r0) && (c1 == r1) then this else EAdd (r0, r1)
  method visit_expr : _ -> expr -> expr =
    fun env this ->
    match this with
    | EConst c0 as this ->
        self#visit_EConst env this c0
    | EAdd (c0, c1) as this ->
        self#visit_EAdd env this c0 c1
end
```

Figure 6: A visitor of the endo variety

```
type expr =
  | EConst of (int[@opaque])
  | EAdd of expr * expr
  [@@deriving visitors { variety = "reduce" }]
```

```
class virtual ['self] reduce = object (self : 'self)
  inherit [_] VisitorsRuntime.reduce
  method visit_EConst : _ -> _ -> 's =
    fun env -> fun c0 -> let s0 = (fun this -> self#zero) c0 in
      s0
  method visit_EAdd : _ -> expr -> expr -> 's =
    fun env c0 c1 ->
    let s0 = self#visit_expr env c0 in
    let s1 = self#visit_expr env c1 in
    self#plus s0 s1
  method visit_expr : _ -> expr -> 's =
    fun env this ->
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd (c0, c1) ->
        self#visit_EAdd env c0 c1
end
```

Figure 7: A visitor of the `reduce` variety

```
let size : expr -> int =
  let v = object
    inherit [_] reduce as super
    inherit [_] VisitorsRuntime.addition_monoid
    method! visit_expr env e =
      1 + super # visit_expr env e
  end in
  v # visit_expr ()
```

This follows Figure 7.

Figure 8: Computing the size of an expression using a `reduce` visitor

2.7 `reduce` visitors

Whereas an `iter` visitor returns no result and a `map` visitor returns a data structure, a `reduce` visitor returns a "summary" of a data structure, so to speak. The summary of a term is computed by combining the summaries of its subterms. This requires summaries to inhabit a monoid, that is, a type equipped with a binary operation `plus` and its neutral element `zero`.

Figure 7 shows a `reduce` visitor for arithmetic expressions. In `visit_EAdd`, the summaries produced by the two recursive calls are combined using a call to `self#plus`. In `visit_EConst`, there are no recursive calls, so there is nothing to combine: the result is `self#zero`.

The virtual methods `zero` and `plus` are declared in the class `VisitorsRuntime.reduce`, which is automatically inherited. The type of the monoid elements, at this point, is undetermined: it is up to the (user-defined) subclasses of the class `reduce` to decide what this type should be and what the monoid operations should be.

As an example application, Figure 8 shows how to compute the size of an expression using a `reduce` visitor. We inherit the class `reduce`. We also inherit the class `VisitorsRuntime.addition_monoid`, which defines the methods `zero` and `plus` as the integer `0` and integer addition, respectively. There remains to override the method `visit_expr` so as to indicate that every node contributes 1 to the size of an expression.

Incidentally, this code is written in such a manner that a single visitor object is created initially and serves in every call to the function `size`. This style can be used when the visitor object is immutable and when the function that one wishes to define is monomorphic. When it cannot be used, one begins with `let size (e : expr) : int = ...` and ends with `v # visit_expr () e`. That style causes a new visitor object to be created every time `size` is called.

The size of an expression can also be computed using an `iter` visitor equipped with mutable state, in the style of Figure 2. It is mostly a matter of style whether such a computation should be performed using `iter` or `reduce`.

An `iter` visitor is in fact a special case of a `reduce` visitor, instantiated with the `unit` monoid. Thus, in principle, one could forget `iter` and always work with `reduce`. Nevertheless, it is preferable to work with `iter` when it is applicable, for reasons of clarity and efficiency.

2.8 `mapreduce` visitors

A `mapreduce` visitor performs the tasks of a `map` visitor and a `reduce` visitor at once. An example appears in Figure 9. Every visitor method returns a pair of a transformed term and a summary. In other words, it returns a pair of the results that would be returned by a `map` visitor method and by a `reduce` visitor method.

Like a `reduce` visitor, a `mapreduce` visitor performs a bottom-up computation. Like a `map` visitor, it constructs a new tree. By default, these two tasks are independent of one another. However, by overriding one or more methods, it is easy to establish a connection between them: typically, one wishes to exploit the information that was computed about a subtree in the construction of the corresponding new subtree. As an example, Figure 10 shows how to transform an arithmetic expression into an arithmetic

```
type 'info node =
  | EConst of int
  | EAdd of 'info expr * 'info expr

and 'info expr =
  { info: 'info; node: 'info node }

[@@deriving visitors { variety = "mapreduce" }]
```

```
class virtual ['self] mapreduce = object (self : 'self)
  inherit [_] VisitorsRuntime.mapreduce
  method virtual visit_'info : _
  method visit_EConst : _ -> int -> ('info_1 node * 's) =
    fun env c0 ->
    let (r0, s0) = self#visit_int env c0 in
    ((EConst r0), s0)
  method visit_EAdd :
    _ -> 'info_0 expr -> 'info_0 expr -> ('info_1 node * 's) =
    fun env c0 c1 ->
    let (r0, s0) = self#visit_expr env c0 in
    let (r1, s1) = self#visit_expr env c1 in
    ((EAdd (r0, r1)), (self#plus s0 s1))
  method visit_node : _ -> 'info_0 node -> ('info_1 node * 's) =
    fun env this ->
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd (c0, c1) ->
        self#visit_EAdd env c0 c1
  method visit_expr : _ -> 'info_0 expr -> ('info_1 expr * 's) =
    fun env this ->
    let (r0, s0) = self#visit_'info env this.info in
    let (r1, s1) = self#visit_node env this.node in
    ({ info = r0; node = r1 }, (self#plus s0 s1))
end
```

Figure 9: A visitor of the `mapreduce` variety

This follows Figure 9.

```
let annotate (e : _ expr) : int expr =
  let v = object
    inherit [_] mapreduce as super
    inherit [_] VisitorsRuntime.addition_monoid
    method! visit_expr env { info = _; node } =
      let node, size = super#visit_node env node in
      let size = size + 1 in
      { info = size; node }, size
    method visit_'info _env _info =
      assert false (* never called *)
  end in
  let e, _ = v # visit_expr () e in
  e
```

Figure 10: Decorating every subexpression with its size

expression where every subexpression is annotated with its size. (This example uses a parameterized type of decorated expressions, which is explained in §2.12.1. We suggest reading the explanations there first.) The transformation is carried out in one pass and in linear time. As in Figure 8, we use the addition monoid to compute integer sizes. This time, however, the visitor methods return not just a size, but a pair of a new expression and a size. The method `visit_expr` is overridden so as to store the size of the subexpression, `size`, in the `info` field of the new expression. Because the overridden method `visit_expr` does not call `visit_'info`, the latter method is never called: we provide a dummy definition of it.

Other example applications of `mapreduce` visitors include:

- decorating every subterm of a λ-term with the set of its free variables;
- decorating every internal node of abstract syntax tree with a region in the source code (assuming that every leaf carries such a region already).

2.9 `fold` visitors

The varities of visitors presented up to this point differ in the computation that they perform on the way up, after the recursive calls. As we have seen, `iter` visitors perform no computation at all; `map` and `endo` visitors reconstruct a term; `reduce` visitors perform a series of monoid operations. Each variety follows a baked-in pattern, which has been programmed ahead of time, and cannot be changed. What if a different form of computation, which has not been envisioned by the author of the `visitors` syntax extension, is needed?

This is where `fold` visitors come in. A `fold` visitor declares virtual methods that are called on the way up and can be overridden by the user (in a subclass) so as to implement the desired computation. Figure 11 shows a `fold` visitor. Two virtual methods, `build_EConst` and `build_EAdd`, are declared. They are invoked by `visit_EConst` and `visit_EAdd`, respectively.

In a `fold` visitor, the return type of the visitor methods is not fixed ahead of time. It is up to the (user-defined) subclasses of the visitor class to decide what this type should be.

As an example application, Figure 12 shows how a `fold` visitor can be used to convert the visited data structure to an entirely different format. In this example, the type `person` is a record type, whose fields are `firstname` and `surname`. The type `crowd` is isomorphic to a list of persons, but, for some reason, it is declared as an algebraic data type equipped with its own data constructors, `Nobody` and `Someone`. Suppose we wish to convert a `crowd` to a list of pairs of strings. We can do so by creating a visitor object that inherits the class `fold` and provides concrete implementations of the methods `build_person`, `build_Nobody`, and `build_Someone`. Our implementation of `build_person` simply allocates a pair, while our implementations of `build_Nobody` and `build_Someone` respectively build an empty list and a nonempty list. Thus, the return type of the methods `build_person` and `visit_person` is `string * string`, while the return type of the methods `build_Nobody`, `build_Someone`, and `visit_crowd` is `(string * string) list`. In a `fold` visitor, not all methods need have the same return type!

If we had chosen to return `f ^ s` instead of `(f, s)` in `build_person`, then a crowd would be converted to a `string list`. A `fold` visitor offers great flexibility.

All previous varieties of visitors are special cases of `fold` visitors. The specialized varieties of visitors are more convenient to use, when they can be used, because they do not require the user to provide `build_` methods. Yet, `fold` visitors are in principle more versatile.[4]

2.10 Visitors of arity two

The visitors introduced so far traverse one tree at a time. There are situations where one wishes to simultaneously traverse two trees, which one expects have the same structure. For this purpose, one can use a visitor of arity 2. Every variety except `endo` is available at arity 2.

[4]This would be true in an untyped setting, but is not quite true in OCaml, due to restrictions imposed by OCaml's type discipline (§4.5).

```
type expr =
  | EConst of (int[@opaque])
  | EAdd of expr * expr
  [@@deriving visitors { variety = "fold" }]
```

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

Figure 11: A visitor of the `fold` variety

```
type person = {
  firstname: string[@opaque];
  surname:   string[@opaque]
}

and crowd =
  | Nobody
  | Someone of person * crowd
[@@deriving visitors { variety = "fold" }]

let convert : crowd -> (string * string) list =
  let v = object
    inherit [_] fold
    method build_person  () f s = (f, s)
    method build_Nobody  ()     = []
    method build_Someone () p c = p :: c
  end
  in v # visit_crowd ()
```

Figure 12: Converting towards unrelated types using a `fold` visitor

```
type expr =
  | EConst of int
  | EAdd of expr * expr
  [@@deriving visitors { variety = "iter2"; concrete = true }]
```

```
class ['self] iter2 = object (self : 'self)
  inherit [_] VisitorsRuntime.iter2
  method visit_EConst : _ -> int -> int -> unit =
    fun env c0_0 c0_1 -> let r0 = self#visit_int env c0_0 c0_1 in
      ()
  method visit_EAdd : _ -> expr -> expr -> expr -> expr -> unit =
    fun env c0_0 c0_1 c1_0 ->
    fun c1_1 ->
    let r0 = self#visit_expr env c0_0 c0_1 in
    let r1 = self#visit_expr env c1_0 c1_1 in
    ()
  method fail_expr : _ -> expr -> expr -> unit =
    fun env -> fun this_0 -> fun this_1 -> VisitorsRuntime.fail ()
  method visit_expr : _ -> expr -> expr -> unit =
    fun env this_0 this_1 ->
    match (this_0, this_1) with
    | (EConst c0_0, EConst c0_1) ->
        self#visit_EConst env c0_0 c0_1
    | (EAdd (c0_0, c1_0), EAdd (c0_1, c1_1)) ->
        self#visit_EAdd env c0_0 c0_1 c1_0 c1_1
    | (this_0, this_1) ->
        self#fail_expr env this_0 this_1
end
```

Figure 13: A visitor of the `iter2` variety

```
let equal : expr -> expr -> bool =
  VisitorsRuntime.wrap2 (new iter2 # visit_expr ())
```

This follows Figure 13.

Figure 14: Determining whether two expressions are syntactically equal

```
let tag : expr -> int = function
  | EConst _ -> 0
  | EAdd _   -> 1

exception Different of int

let compare (i1 : int) (i2 : int) : unit =
  if i1 <> i2 then
    raise (Different (if i1 < i2 then -1 else 1))

class compare = object
  inherit [_] iter2
  method! visit_int  _ i1 i2 = compare i1 i2
  method! fail_expr () e1 e2 = compare (tag e1) (tag e2)
end

let compare (e1 : expr) (e2 : expr) : int =
  try new compare # visit_expr () e1 e2; 0 with Different c -> c
```

This follows Figure 13.

Figure 15: Determining which way two expressions are ordered

```
type unop =
  | UnaryMinus

and binop =
  | BinaryMinus
  | BinaryAdd
  | BinaryMul
  | BinaryDiv

and expr =
  | EConst of int
  | EUnOp of unop * expr
  | EBinOp of expr * binop * expr

[@@deriving visitors { variety = "iter" },
            visitors { variety = "map"  }]
```

Figure 16: Visitors for a family of types

As an illustration, Figure 13 shows an `iter2` visitor. There, the method `visit_expr` expects an environment and two expressions. These expressions must have identical structure: indeed, if `visit_expr` finds that they exhibit different tags at the root, say `EConst` versus `EAdd`, then it invokes the method `fail_expr`, whose default implementation calls the function `VisitorsRuntime.fail`. This function throws the exception `VisitorsRuntime.StructuralMismatch`.

In Figure 13, we have added the optional parameter `concrete = true` to indicate that the generated class should not be virtual. (By default, every generated class is declared `virtual`.) We do this because, in the illustration that follows, we wish to instantiate this class.

As an illustration, in Figure 14, we use an `iter2` visitor to write a function that tests whether two expressions are syntactically equal. This is just a matter of performing a synchronous traversal of the two expressions and detecting a `StructuralMismatch` exception: if this exception is raised, one must return `false`, otherwise one must return `true`. We rely on the fact that the method `visit_int`, which is inherited from the class `VisitorsRuntime.iter2`, fails when its two integer arguments are unequal.

The convenience functions `VisitorsRuntime.wrap` and `VisitorsRuntime.wrap2` run a user-supplied function (of arity 1 or 2, respectively) within an exception handler and return a Boolean result, which is `true` if no exception was raised. Here, we run the function `new iter2 # visit_expr ()`, whose type is `expr -> expr -> unit`, in the scope of such a handler.

Naturally, to test whether two expressions are syntactically equal, one could also use the primitive equality operator `=`. Alternatively, one could exploit `ppx_deriving` and annotate the type definition with `[@@deriving eq]`. Visitors offer greater flexibility: for instance, if our arithmetic expressions contained variables and binders, we could easily define an operation that tests whether two expressions are α-equivalent.

As a second illustration, in Figure 15, we use an `iter2` visitor to write a lexicographic ordering function for expressions. Again, this involves a synchronous traversal of the two expressions. When a mismatch is detected, however, one must not raise a `StructuralMismatch` exception: instead, one must raise an exception that carries one bit of information, namely, which of the two expressions is strictly "smaller" in the ordering. The exception `Different` is introduced for this purpose; it carries an integer return code, which by convention is either -1 or +1. Two visitor methods must be overridden, namely `visit_int`, which is in charge of detecting a mismatch between two integer constants, and `fail_expr`, which is invoked when a mismatch between (the head constructors of) two expressions is detected. The auxiliary function `tag` returns an integer code for the head constructor of an expression. It must be hand-written. One could in principle write such a function once and for all by using the undocumented operations in OCaml's `Obj` module, but that is discouraged.

2.11 Visitors for a family of types

Visitors can be generated not just for one type definition, but for a family of type definitions. In Figure 16, we propose a definition of arithmetic expressions that involves three algebraic data types, namely `unop`, `binop`, and `expr`. We request the generation of two visitors, namely an `iter` visitor and a `map` visitor. This causes the generation of just two classes, named `iter` and `map`, respectively. Each of these classes has visitor methods for every type (namely `visit_unop`, `visit_binop`, `visit_expr`) and for every data constructor (namely `visit_UnaryMinus`, `visit_BinaryMinus`, and so on).

Note that, for the `[@@deriving visitors { ... }]` annotation to apply to the entire family, as opposed to just the type `expr`, the types `unop`, `binop`, and `expr` in Figure 16 are declared simultaneously: that is, their declarations are separated with the keyword **`and`**.

2.12 Visitors for parameterized types

Visitors can be generated for parameterized types, too. However, there are two ways in which this can be done. Here is why.

To visit a data type where some type variable `'a` occurs, one must know how to visit a value of type `'a`. There are two ways in which this information can be provided. One way is to assume that there is a **virtual visitor method** `visit_'a` in charge of visiting a value of type `'a`. Another way is to pass a **visitor function** `visit_'a` as an argument to every visitor method.

These two approaches differ in their expressive power. The virtual-visitor-method approach implies that the visitor methods must have monomorphic types: roughly speaking, the type variable `'a` appears free in the type of every visitor method. The visitor-function approach implies that the visitor methods can have polymorphic types: roughly speaking, each method independently can be polymorphic in `'a`. For this reason, we refer to these two approaches as the **monomorphic** approach and the **polymorphic** approach, respectively.

The monomorphic approach offers the advantage that the type of every method is inferred by OCaml. Correct and most general (monomorphic) types are obtained even if certain hand-written visitor methods (provided via the `ancestors` parameter) have unconventional types.

The polymorphic approach offers the advantage that visitor methods can receive polymorphic types. If the type `container` is parameterized with a type variable `'a`, then the method `visit_container` can be assigned a type that is universally quantified in `'a`, of the following form:

```
method visit_container:
  'env 'a .
  ('env -> 'a -> ...) ->
  'env -> 'a container -> ...
```

The types of the `visit_list` methods, shown later on (§2.13, Figure 22), follow this pattern. Because `visit_container` is polymorphic, taking multiple instances of the type `container`, such as `apple container` and `orange container`, and attempting to generate visitor methods for these types, poses no difficulty. This works even if the definition of `'a container` mentions other instances of this type, such as `('a * 'a) container`. In other words, in the polymorphic approach, irregular algebraic data types (§5.4) are supported.

One downside of the polymorphic approach is that, because polymorphic types cannot be inferred by OCaml, the `visitors` syntax extension must generate polymorphic type annotations. Therefore, it must be able to predict the type of every visitor method. This requires that any visitor methods inherited via `ancestors` adhere to a certain convention (§2.12.2, §2.13, Figure 22).

In summary, both the monomorphic approach and the polymorphic approach are currently supported (§2.12.1, §2.12.2). The parameter `polymorphic` allows choosing between them. As a rule of thumb, we suggest setting `polymorphic = true`, as this produces visitors that compose better.

```
type 'info node =
  | EConst of int
  | EAdd of 'info expr * 'info expr

and 'info expr =
  { info: 'info; node: 'info node }

[@@deriving visitors { variety = "map" }]
```

```
class virtual ['self] map = object (self : 'self)
  inherit [_] VisitorsRuntime.map
  method virtual visit_'info : _
  method visit_EConst : _ -> int -> 'info_1 node =
    fun env -> fun c0 -> let r0 = self#visit_int env c0 in
      EConst r0
  method visit_EAdd :
    _ -> 'info_0 expr -> 'info_0 expr -> 'info_1 node =
    fun env c0 c1 ->
    let r0 = self#visit_expr env c0 in
    let r1 = self#visit_expr env c1 in
    EAdd (r0, r1)
  method visit_node : _ -> 'info_0 node -> 'info_1 node =
    fun env this ->
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd (c0, c1) ->
        self#visit_EAdd env c0 c1
  method visit_expr : _ -> 'info_0 expr -> 'info_1 expr =
    fun env this ->
    let r0 = self#visit_'info env this.info in
    let r1 = self#visit_node env this.node in
    { info = r0; node = r1 }
end
```

Figure 17: A “monomorphic-method” visitor for a parameterized type of decorated expressions

This follows Figure 17.

```
let strip (e : _ expr) : unit expr =
  let v = object
    inherit [_] map
    method visit_'info _env _info = ()
  end in
  v # visit_expr () e

let number (e : _ expr) : int expr =
  let v = object
    inherit [_] map
    val mutable count = 0
    method visit_'info _env _info =
      let c = count in count <- c + 1; c
  end in
  v # visit_expr () e
```

Figure 18: Working with different types of decorations

2.12.1 Monomorphic visitor methods for parameterized types

We begin with an example of the **monomorphic** mode. This mode can be explicitly requested by writing `polymorphic = false` as part of the `[@@deriving visitors { ... }]` annotation. It is the default mode.

In Figure 17, we define a variant of arithmetic expressions where every tree node is decorated with a value of type `'info`. We request the generation of a `map` visitor, whose code is shown in the second part of Figure 17. The generated code has exactly the same structure as in the previous sections. The only new feature is that the class `map` now has a virtual method, `visit_'info`. The general rule is, for each type parameter, there is one virtual method, named after it.

The visitor methods are **not** declared polymorphic in a type variable `'info`, or in two type variables `'info1` and `'info2`, as one might perhaps expect. In fact, they must not be declared polymorphic: indeed, the user who implements `visit_'info` in a subclass of `map` may wish to provide an implementation that expects and/or produces specific types of information.

As a result, a visitor **object** is monomorphic: its method `visit_'info` must have type `info1 -> info2` for certain specific types `info1` and `info2`. Fortunately, because it is parameterized over `'self`,[5] the visitor **class** is polymorphic: two distinct visitor objects can have distinct types.

Although every **automatically generated** method is monomorphic, a visitor class can nevertheless **inherit** polymorphic methods from a parent class, whose name is specified via the `ancestors` parameter (§5.1). For instance, the `visit_list` methods provided by the classes `VisitorsRuntime.iter`, `VisitorsRuntime.map`, and so on, are polymorphic in the types of the list elements. (See §2.13 for more information on the treatment of preexisting types.)

Figure 18 presents two example uses of the class `map` defined in Figure 17. In the first example, we define a function `strip`, of type `'info expr -> unit expr`, which strips off the decorations in an arithmetic expression, replacing them with unit values. In the second example, we define a function `number`, of type `'info expr -> int expr`, which decorates each node in an arithmetic expression with a unique integer number.[6]

2.12.2 Polymorphic visitor methods for parameterized types

We continue with an example of the **polymorphic** mode. This mode can be explicitly requested by writing `polymorphic = true` as part of the `[@@deriving visitors { ... }]` annotation. It is available for all varieties of visitors except `fold` and `fold2`. The reason why it seems difficult to come up with a satisfactory polymorphic type for `fold` visitor methods is explained later on (§4.5).

In Figure 19, we again have arithmetic expressions where every tree node is decorated with a value of type `'info`. We again request a `map` visitor but, this time, we specify `polymorphic = true`. In order to make the generated code shorter, we specify `data = false` (§5.1), which causes the methods `visit_EConst` and `visit_EAdd` to disappear. (They are inlined at their call sites.)

The generated code is the same as in the previous section, except that `visit_'info` is not a method any more. Instead, it is a function, which is passed as an argument to every visitor method.

The class `map` does not have any virtual methods. It can thus be declared as concrete class: to this end, we specify `concrete = true` (§5.1). Without this indication, it would be declared `virtual`.

Because `visit_'info` is an argument of every visitor method, every visitor method can be declared polymorphic in the type variables `'env`, `'info_0` and `'info_1`, where the function `visit_'info` has type `'env -> 'info_0 -> 'info_1`. The fact that we would like every method to have a polymorphic type cannot be inferred by OCaml. For this reason, the generated code contains explicit polymorphic type annotations.

Figure 20 presents two example uses of the class `map` defined in Figure 19. As in Figure 18, we define two functions `strip` and `number`. A striking feature of this code is that a single visitor object `v`

[5] We explain in §4.2 why all visitor classes are parameterized over `'self`.

[6] Because the `info` field appears before the `node` field in the definition of the type `expr`, and because fields are visited left-to-right, we get a prefix numbering scheme. By exchanging these fields, we would get postfix numbering.

```
type 'info node =
  | EConst of int
  | EAdd of 'info expr * 'info expr

and 'info expr =
  { info: 'info; node: 'info node }

[@@deriving visitors { variety = "map"; polymorphic = true;
                       concrete = true; data = false }]
```

```
class ['self] map = object (self : 'self)
  inherit [_] VisitorsRuntime.map
  method visit_node :
    'env 'info_0 'info_1 .
    ('env -> 'info_0 -> 'info_1) ->
    'env -> 'info_0 node -> 'info_1 node =
    fun visit_'info env this ->
    match this with
    | EConst c0 ->
        let r0 = self#visit_int env c0 in
        EConst r0
    | EAdd (c0, c1) ->
        let r0 = self#visit_expr visit_'info env c0 in
        let r1 = self#visit_expr visit_'info env c1 in
        EAdd (r0, r1)
  method visit_expr :
    'env 'info_0 'info_1 .
    ('env -> 'info_0 -> 'info_1) ->
    'env -> 'info_0 expr -> 'info_1 expr =
    fun visit_'info env this ->
    let r0 = visit_'info env this.info in
    let r1 = self#visit_node visit_'info env this.node in
    { info = r0; node = r1 }
end
```

Figure 19: A “polymorphic-method” visitor for a parameterized type of decorated expressions

This follows Figure 19.

```
let v = new map

let strip : _ expr -> unit expr =
  let visit_'info _env _info = () in
  fun e ->
    v # visit_expr visit_'info () e

let number : _ expr -> int expr =
  let visit_'info count _info =
    let c = !count in count := c + 1; c in
  fun e ->
    let count = ref 0 in
    v # visit_expr visit_'info count e
```

Figure 20: Working with different types of decorations

```
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 : _ -> int -> unit =
    fun env -> fun c0 -> let r0 = self#visit_int env c0 in
      ()
  method visit_EAdd : _ -> expr list -> unit =
    fun env c0 -> let r0 = self#visit_list self#visit_expr env c0 in
      ()
  method visit_expr : _ -> expr -> unit =
    fun env this ->
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd c0 ->
        self#visit_EAdd env c0
end
```

Figure 21: Using preexisting (parameterized) types, such as `int` and `list`

is now allocated, once and for all, and is used in every subsequent call to `strip` and `number`. The two `visit_'info` functions are also defined once and for all.[7]

In the definition of `number`, we choose to store the current count in a reference, `count`, and to let `count` play the role of the "environment". Thus, we initially pass `count` to `visit_expr`, and `visit_'info` receives `count` as its "environment" argument.

The polymorphic type annotations that are automatically generated (Figure 19) follow a certain fixed convention. If the user throws in hand-written visitor methods via the `ancestors` parameter, then those hand-written methods must adhere to the same convention (or the generated code would be ill-typed). This convention is illustrated in the next section (§2.13) with the example of the `visit_list` methods.

A type variable is not allowed to occur under an `[@opaque]` annotation (§5.5). Indeed, annotating a type variable `'a` with `[@opaque]` would cause special-purpose visit code to be generated, whose type is not as polymorphic as required by the above convention.

2.13 Dealing with references to preexisting types

A type definition can contain references to the types that are being defined, also known as **local** types. For instance, in Figure 1, the definition of `EAdd` contains two references to a local type, namely `expr`.

A type definition can also contain references to preexisting types, also known as **nonlocal** types. For instance, in Figure 1, the definition of `EConst` contains a reference to a nonlocal type, namely `int`, which happens to be one of OCaml's primitive types. In Figure 21, the definition of `EAdd` contains a reference to a parameterized nonlocal type, namely `list`, which happens to be defined in OCaml's standard library.

The treatment of local types has been illustrated in the previous sections. In short, for every local type, a visitor method is called, and is defined: for instance, for the local type `expr`, we define the (concrete) method `visit_expr`.

The treatment of nonlocal types is the same, except the visitor method is not defined, nor declared. That is, it is called, but is neither defined (as a concrete method) or declared (as a virtual method). Therefore, its definition must be provided by an ancestor class.

[7] Although we have nested these functions inside the definitions of `strip` and `number`, they are closed, so they could be hoisted out to the top level, if desired.

```
class ['self] iter : object ('self)
  method private visit_list: 'env 'a .
    ('env -> 'a -> unit) -> 'env -> 'a list -> unit
end
class ['self] map : object ('self)
  method private visit_list: 'env 'a 'b .
    ('env -> 'a -> 'b) -> 'env -> 'a list -> 'b list
end
class ['self] endo : object ('self)
  method private visit_list: 'env 'a .
    ('env -> 'a -> 'a) -> 'env -> 'a list -> 'a list
end
class virtual ['self] reduce : object ('self)
  inherit ['s] monoid
  method private visit_list: 'env 'a .
    ('env -> 'a -> 's) -> 'env -> 'a list -> 's
end
class virtual ['self] mapreduce : object ('self)
  inherit ['s] monoid
  method private visit_list: 'env 'a 'b .
    ('env -> 'a -> 'b * 's) -> 'env -> 'a list -> 'b list * 's
end
class ['self] iter2 : object ('self)
  method private visit_list: 'env 'a 'b .
    ('env -> 'a -> 'b -> unit) -> 'env -> 'a list -> 'b list -> unit
end
class ['self] map2 : object ('self)
  method private visit_list: 'env 'a 'b 'c .
    ('env -> 'a -> 'b -> 'c) -> 'env -> 'a list -> 'b list -> 'c list
end
class virtual ['self] reduce2 : object ('self)
  inherit ['s] monoid
  method private visit_list: 'env 'a 'b .
    ('env -> 'a -> 'b -> 's) -> 'env -> 'a list -> 'b list -> 's
end
class virtual ['self] mapreduce2 : object ('self)
  inherit ['s] monoid
  method private visit_list: 'env 'a 'b 'c .
    ('env -> 'a -> 'b -> 'c * 's) ->
    'env -> 'a list -> 'b list -> 'c list * 's
end
```

Figure 22: Conventional types of polymorphic visitor methods

For most of OCaml primitive or built-in types, support is provided by the module `VisitorsRuntime`. This module contains several classes named `iter`, `map`, and so on; each of them supplies methods named `visit_int`, `visit_list`, and so on.[8] As is evident in Figures 1, 5, and so on, the generated visitor automatically inherits from the appropriate class in `VisitorsRuntime`, so it receives default implementations of the methods `visit_int`, `visit_list`, and so on.

The visitor methods for parameterized data types (`array`, `Lazy.t`, `list`, `option`, `ref`, `result`) are polymorphic, so it is not a problem if both lists of apples and lists of oranges need to be traversed. The types of these methods follow a strict **convention**, illustrated in Figure 22 with the example of `visit_list`.

The `visit_list` methods in the classes `VisitorsRuntime.iter`, `VisitorsRuntime.map`, and so on, have been hand-written, but could equally well have been generated, with `polymorphic = true`: their types would be the same. This illustrates the fact that, with `polymorphic = true`, **visitors are compositional**. That is, if the definition of the type `'b bar` refers to the type `'a foo`, then one can **separately** generate a visitor class for `'a foo` and generate a visitor class for `'b bar`, which inherits the previous class.

At a primitive type, it is advisable to carefully consider what behavior is desired. On the one hand, perhaps the inherited method `visit_int` need not be invoked in the first place; this behavior can be obtained by using `(int[@opaque])` instead of `int`. (See §5.5 for details.) This is done, for instance, in Figure 7, where one can check that no call to `visit_int` is generated. On the other hand, when one decides to use an inherited method, one should make sure that one understands its behavior. The methods `visit_array` and `visit_ref` in the class `VisitorsRuntime.map`, for instance, perform a copy of a mutable memory block: one should be aware that such a copy is taking place. If this behavior is undesirable, it can be overridden.

It is possible to inherit as many classes as one wishes, beyond those defined in `VisitorsRuntime`. This is done via the `ancestors` parameter (§5.1). It is also possible to **not** inherit any methods from `VisitorsRuntime`. This is done via the `nude` parameter (§5.1).

2.14 Generating visitors for preexisting types

Because the `[@@deriving visitors { ... }]` annotation must be attached to the type definition, it may seem as if it is impossible to generate a visitor for a type whose definition is out of reach. Suppose, for instance, that the type `expr` of arithmetic expressions is defined in a module `Expr`, which, for some reason, we cannot modify. Can we generate a visitor for this type?

Fortunately, the answer is positive. The basic trick, documented in the `ppx_deriving` README, consists in defining a new type `expr` of arithmetic expressions and to explicitly declare that it is equal to the preexisting type `Expr.expr`, as follows.

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

As can be seen above, the new definition of `expr` can be annotated with `[@@deriving visitors { ... }]`, yielding a visitor for the new type `expr`, which by definition, is equal to the preexisting type `Expr.expr`. Thus, this visitor class be used to traverse expressions of type `Expr.expr`.

This approach works, but requires repeating the definition of the type `expr`. This duplication can be eliminated thanks to the `ppx_import` syntax extension, as follows:

[8] As an exception to this rule, the classes `VisitorsRuntime.fold` and `VisitorsRuntime.fold2` do not supply any methods, because we do not wish to prematurely fix the types of the visitor methods. Please consult `VisitorsRuntime.ml` to check which methods exist and what they do.

```
type expr =
  [%import: Expr.expr]
  [@@deriving visitors { variety = "iter" }]
```

This expands to the code shown previously. (To use ppx_import and visitors simultaneously, if you are using Dune, include (preprocess (staged_pps ppx_import visitors.ppx)) in your dune file.) As icing on the cake, ppx_import allows decorating the type definition, on the fly, with new attributes. In the following examples, we replace all occurrences of int with int[@opaque] (§5.5), so as to ensure that the generated visitor does not invoke the method visit_int:

```
type expr =
  [%import: Expr.expr [@with int := int[@opaque]]]
  [@@deriving visitors { variety = "iter" }]
```

3 Advanced examples

3.1 Visitors for open and closed data types

The algebraic data types of arithmetic expressions shown in the previous section (§1) are **closed**. That is, the type expr is recursive: an expression of type expr has subexpressions of type expr.

It is often desirable, for greater flexibility, to first define an **open** type of arithmetic expressions. Such a type, say oexpr, is parameterized over a type variable 'expr. It is not recursive: an expression of type 'expr oexpr has subexpressions of type 'expr. It is shown in Figure 23. Naturally, we may request the generation of visitors for the type oexpr. In Figure 23, we generate a class of map visitors, which we name omap. (This is an example of using an explicit name parameter.) As explained earlier (§2.12.1), because the type oexpr is parameterized over the type variable 'expr, the visitor class has a virtual method, visit_'expr. In this example, we use the monomorphic mode (§2.12.1), but could just as well use the polymorphic mode (§2.12.2): this is left as an exercise for the reader.

A closed (recursive) type of expressions, expr, can then be defined in terms of expr. This is done in Figure 24. In type-theoretical terms, one would like to define expr as the fixed point of the functor oexpr [3]. That is, roughly speaking, one would like to define **type expr = expr oexpr**. This is not accepted by OCaml, though;[9] we work around this limitation by making expr an algebraic data type whose single data constructor is named E.[10]

Let us now construct a visitor class for the type expr. It is easy to do so, by hand, in a few lines of code.We define a class map, a subclass of omap, and provide a concrete implementation of the virtual method visit_'expr. In the definition of the type expr, the type variable 'expr is instantiated with expr, so the method visit_'expr expects an argument of type expr and must return a result of type expr. We deconstruct the argument using the pattern E e. Therefore, the variable e has type expr oexpr and is a suitable argument to the method visit_oexpr. After this call, we perform the same step in reverse: the result of the call has type expr oexpr, so we wrap it in an application of the data constructor E and obtain a result of type expr.

The visitor class map can now be used to implement transformations of arithmetic expressions, that is, functions of type expr -> expr. As an example, let us implement a transformation whose effect is to double every integer constant in an arithmetic expression. This is done in Figure 26. As expected, it suffices to construct a visitor object that inherits map and overrides the method visit_EConst.

[9] It would be accepted by OCaml with the command line switch -rectypes, which instructs the typechecker to tolerate equirecursive types. However, this is not a good idea, as it causes the typechecker to suddenly accept many meaningless programs and infer bizarre types for them.

[10] We mark this algebraic data type [@@unboxed], which guarantees that there is no runtime cost associated with the data constructor E. Although expr and expr oexpr are considered distinct types by the OCaml typechecker, they have the same runtime representation.

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

```
class virtual ['self] omap = object (self : 'self)
  inherit [_] VisitorsRuntime.map
  method virtual visit_'expr : _
  method visit_EConst : _ -> int -> 'expr_1 oexpr =
    fun env -> fun c0 -> let r0 = self#visit_int env c0 in
      EConst r0
  method visit_EAdd : _ -> 'expr_0 -> 'expr_0 -> 'expr_1 oexpr =
    fun env c0 c1 ->
    let r0 = self#visit_'expr env c0 in
    let r1 = self#visit_'expr env c1 in
    EAdd (r0, r1)
  method visit_oexpr : _ -> 'expr_0 oexpr -> 'expr_1 oexpr =
    fun env this ->
    match this with
    | EConst c0 ->
        self#visit_EConst env c0
    | EAdd (c0, c1) ->
        self#visit_EAdd env c0 c1
end
```

Figure 23: An open type of arithmetic expressions

```
type expr =
  E of expr oexpr [@@unboxed]

class ['self] map = object (self : 'self)
  inherit [_] omap
  method visit_'expr env (E e) =
    E (self#visit_oexpr env e)
end
```

This follows Figure 23.

Figure 24: A closed type of arithmetic expressions

```
open Hashcons

type hexpr =
  H of hexpr oexpr hash_consed [@@unboxed]

let table =
  create 128

let h (e : hexpr oexpr) : hexpr =
  H (hashcons table e)

class ['self] hmap = object (self : 'self)
  inherit [_] omap
  method visit_'expr env (H { node = e; _ }) =
    h (self#visit_oexpr env e)
end
```

This follows Figure 23.

Figure 25: A closed type of hash-consed arithmetic expressions

This follows Figures 23 and 24.

```
let double : expr -> expr =
  let v = object
    inherit [_] map
    method! visit_EConst _env k =
      EConst (2 * k)
  end in
  v # visit_'expr ()
```

Figure 26: A transformation of ordinary arithmetic expressions

This follows Figures 23 and 25.

```
let double : hexpr -> hexpr =
  let v = object
    inherit [_] hmap
    method! visit_EConst _env k =
      EConst (2 * k)
  end in
  v # visit_'expr ()
```

Figure 27: A transformation of hash-consed arithmetic expressions

This follows Figures 23, 24, and 25.

```
let import : expr -> hexpr =
  let v = object (self)
    inherit [_] omap
    method visit_'expr _env (E e) =
      h (self#visit_oexpr _env e)
  end in
  v # visit_'expr ()

let export : hexpr -> expr =
  let v = object (self)
    inherit [_] omap
    method visit_'expr _env (H { node = e; _ }) =
      E (self#visit_oexpr _env e)
  end in
  v # visit_'expr ()
```

Figure 28: Conversions between ordinary and hash-consed arithmetic expressions

3.2 Visitors for hash-consed abstract syntax trees

On top of the open data type `oexpr` of the previous section (§3.1), one can define not just the closed data type `expr` of ordinary arithmetic expressions, but also other closed data types of expressions where every node is annotated with information.

As an example, let us define a type `hexpr` of hash-consed (that is, maximally-shared) arithmetic expressions. We use Filliâtre and Conchon's library [2], which can be found in `opam` under the name `hashcons`.

The definition of the type `hexpr` appears in Figure 25. It is analogous to the definition of the type `expr` (Figure 24), with an added twist: instead of taking the fixed point of the functor `_ oexpr`, we take the fixed point of the functor `_ oexpr hash_consed`. By looking up the definition of the type `hash_consed` in `hashcons.mli`, one finds that this means that every node in an arithmetic expression carries certain information (namely a unique tag and a hash) that are used to enforce maximal sharing.

Enforcing maximal sharing requires maintaining a mutable table where all arithmetic expressions ever constructed are stored. (This is in fact a weak hash table.) We initialize such a table by calling the function `Hashcons.create`. This table is then populated by the function `h`, a smart constructor. This function takes a candidate expression of type `hexpr oexpr` and returns an expression of type `hexpr`, which is either allocated anew or found in the table (should an identical expression already exist).

We can now construct a visitor class for the type `hexpr`. As in the previous section (§3.1), we do so by hand in a few lines of code. The overall structure of this code is the same as in Figure 24. The only difference is that the method `visit_'expr` must now traverse two levels of type structure, corresponding to `_ oexpr hash_consed`. It deconstructs this structure by using the pattern `H { node = e ; _ }`,[11] and reconstructs it by applying the smart constructor `h`.

A function `double` can be defined for hash-consed arithmetic expressions in exactly the same manner as we defined `double` for ordinary arithmetic expressions: compare Figure 26 and Figure 27.

The visitor class `omap` for open arithmetic expressions (Figure 23) can be exploited to define conversions between different types of arithmetic expressions. This is illustrated in Figure 28. There, the function `import` converts an ordinary expression to a hash-consed expression, thereby imposing maximal sharing. Conversely, the function `export` converts a hash-consed expression into an ordinary expression, thereby abandoning all sharing (and possibly causing an exponential explosion). The implementation of these functions is simple: it is just a matter of overriding `visit_'expr` so as to deconstruct one kind of expression and reconstruct the other kind.

3.3 From visitors to iterators

It is possible to use a visitor to implement an iterator, that is, an object that traverses a container data structure and produces its elements on demand. The story is told in a blog post [5].

4 Little-known aspects of OCaml objects

In this section, we document a few relatively little-known aspects of OCaml's class and object system that play an essential role in our visitors.

4.1 Type inference for concrete and virtual methods

It is well-known that OCaml can infer the type of a concrete method. In the following simple example, it infers that the field `x` has type `int` and that (therefore) the methods `get` and `set` must have types `int` and `int -> unit`, respectively:

[11] The `node` field is part of the record type `hash_consed`; see `hashcons.mli`.

```
class int_cell = object
  val mutable x = 0
  method get = x
  method incr y = x <- x + y
end
```

It is perhaps lesser known that **OCaml can also infer the type of a virtual method**, based on the manner in which this method is used. In the following variant of the previous example, it infers that the method `check` must have type `int -> int`, as it receives an integer argument and produces a result that is stored in the field `x`.

```
class virtual int_cell = object (self)
  val mutable x = 0
  method get = x
  method incr y = x <- self#check (x + y)
  method virtual check: _
end
```

The type annotation `_` that appears in the declaration of the method `check` stands for an unconstrained type variable. It lets the OCaml typechecker infer a most general monomorphic type for this method. A polymorphic type cannot be inferred. If we wished for the method `check` to have type `'a . 'a -> 'a`, then we would have to explicitly annotate the declaration with this polymorphic type.

4.2 Virtues of self-parameterized classes

Popular belief holds that inferring a method's type fails if "some type variables are unbound in this type". For instance, in the following variant of the previous example, OCaml infers that the method `check` has type `'a -> int`, where the type variable `'a` is unconstrained:

```
class virtual int_cell = object (self)
  val mutable x = 0
  method get = x
  method set y = x <- self#check y
  method virtual check: _
end
```

At that point, it fails with a type error message:

```
Error: Some type variables are unbound in this type:
  class virtual int_cell :
    object
      val mutable x : int
      method virtual check : 'a -> int
      method get : int
      method set : 'a -> unit
    end
The method check has type 'a -> int where 'a is unbound
```

In this case, the OCaml manual says, "the class should be parametric", and "the type parameters must be [subject] somewhere in the class body to a type constraint". In the above example, one might choose to parameterize the class over a type variable `'a` and to add a type annotation that requires `'a` to be the domain of the virtual method `check`:

```
class virtual ['a] int_cell = object (self)
  val mutable x = 0
  method get = x
  method set y = x <- self#check y
  method virtual check: 'a -> _
end
```

This eliminates the type error: this code is well-typed. One problem with this approach, though, is that further changes to the code might require introducing further type parameters. Suppose for instance that, instead of initializing the field `x` with the value `0`, we wish to parameterize the class over an initial value `init`. We modify the code as follows:

```
class virtual ['a] cell (init) = object (self)
  val mutable x = init
  method get = x
  method set y = x <- self#check y
  method virtual check: 'a -> _
end
```

Unfortunately, this modification makes the code ill-typed again:

```
Error: Some type variables are unbound in this type:
  class virtual ['a] cell :
    'b ->
    object
      val mutable x : 'b
      method virtual check : 'a -> 'b
      method get : 'b
      method set : 'a -> unit
    end
The method check has type 'a -> 'b where 'b is unbound
```

A natural reaction to this type error message might be to parameterize the class over both `'a` and `'b`, as follows:

```
class virtual ['a, 'b] cell (init) = object (self)
  val mutable x = init
  method get = x
  method set y = x <- self#check y
  method virtual check: 'a -> 'b
end
```

This eliminates the type error: this code is well-typed. However, it seems unfortunate that one cannot depend on the typechecker to infer the types of all methods. Instead, it seems that one must introduce an unpredictable number of type parameters, as well as an unpredictable amount of explicit type annotations, for the code to be accepted.

Fortunately, there is a solution to this problem.

Let us first note that, in the above example, even though both the argument type and result type of the method `check` are undetermined, **it is not necessary to introduce two type parameters** `'a` and `'b`. Indeed, one type parameter suffices, provided this parameter determines both the argument type and result type of `check`. To illustrate this, let us parameterize the class over a type variable `'check`, and provide a type annotation that equates `'check` with the type of the method `check`:

```
class virtual ['check] cell (init) = object (self)
  val mutable x = init
  method get = x
  method set y = x <- self#check y
  method virtual check: 'check
end
```

This code is well-typed, too. Its inferred type is as follows:

```
class virtual ['check] cell :
  'b ->
  object
    constraint 'check = 'a -> 'b  (* a type equation *)
    val mutable x : 'b
    method virtual check : 'check
    method get : 'b
    method set : 'a -> unit
  end
```

Because the typechecker infers and records the type equation `'check = 'a -> 'b`, instantiating the type variable `'check` with a concrete type suffices to determine the values of both `'a` and `'b`. For instance, if `'check` is instantiated with `float -> int`, then `'a` must be `float` and `'b` must be `int`. For this reason, there is no need to parameterize the class over `'a` and `'b`. Parameterizing it over `'check` suffices.

This remark alone does not quite solve our problem yet. It still seems as if, for a class definition to be accepted, one must introduce an unpredictable number of type parameters, as well as an unpredictable amount of explicit type annotations and/or type equations.

Fortunately, there is a general way out of this problem. In fact, **one type parameter always suffices**. The trick is to constrain this type parameter to be the type of "self". Indeed, in OCaml, the type of "self" is an OCaml object type that lists the names and types of all (public) methods. Therefore, fixing the type of "self" is enough to determine the type of every method.

We modify the example as follows. We parameterize the class over a single type variable, named `'self`, which we constrain to be the type of "self", via the type annotation `self : 'self`.

```
class virtual ['self] cell (init) = object (self : 'self)
  val mutable x = init
  method get = x
  method set y = x <- self#check y
  method virtual check: _
end
```

Even though, this time, we have not given the type of the method `check`, this code is well-typed. **In a self-parameterized OCaml class, a monomorphic type can be inferred for every method**, be it concrete or virtual. In other words, in a self-parameterized class, the OCaml typechecker never complains that "some type variables are unbound".

In the `visitors` package, this remark solves several problems. First, we never need to wonder how many type parameters a class should have, and what they should be: the answer is always one, namely `'self`. Second, we never need to annotate a method with its type: every virtual method can be annotated with the wildcard `_`. These properties are of utmost importance, as we cannot in general predict the types of the generated methods.

Since self-parameterized classes seem so great, one may wonder whether, in everyday life, it would be a good idea to parameterize every class over `'self` in this manner. The answer is, it probably would

not. Such an approach leads to verbose class types, as the type of "self" contains a list of all (public) methods. It also gives rise to recursive object types, of the form `'self c as 'self`, where `c` is a class.

4.3 Simplifying the type of a self-parameterized class

We have used in §2.4 a few rules that allow an inferred class type to be manually simplified. In going from Figure 3 to Figure 4, we have used the well-known property that private methods can be omitted in a class type. We have also exploited the perhaps lesser-known fact that, in a self-parameterized class, the constraint that bears on the type parameter `'self` can be omitted, as it is implicit in OCaml that the type of "self" must be an object type that lists the public methods.

There remains to explain the surprising use of the "`'monomorphic.`" prefix in Figure 4. On the face of it, this is a universal quantification over a type variable, named `'monomorphic`, which does **not** appear in the type of the method. From a purely logical standpoint, such a universal quantification should be superfluous. Yet, here, it is required, due to the following peculiarity of OCaml [6]:

> In a class type, if the type of a method exhibits a free variable `'a`, if this method type has no explicit universal quantifiers, and if the type variable `'a` does not appear in an explicit type constraint, then, by convention, OCaml considers that the method type is universally quantified in `'a`.

We must work around this syntactic convention: in Figure 3, for instance, the method `visit_expr` has monomorphic type `'env -> expr -> unit`, where the type variable `'env` is connected with `'self` via an implicit constraint and (like `'self`) is quantified at the level of the class. This method does **not** have polymorphic type `'env. 'env -> expr -> unit`. The logically redundant quantification over `'monomorphic` serves as a syntactic mark that we really intend the type variable `'env` to appear free in the method type.

4.4 Monomorphic methods, polymorphic classes

Even if a method is monomorphic, the class that contains this method can be polymorphic. In the following example, the `identity` method is annotated with a monomorphic type, which implies that, if `o` is an object of class `c`, then the method `o#identity` cannot be applied both to apples and to oranges. However, the class `c` is polymorphic, which means that two distinct instances of this class can be used (separately) with apples and with oranges.

```
class ['self] c = object (_ : 'self)
  method identity (x : 'a) : 'a = x
end

let b : bool =
  new c # identity true
let i : int =
  new c # identity 0
```

This (well-known) property of classes is exploited in the `visitors` package. Although (in monomorphic mode, §2.12.1) every generated visitor method is monomorphic, the visitor classes are polymorphic, so (distinct) visitor objects can be used at many different types.

4.5 Where the expressiveness of OCaml's type system falls short

We have noted earlier (§2.9) that, among the varieties of visitors that we have presented, `fold` visitors are in principle the most general. This raises the question: is it possible to define `map` and `reduce` as subclasses of `fold`, equipped with appropriate `build_` methods?

```
(* Direct definitions of [map], [reduce], and [fold]. *)
class virtual ['self] reduce = object (self: 'self)
  method private visit_option: 'a .
    ('env -> 'a -> 'z) -> 'env -> 'a option -> 'z
  = fun f env ox ->
      match ox with None -> self#zero | Some x -> f env x
  method private virtual zero: 'z
end
class ['self] map = object (_ : 'self)
  method private visit_option: 'a 'b .
    ('env -> 'a -> 'b) -> 'env -> 'a option -> 'b option
  = fun f env ox ->
      match ox with None -> None | Some x -> Some (f env x)
end
class virtual ['self] fold = object (self : 'self)
  method private visit_option: 'a .
    ('env -> 'a -> 'r) -> 'env -> 'a option -> 's
  = fun f env ox ->
      match ox with
      | None   -> self#build_None env
      | Some x -> self#build_Some env (f env x)
  method private virtual build_None: 'env -> 's
  method private virtual build_Some: 'env -> 'r -> 's
end
(* A successful definition of [reduce] in terms of [fold]. *)
class virtual ['self] reduce_from_fold = object (self : 'self)
  inherit [_] fold
  method private build_None _env   = self#zero
  method private build_Some _env z = z
  method private virtual zero: 'z
end
(* An unsatisfactory definition of [map] in terms of [fold]. *)
class ['self] map_from_fold = object (_ : 'self)
  inherit [_] fold
  method private build_None _env   = None
  method private build_Some _env x = Some x
end
```

Figure 29: An unsatisfactory definition of `map` as a subclass of `fold`

```
class ['self] map_from_fold : object ('self)
  method private visit_option : 'a .
    ('env -> 'a -> 'b) -> 'env -> 'a option -> 'b option
  method private build_None : 'env -> 'b option
  method private build_Some : 'env -> 'b -> 'b option
end
```

Figure 30: The type of the class `map_from_fold` (Figure 29)

Doing so would be more economical: that is, it would significantly reduce the redundancy between the classes `map`, `reduce`, and `fold`. When these classes are automatically generated, code duplication is not so much of a problem. However, there are situations where (parts of) visitor classes must be hand-written, and where duplication becomes painful.

Furthermore, defining `map` and `reduce` as subclasses of `fold` would give rise to new patterns of customization. It would become possible to define a subclass of `map` or `reduce` and override just one `build_` method so as to obtain custom behavior.[12]

In an untyped setting, the question can be answered positively: `map` and `reduce` can be defined in terms of `fold`. Unfortunately, in the setting of OCaml's type system, the answer is negative: although `reduce` can be defined in terms of `fold`, `map` cannot.

The situation is illustrated in Figure 29. As an example, we define visitor methods, by hand, for the type `'a option`.

The methods `visit_option` in the classes `reduce` and `map` are identical to those found in the classes `VisitorsRuntime.reduce` and `VisitorsRuntime.map`. We note that the method `visit_option` in the class `reduce` is polymorphic in `'a`, whereas `visit_option` in the class `map` is polymorphic in `'a` and `'b`.

There is some similarity between these methods, which is why we define a class `fold`, whose method `visit_option` contains calls to the virtual methods `build_None` and `build_Some`. We assign to this method the most general type that is expressible in OCaml's type system. It is polymorphic in `'a`. The type variable `'r`, which represents the result of the function `f`, and the type variable `'s`, which represents the result of the methods `visit_option`, `build_None`, and `build_Some`, cannot be universally quantified at the level of a method, since they appear in the types of several methods. They must be (implicitly) quantified at the level of the class.

Is this definition of `fold` satisfactory? To test this, let us now attempt to propose new definitions of `reduce` and `map` as subclasses of `fold`. We define two more classes, `reduce_from_fold` and `map_from_fold`, which both inherit `fold` and provide suitable definitions of the methods `build_None` and `build_Some`.

In `reduce_from_fold`, everything works fine. The type parameters `'r` and `'s` in `fold` are both instantiated with `'z`. As a result, the method `visit_option` in the class `reduce_from_fold` has type `'a. ('c -> 'a -> 'z) -> 'c -> 'a option -> 'z`, just as in the class `reduce`.

In `map_from_fold`, a problem arises. The code is well-typed, but its type is less general than desired. The OCaml typechecker infers that the type parameters `'r` and `'s` in `fold` must be respectively instantiated with `'b` and `'b option`, where the type variable `'b` must be quantified at the level of the class. Therefore, the type of `map_from_fold` is as shown in Figure 30. This type is **not** polymorphic in `'b`, thus strictly less general than the type of the method `visit_option` in the class `map` (Figure 29).

What would it take to repair this problem? Apparently, the type of the method `visit_option` in the class `fold` is not general enough. Whereas the type variables `'r` and `'s` currently have kind `*`, it seems that they should have kind `* -> *`. The type of the class `fold` should be as follows:

```
class ['self] fold : object ('self)
  method private visit_option: 'a 'b .
    ('env -> 'a -> 'r['b]) -> 'env -> 'a option -> 's['b]
  method private virtual build_None: 'b . 'env -> 's['b]
  method private virtual build_Some: 'b . 'env -> 'r['b] -> 's['b]
end
```

This is not valid OCaml: we use our own fictitious notation. We write `'r[b]` for the type-level application of `'r` to `'b`. The type of each method is universally quantified in `'b`, as desired. The type of `visit_option` seems to have the desired generality. By instantiating both `'r` and `'s` with the type-level function `fun 'b -> 'z`, we obtain the type of `visit_option` in the class `reduce`. By instantiating

[12] At present, this must be done by overriding a `visit_` method, thereby causing more code duplication than necessary.

`'r` and `'s` with the type-level functions `fun 'b -> 'b` and `fun 'b -> 'b option`, respectively, we obtain the type of `visit_option` in the class `map`.

This suggests that higher kinds, type-level functions, and type-level β-reduction might be valuable features, not just in functional programming languages, but also in an object-oriented programming setting.

`ancestors`	(list of strings)	A list of classes that the generated class should inherit. This is an optional parameter; its default value is the empty list. Unless `nude` is `true`, the class `VisitorsRuntime.<variety>` is implicitly prepended to this list. Every ancestor class must have exactly **one** type parameter, which is typically (but not necessarily) the type of "self".
`build_prefix`	(string)	The prefix that is used in the name of the build methods in `fold` and `fold2` visitors (§2.9). This is an optional parameter, whose default value is "`build_`".
`concrete`	(Boolean)	If this parameter is `true` then the generated class is declared concrete; otherwise, it is declared virtual. This is an optional parameter; its default value is `false`.
`data`	(Boolean)	If this parameter is `true` then one visitor method is generated for every data constructor (§5.3). Otherwise, this method is not generated (it is inlined instead). This is an optional parameter; its default value is `true`.
`fail_prefix`	(string)	The prefix that is used in the name of the failure methods in visitors of arity two (§2.10). This is an optional parameter, whose default value is "`fail_`".
`irregular`	(Boolean)	If this parameter is `true` then the regularity check (§5.4) is disabled; otherwise, it is enabled. This is an optional parameter; its default value is `false`.
`name`	(string)	The name of the generated class. This is an optional parameter; its default value is `<variety>`.
`nude`	(Boolean)	If this parameter is `true` then the class `VisitorsRuntime.<variety>` is **not** implicitly prepended to the list `ancestors`. This is an optional parameter; its default value is `false`.
`polymorphic`	(Boolean)	If this parameter is `true` then type variables are handled by virtual visitor methods, and generated methods are monomorphic (§2.12.1); otherwise, type variables are handled by visitor functions, and generated methods are polymorphic (§2.12.2). This is an optional parameter, whose default value is `false`.
`public`	(list of strings)	This is an optional parameter. If absent, then every method in the generated class is declared public. If present, then every method in the generated class is declared private, except those whose name appears in the list: those are declared public.
`variety`	(string)	The variety of visitor that should be generated. The supported varieties are `iter` (§2.2), `map` (§2.5) and `endo` (§2.6), `reduce` (§2.7), `mapreduce` (§2.8), `fold` (§2.9), `iter2`, `map2`, `reduce2`, `mapreduce2`, `fold2` (§2.10).
`visit_prefix`	(string)	The prefix that is used in the name of visitor methods. This is an optional parameter, whose default value is "`visit_`". If this prefix is changed then the classes provided by the library `VisitorsRuntime` become useless: in that case, one might wish to also specify `nude = true`, so as to not inherit these classes.

Figure 31: Parameters of `[@@deriving visitors { ... }]`

5 Reference

5.1 Parameters

The parameters that can be passed as part of the `[@@deriving visitors { ... }]` annotation, inside the curly braces, are described in Figure 31.

5.2 How to examine the generated code

The generated code is conceptually inserted into the user's source code just after the type definition that is decorated with `[@@deriving visitors { ... }]`.

It can be useful to inspect the generated code, so as to understand how it works, what are the arguments and result of each method, and so on. This can be especially useful when the generated code is ill-typed (which can happen, for instance, when arbitrary user code is inherited via the `ancestors` parameter).

The file `Makefile.preprocess` offers a recipe that builds a file named `%.processed.ml` out of the source file `%.ml`. This file contains just the generated code. The recipe relies on `sed`, `perl`, and `ocp-indent` to extract and beautify the code. This file is installed with the `visitors` package. If the environment variable `OPAM_SWITCH_PREFIX` is defined, the full path of this file should be `$OPAM_SWITCH_PREFIX/lib/visitors/Makefile.preprocess`. If `ocamlfind` is installed, this path can also be written under the form `$(ocamlfind query visitors)/Makefile.preprocess`.

5.3 Structure of the generated code

The `[@@deriving visitors { ... }]` annotation applies to a type definition. A type definition may define several types, and these types may be parameterized. A local type is one that is defined as part of this type definition, whereas a nonlocal type is one that preexists (§2.13).

The generated code consists of **a single class**, whose name is `<variety>`, that is, the value of the `variety` parameter (§5.1). This class has **one type parameter**, namely `'self`, the type of "self" (§4.2). It has no fields. It **inherits** from the class `VisitorsRuntime.<variety>` (unless the parameter `nude` is `true`, §5.1) and from the classes listed via the `ancestors` parameter (§5.1), in that order. To find out which methods exist in the class `VisitorsRuntime.<variety>`, please consult `VisitorsRuntime.ml`.

In the following, the index `i` ranges from 0 (included) to `<arity>` (excluded), where `<arity>` is the arity of the generated visitor (thus, either 1 or 2).

The following **concrete methods** are **defined**:

- for every local type `??? foo`, a visitor method.

method name:	`visit_foo`	
arguments:	`visit_'a`, ...	a visitor function for each type param. of `foo` (only if `polymorphic = true`)
	`env`	an environment of type `'env`
	`this_0`, `this_1`, ...	for each `i`, a value of type `??? foo`
invoked:	on the way down	
example:	Figure 1	

- for every data constructor `Foo` of a local sum type `??? foo`, a visitor method.

method name:	`visit_Foo`	
arguments:	`visit_'a`, ...	a visitor function for each type param. of `foo` (only if `polymorphic = true`)
	`env`	an environment of type `'env`
	`this`	a data structure of type `??? foo` (only in an `endo` visitor)
	`c0_0`, `c0_1`, ...	for each `i`, the first component of a `Foo` value
	`c1_0`, `c1_1`, ...	for each `i`, the next component of a `Foo` value
	...	... and so on
invoked:	on the way down	
example:	Figure 1	

If the parameter `data` is `false`, then this method is **not** generated (§5.1). It is inlined instead. The behavior is the same, but cannot be overridden on a per-data-constructor basis.

- if the visitor has arity two (§2.10), for every local type `foo`, a failure method.

method name:	`fail_foo`	
arguments:	`env`	an environment of type `'env`
	`this_0`, `this_1`, ...	for each `i`, a value of type `??? foo`
invoked:	when two distinct data constructors `Foo` and `Bar` are found	
example:	Figure 13	

The following **virtual methods** are **declared**:

- for every type parameter `'foo` of a local type, a visitor method. (Only if `polymorphic = true`.)

method name:	`visit_'foo`	
arguments:	`env`	an environment of type `'env`
	`this_0`, `this_1`, ...	for each `i`, a value of type `'foo`
invoked:	on the way down	
example:	Figure 17	

- if this is a `reduce` visitor (§2.7) or a `mapreduce` visitor (§2.8), the monoid methods.

method name:	`zero`
arguments:	none
result:	a summary
example:	Figure 7

method name:	`plus`
arguments:	two summaries
result:	a summary
example:	Figure 7

- if this is a `fold` visitor (§2.9), for every local record type `foo`, a build method.

method name:	`build_foo`	
arguments:	`env`	an environment of type `'env`
	`r_0`	the result of the first recursive call
	`r_1`	the result of the next recursive call
		... and so on
invoked:	on the way up	
example:	none in this document	

- if this is a `fold` visitor (§2.9), for every data constructor `Foo` of a local sum type, a build method.

method name:	`build_Foo`	
arguments:	`env`	an environment of type `'env`
	`r_0`	the result of the first recursive call
	`r_1`	the result of the next recursive call
		... and so on
invoked:	on the way up	
example:	Figure 11	

The following methods are **called**, therefore are expected to exist. These methods are neither defined nor declared: their definition or declaration must be inherited from a parent class. These methods can have a polymorphic type.

- for every nonlocal type `??? foo`, a visitor method.

method name:	`visit_foo`	
arguments:	`visit_0`, ...	for each actual type parameter in `??? foo`, a visitor function for this type
	`env`	an environment of type `'env`
	`this_0`, `this_1`, ...	for each `i`, a value of type `??? foo`
invoked:	on the way down	
example:	Figure 21	

All of the above methods are parameterized with an environment `env`, which is propagated in a top-down manner, that is, into the recursive calls. The environment is not returned out of the recursive calls, therefore not propagated bottom-up or left-to-right. The type of this environment is undetermined: it is a type variable. There is no a priori constraint that the type of the environment should be the same in every method: it is possible for unrelated visitor methods to expect environments of unrelated types.

The result types of the visitor methods, build methods, and failure methods depend on the parameter `variety` (§5.1). In an `iter` visitor, every method has result type `unit`. In a `map` or `endo` visitor, the visitor method associated with the type `foo` has result type `??? foo`.[13] In a `reduce` visitor, every method has result type `'s`, if `'s` is the monoid, that is, if the methods `zero` and `plus` respectively have type `'s` and `'s -> 's -> 's`. In a `mapreduce` visitor, the visitor method associated with the type `foo` has result type `??? foo * 's`, if `'s` is the monoid. In a `fold` visitor, it is up to the user to decide what the result types of the visitor methods should be (subject to certain consistency constraints, naturally). In particular, two visitor methods `visit_foo` and `visit_bar` can have distinct result types; this is illustrated in Figure 12.

5.4 Supported forms of types

The following forms of type definitions are supported:

- Definitions of **type abbreviations** (also known as type synonyms).
- Definitions of **record types**.
  Mutable fields are supported.
- Definitions of **sum types** (also known as variant types and as algebraic data types).
  Data constructors whose arguments form an "inline record" are supported.

Definitions of abstract types and of extensible sum types are not supported.

Definitions of **parameterized types** are supported. In monomorphic mode (§2.12.1), only **regular** parameterized types are permitted, whereas in polymorphic mode (§2.12.2), arbitrary parameterized types are permitted. A parameterized type is regular if, within its own definition, it is applied only to its formal parameters. For instance, the well-known definition of lists is regular:

[13] The question marks `???` stand for the type parameters of `foo`, which can be a parameterized type. In a `map` visitor, the type parameters that appear in the method's argument type and in the method's result type can differ. In an `endo` visitor, they must be the same.

```
type 'a list =
| []
| (::) of 'a * 'a list
```

whereas the following definition of a random access list [4, §10.1.2] is not:

```
type 'a seq =
| Nil
| Zero of      ('a * 'a) seq
| One of 'a * ('a * 'a) seq
```

Irregular data types are also known as "nonuniform" [4, §10.1] or "nested" data types [1].
Existential types and generalized algebraic data types (GADTs) are currently not supported.
In the right-hand side of a type definition, the following forms of types are supported:

- Type constructors, possibly applied to a number of types, such as `foo` and `('a * 'b) bar`.
- Type variables, such as `'foo`.
- Tuple types, such as `int * expr`.

The unsupported forms of types include anonymous type variables (`_`), function types (`int -> unit`), object types (`<get: int>` and `#point`), recursive types (`int -> 'a as 'a`), polymorphic variant types (`[ 'A| 'B ]`), universal types (`'a. 'a -> 'a`), and packaged module types (`(module S)`). If these forms appear in a type definition, they must be marked `@opaque` (§5.5).

In theory, at each arity, the tuple type constructor could be viewed as a parameterized nonlocal type constructor. At arity 2, for instance, the pair type `'a * 'b` could be treated as a nonlocal type `('a, 'b) tuple2`. Then, to traverse a value of this type, one would invoke a method `visit_tuple2`, which necessarily would be inherited from a parent class. That would be somewhat inconvenient, as these (polymorphic) methods would have to be manually written, at each arity. Instead, special treatment for tuple types is built-in. There are no visitor methods or build methods for tuples; instead, ad hoc code is generated. This means that the behavior of a visitor at a tuple type is fixed: it cannot be overridden in a subclass. The behavior of a `fold` visitor at a tuple type is to rebuild a tuple, just like a `map` visitor would do.

5.5 Opaque components

One sometimes wishes for a component of a data structure **not** to be visited, either for efficiency reasons, or because this component is of an unsupported type. This can be requested by annotating the type of this component with the attribute `[@opaque]`. This is done, for instance, if Figure 7, where the integer argument of the data constructor `EConst` is marked opaque. (Note the parentheses, which are required.) The effect of this annotation is that this component is not visited: in the method `visit_EConst`, instead of a call to `self#visit_int`, we find a call to `self#zero`, as this is a `reduce` visitor.

Generating a visitor of arity two for a data structure with `@opaque` components requires some care. The methods `visit_int` defined in the classes `VisitorsRuntime.iter2`, `VisitorsRuntime.map2`, and so on, raise a `StructuralMismatch` exception when their two integer arguments differ. If `int` is replaced with `(int[@opaque])`, then these methods are not invoked, so no exception is raised. It is up to the user to decide which behavior is desired. Furthermore, it should be noted that `map2` and `fold2` visitors follow an arbitrary convention at `@opaque` components: they return the first of their two arguments. Again, it is up to the user to decide whether this behavior is appropriate.

In `polymorphic` mode (§2.12.2), a type variable must not occur under an `[@opaque]` annotation.

If desired, instead of `[@opaque]`, one can use the more verbose forms `[@visitors.opaque]` and `[@deriving.visitors.opaque]`.

5.6 Alternate method names

The methods associated with the type `foo` are normally named after this type: this includes the visitor method `visit_foo`, the build method `build_foo`, and the failure method `fail_foo`. This base name can be altered via an attribute. If `foo` is a local type, then a `[@@name]` attribute must be attached to the declaration of this type. If `foo` is a nonlocal type, then a `[@name]` attribute must be attached to every reference to this type. For instance, in the following example, the visitor method associated with the type `cloud` is named `visit_nuage` instead of `visit_cloud`:

```
type cloud =
  | Point of (float[@name "real"]) * (float[@name "real"])
  | Clouds of cloud list
  [@@name "nuage"]
  [@@deriving visitors { variety = "map"; ancestors = ["base"] }]
```

Furthermore, the visitor method associated with the type `float` (which is not generated, and should be inherited from the class `base`) is assumed to be named `visit_real` instead of `visit_float`.

The methods associated with the data constructor `Foo` are normally named after it: this includes the visitor method `visit_Foo` and the build method `build_Foo`. This base name can be altered via a `[@name]` attribute, which must be attached to the data constructor. For instance, in the following example, the visitor methods associated with the data constructors `[]` and `::` are named `visit_nil` and `visit_cons`:

```
type 'a mylist = 'a list =
  | []                      [@name "nil"]
  | (::) of 'a * 'a mylist [@name "cons"]
  [@@deriving visitors { variety = "map" }]
```

If desired, instead of `[@name]`, one can use `[@visitors.name]` or `[@deriving.visitors.name]`.

5.7 Alternate construction code in `map` visitors

By default, a `map` visitor for an algebraic data type constructs a new data structure, by applying a data constructor or by allocating a new record. Sometimes, though, it is illegal to do so: this is the case, for instance, if the type has been declared **`private`**. In such a situation, an alternate constructor can be provided via a `[@build]` attribute (which must be attached to a data constructor) or via a `[@@build]` attribute (which must be attached to a record type declaration). Such an attribute carries an OCaml expression, which should represent a constructor function.

For instance, suppose that the module `Point` has the following signature:

```
type point = private { x: int; y: int }
val make: int -> int -> point
```

Suppose that, outside of this module, one wishes to generate a `map` visitor for the type `Point.point`. One would like to do this in the style that was presented earlier (§2.14), but a naïve attempt fails: the generated visitor is illegal, because allocating a record of type `Point.point` is forbidden. To circumvent this problem, one should indicate, via a `[@@build]` attribute, that the constructor function `Point.make` should be used:

```
type point = Point.point = private { x: int; y: int }
  [@@build Point.make]
  [@@deriving visitors { variety = "map" }]
```

The OCaml expression carried by a `[@build]` attribute can refer to the local variable `self`, which represents the visitor object, and to the local variable `env`, which represents the environment received by the visitor method.

`[@build]` attributes influence not only `map` visitors, but also `endo` and `mapreduce` visitors.

Instead of `[@build]`, one can use `[@visitors.build]` or `[@deriving.visitors.build]`.

References

[1] Richard Bird and Lambert Meertens. Nested datatypes. In *Mathematics of Program Construction (MPC)*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998.

[2] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *ACM Workshop on ML*, pages 12–19, 2006.

[3] Bartosz Milewski. Understanding F-algebras, October 2013.

[4] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[5] François Pottier. From visitors to iterators, March 2017.

[6] François Pottier. Ocaml issue 7465, January 2017.