Visitors Unchained

FRANÇOIS POTTIER, Inria Paris, France

Traversing and transforming abstract syntax trees that involve name binding is notoriously difficult to do in a correct, concise, modular, customizable manner. We address this problem in the setting of OCaml, a functional programming language equipped with powerful object-oriented features. We use visitor classes as partial, composable descriptions of the operations that we wish to perform on abstract syntax trees. We introduce `visitors`, a simple type-directed facility for generating visitor classes that have no knowledge of binding. Separately, we present `alphaLib`, a library of small hand-written visitor classes, each of which knows about a specific binding construct, a specific representation of names, and/or a specific operation on abstract syntax trees. By combining these components, a wide range of operations can be defined. Multiple representations of names can be supported, as well as conversions between representations. Binding structure can be described either in a programmatic style, by writing visitor methods, or in a declarative style, via preprogrammed binding combinators.

CCS Concepts: • **Software and its engineering → Object oriented languages**; **Functional languages**;

Additional Key Words and Phrases: abstract syntax trees, names, binding constructs, traversals, boilerplate

ACM Reference Format:
François Pottier. 2017. Visitors Unchained. *Proc. ACM Program. Lang.* 1, ICFP, Article 28 (September 2017), 28 pages.
https://doi.org/10.1145/3110272

1 INTRODUCTION

Weirich *et al.* [2011] suggest that "name binding is one of the most annoying parts of language implementations", and put forth two reasons why this is so. First, it is boring, "requiring boilerplate that must be maintained as the implemented language evolves". Second, it is difficult to get right, as it usually involves "subtle invariants that pervade the system". Cheney [2005] coins the term "nameplate" for this "particularly intractable kind of boilerplate [that has] to do with names, name-binding, and fresh name generation".

Getting rid of nameplate is nontrivial because nameplate is typically a mixture of several different kinds of boilerplate code, which are concerned with different aspects of the problem. It can be difficult to recognize how to best disentangle these aspects, modularly decompose the boilerplate into several parts, and deal with each part using the tools and architectural principles that are most suitable for this part.

Part of the nameplate is just boilerplate. This part is concerned with the traversal and transformation of ordinary algebraic data structures, and has nothing to do with names or binding. It is conceptually simple. Yet, because its structure is dictated by a user-defined data type, it cannot just be written once and placed in a traditional library. Instead, some form of datatype-generic programming or type-directed code generation is required. This kind of boilerplate is well-identified:

2475-1421/2017/9-ART28
https://doi.org/10.1145/3110272

in the Haskell world, several ways of eliminating it have been proposed. For instance, Cheney's work [2005] uses "Scrap your Boilerplate" [Lämmel and Peyton Jones 2005], while Weirich *et al.* [2011] rely on RepLib [Weirich 2006]. One contribution of this paper is to deal with this boilerplate, in the setting of OCaml, by exploiting automatically-generated visitor classes. We view visitor classes as incomplete, composable, open-ended descriptions of operations on user-defined data structures. Our visitors come in a limited number of "varieties", such as `iter`, `map`, and `iter2`: fortunately, we find that these three varieties are sufficient for our needs.

The rest of the nameplate is concerned with names, binding constructs, of which there is a wide range in the literature, and with operations on abstract syntax trees, of which there is also a wide range: collecting free names, converting between surface and internal representations of names, testing for α-equivalence, performing substitution, and many more come to mind. We argue that, there, too, it is possible to distinguish and separate several concerns.

One concern is the treatment of binding constructs. It has been argued in the literature that, instead of baking in support for a limited set of binding constructs, one could offer a domain-specific language, a set of binding combinators, which allow a declarative description of the binding structure. Cαml [Pottier 2006] and Unbound [Weirich et al. 2011] offer examples of such domain-specific languages. A second contribution of this paper is to propose that the meaning and implementation of these binding combinators can be made independent of the representation of names. Indeed, what is the meaning of a binding construct? It is summed up by the manner in which an environment is extended with suitable names at suitable points. Thus, to define the meaning of a binding construct or binding combinator, it suffices to write traversal code for it, which can be independent of the representation of names and environments, as long as it has access to a single operation, namely `extend`, the operation of extending an environment with a (bound) name. In this paper, we implement this traversal code, too, in the form of visitor classes, which are hand-written.

The remaining concern, then, is to deal with concrete representations of bound names, free names, and environments, and with concrete operations on these entities. Examples of concrete operations are "collecting the free names of a term in nominal representation" and "substituting a term for a variable in a term in de Bruijn's representation" and "converting a term from nominal representation to de Bruijn's representation". As the issue of traversal (of sums, products, and binding constructs) has been set aside, defining one such concrete operation is a matter of: (1) defining how an environment is extended with a bound name (that is, providing `extend`) and (2) defining how a free name must be handled. This is done by defining a "kit", a visitor class with two methods.

Our approach to defining concrete operations on terms is modular, as it separates the three components described above, namely: an auto-generated visitor for ordinary data (sums and products), visitors for binding constructs or combinators, and "kits", that is, visitors that know how to extend and look up a concrete environment so as to perform a concrete operation. The last two components can be programmed once and for all and placed in a library. We present one such library, `alphaLib` [Pottier 2017a], which we are developing. This library, currently at an early stage of development, allows easily constructing a toolbox of operations on terms in the "nominal" representation, where both bound names and free names are represented as "atoms" with a unique integer identity [Cheney 2005; Shinwell et al. 2003]. It also offers conversions to and from the "raw" representation, where all names are represented as strings.

Our approach is not only modular, but also open-ended, that is, customizable. If special behavior is needed when traversing or constructing a user-defined data type, this can be specified by providing a suitable visitor method. As an example, the `visitors` documentation [Pottier 2017b] shows how to construct visitors for hash-consed data structures. The technique presented there can be combined with the ideas presented in this paper, so one can easily manipulate hash-consed abstract

syntax with binding, if so desired. Similarly, if an unanticipated binding construct or operation is needed, it can often be programmed by the end user, outside of `alphaLib`, by providing a few methods.

That said, there are limits to the expressiveness of our approach. One limitation is that not every operation can be viewed as an instance of a visitor. For instance, we have encountered difficulty in encoding a recursive binding combinator, in the style of Unbound's Rec, in terms of visitors. Another limitation is that not every binding construct can be described purely in terms of successive environment extensions. For instance, it seems that one cannot in such a way describe a binding construct that allows a name to have multiple binding occurrences.

In summary, the contributions of this paper are as follows:

(1) We introduce `visitors` [Pottier 2017b], a syntax extension for OCaml, which makes it easy to obtain automatically-generated visitor classes for user-defined data types (§2). This is a simple, general-purpose facility, without any knowledge of names and binding.
(2) We show that the meaning of a binding construct (§4) or binding combinator (§8) can be defined in the form of a hand-written visitor class and that this definition can be representation-independent.
(3) By combining these pieces, we achieve a declarative way for an end user to describe abstract syntax with (possibly complex, custom forms of) binding and to obtain visitor classes for this syntax (§3, §8).
(4) We show that abstract syntax, too, can be representation-independent, and that this allows conversions between representations to be defined as instances of a `map` visitor (§5).
(5) We present `alphaLib` [Pottier 2017a], an OCaml library that offers both a small set of binding combinators in the style of Unbound (§8) and a set of ready-made "kits" (§5) which make it easy to define a toolbox of common operations on terms (§6, §7).

2 VISITORS

2.1 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 carry out the grunt work of going down into the data structure and coming back up. Nevertheless, by defining a subclass where one method or a few methods are overridden, a wide range of nontrivial behaviors can be obtained. Therefore, visitors allow many operations on this data structure to be defined with little effort.

The Visitor pattern is a well-known software design pattern. Gamma *et al.* [1995] write that a visitor "represents an operation to be performed on the elements of a [data] structure" and that this pattern allows "defining a new operation without changing the classes of the elements on which it operates." In a functional programming language, naturally, one takes for granted that new operations can be defined without altering existing data type definitions. Nevertheless, visitor classes remain an interesting design pattern, for two reasons:

- They allow *constructing incomplete descriptions* of operations and *composing* them via multiple inheritance.
- Visitors for algebraic data types can be automatically generated. This eliminates hand-written boilerplate code, making visitors lightweight and attractive. Furthermore, this gives visitors a *declarative* flavor, as the structure and behavior of a visitor are directed by a type definition.

In this paper, we deal with abstract syntax that involves name binding. In our approach, visitors for binding constructs must be manually implemented, because that is the only way for these constructs to acquire their intended meaning. Fortunately, this is done once and for all by a library

writer. These binding constructs are then exploited by the end user as part of seemingly-ordinary data type definitions, for which visitors are automatically generated. The end user is thus presented with a declarative approach to abstract syntax with binding.

2.2 Automated Generation of Visitors

Visitors have extremely regular structure. As a result, whereas implementing a visitor by hand is boring and possibly error-prone, automatically generating one is often possible. We have written and released a syntax extension for OCaml, known as `visitors` [Pottier 2017b], which makes it easy for the programmer to request the automatic generation of visitor classes for a user-defined data type or group of data types.

At present, `visitors` supports (parameterized) algebraic data types (which may involve tuples, records with immutable and mutable fields, sums, and recursion) and (parameterized) type abbreviations. Function types are not supported, but (if desired) could be wrapped in a user-defined abstract type, for which a visitor could be manually implemented. Nested algebraic data types [Bird and Meertens 1998] and generalized algebraic data types [Xi et al. 2003] are currently not supported, but might be in the future (§10).

Technically, `visitors` is a plugin for `ppx_deriving`, which itself is a syntax extension for the OCaml compiler. Both `visitors` and `ppx_deriving` are implemented in OCaml. They have access to the compiler's internal representation of abstract syntax trees, and transform an abstract syntax tree into an abstract syntax tree. At the time of writing, `visitors` consists of 530 nonblank noncomment lines of OCaml code. It has no knowledge of or special support for name binding. It is a simple, general-purpose tool.

The `visitors` package allows generating several varieties of visitor classes, which differ in the bottom-up computation that they perform and (therefore) in their result types:

- An `iter` visitor 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 returns a data structure, typically a transformed version of the data structure that it has received as an argument.

There are other varieties of visitors, which are not needed in this paper. The visitors mentioned above have arity one: they traverse one data structure. The `visitors` package can also generate visitors of arity two, which simultaneously traverse two data structures of identical shape. This is used, for instance, when deciding whether two terms are α-equivalent. In this paper, `iter`, `map`, and `iter2` visitors are used.

2.3 Example

In this section, we present the main features of the `visitors` package via an example. More examples, which have to do with name binding, appear further on in the paper. For more information, the reader is referred to the documentation [Pottier 2017b], which contains many examples of visitors and their uses.

Figure 1 presents the definition of an algebraic data type of arithmetic expressions. These expressions are built out of integer constants and binary additions: $e ::= c \mid e + e$. Therefore, two data constructors are introduced, namely `EConst`, which carries an integer, and `EAdd`, which carries two subexpressions. Furthermore, for the sake of this example, every abstract syntax tree node is decorated with a piece of information of type `'info`. This requires distinguishing two data types, namely `'info expr` and `'info expr_desc`. An "expression" is a pair of a decoration of type `'info` and a descriptor. A "descriptor" is an application of either `EConst` or `EAdd`. The data types `'info expr` and `'info expr_desc` are parameterized over the type variable `'info`, so that the

```
type 'info expr_desc =
  | EConst of int
  | EAdd of 'info expr * 'info expr
and 'info expr =
  { info: 'info; desc: 'info expr_desc }
[@@deriving visitors { variety = "map" }]
```

```
class virtual ['self] map = object (self : 'self)
  inherit [_] VisitorsRuntime.map
  method virtual visit_'info : _
  method visit_EConst env _visitors_c0 =
    let _visitors_r0 = self#visit_int env _visitors_c0 in
    EConst _visitors_r0
  method visit_EAdd env _visitors_c0 _visitors_c1 =
    let _visitors_r0 = self#visit_expr env _visitors_c0 in
    let _visitors_r1 = self#visit_expr env _visitors_c1 in
    EAdd (_visitors_r0, _visitors_r1)
  method visit_expr_desc env _visitors_this =
    match _visitors_this with
    | EConst _visitors_c0 ->
        self#visit_EConst env _visitors_c0
    | EAdd (_visitors_c0, _visitors_c1) ->
        self#visit_EAdd env _visitors_c0 _visitors_c1
  method visit_expr env _visitors_this =
    let _visitors_r0 = self#visit_'info env _visitors_this.info in
    let _visitors_r1 = self#visit_expr_desc env _visitors_this.desc in
    { info = _visitors_r0; desc = _visitors_r1 }
end
```

Fig. 1. A visitor for a parameterized type of decorated expressions

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

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

Fig. 2. Working with different types of decorations

user may choose what type of information the tree nodes should be decorated with, and may work with different types of information at different times, if desired.

2.3.1 Generating a Visitor. The last line in the upper part of Figure 1 contains an OCaml attribute, `[@@deriving visitors { ... }]`, which is interpreted by the `visitors` syntax extension as a request to generate a visitor class. The parameter `variety = "map"` is interpreted as an indication that we wish to generate a `map` visitor. The generated class is implicitly inserted into the code immediately after the type definition. In normal use, this code remains invisible to the user. Here, it is shown in the lower part of Figure 1.

2.3.2 Structure of a Visitor. As is evident in the lower part of Figure 1, there is one visitor method per data type: here, these methods are `visit_expr_desc` and `visit_expr`. There is one visitor method per data constructor: here, these methods are `visit_EConst` and `visit_EAdd`.

In OCaml, the expression `o#foo` denotes a selection of the method `foo` of object `o`. Such an expression typically has a function type and can be applied to a suitable number of arguments. In a generated visitor class, the variable `self` is bound (on the first line of the generated code) to the visitor object that is being defined. Thus, `self#visit_expr env c` can be understood as a recursive method call with two arguments `env` and `c`.

There is one visitor method per type parameter: here, `visit_'info`. This method is in charge of processing and transforming a "piece of information". It is a virtual method, so the choice of an appropriate treatment is deferred to a subclass. This is illustrated further on (§2.3.6). Let us note that, instead of declaring a virtual method `visit_'info`, a different convention would be to parameterize every visitor method with a function `visit_'info`. Both conventions are sensible, and, in a typed setting, have incomparable expressiveness. In the future, we hope to let the user choose between these approaches, at a fine level of granularity (§10).

Every method takes an environment `env` as an argument, and transmits it down to its callees. The environment is never extended or looked up in a generated method, so it may seem useless. Yet, as becomes evident further on (§4), a hand-written visitor method can update or exploit the environment, so, by combining generated code and hand-written library code, one obtains visitors that use the environment in nontrivial ways. Therefore, the presence of an environment parameter is a key feature of visitors. It allows a "binding construct" to be defined as a construct whose traversal causes the environment to be extended.

As this is a `map` visitor, the methods `visit_EConst` and `visit_EAdd` respectively return `EConst` and `EAdd` descriptors. The method `visit_expr` returns an expression, that is, a two-field record. In a different variety of visitor, these three lines of code would be different. In an `iter` visitor, every method would return a unit value `()`.

2.3.3 Type Structure of a Visitor. The code in the lower part of Figure 1 looks almost as though it is untyped. Indeed, for maximum simplicity in the implementation of the `visitors` package, we generate as few type annotations as OCaml will allow. It turns out, somewhat to our surprise, that it is possible to set things up so that no type annotations at all are needed. By this, we mean that none of the methods, be it concrete or virtual, is annotated with its type.

For the purposes of examining and understanding the generated code, we believe that the absence of type annotations can be a good thing, as it removes a potentially considerable amount of clutter and helps understand the structure of the code by pure inspection. We have taken some advantage of this when presenting the structure of a visitor (§2.3.2). We have briefly discussed what each method does (so someone who wishes to call this method knows what behavior to expect) and what it is supposed to do (so someone who wishes to override this method knows what behavior

to respect), but have avoided the question: "what is the type of this method?". It is now time to address this question.

Let us first note that every generated visitor method receives a monomorphic type, as opposed to a polymorphic type scheme. This is imposed by the fact that, in the absence of any type annotations, the type of every method is inferred, and OCaml infers monomorphic method types only. Although the `visitors` package can optionally annotate the generated methods with polymorphic types, in this paper, we do not make use of this feature.

What is the type of the visitor method `visit_expr`? This method takes an environment and an expression and returns an expression. Its type must be monomorphic: it cannot involve any universal quantifiers. Therefore, its type must be of the form `_ -> _ expr -> _ expr`, that is, `'env -> 'info1 expr -> 'info2 expr`, where `'env`, `'info1`, and `'info2` are type variables. Similarly, the method `visit_expr_desc` must have type `'env -> 'info1 expr_desc -> 'info2 expr_desc`. Because these methods are mutually recursive, their types share the free type variables `'env`, `'info1`, and `'info2`.

Which constraints (if any) bear on these type variables? There is no constraint on the type variable `'env`. Indeed, because the environment is never used in the generated code, only carried around, its type remains undetermined. It may be (partially or completely) determined in a subclass, and must be fully determined by the time a visitor object is created. This is illustrated further on (§2.3.6).

There is no equality constraint between the type variables `'info1` and `'info2`. Indeed, in the code of the method `visit_expr`, the value `this.info` is not directly embedded into the expression that is returned. Instead, this value goes through a call to the method `visit_'info`. Thus, all we have is a constraint that `visit_'info` should have type `'env -> 'info1 -> 'info2`. Just like `'env`, the type variables `'info1` and `'info2` remain undetermined at this point.

Where are `'env`, `'info1`, and `'info2` bound? They are *not* universally quantified at the level of individual methods. Indeed, we have emphasized that every method has monomorphic type. Furthermore, we can see that they are shared between methods, that is, they appear free in the types of several methods. The answer, therefore, is that they are universally quantified at the level of the class. That is, the class, as a whole, is polymorphic in `'env`, `'info1`, and `'info2`. Thus, in two distinct subclasses, these type variables can be instantiated in distinct ways (§2.3.6).

2.3.4 OCaml Idiosyncrasies. Let us say a few words about a couple tricks that allow us to dispense with type annotations. These comments can safely be skipped by a reader who is not interested in the technicalities of OCaml's classes and objects.

First, although it is well-known that the type of a concrete method can be inferred, it is perhaps lesser-known that the type of a virtual method can be inferred, too. The declaration "`method virtual visit_'info: _`" in Figure 1 can be interpreted as a request for OCaml to infer a monomorphic type for `visit_'info`. Technically, each occurrence of a wildcard _ represents a unique anonymous type variable.

Second, although the type variables `'env`, `'info1`, and `'info2` appear free in the types of certain methods, we do *not* need to explicitly parameterize the class over them. Instead, the class `map` has exactly one type parameter, namely `'self`. The type annotation `(self : 'self)` means that `'self` stands for the type of the visitor object, `self`. This is in fact the most general way of parameterizing a class: *if a class is parameterized over* `'self`*, then it does not need any other type parameters.* In generated code, this property is extremely useful, as it relieves us from the burden of guessing how many type parameters are needed and where they should be mentioned in type annotations. We simply parameterize every class over `'self`, and let OCaml work its magic.

How is this possible? This may seem to contradict the popular wisdom according to which "a type variable that appears free in the type of some method must be [made] a type parameter of the class" [Leroy et al. 2016, §3.10, "Parameterized classes"] [Minsky et al. 2013, Chapter 12, "Classes"]. In reality, OCaml's type discipline [Rémy and Vouillon 1998] is more flexible than stated in the previous sentence. It is able to internally assign the class a *constrained* type scheme, and to do so, imposes the following relaxed rule: "a type variable that occurs free in the type of some method must be *related by an equality constraint* to a type parameter of the class". Fortunately, the type annotation `(self : 'self)` imposes an equality constraint between the type parameter `'self` and a structural object type where the type of every method is listed. Thus, provided this type annotation is given and `'self` is a type parameter of the class, the rule is automatically satisfied.

2.3.5 Inherited Behavior in a Visitor. By convention, every automatically generated `map` visitor inherits from the class `VisitorsRuntime.map`. This class has been hand-written, once and for all. It offers visitor methods for most of OCaml's primitive types, such as `int`, `bool`, and so on. This includes parameterized types, such as `'a array`, `'a list`, and `'a ref`.

The generated code in the lower part of Figure 1 relies on this feature. For instance, because the data constructor `EConst` carries an argument of type `int`, the generated method `visit_EConst` contains a call to the method `visit_int`. This method is neither defined nor declared in the generated class: it is inherited from the class `VisitorsRuntime.map`, where it is defined as a one-liner: "**`method`** `visit_int env i = i`".

To traverse containers, such as lists and arrays, the class `VisitorsRuntime.map` offers polymorphic visitor methods. For instance, the method `visit_list` has polymorphic type `'env 'a 'b . ('env -> 'a -> 'b) -> 'env -> 'a list -> 'b list`. By convention, when a value of type `foo bar` must be traversed, where the parameterized type `bar` preexists, the `visitors` package generates a call of the form `(self#visit_bar (self#visit_foo) env ...)`.[1] This allows lists of anything to be traversed. For example, if the data constructor `EAdd` carried a list of subexpressions, instead of two subexpressions, then `visit_EAdd` would contain a call of the form `(self#visit_list (self#visit_expr) env ...)`.

At this point, let us formulate a couple of remarks. First, inheritance (without overriding) is used here as a means of composing hand-written code and automatically generated code. Second, considering that generated methods currently must be monomorphic (§2.3.3), composition with hand-written code is our only way of injecting polymorphic methods into the mix.

Because this mechanism is so useful, it is made available to the user. The `visitors` package accepts a parameter, `ancestors`, which allows the user to list any number of classes that the generated visitor class should inherit. This enables a form of customization: if traversing a value of a (possibly parameterized) type `foo` requires custom behavior, then the method `visit_foo` can be hand-written by the user and inherited by automatically generated visitors. In this paper, we use this mechanism to provide custom behavior for binding constructs (§4, §8).

2.3.6 Using a Visitor. Figure 2 presents two uses of the visitor class `map` obtained in Figure 1. The function `strip`, of polymorphic type `'info . 'info expr -> unit expr`, strips off the decorations in an arithmetic expression, replacing them with unit values. The function `number`, of polymorphic type `'info . 'info expr -> int expr`, decorates each tree node in an arithmetic expression with a unique integer number.

[1] In contrast, when a value of type `'info expr` must be traversed, where the parameterized type `expr` is part of the type definition at hand, the `visitors` package generates a call of the form `(self#visit_expr env ...)`. The treatment of `'info` is specified via the virtual method `visit_'info`, not by parameterizing `visit_expr` with a function `visit_'info`. We plan to revisit this design choice in the future (§10).

In both examples, a concrete implementation of the virtual method `visit_'info` is provided, so as to define how decorations must be transformed. In the second example, a mutable field `count` is declared, as the visitor must thread a current state through the computation. The fact that OCaml allows side effects (and does not keep track of them in the types) makes visitors more versatile than they may at first seem.

In both examples, we are able to focus on the treatment of decorations, and do not need to specify how expressions are traversed. Visitor-based code is not only concise, but also robust in the face of evolution: if the syntax of expressions is altered by adding new data constructors or new arguments to existing data constructors, then the code in Figure 2 requires no change.

These examples highlight the fact that, because the class `map` of Figure 1 is polymorphic, two distinct visitor objects can have distinct types. The visitor used in the definition of `strip` produces expressions of type `unit expr`, whereas the one used in `number` produces expressions of type `int expr`. Finally, these examples highlight the fact that the type variables `'info1` and `'info2` can be instantiated with distinct types. This allows defining heterogeneous transformations: `strip` maps possibly-decorated expressions to undecorated expressions, while `number` maps possibly-decorated expressions to integer-decorated expressions. In this paper, we exploit this flexibility when converting between several representations of names (§5).

3 VISITORS FOR ABSTRACT SYNTAX WITH BINDING

The abstract syntax of arithmetic expressions used as an example in the previous section is quite minimal. In type-theoretic terms, it involves products, sums, and recursion. The `visitors` package knows about these concepts and is able to generate code for them. In order to deal with richer, real-world syntaxes, though, we need a way of constructing visitors for abstract syntax that involves names and binding constructs. Furthermore, we would like to find an approach that enjoys the following properties:

- **Light weight:** an end user does not have to implement visitor classes or methods.
- **Declarativity:** an end user expresses binding structure as part of type definitions.
- **Extensibility:** a library writer can programmatically define a new binding construct.
- **Representation independence:** a binding construct is defined without knowledge of the concrete representation(s) of names and environments.

3.1 Hypothesis: A Binding Construct = A Type with its Visitors

In order to imagine how to meet these requirements, recall how the `visitors` package deals with user-defined types (§2.3.5). The package has no built-in knowledge of or support for these types. When a value of a user-defined type `foo` must be traversed, the package blindly generates a call to the method `visit_foo`. It is up to the user to ensure that a suitable implementation of this method is inherited from some parent class. The package lets the user provide a list of parent classes via the `ancestors` parameter.

A key idea is that a binding construct can be defined in the same guise: *a binding construct is a user-defined type, accompanied with hand-written visitor methods.* These methods define the "meaning" of the binding construct (that is, which names are considered bound in which subterms) by passing suitably extended environments down to their callees.

Let us for a moment hypothesize that someone has defined for us, in a library module named `BindingForms`, the simplest possible binding construct: an abstraction of one name in one term. (This definition is shown in §4.)

This module must provide a type `(_, _) abs`, accompanied with several visitor classes. In `alphaLib`, as mentioned earlier (§2.2), three varieties of visitors are needed, namely `iter`, `map`,

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

Fig. 3. The abstract syntax of λ-calculus (`Term.ml`)

`iter2`. However, for the sake of presentation, we restrict our attention throughout the paper to `map` visitors, as they suffice to illustrate our ideas. Thus, let us assume that this module provides just a type `(_, _) abs` and a visitor class `BindingForms.map` where the method `visit_abs` is defined.

The type `('bn, 'term) abs` is the type of an abstraction. In other words, a value of type `('bn, 'term) abs` can be thought of as a pair of a name `x` and a term `t`, with the convention that `x` is considered bound in `t`. In fact, we will see that an abstraction actually *is* represented in memory as a pair (§4): the type `('bn, 'term) abs` is defined as a synonym for `'bn * 'term`. The type variables `'bn` and `'term` stand for the concrete types of bound names and terms.

As an end user, one does not need to know how the method `visit_abs` is defined, and does not even need to ever explicitly invoke it. One just uses the type `(_, _) abs` as part of an algebraic data type definition. As an example, let us examine how one would define the abstract syntax of the untyped λ-calculus.

3.2 Example: λ-Calculus

Figure 3 presents the abstract syntax of the untyped λ-calculus. This is an ordinary algebraic data type definition. The type `(_, _) abs` is used to indicate that, in the term `TLambda (x, t)`, the name `x` is considered bound in the subterm `t`.

This type definition carries a `[@@deriving visitors { ... }]` annotation, which requests the generation of a `map` visitor class, named `map`. The `ancestors` parameter is used to indicate that the generated visitor should inherit the class `BindingForms.map`. Thus, the library method `visit_abs` is combined with the generated code. The `public` parameter is explained later on (§7).

This definition is representation-independent: the type `('bn, 'fn) term` is parametric in the type `'bn` of bound names and in the type `'fn` of free names. In `TVar x`, the name `x` has type `'fn`, whereas in `TLambda (x, t)`, the name `x` has type `'bn`.

Parameterizing the type of terms over `'bn` and `'fn` in this manner is not mandatory. We make this design decision because it allows us to work with different representations of names at different times. Indeed, virtually every program that manipulates abstract syntax uses at least two representations, namely the raw representation, where all names are represented as strings, and some internal representation, which supports more efficient implementations of comparison, substitution, and so on. By exploiting a `map` visitor, we are able to convert between representations (§5).

In our setting, raw terms are obtained by instantiating both `'bn` and `'fn` with `string`, whereas nominal terms are obtained by instantiating both of them with `Atom.t`. (The module `Atom`, which is part of `alphaLib`, defines "atoms" with unique identity.) This is reflected in the definition of the type abbreviations `raw_term` and `nominal_term` in Figure 3. More representations could be defined, if desired: for instance, de Bruijn terms would be obtained by instantiating `'bn` with `unit`

```
(* An empty type, used below. *)
type void
(* An abstraction is stored in memory as a pair of a name and a term. *)
type ('bn, 'term) abs = 'bn * 'term

class virtual ['self] map = object (self : 'self)
  (* A visitor method for the type abs. *)
  method private visit_abs: 'term1 'term2 .
    _ ->
    ('env -> 'term1 -> 'term2) ->
    'env -> ('bn1, 'term1) abs -> ('bn2, 'term2) abs
  = fun visit_'bn visit_'term env (x1, t1) ->
      let env, x2 = self#extend env x1 in
      let t2 = visit_'term env t1 in
      x2, t2
  (* A hook that defines how to extend the environment. *)
  method private virtual extend: 'env -> 'bn1 -> 'env * 'bn2
  (* A dummy method, never called. *)
  method private visit_'bn (_ : void) (_ : void) = assert false
end
```

Fig. 4. Defining a binding construct: an abstraction of one name in one term (`BindingForms.ml`)

and `'fn` with `int`, whereas locally nameless terms would be obtained by instantiating `'bn` with `unit` and `'fn` with `Atom.t`.

In summary, under the hypothesis that the binding construct `abs` can indeed be defined in a library (§3.1), we have achieved "light weight" and "declarativity". An end user expresses binding structure via type definitions, without writing any code. For free, she gets a `map` visitor, which she can subsequently use to implement several `map`-like operations on terms, such as `copy`, `import`, `export`, `subst`, and so on (§5). There remains to explain how we achieve "extensibility" and "representation independence" as well, that is, how a new binding construct can be defined programmatically and without knowledge of the concrete representations of names and environments. This justifies our hypothesis at the same time.

4 DEFINING A BINDING CONSTRUCT

In this section, we explain how the simplest possible binding construct, `abs`, is defined. We return later on to the question of defining more complex binding constructs (§8).

The key idea spelled out earlier (§3.1) was that, to define such a binding construct, it suffices to define the type `('bn, 'term) abs` and its visitor method `visit_abs`. The definition of the type `('bn, 'term) abs` describes the layout of an abstraction `(x, t)` in memory, whereas the visitor method `visit_abs` is responsible for extending the environment with the name `x` when visiting the term `t`. A second key idea is that *the definition of* `visit_abs` *can be representation-independent.* In order to extend the environment with a new name, there is no need to know how names and environments are represented. It suffices to assume a virtual method `extend`, of which suitable implementations can be provided at a later time.

Figure 4 presents the definition of the type `('bn, 'term) abs`, as well as a class that contains a visitor method for this type, `visit_abs`. These definitions are placed in a file named `BindingForms.ml`, so this class is referred to from the outside as `BindingForms.map`.

The Type `abs`. The type `('bn, 'term) abs` is defined as a synonym for `'bn * 'term`. Thus an abstraction of one name `x` in one term `t` is represented in memory as a pair `(x, t)` of a name and a term. The type parameter `'bn`, for "bound name", determines how a name in a binding position is represented.

The Method `visit_abs`. When applied to an abstraction `(x1, t1)`, the method `visit_abs` visits the term `t1` in an environment that has been extended with information about the name `x1`. This reflects the fact that `x1` is considered bound in `t1`. Because this is a `map` visitor, the method `visit_abs` must return a new abstraction `(x2, t2)`. The name `x2` is computed by `extend` from `env` and `x1`, while the term `t2` is the result of visiting `t1`.

As desired, `visit_abs` is representation-independent. The author of this code does not know how names and binders are represented in the term that is traversed or in the term that is constructed. Neither does she know what information is contained in the environment or *how* the environment is extended. What she does know is *with what names* and *where* the environment must be extended. Specifically, when visiting an abstraction `(x1, t1)`, the environment must be extended *with the name* `x1` and *while visiting the term* `t1`. This, in the end, is the essential meaning of the informal sentence: "the name `x1` is bound in the term `t1`".

Because `visit_abs` does not know *how* to extend the environment, it must delegate this task. We adopt the convention that this is done via a call to the virtual method `extend`. This method receives a name `x1` and an environment `env` and is supposed to return a pair of a name `x2` and an extended environment.

The types of `x1`, `x2`, and `env` are respectively represented by the type variables `'bn1`, `'bn2`, and `'env`. These types are undetermined: they are intended to be fixed later on, in a subclass. The methods `visit_abs` and `extend` are *not* polymorphic in `'bn1`, `'bn2`, and `'env`: these type variables appear free in their types.

Whereas the input term `t1` has type `'term1`, the output term `t2` has type `'term2`. There is no equality constraint between `'term1` and `'term2`: the input and output of a `map` visitor need not have the same type. For instance, later on (§5), we use a `map` visitor to implement an `import` operation, which transforms a raw term into a nominal term. The method `visit_abs` is polymorphic in `'term1` and `'term2`. This allows the end user to use several distinct instances of `abs` as part of a single type definition. For instance, an explicitly-typed syntax for System F involves type-in-type and type-in-term abstractions, therefore requires two distinct instances of `abs`.

The Parameters of `visit_abs`. Because the type `('bn, 'term) abs` is parametric in `'bn` and `'term`, the method `visit_abs` must be parameterized with two visitor functions, `visit_'bn` and `visit_'term`. Indeed, the situation is the same as in our explanation of `visit_list` (§2.3.5).

The role of `visit_'term` should be intuitively clear: it is a function that traverses (and transforms) a subterm. Previously (§3.2), we have used the type `abs` as part of the definition of the type `term` and requested the generation of a visitor for terms. There, the generated code contains a call to `visit_abs` where the formal parameter `visit_'term` is instantiated with an actual argument of the form `(self#visit_term ...)`. Thus, the call to `visit_'term` inside `visit_abs` can be thought of as "a recursive call" which traverses the subterm `t1`.

The parameter `visit_'bn` plays no role: it is unused in `visit_abs`. Due to our conventions, `visit_abs` must expect this parameter. Yet, there is no need for it, because we deal with the name `x1` by using `extend` instead of `visit_'bn`. We provide a dummy method `visit_'bn` and declare that it

```
(* An import environment maps strings to atoms. *)
module StringMap = Map.Make(String)
type env = Atom.t StringMap.t
let empty : env = StringMap.empty

(* When the scope of [x] is entered, the environment is extended
   with a mapping of [x] to a fresh atom [a]. *)
let extend (env : env) (x : string) : env * Atom.t =
  let a = Atom.fresh x in
  let env = StringMap.add x a env in
  env, a

(* When an occurrence of [x] is found, the environment is looked up. *)
exception Unbound of string
let lookup (env : env) (x : string) : Atom.t =
  try StringMap.find x env with Not_found -> raise (Unbound x)

(* Group the above instructions in a little class -- a kit. *)
class ['self] map = object (_ : 'self)
  method private extend = extend
  method private visit_'fn = lookup
end
```

Fig. 5. The "kit" used in defining an `import` operation for any syntax (`KitImport.ml`)

takes arguments of the empty type `void`, which gives us a compile-time guarantee that this method cannot be invoked. In the generated visitor (§3.2), the formal parameter `visit_'bn` is instantiated with `self#visit_'bn`. That said, we may forget about `visit_'bn`.

In summary, our proposed API for defining a new binding construct is as follows. First, define a (parameterized) type, so as to give a name to this construct and define its representation in memory. Then, assuming the availability of an `extend` method, implement a visitor method for this type.

Returning to the visitor class that was generated in §3.2 for the type `('bn, 'fn) term`, we now see that this class, named `map`, has three virtual methods, namely `extend`, whose declaration is inherited from the class `BindingForms.map`, and `visit_'bn` and `visit_'fn`, which are declared by the `visitors` package because the type `('bn, 'fn) term` is parametric in `'bn` and `'fn`. Because a dummy implementation of `visit_'bn` is inherited from the class `BindingForms.map`, two virtual methods remain, namely `extend` and `visit_'fn`. Therefore, in order to define a term-to-term transformation based on the class `map`, it suffices to provide implementations of these two methods. This is the topic of the next section (§5).

5 IMPLEMENTING OPERATIONS ON TERMS

In this section, we show how to construct a term transformation, that is, a function that takes a term and produces a term. We assume that a `map` visitor, such as the one constructed previously (§3), is at hand. That is all we require: we do not need any knowledge of the syntax of terms.

```
let import_term env t =
  (object
    inherit [_] map                    (* auto-generated; see Figure 3 *)
    inherit [_] KitImport.map (* provided by a library; see Figure 5 *)
  end) # visit_term env t
```

Fig. 6. The "glue" used in defining an `import` operation for the λ-calculus of Figure 3

As explained above, to implement one term transformation, one must provide implementations of the virtual methods `extend` and `visit_'fn`. For clarity, we place these two method definitions in a little standalone class, which, following Pouillard and Pottier [2012, §5.4], we call a "kit".

In this section, we illustrate the construction of an `import` operation on terms. This operation, sometimes known as "name resolution", typically takes place immediately after parsing. It converts a raw term, where every name is represented as a string, to a nominal term, where every name is represented as an atom. It fails if it encounters a reference to a name that has not been properly bound.

5.1 The Kit

The `import` kit, shown in Figure 5, is placed in the file `KitImport.ml`, so the little class `map` defined in this kit is referred to from the outside as `KitImport.map`.

Because the purpose of an `import` operation is to transform strings to atoms, the environment, in this case, must be a finite (immutable) map of strings to atoms. An efficient implementation of maps as balanced binary search trees is found in OCaml's standard library module `Map`.

When the scope of a name `x` is entered, the environment must be extended with a mapping of the string `x` to a fresh atom `a`. We define the method `extend` to do this. When this kit is later combined with the `map` visitor of Figure 3, the method `extend` is invoked at every `TLambda` node by the visitor method `visit_abs`, which itself is invoked by the generated method `visit_TLambda`.

When a free name occurrence `x` is found, the environment must be looked up, so as to find which atom corresponds to the string `x`. If there is no entry for `x` in the environment, then an exception must be raised. We define the method `visit_'fn` to do this. When this kit is later combined with the `map` visitor of Figure 3, the method `visit_'fn` is invoked at every `TVar` node by the generated method `visit_TVar`.

A kit is independent of the syntax of terms, so it can be written once and for all and placed in a library. `alphaLib` includes a set of such kits. A kit is also binding-construct-independent: it does not care which binding constructs are used in the syntax of terms. A kit is usually not representation-independent: it may assume specific representations of bound names, free names, and environments. For instance, in the `import` kit, the source representation is the raw representation, the target representation is the nominal representation, and the environment is a map of strings to atoms.

5.2 The Glue

The "glue" code that is required to complete the definition of the function `import_term` appears in Figure 6. The term visitor of Figure 3 and the import kit of Figure 5 are combined by multiple inheritance. When the function `import_term` is invoked, a fresh visitor object is allocated and its `visit_term` method is called so as to carry out the desired transformation.

The function `import_term` has type `KitImport.env -> raw_term -> nominal_term`. This offers an example of a conversion between representations.

```
module type OUTPUT = sig
  type ('bn, 'fn) term
  type raw_term = (string, string) term
  type nominal_term = (Atom.t, Atom.t) term
  (* Queries about the free atoms of a term. *)
  val fa_term: nominal_term -> Atom.Set.t
  val filter_term: (Atom.t -> bool) -> nominal_term -> Atom.t option
  val closed_term: nominal_term -> bool
  val occurs_term: Atom.t -> nominal_term -> bool
  (* Queries about the bound atoms of a term. *)
  val ba_term: nominal_term -> Atom.Set.t
  val avoids_term: Atom.Set.t -> nominal_term -> bool
  val guq_term: nominal_term -> bool
  (* Renaming some or all of the bound atoms of a term. *)
  val copy_term: nominal_term -> nominal_term
  val avoid_term: Atom.Set.t -> nominal_term -> nominal_term
  (* Conversions between raw and nominal terms. *)
  exception Unbound of string
  val import_term: KitImport.env -> raw_term -> nominal_term
  val export_term: KitExport.env -> nominal_term -> raw_term
  val show_term: nominal_term -> raw_term
  (* Comparing terms up to alpha-equivalence. *)
  val equiv_term: nominal_term -> nominal_term -> bool
  (* Substituting terms for variables in terms. *)
  val subst_TVar_term:
    (nominal_term -> nominal_term) ->
    nominal_term Atom.Map.t -> nominal_term -> nominal_term
end
```

Fig. 7. A toolbox of operations

The glue code must come after the definition of terms. In the simplest scenario, it is written by the end user. Unfortunately, this requires the user to write about 5 lines of boilerplate per operation. Since there can be over 15 operations of interest, this is rather unpleasant. Ways of eliminating the glue are discussed later on (§7). In the meantime (§6), we review a set of common operations on terms that can be defined on top of visitors.

6 A TOOLBOX OF OPERATIONS ON NOMINAL TERMS

In the previous section (§5), we have presented a general recipe for constructing operations on terms and illustrated it with the definition of an `import` operation. This recipe can be used to build many common operations on terms. The required kits are defined once and for all as part of `alphaLib`. The glue, in the simplest scenario, is written by the end user. The set of operations is not closed: if desired, more kits can be implemented by the end user. In this section, we review the operations that are offered, as of today, by `alphaLib` (§6). We have focused for the time being on the nominal representation, because we believe that it strikes a good compromise between conceptual simplicity, ease of implementation, and efficiency (§9). Nevertheless, it would be easy, by writing more kits, to extend the library with support for other representations.

Our current set of operations is shown in Figure 7. (The signature `OUTPUT` is the result signature of the functor `Toolbox.Make`, which is discussed in §7.1.) It is divided in the following groups.

Free Atoms. "`fa_term t`" computes the set of the free atoms of the term `t`. "`filter_term p t`" tests whether the term `t` has a free atom that satisfies the property `p`. Both are constructed on top of an `iter` visitor. Although `filter_term` could be defined in terms of `fa_term`, it is preferable to implement it directly, so as to ensure that iteration is stopped as soon as a free atom that satisfies `p` is found. (This is done by raising and catching an exception.) "`closed_term t`" tests whether the term `t` is closed, while "`occurs_term a t`" tests whether the atom `a` occurs free in the term `t`. These operations are special cases of `filter_term`.

Bound Atoms. "`ba_term t`" computes the set of the bound atoms of the term `t`, by which we mean, the set of atoms that occur in a binding position in `t`. This function is "ill-behaved" in the sense that it does not respect α-equivalence [Gabbay and Pitts 2002, Example 6.12]. Therefore, one might argue that this function should not be exposed to the user (and, in a "safe" language, it should be impossible for the user to define it). We deliberately offer a low-level, unsafe toolbox, because we do not know how to enforce "safety" without compromising efficiency. In such a setting, exposing this function makes sense. It is especially useful in debugging assertions, where one wishes to ensure that the bound atoms of a term are disjoint with some set of in-scope atoms. Indeed, our substitution function (§6) requires such a property.

"`avoids_term bad t`" tests whether the set of bound atoms of the term `t` is disjoint with the set `bad` *and* there is no "shadowing" within `t` (that is, no atom is bound twice along a branch in `t`). "`guq_term t`" tests whether the bound atoms of the term `t` are pairwise distinct (that is, no atom is bound twice anywhere in `t`). The mnemonic `guq` stands for "globally unique". Like `ba_term`, these functions do not respect α-equivalence, and are typically used as part of debugging assertions.

Freshening. "`copy_term t`" returns a term that is α-equivalent to `t` and where every bound name has been replaced with a fresh name. "`avoid_term bad t`" returns a term that is α-equivalent to `t` and where every bound name in the set `bad` has been replaced with a fresh name. It is a possibly less expensive variant of `copy_term`.

Conversions between Raw and Nominal Terms. "`import_term env t`" converts the raw term `t` to an α-equivalent nominal term, in the environment `env`, whose domain must include the free names of `t`. Its code has been presented earlier (§5). This function is not specialized to an empty environment: a scenario where a nonempty environment is needed is a read-eval-print loop, where the user can make definitions in an incremental manner, and each new definition must be interpreted in the scope of all previous definitions.

"`export_term env t`" converts the term `t` back to a raw term, for display. This is done in the scope of an export environment `env`, which maps atoms to strings. In a read-eval-print loop, a nonempty export environment might be needed, say, to print a type, because a type may contain free references to previously-defined types.

"`show_term t`" returns a raw term obtained by converting every (bound and free) atom in `t` to a string that gives away the atom's internal identity. This function is intended to be used for debugging. It is nevertheless a correct way of converting a closed nominal term to a raw term.

Comparing Terms up to α-Equivalence. "`equiv_term t1 t2`" tests whether the terms `t1` and `t2` are α-equivalent. This function is constructed on top of an `iter2` visitor. Its implementation can be understood as a conversion of both terms to a locally nameless representation, followed with an equality test in that representation. For efficiency, the conversions and the equality test are fused. An ordering function that respects α-equivalence could be defined in the same style.

```
module type INPUT = sig
  (* The type of terms. *)
  type ('bn, 'fn) term
  (* A map visitor. *)
  class virtual ['self] map : object ('self)
    method visit_term : 'env -> ('bn1, 'fn1) term -> ('bn2, 'fn2) term
    method private visit_TVar : 'env -> 'fn1 -> ('bn2, 'fn2) term
    method private virtual extend : 'env -> 'bn1 -> 'env * 'bn2
    method private virtual visit_'fn : 'env -> 'fn1 -> 'fn2
  end
  (* ... [iter] and [iter2] visitors are required too, but not shown. *)
end
```

Fig. 8. What is needed to build an operations toolbox?

```
module Make
  (Term : INPUT)  (* see Figure 8 *)
: OUTPUT          (* see Figure 7 *)
    with type ('bn, 'fn) term = ('bn, 'fn) Term.term
```

Fig. 9. Signature of the functor `Toolbox.Make`

Substitution. "`subst_TVar_term copy sigma t`" applies the substitution `sigma` to the term `t`, replacing a variable `(TVar x)` with the term `(copy u)` if `sigma` maps the atom `x` to the term `u`. A substitution `sigma` is a finite map of atoms to terms. This operation is implemented on top of a `map` visitor. The required glue involves not only combining the generated visitor with a kit, as usual, but also overriding the visitor method `visit_TVar` so as perform the desired replacement at a free variable.

By design, this operation is not automatically capture-avoiding (§9). It has the precondition that the bound atoms of `t` must be fresh for `sigma`, that is, disjoint with the domain and codomain of `sigma`. The "codomain" part of this requirement guarantees that the free atoms of a term `u` that appears in the codomain of `sigma` cannot be captured by a binder in `t` when `u` is grafted into `t`. The "domain" part guarantees that an atom in the domain of `sigma` cannot be masked by a binder within `t`. Thus, it is safe to apply the substitution `sigma`, unrestricted, to every variable in the term `t`.

The subterms `u` that are grafted into `t` are "copied" on the fly using the user-specified function `copy`. If `copy` is the identity function, no copying takes place. If it is `copy_term`, then every bound atom inside a grafted subterm is freshened. This feature can be used to ensure that the result of the substitution satisfies "global uniqueness".

7 GETTING RID OF THE GLUE

As noted earlier (§5), in the approach that we have explained so far, although visitors are generated and kits are ready-made, the end user still must write (or copy and paste) some glue code. This is rather unpleasant, because the user must work more than we would like, and because this exposes some implementation details that we would rather hide, such as which visitors and kits are used.

7.1 Glue as a Functor

To remedy this problem, the first idea that comes to mind is simple: since the glue is always the same, or almost the same, can't it be written once and made part of `alphaLib`? Although it cannot be an ordinary module, because it depends on the end user's definition of terms, perhaps it can be a functor.

7.1.1 Defining a Functor. Let us clarify what the glue code depends on, that is, which user-defined types, classes, and methods it relies upon. More precisely, in an object-oriented setting, we must think both about the methods that this code requires and the methods that it offers. Let us for instance go back to the glue code that was shown in Figure 6. Its interaction with its environment can be summarized as follows:

- There must exist a type `(_, _) term`. Indeed, although (thanks to type inference) this type is not explicitly mentioned in the code, it does occur in the type of the function `import_term`. We expect the type `(_, _) term` to be defined by the user, as in Figure 3.
- There must exist a class `map`. We expect this class to be automatically generated for the user-defined type `term` via a `[@@deriving visitors { ... }]` annotation.
- The class `map` must have a method `visit_term`. Indeed, this method is explicitly invoked in the code. This method must be public: in OCaml, a private method cannot be invoked from the outside of an object. Note that this method is named after the user-defined type `term`.
- The class `map` must have a method `visit_TVar`. Indeed, the glue code for `subst_term` (§6) overrides this method, and in OCaml, one cannot override a method which one does not know exists.
- For the same reason, the class `map` must have two virtual methods `extend` and `visit_'fn`. Indeed, every kit provides implementations of these methods.

Fortunately, OCaml's module system is sufficiently powerful to express these considerations. The glue code can be packaged in a functor, `Toolbox.Make`, whose input signature, shown in Figure 8, summarizes the above elements. In the body of the functor, we place the hundred-or-so lines of glue code required to build our toolbox of common operations (§6). The output signature of the functor has been presented already in Figure 7. The type of the functor `Toolbox.Make` is shown in Figure 9.

Mentioning the virtual methods `extend` and `visit_'fn` in the signature `INPUT` (Figure 8) amounts to advertising the functor is willing to tolerate the presence of these virtual methods. In other words, the functor provides its own implementations of these methods. If we did not list these methods in the input signature, the functor itself would still be well-typed, but its application to the class `map` of Figure 3 would become ill-typed, as OCaml would complain that the existence of a virtual method cannot be hidden.

Although Figure 8 explicitly shows only the class `map`, several visitor classes are required by the functor. The visitor varieties `iter`, `map`, and `iter2` are sufficient to implement all of the operations listed in Figure 7.

7.1.2 Using the Functor. To use this functor, an end user writes a functor application. If `Term` is the user-defined module in Figure 3, this takes the form "`module T = Toolbox.Make(Term)`". Thereafter, the operations on terms are accessible under the names `T.fa_term`, `T.ba_term`, and so on.

As a technical remark, let us note that the class `map` of Figure 3 has several methods that are *not* mentioned in the signature `INPUT`. Examples of such methods include `visit_TLambda` and `visit_TApp`. For the functor application `Toolbox.Make(Term)` to be valid, these methods must be private. Indeed, in OCaml, a private method can become hidden during signature matching, that

```
(* Naming conventions. *)
#define VISIT(term)      CONCAT(visit_, term)
#define IMPORT_FUN(term) CONCAT(import_, term)
(* Glue code. *)
#define IMPORT(term)             \
  let IMPORT_FUN(term) env t =  \
    (object                     \
      inherit [_] map           \
      inherit [_] KitImport.map \
    end) # VISIT(term) env t
```

Fig. 10. Glue macros for `import`

is, when an actual signature is compared against an expected signature [Leroy et al. 2016, §3.6, "Private methods] [Minsky et al. 2013, Chapter 12, "Classes"], whereas a public method can never be hidden. The parameter `public = ["visit_term"]` in Figure 3 serves this purpose. It instructs the `visitors` package that all generated methods except `visit_term` should be declared as private.

7.1.3 Evaluation. This functor-based approach to eliminating the glue may seem successful: the end user effectively now has to write zero lines of glue code. The only burden that remains on her shoulders is to write the `[@@deriving visitors { ... }]` annotations and the functor application.

The main shortcoming of this approach, however, is that it makes strong assumptions about the user's abstract syntax. Indeed, the functor assumes that there is only one syntactic category, that this syntactic category is named `term`, and that the data constructor that represents a variable is named `TVar`.

If there are two syntactic categories, say `term` and `typ`, one might attempt to get away with two functor applications. However, applying the functor to `typ` requires several renamings: for the purposes of the functor application, the type `typ` should be renamed `term`, and the method `visit_typ` should be renamed `visit_term`. Although renaming a type is possible, renaming a method is not. One can simulate such a renaming by delegation, but that is awkward. As the last straw that breaks the camel's back, the user may need both type-in-type and type-in-term substitution functions, and that is not permitted by the functor as it stands.

In conclusion, the functor `Toolbox.Make` appears to be a satisfactory solution when it is applicable, that is, in the simple case where there is just one syntactic category and the user is willing to use the names `term` and `TVar`.

7.2 Glue as Macros

Outside of this simple case, the functor-based approach appears to break down. We cannot think of a way of salvaging it, so suggest resorting to macros instead. cppo is an OCaml analogue of the infamous C preprocessor. It is easy to turn all of our glue code into a set of cppo macro definitions, such as the one in Figure 10, which corresponds to the glue code shown earlier in Figure 6. To understand these macro definitions, it suffices to know that **`CONCAT`** is cppo's analogue of cpp's concatenation operator ##. Thanks to this macro definition, an end user need write just `IMPORT(term)` to obtain the function `import_term`, and `IMPORT(typ)` to obtain the function `import_typ`, without further fuss. It is possible to propose a macro that builds an entire toolbox at once, so the user does not have to write one macro invocation per operation.

These macro definitions can be used in the body of the functor that was discussed previously (§7.1). This means that offering both a glue functor and glue macros to the end user does not require code duplication in the library. This also allows us to check, when the library is compiled, that the macros expand to well-formed code.

This macro-based approach to getting rid of the glue seems reasonably effective, albeit rather inelegant. Its main drawback is perhaps that, if the visitor classes provided by the user do not have the shape expected by the macros, a rather unclear type error is reported in the expanded code (which the user cannot see).

8 DEFINING COMPLEX BINDING CONSTRUCTS

So far, we have defined and used only the simplest possible binding construct, namely `abs`, an abstraction of one name in one term. This construct is not built-in: it is defined programmatically (§4). In this programmatic style, a wide range of more elaborate binding constructs can be defined by a library writer or by the end user (§8.1). However, although this programmatic approach to defining binding constructs is relatively simple, it requires a certain amount of repetitive code. Therefore, we sketch a library of combinators, inspired by Unbound [Weirich et al. 2011], which allow the end user to describe binding structure in a declarative style (§8.2).

8.1 A Programmatic Approach

Let a "simple definition" be a triple `(x, t, u)` of a name `x` and two subterms `t` and `u`, with the convention that `x` is bound in `u`, not in `t`. (This is analogous to ML's **`let`**.) To model this, following our API (§4), one would first define a type `('bn, 't, 'u) def` as a synonym for `'bn * 't * 'u`, then define the visitor method `visit_def`. This method should visit `t` under the outside environment and visit `u` under an environment that has been extended with the name `x`. We omit the code: reconstructing it should be an easy exercise.

Many binding constructs can be defined in this programmatic style. For instance, "multiple definitions", which involve not just one pair `(x, t)` but a list of such pairs $(\mathtt{x}_i, \mathtt{t}_i)$, followed with a single term `u`, can be modeled. The visitor method should visit every term $\mathtt{t}_i$ under the outside environment and visit `u` under an environment that has been extended with all of the names $\mathtt{x}_i$'s. If there is a condition that the $\mathtt{x}_i$'s should be pairwise distinct, it is up to the user to initially enforce this condition, say, during parsing, or immediately after parsing. This condition is then naturally preserved by all of the operations in our toolbox (§6).

"Telescopes" [Weirich et al. 2011, §3.2] in appearance have the same structure as multiple definitions. However, in a telescope, the name $\mathtt{x}_i$ is in scope not only in `u`, but also in the terms $\mathtt{t}_{i+1}$, $\mathtt{t}_{i+2}$, and so on. (This is analogous to Scheme's **`let*`**.) This can again be easily modeled with an appropriate visitor method.

"Recursive multiple definitions" have the same structure as multiple definitions and telescopes, but have the convention that every name $\mathtt{x}_i$ is in scope in every term $\mathtt{t}_i$ as well as in `u`. (This is analogous to ML's **`let rec`**.) This can again be modeled with an appropriate visitor method. The most natural implementation walks the list of definitions twice: once to find all of the names $\mathtt{x}_i$ and extend the environment, and once to visit all of the terms $\mathtt{t}_i$ in this extended environment. In the case of a `map` visitor, these two traversals produce two lists, which are then zipped to produce a list of pairs of a transformed name and a transformed term. Again, if there is a condition that the $\mathtt{x}_i$'s should be pairwise distinct, then it must separately enforced.

8.2 A Declarative Approach

Although defining a binding construct by programming its visitor methods is a relatively simple and powerful style, it still requires a certain amount of repetitive code, which is boring and easy to get wrong. Therefore, a more declarative approach to describing binding structure seems desirable.

Fortunately, such a declarative approach can be constructed on top of the programmatic approach. Following Weirich *et al.* [2011], we propose building a library of binding combinators, each of which is accompanied with suitable visitor methods. The end user then declares the desired binding structure simply by using these combinators, without writing any code.

`alphaLib` offers such a combinator library. This is a preliminary design: although it is simple and effective, it lacks an analogue of Unbound's Rec combinator. We discuss the limitations of our approach further on (§8.3).

Taking inspiration from Cαml [Pottier 2006] and Unbound [Weirich et al. 2011], we distinguish two syntactic categories, namely "terms" and "patterns". (Terms are known as "expressions" in the literature.) An intuitive justification for this distinction is that traversing a term simply *requires* an environment, whereas traversing a pattern *requires and extends* an environment, because a pattern binds zero, one or more names. In the following, we let a visitor method for a term receive an ordinary *immutable* environment, while a visitor method for a pattern receives an enriched *mutable* context. (We use different words, namely "environment" and "context", to distinguish these notions.) Extending the context via a side effect allows us to fit in the mold of visitors (§2), which imposes that the context be propagated top-down, as opposed to threaded left-to-right.

There is ample room in the design space of binding combinators. We propose the following combinators, which are relatively easy to understand and implement. They are essentially Unbound's combinators, minus Rec.

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

The most precise way of explaining the meaning of these combinators is to examine their visitor methods, shown in Figure 11. Although the type signatures of these methods are quite verbose, their code is very short. At the top of Figure 11 are type definitions for the types `'p abstraction`, `'bn binder`, `'t outer`, and `'p rebind`. Each of these parameterized types is defined as a synonym for its parameter. Thus, these binding combinators have no runtime representation: they should be understood as annotations that can be used as part of a type definition so as to influence the behavior of the generated visitor.

Next in Figure 11 comes the type `'env context`. If `'env` is the type of an ordinary, presumably immutable environment, used while visiting a term, then `'env context` is the type of a mutable context, used while visiting a pattern. A context has two components, namely `outer` and `current`. The field `outer` holds an ordinary environment. This environment is used when an embedded subterm must be visited. The field `current` holds a mutable reference to an ordinary environment. This reference is updated with an extended environment when a binder is encountered. Because a generated visitor guarantees a left-to-right traversal, the environment stored in the field `current` reflects the bound names that appear leftwards of the current point.

```
type 'p abstraction = 'p
type 'bn binder = 'bn
type 't outer = 't
type 'p rebind = 'p
(* A context is used while visiting a pattern. *)
type 'env context = { outer: 'env; current: 'env ref }
(* The meaning of the combinators is given by their visitor methods. *)
class virtual ['self] map = object (self : 'self)
  method private virtual extend: 'env -> 'bn1 -> 'env * 'bn2
  method private visit_abstraction: 'env 'p1 'p2 .
    ('env context -> 'p1 -> 'p2) ->
    'env -> 'p1 abstraction -> 'p2 abstraction
  = fun visit_p env p1 ->
      visit_p { outer = env; current = ref env } p1
  method private visit_binder: _ ->
    'env context -> 'bn1 binder -> 'bn2 binder
  = fun visit_'bn ctx x1 ->
      let env = !(ctx.current) in
      let env, x2 = self#extend env x1 in
      ctx.current := env;
      x2
  method private visit_outer: 'env 't1 't2 .
    ('env -> 't1 -> 't2) ->
    'env context -> 't1 outer -> 't2 outer
  = fun visit_t ctx t1 ->
      visit_t ctx.outer t1
  method private visit_rebind: 'env 'p1 'p2 .
    ('env context -> 'p1 -> 'p2) ->
    'env context -> 'p1 rebind -> 'p2 rebind
  = fun visit_p ctx p1 ->
      visit_p { ctx with outer = !(ctx.current) } p1
end
```

Fig. 11. A simple library of binding combinators (`BindingCombinators.ml`)

Last in Figure 11 come the visitor methods which define the meaning of the four primitive combinators by their action on the environment or context. `visit_abstraction` causes a switch from "term mode" to "pattern mode". Out of an ordinary environment `env`, it creates a context, which is used when visiting the pattern `p1`. The `outer` component is `env`: this records the environment that exists outside of this abstraction. The `current` component is a fresh reference, which is initialized with `env`: this means that no names have been bound yet. `visit_binder` updates the `current` component of the context by extending it with the name `x1`. The `outer` component is unaffected. `visit_outer` switches back from "pattern mode" to "term mode". The `outer` component, an ordinary environment, is used to visit the embedded term `t1`. Finally, `visit_rebind` visits the pattern `p1` in a context whose `outer` component is the current value of `current`. Thus, the effect of the combinator `rebind` is to locally change the meaning of `outer`: the names that are bound leftwards of rebind(p) are in scope in any subterms that are embedded in p.

```
#define tele     ('bn, 'fn) tele
#define term     ('bn, 'fn) term
(* The types that follow are parametric in 'bn and 'fn: *)
type tele =
  | TeleNil
  | TeleCons of 'bn binder * term outer * tele rebind
and term =
  | TVar of 'fn
  | TPi  of (tele, term) bind
  | TLam of (tele, term) bind
  | TApp of term * term list
[@@deriving visitors { variety = "map";
                       ancestors = ["BindingCombinators.map"] }]
```

Fig. 12. Defining telescopes in terms of binding combinators

The combinator inner(t) is sugar for rebind(outer(t)): thus, the names that are bound leftwards of inner(t) are in scope in the term t. The combinator bind(p, t) is sugar for abstraction($p \times$ inner(t)): thus, the names bound by the pattern p are in scope in the term t. We omit the OCaml definitions of these combinators.

Because environments and contexts have distinct types, the distinction between terms and patterns is statically enforced by the OCaml typechecker. Attempting to use an ill-formed construction, such as inner(rebind(p)), as part of a type definition, causes a type mismatch in the generated visitor. Unfortunately, the type error message is rather obscure, as the typechecker attempts to unify `'env` with `'env context` and complains that this gives rise to a recursive type.

This little combinator library can be used to define several complex binding constructs in a fully declarative manner. As an example, Figure 12 shows how an end user might declare the syntax of a dependently-typed λ-calculus whose Π and λ forms involve a telescope. The definition is essentially identical to Weirich *et al.*'s [2011, §3.2], with the proviso that their combinator rebind(p_1, p_2) is encoded here as $p_1 \times$ rebind(p_2). Recall that type application in OCaml is reversed, so "`term outer`" means outer(`term`), and so on. As in our earlier example (§3), we parameterize the syntax over `'bn` and `'fn`, so the types of telescopes and terms are representation-independent. The (local) macros `tele` and `term` are not essential: they save us the trouble of repeating the parameters `('bn, 'fn)` everywhere. Without them, the definition would be very much cluttered. This hack is reminiscent of Coq sections.

8.3 Comparison and Limitations

In discussing the expressiveness of our approach, one should distinguish two layers, namely the programmatic layer (§8.1), where a binding construct is defined by programming a visitor method, and the declarative layer (§8.2), where a binding construct is described using a set of combinators. The second layer is built on top of the first, so any limitations inherent in the design of the first layer must be shared by the second layer as well.

The first layer rests upon two ideas, to wit: (1) a binding construct can be defined purely by its visitor method, and (2) this method can be defined in a representation-independent manner by relying solely on the virtual method `extend`. Although these ideas seem to shine by their simplicity, this API for defining binding constructs may be sometimes too restrictive. Because calling `extend` is the only way of extending the environment, this API effectively requires a complex binding

construct to be "linearized", that is, translated on the fly into a sequence of nested single-name abstractions. Furthermore, this API does not allow testing whether two bound names are equal. Because of this, it apparently cannot model binding constructs where a bound name may have multiple binding occurrences, such as ML's disjunction patterns, or the "join patterns" of join-calculus [Fournet and Gonthier 1996]. (It should be noted that, as far as we understand, Unbound does not support these constructs either.) There are various ways in which we might work around this limitation. One approach might be to accept this limitation and transform the surface syntax down into an internal form where multiple binding occurrences are not needed. For instance, in an internal syntax for ML, one might require that each of the names bound by an ML pattern be explicitly listed (once) at the beginning of the pattern. Another approach might be to give the visitor method access to operations other than `extend`, such as testing two bound names for equality, or looking up a bound name in an environment. However, by doing so, one would compromise representation independence: indeed, a nameless representation obviously does not support these operations.

The second layer rests upon a set of combinators. Although the design space of combinators may seem quite large, it is subject to a strong constraint: the meaning of each combinator is defined purely by its visitor method, independently of all other combinators. In other words, the meaning of each combinator must be defined in isolation, in terms of its action on the context. This is not an easy game to play. Although we are able to encode most of Unbound's combinators [Weirich et al. 2011], we find Rec problematic. As noted earlier (§8.1), the most natural way of dealing with a recursive binding construct is to traverse it twice, once to collect its bound names, once to visit its embedded subterms. In the case of a `map` visitor, these two traversals produce two data structures which must then be zipped (combined). Unfortunately, the pattern "visit-then-visit-then-zip" does not seem a good fit for visitors. Although we have a few potential solutions to this problem, we do not deem them mature enough to be presented in this paper, and leave the Rec combinator for future work.

Our combinators are closely related to those of Cαml [Pottier 2006]. Cαml has primitive abstractions (written with angle brackets) and `inner` and `outer` combinators. Cαml's abstractions are implicitly recursive, so *all* of the names bound in a pattern are in scope in every `inner` subterm. Thus, Cαml can model "recursive multiple definitions" (§8.1), whereas our combinators cannot. Conversely, Cαml lacks rebind, so it cannot model telescopes. In terms of implementation, Cαml is a monolithic code generator. It is not open-ended in any way: it offers just one representation of names (namely, the nominal representation) and a fixed set of operations.

In comparison with Unbound [Weirich et al. 2011], our combinator language lacks Rec. On the other hand, our combinator library is representation-independent: it allows working with several representations of names and converting between representations, whereas Unbound's metatheory and implementation seem tied to the locally nameless representation. There may be a connection between these remarks: Unbound's assumption of a specific representation, where a bound name is represented as a pair of integers $j@k$, seems to open the door to a simple and direct treatment of Rec, which does not require our complicated "visit-then-visit-then-zip" pattern.

9 RELATED WORK

Concrete Representations of Names. Many representations of names have been proposed in the literature, for use in programming and in proving. The locally nameless representation, where free names are represented as atoms and bound names are represented as de Bruijn indices, is one of the most popular representations. Aydemir *et al.* [2008] advocate its use in proof assistants, noting that it does not need the "shift" operations that are required in de Bruijn's representation,

that it is nevertheless canonical (that is, α-equivalence coincides with equality), and that substitution can be defined by structural recursion, without "freshening", yet without risk of capture. When programming, as opposed to proving, these advantages do not seem decisive. Nevertheless, Unbound [Weirich et al. 2011] uses this representation both in its formal description and in its implementation. A drawback of this representation is its heavy reliance on the operations of "opening" and "closing" a binding construct, which, if implemented naïvely, are costly.

The nominal representation, where names are represented as atoms, has been widely used as well. It is used, for instance, by the Glasgow Haskell compiler [Peyton Jones and Marlow 2002], by FreshOCaml [Shinwell 2006], by FreshLib [Cheney 2005] and Cαml [Pottier 2006]. It is really three representations in one, as one can choose to work with either: (1) arbitrary terms; or (2) terms that have "no shadowing", that is, where no atom is bound twice along a single branch; or (3) terms that enjoy "global uniqueness", that is, where no atom is bound twice anywhere. It is convenient to assume the strongest invariant (that is, global uniqueness) when descending into a term and to enforce the weakest invariant (that is, none at all) when coming back up and constructing a new term. In this scenario, both "opening" and "closing" are no-ops, so simplicity and efficiency are both achieved. A `copy_term` operation is used, when necessary, to go from the weakest invariant back to the strongest one.

Our substitution operation, `subst_term`, is not a "sledgehammer" [Peyton Jones and Marlow 2002]: it does not perform systematic freshening. The sledgehammer is obtained by composing `copy_term` and `subst_term`. Peyton Jones and Marlow's "rapier", where freshening is performed only when necessary to avoid shadowing and is combined with an application-specific operation (say, inlining), can be programmed by a user of our library, as we expose the concrete representation of terms: the type `nominal_term` is not abstract.

Binding Specification Languages. Many domain-specific languages for describing binding structure have appeared in the literature. LF signatures [Harper et al. 1993], λ-tree syntax [Miller 2000], and nominal signatures [Urban et al. 2004] allow abstractions of one name in one term. In Talcott's binding structures [1993], each operator carries a fixed number of variables as well as a fixed number of subterms, and a fixed binding relation tells which variables are considered bound within which subterms. More generally, Ott [Sewell et al. 2010] and Knot [Keuchel et al. 2016] allow defining auxiliary name-collecting functions, which are then used to specify which sets of names are in scope in which subterms. Still more generally, it seems, InBound [Keuchel and Schrijvers 2015] exploits inherited and synthesized attributes to express the flow of bound variables and contexts. This seems somewhat similar to our "programmatic style" (§8.1), where this flow is programmed in OCaml. However, our approach is more limited in expressive power: because the computation must be expressed as an instance of a visitor, it must be performed in one pass.

FreshOCaml [Shinwell 2006] offers a generalized form of nominal signatures, where the left-hand side of an abstraction is not necessarily a name, but can be a value of arbitrary type. This allows describing constructs that bind a variable number of names as well as constructs where one name has multiple binding occurrences. Cαml [Pottier 2006] and Unbound [Weirich et al. 2011] offer richer, combinator-based specification languages, which we have discussed earlier (§8.3).

Neron *et al.* [2015] describe binding structure by translating syntax to a more abstract data structure, a "scope graph", which involves nested scopes, declarations of names, and references to names. The process of finding out where a name is declared, or "name resolution", amounts to searching for certain paths in this graph. There is freedom both in the definition of which paths are admissible and in the definition of the translation of programs to scope graphs. The authors wish to describe this translation in a declarative manner, by using a domain-specific language such as NaBL [Konat et al. 2013].

Generic Programming. Gibbons [2006] explains datatype-generic programming and discusses its connection with four classic object-oriented design patterns, including Composite and Visitor. Jeuring *et al.* [2008] offer a survey of several libraries for generic programming in Haskell. Cheney's FreshLib [2005] is based on SYB [Lämmel and Peyton Jones 2005], while Unbound [Weirich et al. 2011] is built on top of RepLib [Weirich 2006].

Allais *et al.* [2017] define several operations on λ-terms, including renaming, substitution, and printing, as instances of a generic evaluation function, which can be compared to a `fold` visitor. (The special case of "syntactic" operations can be compared to a `map` visitor.) By exploiting Agda's type system, they are able to statically ensure that object-level terms are well-scoped and to prove properties of programs (e.g., "applying a renaming is the same as viewing this renaming as a substitution and applying this substitution").

In the OCaml world, `ppx_deriving` allows annotating a type definition with `[@@deriving ...]` and offers facilities for type-directed code generation. These facilities are exploited by "plug-ins", which are written in OCaml and have access to the OCaml compiler's internal representations. `ppx_deriving` comes with a set of plug-ins that generate monolithic recursive functions, such as `eq`, `show`, `map`, `fold`, and so on. `ppx_deriving_morphism` is a plug-in that produces `map` and `fold` visitors. It does not exploit OCaml's classes and objects: instead, visitors are expressed as records-of-functions. This lets an end user override one "method" and "inherit" default behavior elsewhere. Unfortunately, whereas object types are open-ended (one can invoke a method without knowing which other methods exist), record types are closed (the fields of a record type are fixed when the type is declared). Thus, this approach does not allow multiple visitor "classes" to be combined. `ppx_traverse` is a plug-in that produces visitor classes: `iter`, `map`, and more. It is closely related with `visitors`, but lacks several key features, such as the generation of one method per data constructor (which we use to get custom behavior at one data constructor), the parameterization of every method with an environment (which we use to define the meaning of binding constructs), and the `ancestors` parameter (which we use to get custom behavior at preexisting types). Fan [Zhang and Zdancewic 2013], a general-purpose syntactic metaprogramming system for OCaml, could most likely be used to perform type-directed generation of visitor classes or (more directly) to develop an embedded domain-specific language for describing binding structure.

Libraries and Tools for Dealing with Nameplate. FreshLib [Cheney 2005], Unbound [Weirich et al. 2011], and Bound are Haskell libraries that help deal with names and binding. In the OCaml world, to the best of our knowledge, there are no such libraries, perhaps due to OCaml's lack of metaprogramming facilities. Cαml [Pottier 2006] is a code generator. It is not customizable: it supports a fixed binding specification language, a fixed representation of names, and a fixed set of operations. This criticism also applies to Ott [Sewell et al. 2010], which can produce not only OCaml code, but also definitions for several proof assistants. In the world of proof assistants, several tools can generate definitions and theorems about abstract syntax with binding. They include Nominal Isabelle [Urban 2008], LNgen [Aydemir and Weirich 2010], GMeta [Lee et al. 2012], AutoSubst [Schäfer et al. 2015], and Needle [Keuchel et al. 2016]. Pouillard and Pottier [2012] propose a representation-independent API for working with names and binding, expressed in Agda, and use logical relations to prove the correctness of several representations with respect to this API.

Dedicated Programming Languages. Several programming languages with built-in support for binding have been proposed, including λProlog [Nadathur and Miller 1988], FreshOCaml [Shinwell 2006], and Beluga [Pientka 2008]. Ferreira and Pientka [2017] implement such a language as a lightweight syntactic extension of OCaml. At the OCaml level, variables are represented as de Bruijn indices, and GADTs are used to statically ensure that terms are well-scoped. These low-level

details are hidden from the user, who is presented with the illusion of built-in support for binding, in the style of higher-order abstract syntax. It would be interesting to extend the `visitors` package with support for GADTs, as our two approaches could then possibly interoperate.

10 FUTURE WORK

In this paper, the `visitors` package has been used to generate *monomorphic* methods. However, the `visitors` package also supports generating *polymorphic* methods, which are explicitly annotated with a polymorphic type. This helps improve compositionality and allows generating visitors for nested algebraic data types. It would be worth investigating whether this feature allows traversing well-scoped syntax in the style of Bird and Paterson [1999]. At this time, generalized algebraic data types (GADTs) are not supported by the `visitors` package. It would be desirable to add such a feature, as GADTs are commonly used to represent well-typed syntax [Xi et al. 2003].

`alphaLib` [Pottier 2017a] is still at a preliminary stage. The convenience of the library in "real-world" applications should be assessed. More kits should be written, so as to support more operations and more representations of names. We should investigate how to best support multiple sorts of names. More research should be carried out so as to strengthen the set of binding combinators, which currently lacks Unbound's Rec and cannot deal with multiple binding occurrences of a name. Finally, it would be desirable to prove that the library correctly implements (a subset of) Unbound's binding combinators, for which a formal semantics has been given by Weirich *et al.* [2011].

REFERENCES

Guillaume Allais, James Chapman, Conor McBride, and James McKinna. 2017. Type-and-scope Safe Programs and Their Proofs. In *Certified Programs and Proofs (CPP)*. 195–207.

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *Principles of Programming Languages (POPL)*. 3–15.

Brian Aydemir and Stephanie Weirich. 2010. *LNgen: Tool Support for Locally Nameless Representations.* Technical Report MS-CIS-10-24. University of Pennsylvania Department of Computer and Information Science.

Richard Bird and Lambert Meertens. 1998. Nested Datatypes. In *Mathematics of Program Construction (MPC) (Lecture Notes in Computer Science)*, Vol. 1422. Springer, 52–67.

Richard Bird and Ross Paterson. 1999. de Bruijn Notation as a Nested Datatype. *Journal of Functional Programming* 9, 1 (1999), 77–91.

James Cheney. 2005. Scrap your nameplate. In *International Conference on Functional Programming (ICFP)*. 180–191.

Francisco Ferreira and Brigitte Pientka. 2017. Programs Using Syntax with First-Class Binders. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 10201. Springer, 504–529.

Cédric Fournet and Georges Gonthier. 1996. The Reflexive Chemical Abstract Machine and the Join-Calculus. In *Principles of Programming Languages (POPL)*. 372–385.

Murdoch J. Gabbay and Andrew M. Pitts. 2002. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing* 13, 3–5 (2002), 341–363.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley.

Jeremy Gibbons. 2006. Datatype-generic programming. In *International Spring School on Datatype-Generic Programming (Lecture Notes in Computer Science)*, Vol. 4719. Springer, 1–71.

Robert Harper, Furio Honsell, and Gordon D. Plotkin. 1993. A Framework for Defining Logics. *Journal of the ACM* 40, 1 (1993), 143–184.

Johan Jeuring, Sean Leather, José Pedro Magalhães, and Alexey Rodriguez Yakushev. 2008. Libraries for Generic Programming in Haskell. In *Advanced Functional Programming (Lecture Notes in Computer Science)*, Vol. 5832. Springer, 165–229.

Steven Keuchel and Tom Schrijvers. 2015. InBound: simple yet powerful specification of syntax with binders. (2015). Unpublished.

Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. 2016. Needle & Knot: Binder Boilerplate Tied Up. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 9632. Springer, 419–445.

Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. 2013. Declarative Name Binding and Scope Rules. In *Software Language Engineering (Lecture Notes in Computer Science)*, Vol. 7745. Springer, 311–331.

Ralf Lämmel and Simon Peyton Jones. 2005. Scrap your boilerplate with class: extensible generic functions. In *International Conference on Functional Programming (ICFP)*. 204–215.

Gyesik Lee, Bruno C. d. S. Oliveira, Sungkeun Cho, and Kwangkeun Yi. 2012. GMeta: A Generic Formal Metatheory Framework for First-Order Representations. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 7211. Springer, 436–455.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2016. The OCaml system: documentation and user's manual. (2016).

Dale Miller. 2000. Abstract Syntax for Variable Binders: An Overview. In *Computational Logic (Lecture Notes in Computer Science)*, Vol. 1861. Springer, 239–253.

Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. 2013. *Real World OCaml: Functional programming for the masses*. O'Reilly.

Gopalan Nadathur and Dale Miller. 1988. An Overview of Lambda-Prolog. In *Logic Programming*. 810–827.

Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of name resolution. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 9032. Springer, 205–231.

Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming* 12, 4&5 (2002), 393–433.

Brigitte Pientka. 2008. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Principles of Programming Languages (POPL)*. 371–382.

François Pottier. 2006. An overview of Cαml. In *ACM Workshop on ML (Electronic Notes in Theoretical Computer Science)*, Vol. 148. 27–52.

François Pottier. 2017a. `AlphaLib`. https://gitlab.inria.fr/fpottier/alphalib. (2017).

François Pottier. 2017b. The `visitors` package. https://gitlab.inria.fr/fpottier/visitors. (2017).

Nicolas Pouillard and François Pottier. 2012. A unified treatment of syntax with binders. *Journal of Functional Programming* 22, 4–5 (2012), 614–704.

Didier Rémy and Jérôme Vouillon. 1998. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems* 4, 1 (1998), 27–50.

Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *Interactive Theorem Proving (ITP) (Lecture Notes in Computer Science)*, Vol. 9236. Springer, 359–374.

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. 2010. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming* 20, 1 (2010), 71–122.

Mark R. Shinwell. 2006. Fresh O'Caml: nominal abstract syntax for the masses. *Electronic Notes in Theoretical Computer Science* 148, 2 (2006), 53–77.

Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. 2003. FreshML: Programming with Binders Made Simple. In *International Conference on Functional Programming (ICFP)*. 263–274.

Carolyn Talcott. 1993. A Theory of Binding Structures and Applications to Rewriting. *Theoretical Computer Science* 112, 1 (1993), 99–143.

Christian Urban. 2008. Nominal Techniques in Isabelle/HOL. *Journal of Automated Reasoning* 40, 4 (2008), 327–356.

Christian Urban, Andrew Pitts, and Murdoch Gabbay. 2004. Nominal Unification. *Theoretical Computer Science* 323 (2004), 473–497.

Stephanie Weirich. 2006. RepLib: a library for derivable type classes. In *Haskell workshop*. 1–12.

Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. 2011. Binders unbound. In *International Conference on Functional Programming (ICFP)*. 333–345.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Principles of Programming Languages (POPL)*. 224–235.

Hongbo Zhang and Steve Zdancewic. 2013. Fan: compile-time metaprogramming for OCaml. (2013). Unpublished.