Towards Large-scale Refactoring for OCaml

REUBEN N. S. ROWE, University of Kent, UK
SIMON J. THOMPSON, University of Kent, UK

Refactoring is the process of changing the way a program works without changing its overall behaviour. The functional programming paradigm presents its own unique challenges to refactoring. For the OCaml language in particular, the expressiveness of its module system makes this a highly non-trivial task. The use of PPX preprocessors, other language extensions, and idiosyncratic build systems complicates matters further.

We begin to address the question of how to refactor large OCaml programs by looking at a particular refactoring—value binding renaming—and implementing a prototype tool to carry it out. Our tool, Rotor, is developed in OCaml itself and combines several features to manage the complexities of refactoring OCaml code. Firstly it defines a rich, hierarchical way of identifying bindings which distinguishes between structures and functors and their associated module types, and is able to refer directly to functor parameters. Secondly it makes use of the recently developed `visitors` library to perform generic traversals of abstract syntax trees. Lastly it implements a notion of 'dependency' between renamings, allowing refactorings to be computed in a modular fashion. We evaluate Rotor using a snapshot of Jane Street's `core` library and its dependencies, comprising some 900 source files across 80 libraries, and a test suite of around 3000 renamings.

We propose that the notion of dependency is a general one for refactoring, distinct from a refactoring 'precondition'. Dependencies may actually be *mutual*, in that all must be applied together for each one individually to be correct, and serve as declarative specifications of refactorings. Moreover, refactoring dependency graphs can be seen as abstract (semantic) representations.

CCS Concepts: • **Software and its engineering → Software notations and tools**; **Software maintenance tools**; • **Theory of computation → Semantics and reasoning**; **Abstraction**; *Program constructs*; *Functional constructs*;

Additional Key Words and Phrases: Refactoring, Renaming, Dependencies, Binding Structure, OCaml

ACM Reference Format:
Reuben N. S. Rowe and Simon J. Thompson. 2018. Towards Large-scale Refactoring for OCaml. 1, 1 (May 2018), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Refactoring is a necessary and ongoing process in both the development and maintenance of any codebase [Fowler et al. 1999]. Individual refactoring steps are often conceptually very simple (e.g. rename this function from `foo` to `bar`, swap the order of parameters `x` and `y`). However applying them in practice can be complex, involving many repeated but subtly varying changes across the entire codebase. Moreover, refactorings are, by and large, context sensitive, meaning that even powerful low-tech utilities (e.g. `grep` and `sed`) are only effective up to a point.

Take as an example the renaming of a function, which is the refactoring that we focus on in this paper. As well as renaming the function at its definition point, every call of the function

Authors' addresses: Reuben N. S. Rowe, University of Kent, Canterbury, Kent, CT2 7NZ, UK, r.n.s.rowe@kent.ac.uk; Simon J. Thompson, University of Kent, Canterbury, Kent, CT2 7NZ, UK, sjt@kent.ac.uk.

XXXX-XXXX/2018/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

must also be renamed (i.e. many repeated changes). In OCaml [Minsky et al. 2013], the language targeted by our work, the function identifier may be used as a *punned* argument in which case an argument label must be introduced by the renaming (i.e. subtly varying changes). Additionally, scoping must be taken into account: not all occurrences of the string corresponding to the old name should necessarily be replaced (i.e. renaming is context-sensitive). These aspects make performing refactoring by hand and reviewing the resulting changes tedious and error-prone. It is clear that refactoring-oriented, language-aware tool support can therefore provide major improvements in productivity during the refactoring process, as well as during code review after the refactoring has been performed.

Although some tools provide limited support for localised refactoring tasks (e.g. `merlin` can rename an identifier within a given source file), there is currently no general purpose automatic refactoring tool for the OCaml language. Our goal is develop such a tool, and we have begun by implementing a prototype, named Rotor,[1] which supports renaming of value bindings (i.e. both the identifier used at the definition site, as well as all call site identifiers) across an entire codebase. Whilst ostensibly very simple, this already presents the significant challenges that we expect to feature in most, if not all, refactoring tasks for OCaml. Thus, we believe our approach provides a generic foundation for implementing further refactorings.

The OCaml language itself has a number of features that present unique challenges for refactoring tools. These include its expressive module system, as well as its tooling and build ecosystem. We summarise these challenges in Section 2. To manage the complexities of renaming in OCaml, Rotor implements two novel solutions that enable it to break a top-level renaming task into a number of compositional, and *manageable*, subtasks.

Rich, hierarchical identifiers. The OCaml compiler provides datatypes for identifiers, in the `Longident` and `Path` modules, that capture the basic notion of hierarchy in OCaml programs. However the representations implemented by these datatypes do not contain enough information to distinguish the sort of each element (i.e. module, signature, value). Thus, to refer unambiguously to all the different elements that may have to undergo renaming, Rotor defines a richer datatype for identifying the subcomponents of an OCaml program. We give details of this datatype, as well as the design and implementation of Rotor more generally, in Section 5.

Subsidiary refactorings and dependencies. In the process of renaming a binding specified by the user, we often find that other bindings must be renamed as a consequence. One example of when this happens is on encountering an `include` of the binding's parent module(s) within another module. We examine the different possible situations in Section 3. By transitively computing the set of such 'induced' renamings, Rotor breaks a single monolithic renaming into a number of smaller, more straightforward subtasks. In general we may say that a renaming *depends* upon these other renamings that it induces, in the sense that the latter must all be applied in order for the former to be correct. We argue that this concept can be extended beyond renaming, to refactoring more generally. Moreover, our notion of refactoring dependency is distinct from the the notion of refactoring *precondition* introduced by Opdyke [1992], since dependencies may be mutual and have distinct footprints on the codebase as changesets.

We propose treating sets of refactoring dependencies as first-class objects of study. For example, they can be seen as *declarative* specifications of the action of a refactoring upon a given codebase. Furthermore, the set of renaming dependencies of all bindings in a given program can be viewed as an abstraction of the program itself. We also believe that this declarative view of refactoring will work equally well for other languages and paradigms, and thus the contribution of this paper is

[1] Reliable OCaml Tool for OCaml Refactoring.

not just to refactoring OCaml, but to refactoring in general. We explore the concept of refactoring dependencies in more detail in Section 4.

In summary, this paper makes the following contributions.

(1) We develop the first tool for performing a large scale refactoring over a whole OCaml codebase, and evaluate it on an extensive real world example.
(2) We provide a generic framework in which further refactorings can be developed.
(3) We identify a novel abstract concept within the refactoring literature—that of refactoring dependencies—and propose it as general mechanism for specifying and presenting the action of a refactoring upon a codebase.

The rest of the paper is organised as follows. In Section 2 we survey the features of the OCaml language which present particular problems in the context of refactoring. Section 3 then presents a number of examples which illustrate these challenges and motivate our general treatment of refactoring for OCaml. In Section 4 we elaborate our novel concept of refactoring dependencies before describing the design and implementation of our prototype tool, Rotor, in Section 5, with particular emphasis on the features that provide a general framework for large-scale refactoring in OCaml. Section 6 then presents a case study in which we have applied Rotor to a large codebase consisting of a number of the Jane Street [2018] public OCaml libraries. Finally, Section 7 surveys related work and in Section 8 we conclude and discuss future work.

2 CHALLENGES IN REFACTORING OCAML

Many features of the OCaml language present unique, idiosyncratic challenges for refactoring. Perhaps foremost of these is its rich module system, which lifts many concepts found at the function level. For example, structures and functors may be taken to correspond, respectively, with ground values and functions: modules may have recursive definitions and functors may be higher-order, i.e. taking other functors as arguments, returning them as results, or both. Types for modules may be: independently declared; bound to (module type) identifiers; subject to further type constraints; and used as annotations that trigger compiler checks—mirroring the treatment of types for ordinary values. To push the analogy a little further, a structure (type) itself is most like a record (type) in that it may have subcomponents; and these may themselves be modules or module types, allowing complex hierarchies to be built.

These language features endow OCaml with a kind of self-similarity property, meaning many issues that would only appear at a 'whole-program' level in other languages can already appear within a single file OCaml program. Rich, hierarchical module systems like OCaml's are not found in other functional languages, such as Haskell [Peyton Jones 2003] or Erlang [Armstrong et al. 1996]. In some ways, OCaml's module system is more akin to the use of objects, (generic) classes and interfaces in object-oriented languages. Of course, we note that OCaml also contains a 'true' object-oriented subset and so challenges specific to refactoring object-oriented code will also apply to OCaml, possibly interacting with its functional aspects in interesting ways that we have not yet considered. Notwithstanding, although Rotor does not yet address refactorings specific to the object-oriented part of the language, OCaml's object-oriented features do not interfere with renamings of the functional components.

OCaml's module system also incorporates other features that go beyond the lifting of simple functional concepts. For one, the `include` keyword allows (a subset of) the bindings of one module to be taken wholesale and re-exported as bindings defined, at the point of inclusion, by the including module. This can have far-reaching consequences for renaming a given binding within a large codebase, which can thus cascade into a number of different renamings through `include`s of the binding's parent module(s) within other modules. These other renamings may also cascade similarly,

quickly resulting in a blow-up. Modules (and module types) can also be *aliased*, and so a renaming within the aliased module cascades into a renaming within the aliasing module, and vice versa.

Two further language extensions present different challenges (which Rotor does not yet handle) by introducing tighter couplings between the various subsystems of the language than are present in core OCaml. Firstly, module type extraction permits a module expression to stand for its module type as inferred by the compiler. This means that a module type expression may contain a module expression as a subcomponent; in core OCaml module expressions can contain module types, but not vice versa. Secondly, first class modules permit wrapping module expressions up as values that can be passed to or returned from functions; correctly performing refactoring in the presence of this language extension may require additional data-flow analysis.

Whilst the main challenge in developing a refactoring tool for the OCaml language has been the complexity of the module system, Rotor must also work around certain issues arising from the build infrastructure and wider language ecosystem in order to function effectively as a practical tool over large codebases. Firstly, the PPX preprocessor infrastructure [Leroy et al. 2017, Chap. 27] provides a standard syntax for specifying where preprocessors may insert or replace original source code with automatically generated code, along with a plugin architecture for applying them.[2] The use of PPX preprocessors is becoming ever more common and presents a fundamental challenge for refactoring tools, which ultimately must associate the code that is actually compiled with the source files as written and seen by the programmer. Second, Rotor must have knowledge of any packaging strategies that are used to manage the codebase. For example, the `jbuilder` build system creates libraries and associated namespaces by automatically transforming the identifiers of top-level modules. Thus, for code compiled using `jbuilder`, Rotor must know how to map the module identifiers it encounters back into the correct filesystem paths for locating the original source files.

We have, in fact, implemented Rotor in the OCaml language itself. This has proved very useful in ameliorating some of these 'supra-lingual' challenges since it allows existing software solutions already in the ecosystem, as well as the compiler infrastructure, to be reused by and incorporated into Rotor as libraries.

3 MOTIVATING EXAMPLES

In this section, we present some simple example programs that illustrate the surprising number of complexities involved in renaming bindings in OCaml. We use them to motivate Rotor's strategy of dividing a renaming into logically separate subtasks, and our notion of dependencies between refactorings that we explore further in Section 4 below.

3.1 Module Includes and Aliases

Modules can be included within other modules using the `include` keyword. This takes (a subset of) the bindings in the included module and re-exports them as bindings defined, at the point of inclusion, by the including module. For example, consider the program below.

```
1  module A = struct
2    let foo = 1
3    let bar = 2
4    let baz = 3
5  end

6  module B = struct
7    include
8      (A : sig val foo : int
9               val bar : int end)
10   val bar = 4
11 end ;;

12 A.foo + B.foo ;;
13 A.bar + B.bar ;;
```

[2]The previous incarnation of this infrastructure, `camlp4`, is now deprecated.

The two modules `A` and `B` each define some integer value bindings, and then two expressions composed of these bindings are evaluated. For simplicity, we have used bindings to integer values, but the principles we describe apply to bindings of any type.

Module `A` includes module `B`, although notice that the `include` is restricted by an explicit module type annotation so not all of `A`'s bindings are included. Moreover, the module `B` *rebinds* `bar` to a new value. The program contains five distinct bindings: one for each of `foo` and `bar` in each of the modules `A` and `B`, and `baz` in module `A`. What changes must be made to the program source to rename each of them?

- To rename `A.foo`, we need to rename the identifier in the binding on line 2, as well as the reference on line 12 and the binding declaration in the module type annotation at line 8. However since the binding is also included in module `B`, which is then referred to on line 12, we must also rename `B.foo`. Not doing so would result in a malformed program.
- Renaming `B.foo` requires renaming the reference to it on line 12. In contrast to `A.foo`, there is no 'local' definition to be renamed since the binding definition is included from `A`. However, this module `include` still requires `A.foo` to be renamed in order for the refactoring to result in a valid program. That is, the two renamings are *dependent* on one another. Notice that after applying both of these renamings, the behaviour of the program remains the same.
- When renaming `A.bar`, similarly to before, we rename the bound identifier on line 3, the binding declaration on line 9, and the reference to the binding in the expression on line 13. However in this case, even though the binding is included in module `B`, we do not need to rename `B.bar`. This is because `B` rebinds `bar`, and so uses of the identifier `B.bar` refer to this local binding in `B`. Thus, it has no dependency on `A.bar`. Renaming `B.bar` involves renaming the identifier in the binding on line 10 and the identifier that references it on line 13.
- For the binding of `baz` in module `A`, we only need to rename the identifier in the binding on line 4. Since there is no declaration for `baz` in the module type annotation for the `include` of `A` in `B`, there is no need to perform a renaming of `B.baz`, since `B` contains no binding for `baz`.

Similar considerations for renaming arise from the use of module *aliases*, which allow one module to reuse the implementation of another. The following program reproduces a set of dependencies analogous to those illustrated above for module includes. Here, renamings of `C.foo` and `D.foo` are dependent on one another, but module `D` does not contain a binding for `bar` because of the explicit module type restriction on the alias.

```
1  module C = struct
2    let foo = 1
3    let bar = 2
4  end

5  module D = (C : sig val foo : int end) ;;
6  C.foo + D.foo ;;
```

3.2 Module Interfaces

In addition to modules, OCaml also allows module type expressions to be bound to identifiers, which can then be used as module type annotations. This can potentially couple together changes in many different modules across a codebase, if they all implement the same *interface* (i.e. named module type).

Consider the program in Listing 1, which defines a module type `Magma` that encapsulates the mathematical notion of a set with a binary operation. The module type also stipulates a way of generating elements of the datatype. Additionally, it defines two concrete modules that implement this interface and evaluates whether the magma operation of each one is commutative.

If we wanted to rename the `op` binding in either of modules `M1` or `M2`, this would involve renaming the identifier in the bindings on lines 8 and 14, respectively, as well as their references on lines 19

```
1  module type Magma = sig
2    type t
3    val op : t -> t -> t
4    val choose : unit -> t
5  end
6  module M1 : Magma = struct
7    type t = int
8    let op x y = x + y
9    let choose () =
10     Random.int (Random.bits ())
11 end
12 module M2 : Magma = struct
13   type t = float
14   let op x y = x - y
15   let choose () = Random.float
16     (float_of_int (Random.bits ()))
17 end ;;
18 let x,y = M1.(choose (), choose ())
19 in M1.(op x y = op y x) ;;
20 let x,y = M2.(choose (), choose ())
21 in M2.(op x y = op y x) ;;
```

Listing 1. An example program demonstrating the issues for renaming when using module interfaces.

and 21, respectively. However, in doing this we would also have to rename the declaration of op in the `Magma` module type. If this is not also done, then after applying the renaming to `M1` or `M2` these modules would not be correctly implementing the `Magma` interface—they would be missing a definition for op. So renaming `M1.op` or `M2.op` depends on renaming `Magma.op`. Conversely, renaming `Magma.op` depends on renaming *both* `M1.op` and `M2.op`. Thus, renaming a binding in *any* module implementing an interface (transitively) depends on renaming the binding in *all* such modules, as well as in the interface itself.

3.3 Module Type Aliases and Includes

Similarly to the case for modules considered above, module types may be included within and aliased by other module types, e.g.

```
22 module type Monoid = sig
23   include Magma
24   val zero : t
25 end
26 module type Groupoid = Magma
```

In the presence of these module type declarations, the renamings considered in Section 3.2 above would also depend on renaming both `Groupoid.op` and `Monoid.op`. For these renamings, notice that their footprints (the locations in the source code that must be modified) are empty since the bindings are not defined locally to these module types. However, they themselves can be used as interfaces and thus contribute to a further 'cascade' of dependencies on renaming other bindings defined elsewhere in the codebase. This notion of a footprint, and how it relates to a general notion of dependency for refactoring, is discussed in a more formalised way in Section 4.

3.4 Module Type Constraints

OCaml allows module types to be defined by modifying other module types in certain ways, for example by applying equality constraints and (destructive) substitutions to its submodules. For example, consider the following two module types.

```
1 module type S = sig
2   module M : sig val foo : int end
3 end
4 module type T =
5     S with module M = C
```

The module type `T` is defined by modifying `S` to specify that the submodule `M` is equal to the module `C`, defined above in Section 3.1. If the binding `C.foo` is being renamed, we now must generate two other dependent renamings. Firstly `T.M.foo` must be renamed and, as described above, this will

also induce renamings for those modules whose interface is declared to be `T`. Secondly, because the submodule `M` in the module type `S` also declares a binding for `foo`, this too must be renamed. That involves renaming the identifier in the declaration on line 2, and will also induce further renamings for modules implementing the interface `S`. On the other hand renaming `C.bar`, although depending on renaming `T.M.bar`, should not induce a renaming of `S.M.bar` since there is no such binding.

Now consider the two module types below.

```
6  module type TestMagma = sig
7    module M : Magma
8    val idempotent : M.t -> bool
9  end
```

```
10 module type TestM1 =
11     TestMagma with module M := M1
```

`TestM1` is bound to a module type defined by modifying `TestMagma` with a *destructive* substitution of the module `M1` for its submodule `M`. This means that the `TestM1` module does *not* declare a submodule `M`, but does contain all the other bindings declared in `TestMagma` with any references to the substituted module `M` replaced by `M1`. In this case, it leaves only a declaration of the binding `idempotent` with type `M1.t -> bool`. To carry out a renaming of `M1.op`, there is now also a dependency on renaming `TestMagma.M.op` induced by the module substitution. However note that because the substitution is destructive, the module type `TestM1` does *not* declare a binding for `M.op`. Thus there should not be a dependency on renaming `TestM1.M.op`.

3.5 Functors

The use of functors also propagates dependencies when renaming value bindings. Since functors produce modules, which can be aliased, this can lead to having to rename bindings within functor bodies. Indeed, functors themselves can also be aliased since they too are modules. Additionally, renamings in modules that are used as an argument to a functor can induce renamings in other modules that are also used as arguments to that functor.

For example consider the program below, which defines a functor that creates a string conversion function for a pair from string conversion functions for each of its components. It is then used to print out an axiom of Peano arithmetic.

```
1  module Int = struct
2    type t = int
3    let to_string i = int_to_string i
4  end
```

```
5  module String = struct
6    type t = string
7    let to_string s = s
8  end
```

```
9  module Pair (X : sig type t val to_string : t -> string end)
10             (Y : sig type t val to_string : t -> string end) = struct
11   type t = X.t * Y.t
12   let to_string (x, y) = X.to_string x ^ " " ^ Y.to_string y
13 end
14 module P = Pair(Int)(Pair(String)(Int)) ;;
15 print_endline (P.to_string (0, ("!=", 1))) ;;
```

Let us consider the chain of dependencies that are induced, and their footprints, by renaming the `P.to_string` function.

- The module `P` is an alias of the result of applying the `Pair` functor; so as well as having to rename the reference to `P.to_string` on line 15, we must also rename the identifier on line 12 in the binding defined in the body of the functor.

- Since the result of applying the `Pair` functor is itself used as the second argument to another application of the functor, and the `to_string` binding is declared in the signature of the corresponding parameter, we must also rename the binding in this functor parameter. The footprint of this dependency is the declaration of the binding in the signature of the parameter on line 10, and the reference to the binding `Y.to_string` on line 12.
- Renaming the binding in the second parameter now induces a dependency on renaming the `Int.to_string` binding, since the module `Int` is also passed as the second argument to the nested application of the `Pair` functor. For this program, the footprint of this dependency is the binding definition in the `Int` module on line 3.
- The `Int` module is also passed as the first argument to the outer application of the `Pair` functor. Thus, the binding of `to_string` in the first parameter of the functor must be renamed too. This involves renaming the declaration in the signature of the first parameter, on line 9, and the reference to the binding `X.to_string` on line 12.
- Finally, renaming the binding in the first parameter induces a dependency on renaming the `to_string` binding in the `String` module, which is also passed as the first argument to the inner application of the `Pair` functor. The footprint of this dependency is the identifier in the binding defined on line 7.

This example very clearly demonstrates the fact that a renaming may quickly induce a long chain of dependencies; renaming in OCaml is, in general, a highly complex task. In order to control this complexity, in the next section we show how these dependencies can be formalised.

4 REFACTORING DEPENDENCIES

The examples in Section 3 illustrate how an initial renaming may induce other renamings. This happens due to the interaction of a number of OCaml's language features, such as the `include` construct; module (type) aliasing; the use of interfaces (i.e. named module types) and module type constraints; and the application of functors. These renamings can induce further renamings, and so on, based upon the particular structure of the program that the renaming is being applied to.

We see this phenomenon not as an incidental artefact of renaming within OCaml, but as indicative of a more fundamental aspect of the refactoring process itself. To explain and understand this more fully we propose to formulate an abstract notion of *dependency* for refactoring actions, which is novel in the literature on refactoring.

4.1 Distilling a Notion of Dependency for Refactoring

As the examples in Section 3 demonstrate, a renaming *depends* upon the other renamings that it induces in the sense that the latter must all be applied in order for the former to be correct. We have noticed two interesting aspects about these renaming 'dependencies'. One is that induced renamings are *mutually* dependent on the renaming that induced them, and that the network of *direct* dependencies can have complex, non-trivial structure. The other is that the *footprints* of the individual renamings—the particular set of changes in the codebase that they represent—do not overlap.

This suggests that our renaming dependencies constitute something other than refactoring *preconditions*, a concept introduced by Opdyke [1992] in his thesis which was one of the founding projects in the field. A refactoring precondition must be applied, or already hold, *before* the refactoring itself can be applied. It may involve modifying code which is then subsequently modified by the primary refactoring, i.e. the footprints of the precondition and the refactoring may overlap. Also, refactoring preconditions do not generally exist in complex, interrelated networks.

It is thus perhaps tempting to see the set of renamings generated in this way as something intrinsic to, or a procedural artefact of, carrying out the initial renaming and thus only relevant as an implementation detail. However we believe there are at least two very good reasons not to take this view.

(1) We found that in order to render implementing renaming for OCaml even feasible we had to modularise the problem by identifying these 'subsidiary' renamings and then applying them separately. When a particular design choice fundamentally threads itself through the architecture of a system, it can indicate a deeper concept worthy of further elucidation.

(2) Faced with a large, automatically computed patch to a codebase, it can be very difficult for the programmer to determine if applying it really achieves the intended result. An understanding of which renaming subtasks are involved and their interrelationships can actually provide confidence that this is so, and therefore make code review and change management more effective. Alternatively, if the set of dependencies is large, it may help the programmer to realise that a different, more local change is actually what should be enacted.

Although each individual renaming is dependent on the others and therefore, in some sense, a facet of the same 'object' to be renamed, these relationships are not *ipso facto* evident to the programmer. From a high-level point of view, each renaming applies to a logically independent component. Indeed it is the underlying dependencies that determine how the refactoring unfolds, not vice versa. There is therefore utility revealing this structure.

4.2 Widening the Notion of Dependency

The notion of refactoring dependency that we have examined thus far has been informed by, and is tailored to, renaming value bindings in OCaml. However, we expect it to generalise quite naturally to refactoring in a wider sense than simply renaming, and also to apply to other programming languages.

Many of the situations illustrated in Section 3 will also apply, for example, to renaming modules and module types themselves. Furthermore, it is not difficult to imagine that similar notions of dependency will also exist for value types. These are also defined in modules and declared in module types; may alias or reference one another in their definitions; and may also be used in value type annotations. This means that renaming, or even modifying the definition of value types will induce further such refactorings. Moreover, since value types affect the definition of value bindings, modifying value types will induce dependencies on modifying value bindings. We begin to see that the network of dependencies for a general refactoring may be composed of many qualitatively diverse operations.

We have also mentioned, in Section 2, that OCaml's module system has some similarity to programming constructs found in object-oriented languages. It is therefore perhaps reasonable to expect that our notions of refactoring dependency will have some analogue in the context of these other languages. Indeed, OCaml itself also contains true object-oriented features (classes, objects, inheritance, methods, etc) and so further extensions of dependencies to deal with this subset of the language should carry over to other object-oriented settings.

4.3 Abstract Properties of Refactoring Dependencies

In anticipating the more general application of the notion of dependency for refactorings, it behooves us to consider: what are the possible abstract properties that characterise them? We have already hinted at some, above: the footprint of individual dependencies do not overlap, and the direct dependency relation is a graph structure. We also noted that, for renaming value bindings,

direct dependencies appear to always be mutual—thus the dependency graph is *connected*. We do not know if this will be the case in general.

We can also observe a further aspect in our analysis in Section 3. We can ask, for a given refactoring, what are its dependencies with respect to some subset of the program. For example, a dependency induced by a module `include` can be determined by examining the renaming with respect to the including module. That is, each dependency can be linked to a *particular* subcomponent of the program. Of course, each dependency can be determined by analysing the *whole* program, but the direct dependencies (and indeed the footprint) of each refactoring is actually only determined by some *subset* of the program. We can call this subset the 'kernel' of the refactoring for the program. A consequence of this is that the dependency graph can be constructed, node by node, using the kernel of each node.

These observations form the basis of the generic interfaces implemented by our tool, Rotor, and the algorithm it uses for computing refactorings, which are discussed in Section 5.

4.4 Applications

Our concept of refactoring dependency, as far as we are aware, is a novel one. We believe it will contribute to wider research into refactoring and programming languages in the following ways.

4.4.1 Presenting the Results of Refactorings. Refactoring dependencies provide an alternative way to present the results of a refactoring instance to the user. A `diff` patch is a 'flat' presentation and gives little information about relationships between the changes. In contrast, a dependency graph displays much more structure and gives more 'logical' information about the refactoring and its relationship to the code. Combining a dependency graph with a `diff` gives both a concrete and an abstract presentation of the refactoring to the user at the same time.

4.4.2 Declarative Specifications for Refactorings. A refactoring dependency graph can be viewed as the result of 'compiling' the high-level, human description of a refactoring task ("rename this binding") into a *declarative* specification of what it means to carry out the refactoring on a given codebase. Consequently, it can be taken to constitute the 'meaning' of the refactoring with respect to a given program. This contrasts with previous approaches in which refactorings are understood, specified and implemented from an *operational* point of view, and points towards a possible way of treating refactoring *semantically*.

4.4.3 Program Abstraction. We suggested above that the renamings in a graph of mutual renaming dependencies are really the various faces of a single underlying program subcomponent. Considering more dependency graphs consequently captures a view on a greater portion of the program. Thus, we propose that sets of dependency graphs actually represent an abstraction of the program itself. By considering more kinds of dependency, we can capture increasingly finer abstractions.

4.4.4 Verification. Insofar as refactoring dependencies constitute some (semantic) notion of meaning and abstraction for both refactorings and the programs they operate on, they provide a foundation for reasoning about refactorings, and verifying their correctness.

5 ROTOR: A TOOL FOR REFACTORING OCAML

We now describe the implementation of our tool, Rotor. We begin in Section 5.1 with a high level description of Rotor's architecture and basic pipeline, as well as how it accommodates and makes use of the OCaml language infrastructure. We then examine a number of important aspects of its design in more detail. In particular, we address how Rotor:

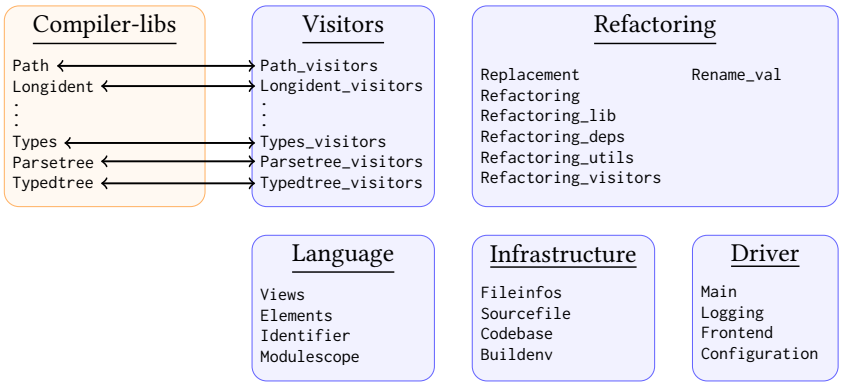

Fig. 1. Diagrammatic representation of Rotor's high-level architecture.

– provides a *generic* framework for implementing and applying refactorings to OCaml code (§5.2);
– identifies different subcomponents of programs (§5.3);
– analyses abstract syntax trees using automatically generated *visitor* classes (§5.4); and
– generates the dependencies and results of applying refactorings (§5.5).

Rotor is available as an open source project online: gitlab.com/trustworthy-refactoring/rotor.

5.1 High-level Architecture

A diagrammatic representation of Rotor's high-level architecture is given in Fig. 1, showing the major components of the tool. These include visitor classes for traversing abstract syntax trees; a generic framework for supporting refactorings, along with the concrete implementation of renaming; datatypes for abstractly representing logical elements of the OCaml language itself; modules for accessing a program's codebase and managing metadata about its compilation environment; and finally, a front end allowing the user to interact with the tool.

5.1.1 Refactoring Pipeline. As input, Rotor takes a codebase (i.e. a program and/or set of libraries), a value binding identifier, and a new name for the binding. As output, it produces a patch file of changes that can be applied to the codebase to effect the renaming. We opted against having Rotor perform the renaming in-place in favour of an approach that more easily facilitates integration with other tools and external pipelines.

In order to minimise the processing time for the refactoring, Rotor first computes the source file dependency graph of the codebase. Rotor computes this itself rather than, e.g., relying on the output of `ocamldep`, which can be ambiguous in the absence of extra information about the build environment of the program (cf. Section 5.1.3 below). This dependency graph can be pre-computed, and incrementally updated according to the changes introduced by refactorings (although note that simply renaming a binding will not affect the source file dependency graph).

Rotor then proceeds to compute the renaming using a worklist algorithm, shown in pseudocode in Fig. 2, with the worklist initially containing the input renaming. While the worklist is non-empty, Rotor takes a renaming from the list, and processes it as follows. It first calculates an (over)approximation of the footprint of the renaming, which consists of the source file(s) that define the binding to be renamed and all the source files that directly depend on them (i.e. refer to subcomponents of the modules defined by them). For each of these files, Rotor computes: i) the textual replacements to be applied to the file; and ii) the bindings declared in the file that need to be renamed as a consequence of the renaming currently being processed (i.e. the renaming

```
procedure WORKLIST(c, r)                          ▷ Codebase c, initial renaming r
    W ← {r}; D ← ∅; R ← ∅                         ▷ Worklist W, Dependencies D, Replacements R
    while W ≠ ∅ do
        r ← CHOOSE(W)
        W ← W \ {r}; D ← D ∪ {r}
        for all f ∈ FOOTPRINT(r, c) do            ▷ File f
            R ← R ∪ REPLACEMENTS(f, r)
            W ← W ∪ (DEPENDENCIES(f, r) \ D)
        end for
    end while
    return (R, D)
end procedure
```

Fig. 2. Rotor's main worklist algorithm.

dependencies). Each renaming dependency that has not already be processed is then added to the worklist.

Once all the renaming dependencies and textual replacements have been computed, Rotor then applies all of the replacement operations to the original source code and produces a diff. Rotor can also produce a log file containing all the renaming dependencies of the given input renaming. It is worth noting that the computation of the dependencies is *independent* of computing the refactoring itself (i.e. the patch to be applied); indeed, the former is a precursor to the latter, although the computations can be interleaved. Rotor can link each individual part of the computed patch to the dependency that produced it; thus it is able to 'explain' in detail to the user what it is doing.

5.1.2 Reuse of the Compiler Infrastructure. Since Rotor is written in OCaml, it is able to make direct use of the OCaml compiler infrastructure via the `compiler-libs` package. We leverage this capability, firstly by reusing the datatypes defined in the compiler that represent the abstract syntax trees (ASTs) of OCaml source files; and secondly, by delegating all parsing and type checking to the compiler itself. That is, we are able to use the interface exposed by `compiler-libs` to directly obtain untyped and typed AST from the source files provided by the user.

The OCaml compiler stores detailed information in the AST regarding the source file locations of each element. Thus an advantage of this approach is that it is very straightforward for Rotor to generate replacement operations as it analyses the ASTs. Rotor also makes heavy use of the functions available in the compiler to look up elements that are in scope at different points in the source code. This is particularly useful for performing renaming, since the compiler performs binding analysis as part of the type checking process. Thus, Rotor can obtain a unique ID number for the value to be renamed (as well as its parent modules), and then simply compare this to those of the identifiers that it finds as it traverses the AST. The OCaml compiler also provides functions for looking up various elements within module types, which is crucial to correctly and accurately generating renaming dependencies.

5.1.3 Managing the Codebase. In order to correctly apply refactorings across a whole codebase Rotor needs to be aware not only of the semantics of the core OCaml language, but also of how a program's codebase is demarcated, organised, and compiled into a working executable or library. Most immediately, Rotor must know where to find the source files over which it is to operate. Moreover, the program to be refactored may make use of external libraries that are not intended to fall within the scope of the refactoring process and for which, most likely, the source code is not

available. However, since Rotor makes use of the OCaml compiler, it must also know where to find these libraries so that it can make them available to the compiler for type checking.

Extended aspects of the OCaml language and a program's build environment also have an impact on the refactoring process. We describe two of these in particular, which have proved vital to consider in the course of developing Rotor.

PPX Preprocessors. OCaml has a preprocessor infrastructure called PPX [Leroy et al. 2017, Chap. 27], the use of which is common in real-world OCaml software development [Frisch 2014]. This means that in order to produce the ASTs that Rotor uses for analysis, it is necessary to know which PPX preprocessors are applied to each source file. Rotor currently sidesteps this issue by relying on the presence of `.cmt` and `.cmti` files. These are artefacts produced by the compiler,[3] which contain a binary representation of the typed AST for a given source file and from which the compiler (and thus also Rotor) can reconstruct ASTs. In the future, we plan to engineer Rotor so that it can determine which preprocessors to apply by reading build configuration files such as `ocamlbuild`'s `_tags` files, or `jbuilder`'s `jbuild` files.

Packaging Strategies. In large codebases, namespacing can become an important issue: it may be desirable to have multiple top-level modules with the same name, as long as they belong to different projects. Although not supported by the OCaml compiler directly, it possible to implement build processes that achieve this using the type level module aliases language extension (see [Leroy et al. 2017, §8.12]).

For example, to compile the modules contained in files `foo.ml`, `bar.ml`, and `baz.ml` into a package called `rotor`, an auxiliary parent file `rotor__.ml` is created containing aliased submodules:

```
module Foo = Rotor__Foo
module Bar = Rotor__Bar
module Baz = Rotor__Baz
```

This is then compiled into a module **`Rotor__`** with the `-no-alias-deps` flag, which instructs the compiler not to try and compile any dependencies—i.e. the modules **`Rotor__Foo`**, **`Rotor__Bar`**, and **`Rotor__Baz`**, which do not (yet) exist. Each of the original source files is then compiled using the following command, in this case for `foo.ml`:

```
ocamlc -no-alias-deps -open Rotor__ -o Rotor__Foo.cmo -c foo.ml
```

to produce the namespaced modules **`Rotor__Foo`**, etc. The result of using this compilation strategy, implemented by e.g. the `jbuilder` build system, is that in the ASTs Rotor will encounter identifiers such as **`Rotor__.Foo`**, **`Rotor__Bar`**, etc., which do not correspond to the names of source files created by the programmer. Instead, Rotor must know that it should associate these with files named `foo.ml`, `bar.ml`, etc., in a package called `rotor`. In short, the strategy implements a non-standard mapping between module names and source files.

Rotor deals with these two issues using a collection of modules for representing and managing information about a program's codebase and build environment.

– The **`Fileinfos`** module encapsulates the metadata associated with a single file in a program. This includes its location on disk, the name of any package or library that it belongs to, and which preprocessors should be applied.

– The **`Sourcefile`** module manages the production of the ASTs for a source file. Since parsing, type checking, or reading ASTs from a `.cmt` or `.cmti` file is a relatively time-consuming process, and not all files in a codebase will need to be analysed for a given refactoring, this is only done once on demand and the results cached in memory. The **`Sourcefile`** module is also responsible for serving the source code of the file from disk, and applying the results of refactorings to it.

[3] The OCaml compiler produces them as a by-product when invoked with the `-bin-annot` flag.

– The `Codebase` module acts as a set of `Fileinfos` values, and thus represents the codebase of a project as a whole. It is also responsible for computing and managing the file dependency graph of a project (cf. Section 5.1.1), as well as project-wide metadata such as external library dependencies.
– The `Buildenv` module holds knowledge about aspects of the build environment. In particular, it knows about packaging strategies and is responsible for providing appropriate mappings between identifiers encountered in the typed AST and source files in the codebase.

5.2 A General Framework for Refactorings

We have designed Rotor with extensibility in mind, in order to make it straightforward to integrate new refactorings as and when they are implemented. This is mediated via a number of generic interfaces and abstract datatypes, which implement the concepts discussed in Section 4. Concrete refactorings are expressed as first class modules. Specifically, we implement a form of the factory design pattern [Gamma et al. 1995], comprising the following elements.

Runtime Representation. The `Refactoring_repr` module provides an abstract datatype whose values are runtime representations of refactorings. The `Refactoring_repr.t` datatype is implemented as a variant type, currently providing representations of value binding renaming and the identity refactoring that leaves a program unchanged. Further constructors can easily be added to represent other refactorings.

Abstract Source Code Replacements. The basic model we implement is that of refactorings as sets of textual replacement operations on source code. Thus, a replacement can be characterised by a location range in a source file and a 'payload' string which replaces the existing code within that range. This abstraction is flexible since it can express both the insertion of new code (using a zero-length range) and the deletion of existing code (using a zero-length payload). The `Replacement` module implements an abstract datatype for the source code replacement operations, and has the basic signature shown in Listing 2.

```
type t (* The abstract type of replacements *)
type payload = string
module Set : Set.S with type elt = t
val mk : Location.t -> payload -> t
val apply : t -> Sourcefile.t -> string
val apply_all : Set.t -> Sourcefile.t -> string
```

Listing 2. The basic signature of the `Replacement` abstract datatype.

We use the compiler's `Location.t` type to represent location ranges in source code. The `apply` function takes a single replacement and a source file representation, and returns the result of applying the replacement to the source code in the file. The `apply_all` function applies a set of replacements to the contents of a source file.

An Interface for Refactorings. We define a common interface `Refactoring.S` for refactorings, shown in Listing 3, so that any module implementing this interface may be used by Rotor as a refactoring. A refactoring must be able to carry out four basic operations:

1) return a runtime representation of itself;
2) return the set of refactoring dependencies that are directly induced by the refactoring with respect to a given source file;
3) return the set of replacements that must be applied to the given source file in order to effect the refactoring; and

```
module Set : Set.S with type elt = Refactoring_repr.t

module type S =
sig
  val repr : Repr.t
  val get_deps : Sourcefile.t -> Set.t
  val process_file : Sourcefile.t -> Replacement.Set.t
  val kernel : Codebase.t -> Fileinfos.t list
end
```

Listing 3. The basic signature of the `Refactoring` module.

4) return a refactoring *kernel* for a given codebase—this is a subset of the codebase (i.e. source files) that contains the footprint of the refactoring, and is sufficient for computing all of the (direct) dependencies of the refactoring.

The notion of a kernel, discussed in Section 4, allows for efficiency: the footprint and dependencies of a refactoring can be computed by iterating over *all* files in the codebase and applying the `process_file` and `get_deps` functions, however this is expensive and in general it will only be necessary to iterate over a subset of the files.

A Factory Module. The final element of our extensible architecture is the `Refactoring_lib` module, which contains a function

```
val of_repr : Refactoring_repr.t -> (module Refactoring.S)
```

mapping runtime representations of refactorings to (first class) module values, which can then be unwrapped and used to compute and apply the results of the appropriate refactoring. Having a factory module to perform this mapping means that adding new refactorings to Rotor requires only that we modify: a) the datatype for runtime representations to capture the new refactoring; and b) the factory module to be aware of a new mapping.

5.3 Identifying Program Elements

OCaml programs have a hierarchical structure, in which both module bindings and module type declarations can be nested within one another and can also contain value bindings or declarations.[4] OCaml uses 'dot notation' for identifiers, which uses the infix operator dot (' . ') to indicate this hierarchical nesting. Take, for example, the `Set` module from OCaml's standard library. This contains a module type `S`, which is the signature of modules created using its `Make` functor. In particular, this signature declares a function named `add`, which could be referred to using the identifier `Set.S.add`. Although it captures the hierarchical structure, the identifier does not specify the *sort* of element at each level.

Whilst sufficient for the purposes of the OCaml compiler—which can disambiguate the information it requires based upon the context in which an identifier is used—Rotor's use of identifiers imposes further requirements. We describe two in particular.

– Rotor needs to be able to reference more intricate program structures. Returning to the example of the OCaml standard library's `Set` module, there is no lookup function[5] in the

[4] In OCaml's terminology, named subcomponents of modules which specify implementations are called 'bindings', whereas named module types and their subcomponents are 'declarations'.

[5] Here, we are referring to the functions in the `Env` module that look up elements in scope at some particular point in the AST; these functions require the caller to know the sort of element to be found, e.g. `Env.lookup_module`, `Env.lookup_modtype`, `Env.lookup_value`, etc.

compiler that will interpret the identifier `Set.S.add` as referring to a declaration of the value `add` in the **`S`** module type within **`Set`**. Instead, one has to first look up `Set.S` based on the knowledge that it refers to a module type; only then can the declaration of `add` be found by looking up a value binding specifically.
- Rotor is not always able to rely on context to give meaning to identifiers. For example, OCaml allows a module and a module type to be bound using the same name in a given scope. We have encountered cases in our test-bed where this occurs, and where it is also crucial that Rotor can identify one rather than other in order to perform a renaming.

Furthermore, Rotor needs to be able to distinguish between structures and functors, between their associated structure and functor module types, and identify functor parameters and arguments.

5.3.1 A Syntax for Rich Identifiers. OCaml interprets the dot '.' in identifiers as a (left associative) infix operator. Rotor enhances OCaml's notion of identifiers by generalising the dot syntax in two ways. Firstly, instead of treating the dot as an infix operator, it uses it as a *prefix* operator on names to indicate an element of a particular sort and introduces new prefix operators to express other sorts (module, module type, value, etc.). Rotor currently uses the operators '.', '#', '%', '*', and ':' to indicate structures, functors, structure types (i.e. signatures), functor types, and values, respectively. Notice that this scheme is extensible: we can easily add representations for other sorts of OCaml language element (e.g. value types, exceptions, classes, objects, etc.) by adding new prefix operators. We also introduce an indexer element of the form `[`i`]` to our syntax for identifiers, whose intended meaning is to stand for the i^{th} parameter of its parent element. We use numerical indices for functor parameters since functor parameter names are not immediately available at application sites, and parameters are permitted to be named differently in module type annotations. Secondly, the hierarchical structure is now represented by the sequencing of prefixed names.

The syntax of Rotor's identifiers is given by the following grammar.

$$\langle \textit{signifier} \rangle ::= \text{`.'} \mid \text{`\#'} \mid \text{`\%'} \mid \text{`*'} \mid \text{`:'}$$

$$\langle \textit{id\_link} \rangle ::= \langle \textit{signifier} \rangle \, \langle \textit{name} \rangle \mid \text{`[`} \, \langle \textit{index} \rangle \, \text{`]'}$$

$$\langle \textit{identifier} \rangle ::= \langle \textit{id\_link} \rangle \mid \langle \textit{id\_link} \rangle \, \langle \textit{identifier} \rangle$$

Where the nonterminal $\langle \textit{name} \rangle$ denotes a standard OCaml (short) identifier, and $\langle \textit{number} \rangle$ denotes a non-negative integer literal. We illustrate this syntax on the examples from the standard library considered above:

- `.Set%S` refers to the **`S`** module type within the **`Set`** module;
- `.Set%S:add` refers to the `add` value declaration within the `.Set%S` module type;
- `.Set#Make` refers to the **`Make`** functor within the **`Set`** module;
- `.Set#Make[1]:compare` refers to the declaration of the `compare` value in the (module) type of the `.Set#Make` functor's first parameter.

Rotor overloads this last example in order to refer to the `compare` binding in the first *argument* to an application of the `.Set#Make` functor (these two uses *can* always be disambiguated by Rotor from the AST context). When the immediate child of a functor (type) element is not a parameter, as in `.Set#Make:add`, Rotor interprets the remainder of the identifier (in this case the `add` value binding) to refer to an element within the *body* of the functor.

5.3.2 A GADT Representation. In building a data structure to represent these identifiers, Rotor has two main constraints that derive from how they are used to guide the analysis of code to be refactored. One is that Rotor performs *top-down* traversals of the AST, and thus when using

(a) Rotor's representation of `.Set%S:add` (b) OCaml's representation of `Set.S.add`

Fig. 3. Representations of hierarchical identifiers in the OCaml compiler and Rotor.

identifiers to guide these traversals (e.g. to find a local binding), Rotor deconstructs them left-to-right. This means that Rotor parses sequencing of sort-signified names as right associative. Thus, for example, Rotor represents the identifier `.Set%S:add` as shown in Fig. 3a. This directly contrasts with the implementation of identifiers in the OCaml compiler, in which the top most parent module is deepest in the data structure. This is illustrated in Fig. 3b, which shows OCaml's representation of the identifier `Set.S.add` as a `Longident.t`.

Although Rotor requires a top-down representation for identifiers, it nonetheless still requires access to the *sort* of the final element in the chain. To achieve this we use generalised abstract datatypes (GADTs), which allow fine-grained control over type parameters in the construction of data values. We implement a datatype `'a Identifer.t` in which the type parameter `'a` specifies the sort of the final subcomponent. This comprises the following.

Phantom types for language elements. We define a number of 'phantom' types to refer to elements of the OCaml language.

```
type _value          = Value          type _parameter      = Parameter
type _structure      = Structure      type _functor        = Functor
type _structure_type = StructureType  type _functor_type   = FunctorType
```

These types are only for use in type checking: we do not create any values of these types at runtime. The reason that we do not leave them completely abstract (i.e. without a definition) is that OCaml requires a concrete definition in order to be able to statically exclude unreachable match cases.

Sorted Atoms. We define a GADT of *atomic* elements whose type parameter is constrained to be one of the language element phantom types.

```
type 'sort atom =
  | Value         : string -> _value          atom
  | Parameter     : int    -> _parameter      atom
  | Structure     : string -> _structure      atom
  | StructureType : string -> _structure_type atom
  | Functor       : string -> _functor        atom
  | FunctorType   : string -> _functor_type   atom
```

The Identifier type. We define a GADT for hierarchical identifiers within a module `Identifer` as follows.

```
type 'sort t =
  | Atomic          : 'a atom          -> 'a t
  | InParameter     : int    * 'a t -> 'a t
  | InStructure     : string * 'a t -> 'a t
  | InStructureType : string * 'a t -> 'a t
  | InFunctor       : string * 'a t -> 'a t
  | InFunctorType   : string * 'a t -> 'a t
```

Although this datatype needn't be defined as a GADT, defining it as such means that in the future it could easily be extended to represent elements whose subcomponents should be constrained to particular language elements, e.g. concrete functor applications:

```
type 'sort t = ...
  | InFunctorApplication : (_functor t * _structure t) * 'a t -> 'a t
```

Moreover, as Rotor is extended to represent other language elements it will become necessary to enforce further constraints on the structure of identifiers (e.g. a class declaration cannot have a module as a subcomponent). Whilst this could be done dynamically at runtime, GADTs allow such constraints to be encoded statically and used to reduce code clutter, since the compiler can determine that certain match cases are not possible. For example, we specify that the identifier for a value binding renaming should be of type `_value Identifier.t`, and so when matching the atomic case we only need to specify a case for the constructor pattern `Atomic (Value v)`. This is one advantage of Rotor's representation for identifiers.

Another advantage of this representation is that we can easily make the sort of an identifier available to Rotor. This contributes to the extensibility of Rotor, since it makes it easier for the factory module to dispatch control to the correct refactoring implementation. For this, we introduce a runtime representation for OCaml language elements, and an additional type for identifiers that existentially quantifies over the sort:

```
type 'sort elem =
  | Value     : _value        elem  | Parameter     : _parameter     elem
  | Structure : _structure    elem  | StructureType : _structure_type elem
  | Functor   : _functor_type elem  | FunctorType   : _functor_type   elem

type _t = Ex : 'a elem * 'a t -> _t
```

The `Identifier` module provides a parsing function for identifiers.

```
val parse : string -> _t
```

This allows for the sort of the result to be pattern-matched, and the identifier it contains to then be passed to code requiring a specific sort.

5.3.3 Grouping Compiler Functions with Ad-hoc Polymorphism. Rotor's use of GADTs to represent language elements allows it to implement a form of ad-hoc polymorphism, grouping families of similar compiler functions that operate over different parts of the AST structures. For example, the compiler contains a number of lookup functions for different elements that return a resolved path to the element and appropriate information on its type. We define a GADT, that we call a 'view', to group together this type information that is represented by different datatypes within the compiler's `Types` module:

```
type 'sort type_view =
  | Value         : Types.value_description   -> _value          type_view
  | Structure     : Types.module_declaration  -> _structure      type_view
  | Functor       : Types.module_declaration  -> _functor        type_view
  | StructureType : Types.modtype_declaration -> _structure_type type_view
  | FunctorType   : Types.modtype_declaration -> _functor_type   type_view
```

We can then define a lookup function, shown in Listing 4, that groups together the compiler's lookup functions for the different sorts of language element. This results in much more uniform implementations for refactorings in Rotor.

```
let lookup : Env.t -> 'a atom -> Path.t * 'a type_view =
  fun env (type a) atm ->
    match atm with
    | Value v ->
      let p, vd = Env.lookup_value (Longident.Lident v) env in
      p, Value vd
    | Parameter _ ->
      invalid_arg "Doesn't correspond to a lookup function"
    | Structure s ->
      let p, md = Env.lookup_module (Longident.Lident s) env in
      p, Structure md
    | StructureType st ->
      let p, mtd = Env.lookup_modtype (Longident.Lident v) env in
      p, Structure mtd
    | Functor f -> ... | FunctorType ft -> ...
```

Listing 4. A lookup function that groups the different compiler lookup functions.

```
class virtual ['self] iter : object ('self)
  method visit_Add       : 'monomorphic. 'env -> expr -> expr -> unit
  method visit_Const     : 'monomorphic. 'env -> int -> unit
  method visit_expr_desc : 'monomorphic. 'env -> expr_desc -> unit
  method visit_expr      : 'monomorphic. 'env -> expr -> unit
end
```

Listing 5. The class type of an `iter` visitor for a simple expression datatype.

5.4 Use of Visitor Classes

The core of Rotor's operation involves performing traversals of various types over the program's abstract syntax trees. For this, we have made use of the recently developed `visitors` syntax extension for OCaml [Pottier 2017]. This automatically generates classes whose methods perform a bottom-up traversal of values of a given set of datatypes. By default, these visitors do not perform any operation in particular, beyond the basic traversal. However by overriding particular methods complex computations can be carried out on these data values. This extension thus provides a basis in OCaml for similar capabilities to those found in Haskell's SYB generic programming library [Lämmel and Peyton Jones 2003], the Strafunski framework [Lämmel and Visser 2003], and the Stratego/XT language [Bravenboer et al. 2008].

The `visitors` library plugs into the `ppx_deriving` PPX framework to provide a preprocessor that inserts default visitor classes for datatypes to which a `[@@deriving visitors ... ]` annotation is attached. For example, consider the following pair of mutually defined datatypes for a simple tagged expression language (cf. [Pottier 2017, §2.3]):

```
type 'a expr_desc = Const of int | Add of 'a expr * 'a expr
and 'a expr = { tag : 'a; desc : 'a expr_desc; }
[@@deriving visitors { variety = "iter" } ]
```

Triggered by the `@@deriving` annotation, the `visitors` PPX preprocessor inserts a class with the type shown in Listing 5. This class produces *iterator* visitors: objects which traverse `expr` and

```
let count_const = object (self : 'self)
  inherit ['self] iter as super
  method! visit_expr ht ({ tag; desc } as e) =
    match desc with
    | Add _   -> super#visit_expr ht e  (* delegate to default traversal *)
    | Const i -> match Hashtbl.find_opt ht tag with
                 | None   -> Hashtbl.replace ht tag i
                 | Some j -> Hashtbl.replace ht tag (i+j) end ;;
let htbl = Hashtbl.create 0 ;;
let e = { tag = "top"; desc = Add (
            { tag = "left";  desc = Add (
              { tag = "left" ; desc = Const 1 },
              { tag = "right"; desc = Const 2 }) },
            { tag = "right"; desc = Const 3; }) } ;;
let () = count_const#visit_expr htbl e in htbl ;;
>> - : (string, int) Hashtbl.t = [ "left" ->  1; "right" -> 5 ]
```

Listing 6. An iterator visitor for the expr and expr_desc datatypes.

expr_desc values, but do nothing else by default. The class has methods for each datatype and each variant constructor with method names built from the names used in the type definition. Each method also takes an 'environment' argument, which can be used to accumulate and pass intermediate data around during the traversal. The `'monomorphic` annotations on the method types indicate that the type of this environment argument is quantified (and inferred) at the level of the whole class, rather than each individual method. By defining classes or objects that override these methods, we may implement bespoke traversals. We could, for example, create an object that traverses an `expr` value and fills a hash table with bindings that return the sum of all the **`Const`** sub-expressions tagged with a given annotation, as shown in Listing 6. The `count_const` object inherits from the automatically generated `iter` visitor class, and overrides the `visit_expr` method. If the expression is a **`Const`**, then it adds the constant value to the accumulated sum of the expression's tag; otherwise it delegates to the default implementation in order to traverse deeper into the data value. Notice how the hash table is passed around as the environment.

The `visitors` library is able to create other kinds of visitor class. In particular, it can also create *reducers* with the annotation `[@@deriving visitors { variety = "reduce" }]`. A reducer class is parameterised by a result type `'res` and contains two further methods, which must be implemented by concrete reducer objects to define a monoid at the `'res` type.

```
method private virtual zero : 'monomorphic . 'res
method private virtual plus : 'monomorphic . 'res -> 'res -> 'res
```

The visitor methods of reducers return a value of type `'res`, with the default implementation returning `zero` at the leaves, and combining the results of subcomponents using `plus`. Thus they 'reduce' a complex data value to a single result of type `'res`. Although the `visitors` extension provides a wide range of visitor classes (including, additionally, the `map`, `mapreduce`, and `fold` visitors), Rotor currently only makes use of the `reduce` variety as we describe below.

5.4.1 Visitors for the OCaml ASTs. The OCaml compiler defines datatypes for both an untyped and a typed AST, contained in modules **`Parsetree`** and **`Typedtree`** respectively. These AST types

```
type tt_structure = Typedtree.structure = {
  str_items     : tt_structure_item list;
  str_type      : (Types_visitors.ty_signature [@opaque]);
  str_final_env : Env_visitors.env_t;
}
and tt_structure_structure_item = Typedtree.structure_item = {
  str_desc : tt_structure_item_desc;
  str_loc  : Location_visitors.location_t;
  str_env  : Env_visitors.env_t
}
and tt_structure_item_descr = Typedtree.structure_item_desc = ...
[@@ deriving visitors { variety = "iter";
        ancestors = [ "Env_visitors.iter"; "Location_visitors.iter";
          "Types_visitors.iter"; ... ] },
      visitors = { variety = "reduce"
        ancestors = [ "Env_visitors.reduce"; "Location_visitors.reduce";
          "Types_visitors.reduce"; ... ] }, ... ]
```

Listing 7. A portion of Rotor's **Typedtree_visitors** module illustrating generation of visitor classes.

refer to other types from a number of different modules, namely the **Asttypes**, **Env**, **Ident**, **Lexing**, **Location**, **Longident**, **Path**, **Primitive**, and **Types** modules.

We use the `visitors` PPX to generate visitor classes for them as follows. We create modules in Rotor named **Parsetree_visitors**, **Typedtree_visitors**, etc., corresponding to each compiler module containing types for the ASTs. In these modules we place copies of the type definitions from the corresponding compiler module, but with a number of crucial modifications.

i) We declare an explicit equality with the corresponding type in the compiler. This allows the visitor classes that are generated to operate over ASTs obtained directly from the compiler.
ii) Since many datatypes in the various modules share the same names, we rename the types in Rotor with prefixes, e.g. with `tt_` in **Typedtree_visitors**, `ty_` in **Types_visitors**, etc. This is to avoid the `visitors` PPX generating more than one method with the same name but with differing signatures, which would cause the generated classes to be ill-formed.
iii) We have to attach `[@opaque]` annotations to subcomponents with types belonging to the **Types** module; this is to prevent the default implementations from actually traversing these types. We do this because the datatypes in this module, which represent OCaml types, can define cyclic structures and thus the default visitor implementations may result in non-terminating traversals. The possibility remains for the developer to override these methods, if they can ensure the resulting traversals will not loop.

We then attach `[@@deriving visitors ... ]` annotations to the datatypes to obtain automatically generated visitor classes for them. We also make use of the `ancestors` parameter to specify the appropriate parent visitor classes that they should inherit from in order to traverse the externally defined types they refer to. Listing 7 shows part of the definition of Rotor's **Typedtree_visitors** module, illustrating these points.

5.4.2 Visitors for Producing Refactorings and Dependencies. We noted that the most useful kind of visitor classes for Rotor are of the `reduce` variety. This is because they allow Rotor to 'reduce' an AST to a single value. Rotor uses these kinds of visitors to analyse ASTs and produce *sets* of

```
class ['self] replacement_reducer =
  object (self : 'self)
    inherit ['self]
      Typedtree_visitors.reduce
    method private zero =
      Replacement.Set.empty
    method private plus rs rs' =
      Replacement.Set.union rs rs'
  end
```

(a) Rotor's reducer for replacement operations.

```
class ['self] dependency_reducer =
  object (self : 'self)
    inherit ['self]
      Typedtree_visitors.reduce
    method private zero =
      Refactoring.Set.empty
    method private plus ds ds' =
      Refactoring.Set.union ds ds'
  end
```

(b) Rotor's reducer for depedencies.

Fig. 4. Reducers in Rotor's library of derived visitors.

replacement operations and refactoring dependencies, which are the two main kinds of results that Rotor's refactorings needs to produce (see Section 5.2). Rotor contains a library of derived visitor classes for concrete refactorings to make use of, including those shown in Fig. 4.

5.5 Generation of Replacements and Dependencies

In this section we detail how Rotor computes a set of textual replacements, i.e. the action of the refactoring on a given codebase, and the dependencies for each renaming. The basic approach is to define visitors for each task that inherit from the basic reducers shown in Fig. 4. The methods of these visitors are overridden to generate replacement operations or representations of renamings (i.e. dependencies) when certain patterns or conditions are encountered within the AST. In actuality, a pair of reducers is defined for each task. This is because slightly different behaviour is required depending on whether the binding to be renamed is *defined* within the AST being traversed, or whether the AST simply *uses* identifiers that reference it.

'Definition' Visitors. These reducers search for the particular subcomponent of the AST that defines the binding to be renamed, and so they constrain the traversal according to the identifier of the binding being searched for. That is, they deconstruct the identifier one element at a time, and at each stage find the corresponding subcomponent of the current AST node (i.e. module, module type, functor parameter, or value binding). They then traverse the subcomponent according to the remainder of the identifier. Because the identifier will stipulate that a module should be either a structure or a functor (similarly with module types), the visitor must check that the module and module type subcomponents it finds are of the correct sort. It does this by examining the type inferred for the subcomponent, which is available in the typed AST.

At each stage of this traversal, the 'definition' visitor must also delegate to the 'use' visitor to process the AST that constitutes the rest of the source file, possibly also including the AST for the subcomponent in which the binding is defined, in the case it is *recursive*.[6] This is because in the remainder of the file, after its definition, (and within recursive definitions themselves) the binding may be referenced. It is the 'use' visitor that knows how to process these references, whether they are referencing a binding defined in the same source file or externally to it.

The 'definition' visitor for dependencies simply traverses the AST delegating to the 'use' visitor at each level. The 'definition' visitor for generating replacements also computes the replacement operation for renaming the local binding when it finds it. Moreover, it performs a *soundness* check

[6] OCaml allows both recursive module and value definitions, signalled explicitly with the `rec` keyword.

to make sure that there is no existing binding of the target name. If this is the case, the resulting refactoring would change the behaviour of the program, and Rotor throws an exception. That the renaming does not shadow an existing binding constitutes a precondition for the refactoring, which is ordinarily checked before the refactoring is carried out. Our work in developing Rotor suggests that the most convenient way to check if such preconditions hold is simply to detect violations as the refactoring is computed. Thus, we do not mean here that Rotor fails in performing the refactoring; rather the exception is used to provide feedback that a precondition does not hold.

'Use' Visitors. These visitors traverse the whole AST looking for references to the binding being renamed. Before beginning the traversal, Rotor calls a function in the compiler to look up the top-most element in the identifier, which returns a unique ID number for it. Then every time an identifier is encountered in the AST, after first calling a compiler normalization function to ensure it is in a canonical form, the visitor can check if it corresponds to the binding to be renamed using this unique ID. On encountering an `include` of the value binding, the 'use' visitor for replacements also performs a soundness check, as described above for the 'definition' visitor.

For computing dependencies, the 'use' visitor maintains a record of the scope it encounters as it traverses the AST. Rotor manages this scope using an abstract datatype defined in the `Modulescope` module (cf. Fig. 1). Every time it enters into the body of a module (type) binding, or the body and parameters of a functor, it registers a new frame in the scope. This allows the visitor to generate the identifier of bindings to be renamed in dependencies. For example, suppose the visitor is searching for occurrences of the identifier `Foo.Bar.baz` and encounters the module alias `module M = Foo` in an AST. It should then generate a dependency on renaming ⟨*parent_id*⟩`.M.Bar.baz`, where the identifier ⟨*parent_id*⟩ corresponds to the scope in which the module alias occurs. A similar situation arises with an `include` of a module (type).

In certain situations, the 'use' visitor must also delegate back to the 'definition' visitor. For example when renaming a binding within a functor parameter, identified by `#Foo[1]:bar` (cf. Section 5.3), the visitor may encounter an application of the functor, e.g., `Foo (struct ... end)`. Here a renaming of the binding `bar` must take place inside the argument to this application, i.e. within the module expression `struct ... end`. The binding is expected to be *defined* within the argument, therefore it is the 'definition' visitor that should process this subcomponent.

6 CASE STUDY

We tested our prototype refactoring tool using a real-world test-bed consisting of a large number of the Jane Street [2018] public libraries. Namely, we took a snapshot of the `core` library and its dependencies. This includes the libraries `core_kernel`, `base`, `stdio`, `fieldslib`, `sexplib`, and a large number of libraries for PPX preprocessing and unit testing, which are used heavily throughout the test-bed. Altogether, the snapshot comprises 77 libraries containing together 869 source files.

These libraries form the core of Jane Street's codebase, which underpins their business activities and runs into millions of lines of code. It is the result of many years of continuous development, and therefore represents both a very robust and broad range of usage of the OCaml language. This makes it ideal as a test-bed for our prototype.

6.1 Harvesting Test Cases

To obtain a suite of test cases for renaming, we harvested the collection of identifiers that is used in the test-bed. We did this by creating a visitor to traverse the ASTs of the source files and collect all the value identifiers that it encountered. We placed two filters on the identifiers that are collected.

Rebuild Failed	Rebuild Succeeded
2336	733
(76.1%)	(23.9%)

(a) Success/failure rate of re-compilation.

	Files	Hunks	Avg. Hunks/File	Depends
Max	35	91	5.7	92
Mean	4.1	6.1	1.2	14.0
Mode	3	3	1	1

(b) Statistics (success cases).

	Files	Hunks	Avg. Hunks/File	Depends
Max	66	304	7.7	864
Mean	5.1	7.9	1.3	48.6
Mode	2	2	1	1

(c) Statistics (failure cases).

Fig. 5. Results of running Rotor over the renaming test suite.

(1) We did not collect any identifiers whose source location has a 'ghost' flag set. This flag indicates that the AST node to which it is attached is not part of the programmer's original source code, but was instead automatically generated either by the compiler or by a preprocessor.

(2) We only collected identifiers that the compiler considers 'persistent'—that is, referring to bindings defined outside the current source file. This is because we needed to harvest identifiers that address bindings from the top level of the program. For non-persistent identifiers we would have had to compute the local source of the binding being referred to, as well as its scope, in order to produce such an address. Moreover, the binding referred to might be defined in an *anonymous* module,[7] and therefore not be addressable at all.

In total, we harvested 5051 of these persistent, non-ghost identifiers. Of these, 963 referenced bindings outside of the test-bed (e.g. in OCaml's standard libraries), and so we discarded them as test cases. A further 1019 referred to infix operators. We also discarded these cases since renaming these operators would require more complex, non-local changes. Specifically we would need to change the structure of the applicative expressions in which the operator is used, in order to move the renamed identifier into the head position of the application. Rotor does not yet support this, although doing so is future work. This left a total of 3069 bindings that formed the cases of our renaming test suite.

6.2 Results and Analysis

For each identifier in our test suite, we ran Rotor to compute the result of renaming the identifier using a fresh name not occurring in the codebase.[8] We then applied the resulting patch to a copy of the test-bed source code and attempted to recompile it. Admittedly this is a rather crude test for success, since it is possible to obtain false positives: cases in which the renaming is computed incorrectly but still compiles, thus producing a valid program but with different behaviour. Nevertheless, it is a good first approximation for a test suite as large as ours.

Our results are shown in Fig. 5. They show that recompilation succeeded for around 24% of our test cases (733 out of 3069). Theoretically we might expect all test cases to succeed, since we chose a fresh identifier for the renaming. One reason that this is not the case is that the renaming might depend on modifying a binding *outside* the codebase, e.g. in the standard library. In this case,

[7] An anonymous module is one that is not bound to a name—this can happen when a complex module expression is used in an `include` statement or as the argument to a functor application.

[8] Imaginatively, we chose the new name to be 'foo'.

the refactoring is actually outside the scope of Rotor's ability; the solution here is for Rotor to signal this to the user. This case might actually be relatively common in our test-bed, since it was developed as an overlay to the standard library. Still, even discounting such cases the failure rate of our test suite seems high. There are a number of other reasons why this might be the case.

- The tool itself does not catch all changes that must be made—Rotor is a prototype, and still requires further development. We have already mentioned in Section 2 that it does not handle module type extraction and first class modules; other corner cases are also problematic, e.g. anonymous modules.
- The refactoring might attempt to rename the identifier in a binding that is automatically generated; even though the source code for the preprocessors is included within the test-bed this will fail since correctly performing the renaming will require modifying the behaviour of the preprocessor itself (i.e. implementing a different name generation scheme).
- We have found that the preprocessors do not always correctly set the 'ghost' flag in the ASTs they generate. This has caused some interesting errors: the preprocessors often set the location of automatically generated identifiers to be that of the original program element that triggered the preprocessor (e.g. a type definition, for which the preprocessor generates a comparison function); the result is that Rotor renames the wrong element (e.g. a type).

In the second case the renaming is clearly out of Rotor's scope of operation, and it is not clear how Rotor could determine that this is the case. In the third one, the failure is not Rotor's 'fault' but it at least seems feasible that Rotor could detect a potential problem by, e.g., checking that there is not more than one non-ghost AST node corresponding to the same source location. These problems demonstrate the sort of complex, pragmatic challenges that arise in applying Rotor to real-world code.

Figures 5b and 5c show statistics that we collected for a number of metrics in both the succeeding and failing test cases. Namely, the number of dependencies for each renaming, the number files in the renaming footprint, the number of `diff` hunks in the resulting patch, and the average number of such hunks per modified file. We show the maximum, mean and mode number in each case. For most of the test cases, the number of files in the footprint is small, and there is only a single change per file. In the successful test cases, the average number of dependencies for a refactoring was 14, but a maximum of 92 was encountered. This gives some indication as to the degree of the complexity and coupling that is present in the test-bed.

The maximum number of dependencies generated for a failing test case was an order or magnitude greater than for succeeding test cases, and the average number was 4.5 times as great. Moreover, although most commonly the computed footprint of a renaming was similar in both successful and failing test cases, the average size of the footprint was slightly higher in the failing cases. Thus, whilst we should not really conclude anything about the codebase or the renamings themselves from the statistics of the failing test cases, they do appear to reflect the common sense expectation that the more complex a renaming is, the 'harder' it is to get right.

7 RELATED WORK

A general survey of refactoring up to 2004 is provided by Mens and Tourwé [2004], while Thompson and Li [2013] specifically look at refactoring for functional languages such as Haskell and Erlang. We mention other related work pertaining to more specific aspects of refactoring.

7.1 Preconditions and dependencies

From the beginning [Griswold and Opdyke 2015] researchers have known that it is not always possible to perform a refactoring: there might be a set of preconditions, expressed as a set of logical

properties, that have to hold before a refactoring can be performed successfully, that is without changing the meaning of a program.

In some approaches—such as early versions of Eclipse—the preconditions are implicit. In the simplest case, a regression test suite can be applied[9] and if any test fails then the refactoring is rejected. This is obviously a crude approximation to correctness, and will permit potentially different programs to be accepted. A stronger test can be provided in some cases; for example, after a renaming the binding structure of the refactored and original programs are compared, and unless they are the same then the refactoring is rejected, with the attendant possibility of generating a counterexample to witness how precisely it fails.

In other cases, as initially advocated by Opdyke [1992], the preconditions are tested statically, that is without performing the refactoring. In practice the static preconditions will need to be decidable, and that will generally require an approximation of the true conditions, thus potentially excluding some valid refactorings.

How should the preconditions be made to hold? One design choice is to leave that to the person refactoring the code, but an alternative is for the tool itself to compensate automatically and enable the refactoring to go ahead; this is discussed in more detail by Thompson and Li [2013]. Of course, in many cases there may well be more than one incompatible compensation, and so, in general, we take the option of requiring manual preparation of the code to meet the precondition. Our approach, in which we make explicit the dependencies of a particular refactoring, provides to the person effecting the refactoring more detailed information, and at a higher level, than previous approaches.

7.2 Implementing refactorings

A refactoring is a transformation of the representation of a source code program to generate another source code program, governed by some conditions on its application. The fact that a readable and recognisable program needs to be produced as output makes for some engineering difficulties, but the key to ensuring that refactorings work as appropriate lies in the representation, which is often modelled on the internal representations of programs within compilers. Tools typically use two kinds of representation, as noted by Mens and Tourwé [2004]: an abstract syntax tree, typically enhanced or annotated to include information about static semantics, types etc., and a graph structure of some kind. The implementation of a refactoring will need to traverse these structures: tree-structured systems will typically use something akin to strategic programming [Bravenboer et al. 2008] that explicitly controls the application of transformations across the nodes of a tree, while graph transformation systems [Heckel 1995] are used to describe refactorings over graphical representations.

7.3 Queries and scripts

In evaluating preconditions it can be useful to represent information about a program in a database, and to extract information using a query language. While SQL and Prolog can be used, the most successful work here, resulting in the codeQuest tool [Hajiyev et al. 2005], uses Datalog to extract information from a relational database. This approach underlies JunGL [Verbaere et al. 2006], which is a high-level language for describing Java refactorings; the language combines a functional approach to transformations with the database of basic program facts. The paper gives particular attention to the problems of renaming, and in doing this shows the queries required to describe suitable preconditions for renaming. A higher-level approach to scripting refactorings is exemplified by the DSL provided by Wrangler [Li and Thompson 2012]: this allows the construction

[9] Assuming that the tests themselves don't need to be refactored!

of composite refactorings built by putting together 'atomic' components, and the tool reports the atomic refactorings performed by the application of a DSL script to a particular program.

7.4 Presenting refactorings

We argue in Section 4 that our dependencies represent a new way of presenting the effect of a refactoring. The usual approach is to provide some sort of visualisation of the effect of a refactoring, typically by means of a visual `diff` of the modules that have changed, as might be provided by an online repository like `github`. A novel form of this, which makes optimal use of screen space, is to use a pixel visualisation of a set of files, using one pixel per character in the files [Eick et al. 1992]; other techniques for displaying software changes are surveyed by Eick et al. [2002]. This issue is relevant to our work because, as Kim et al. [2014] acknowledge, refactorings cause particular problems for the process of code review; recent work on semantic code review by Menarini et al. [2017] may offer some help here.

8 CONCLUSIONS AND FUTURE WORK

We have presented Rotor, a prototype refactoring tool for OCaml, implemented in OCaml itself, that can carry out value binding renamings over large codebases. Furthermore, Rotor provides a generic framework in which further refactorings for OCaml can be implemented and integrated with one another. The design of Rotor is informed by an analysis of how the features of the OCaml language create binding structure and induce dependencies between different program subcomponents. This analysis has in turn led to the formulation of a novel, abstract notion of 'dependency' for refactoring. The complexities that we have identified in implementing renaming highlight the expressiveness of the OCaml language. We have also evaluated Rotor on an extensive test-bed consisting of real-world OCaml code.

There are many directions for future research. Firstly, Rotor should be extended to handle renaming in the presence of language features such as anonymous modules, first class modules, and module type extraction. Further development to better handle the difficulties presented by PPX preprocessors should also be undertaken. Beyond this, more refactorings for OCaml need to be implemented, including: renaming of other language elements (e.g. modules, module types, types, type constructors, classes, named function arguments, function parameters, etc.); structural refactorings such as adding and removing function arguments, type constructors, or class/object methods; and more complex refactorings such as module promotion. To increase the usability of Rotor, we could also support a scripting language for refactorings, as done for the Wrangler tool [Li and Thompson 2012], and integrate it with IDEs in a similar way to the `merlin` tool.

On the theoretical side, we would like to undertake a more formal investigation of refactoring dependencies and also investigate how the correctness of refactorings might be formally verified. This could take advantage of the CakeML project [Kumar et al. 2014], which is a formally verified implementation of a large subset of Standard ML in the HOL4 [2017] theorem prover.

ACKNOWLEDGMENTS

This work was supported by the EPSRC under Grant No. EP/N028759/1.

We would like to thank Jane Street Capital for supporting an internship and access to their codebase as part of a project collaboration. We especially extend thanks to Jérémie Dimino, Leo White, Mark Shinwell, and Thomas Refis of the tools and compilers team for providing their time and expertise, which was instrumental in the early stages of developing our prototype.

We would also like to thank Frédéric Bour, who provided insight into the implementation of the `merlin` tool, and François Pottier, who incorporated a number of features into the `visitors` library that were initially developed as part of this project to produce visitors for the OCaml ASTs.

REFERENCES

Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent Programming in ERLANG* (2nd ed.). Prentice Hall International (UK) Ltd., Hertfordshire, UK.

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Sci. Comput. Program.* 72 (2008), 52–70. Issue 1–2. https://doi.org/10.1016/j.scico.2007.11.003

S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster. 2002. Visualizing Software Changes. *IEEE Trans. Softw. Eng.* 28, 4 (Apr 2002), 396–412. https://doi.org/10.1109/TSE.2002.995435

Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner Jr. 1992. Seesoft – A Tool For Visualizing Line Oriented Software Statistics. *IEEE Trans. Software Eng.* 18, 11 (1992), 957–968. https://doi.org/10.1109/32.177365

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Alain Frisch. 2014. ppx and extension points. (2014). lexifi.com/blog/ppx-and-extension-points Blog Post.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

github [n. d.]. Online host of `git` repositories. ([n. d.]). https://github.com

William G. Griswold and William F. Opdyke. 2015. The Birth of Refactoring: A Retrospective on the Nature of High-Impact Software Engineering Research. *IEEE Software* 32, 6 (2015), 30–38. https://doi.org/10.1109/MS.2015.107

Elnar Hajiyev, Mathieu Verbaere, Oege de Moor, and Kris De Volder. 2005. CodeQuest: Querying Source Code with Datalog. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA.* ACM, New York, NY, USA, 102–103. https://doi.org/10.1145/1094855.1094884

R. Heckel. 1995. *Algebraic Graph Transformations with Application Conditions.* Master's thesis. TU Berlin.

HOL4 2017. Interactive Theorem Prover. (2017). hol-theorem-prover.org

Jane Street. 2018. Standard Library Overlay. (2018). github.com/janestreet/core

jbuilder [n. d.]. A Compsable Build System, version 1.0 (beta). ([n. d.]). github.com/ocaml/dune v1.0+beta19 released 14 March 2018.

Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An Empirical Study of Refactoring Challenges and Benefits at Microsoft. *IEEE Trans. Software Eng.* 40, 7 (2014), 633–649. https://doi.org/10.1109/TSE.2014.2318734

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014.* ACM, New York, NY, USA, 179–192. https://doi.org/10.1145/2535838.2535841

Ralf Lämmel and Simon Peyton Jones. 2003. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of TLDI'03: 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, New Orleans, Louisiana, USA, January 18, 2003.* ACM, New York, NY, USA, 26–37. https://doi.org/10.1145/604174.604179

Ralf Lämmel and Joost Visser. 2003. A Strafunski Application Letter. In *Practical Aspects of Declarative Languages, 5th International Symposium, PADL 2003, New Orleans, LA, USA, January 13-14, 2003, Proceedings.* Springer-Verlag, Heidelberg Berlin, Germany, 357–375. https://doi.org/10.1007/3-540-36388-2_24

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2017. The OCaml System Release 4.06 Documentation and User's Manual. (2017). http://caml.inria.fr/pub/docs/manual-ocaml/

Huiqing Li and Simon J. Thompson. 2012. A Domain-Specific Language for Scripting Refactorings in Erlang. In *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings.* Springer, Berlin, Heidelberg, 501–515. https://doi.org/10.1007/978-3-642-28872-2_34

Massimiliano Menarini, Yan Yan, and William G. Griswold. 2017. Semantics-assisted Code Review: An Efficient Toolchain and a User Study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 – November 03, 2017.* IEEE Computer Society, 554–565. https://doi.org/10.1109/ASE.2017.8115666

Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Trans. Softw. Eng.* 30 (2004), 126–139. Issue 2. https://doi.org/10.1109/TSE.2004.1265817

merlin [n. d.]. Context sensitive completion for OCaml in Vim and Emacs. ([n. d.]). github.com/ocaml/merlin v3.0.5 released 13 November 2017.

Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. 2013. *Real World OCaml: Functional Programming for the Masses.* O'Reilly Media, Sebastopol, CA, USA.

William F. Opdyke. 1992. *Refactoring Object-Oriented Frameworks.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

Simon Peyton Jones (Ed.). 2003. *Haskell 98 Language and Libraries: Revised Report.* Cambridge University Press, Cambridge, UK. haskell.org/onlinereport

François Pottier. 2017. Visitors Unchained. *PACMPL* 1, ICFP (2017), 28:1–28:28. https://doi.org/10.1145/3110272

ppx_deriving [n. d.]. Type-driven code generation for OCaml >=4.02. ([n. d.]). github.com/ocaml-ppx/ppx_deriving v4.2.1 released 21 November 2017.

Simon Thompson and Huiqing Li. 2013. Refactoring Tools for Functional Languages. *Journal of Functional Programming* 23, 3 (2013), 293–350. https://doi.org/10.1017/S0956796813000117

Mathieu Verbaere, Ran Ettinger, and Oege de Moor. 2006. JunGL: A Scripting Language for Refactoring. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. ACM, New York, NY, USA, 172–181. https://doi.org/10.1145/1134311