Generating a corpus of Hazel programs from ill-typed OCaml programs (Extended Abstract)

Patrick Ferris
University of Cambridge
United Kingdom

Anil Madhavapeddy
University of Cambridge
United Kingdom

Abstract

When developing a new programming language, having a large corpus of both correct and incorrect programs allows language designers to test and explore the capabilities of their new language. However, bootstrapping such a corpus of incorrect programs is time consuming and arduous. We therefore explore how to reuse code from more mature languages to generate a corpus of ill-typed code for newer ones. We have developed a compiler to Hazel, an emerging language with typed holes, from the more mature OCaml ecosystem. We find it practical to generate a comprehensive corpus of ill-typed programs for Hazel development, and discuss future larger scale efforts towards bridging ecosystems.

CCS Concepts: • **Software and its engineering → Source code generation**.

Keywords: source code generation, type errors, OCaml, Hazel

1 Introduction

Over the past 70 years, there has been a wonderful bloom of diverse programming languages, each with its own ancestry, variations and features [12]. It can be difficult for new functional programming languages to meet the expectations set by more mature languages with larger, well-established ecosystems. Older languages have benefited from years of compiler development and maintenance, with comprehensive test suites and users with large codebases. These benefits extend to programming language research as well, where older languages benefit from the existence of corpuses of ill-typed programs [10] and comprehensive fuzz testing [5].

In this abstract, we provide an example of reusing an existing language's programs to help improve a new functional programming language. Our goal is to explore novel ways to quickly generate examples of ill-typed programs for the Hazel [3] programming language, which features typed holes and live programming environments to interactively debug type errors. We show how we built an OCaml to Hazel compiler, and why such an approach may be more broadly useful to bridge ecosystems.

2 Background

OCaml is a general-purpose functional programming language that has been around in some form since 1996 [4]. Hazel is a new, pure functional programming language with typed holes [8]. Both OCaml and Hazel share an ML lineage which provides just enough similarity, particularly in their purely functional core, that compiling bidirectionally is quite possible for a large subset of both languages.

Both Hazel and OCaml offer algebraic data types, recursion and pattern-matching. Consider the following simple code fragment in OCaml:

```
type itree =
  | Leaf
  | Branch of int * itree * itree

let rec sum = function
  | Leaf -> Leaf
  | Branch (i, l, r) -> i + sum l + sum r
```

The same type declaration and function expression follows in Hazel instead:

```
type itree =
  + Leaf
  + Branch (Int, itree, itree)
in
let sum : itree -> Int = fun t -> case t
  | Leaf => 0
  | Branch (i, l, r) => i + sum(l) + sum(r)
end in
```

There are some syntactic changes, but otherwise the two are very similar. OCaml is a stable, conservative language with a high degree of backwards compatibility [11]. In contrast, Hazel is a hotbed for programming language research and undergoing rapid development to introduce new features such as live literals for filling holes with GUIs [7], live pattern-matching [13] and bidirectionally typed, collaborative editors [1] (to name but a few). This makes Hazel an exciting language to target, but difficult to keep up with!

Our overall goal is to explore Hazel's *typed holes*, which make it a very developer-friendly choice as a target language. When developing our `hazel_of_ocaml` compiler, any portions of OCaml code that is not straightforward to translate could be transformed into a typed hole, making it easier to check the validity of programs in the online editor. Using OCaml as the source language works well, given the similarities between the two languages and the tendency for many OCaml programs to use straightforward functional programming features. And in particular, this bridge would give us a readymade source of useful programs to bootstrap a more realistic Hazel codebase where none currently exists.

3 Implementation

We first describe how our `hazel_of_ocaml` compiler works and some of the more difficult parts of the compilation. Going forward, we use OCaml version `5.2.0` and Hazel as of Git commit `d7a2b93`.

3.1 A Transformation of Syntax

A majority of the `hazel_of_ocaml` compiler is a syntax-driven transformation, thanks to the shared ML ancestry of OCaml and Hazel. We map the OCaml abstract syntax tree (AST) on to the Hazel AST, raising errors where non-translatable OCaml features are used (e.g. first-class modules). The definition of `itree` earlier is a good example of a syntax transformation, where a sum type uses `|` to separate constructors in OCaml and `+` in Hazel. The tuple argument of the `Branch` case is also syntactically different between the two languages.

Some transformations also require a desugaring of OCaml syntax. For example, disjunctive pattern-matching cases.

```
let rec equal eq l1 l2 =
  match l1, l2 with
  | [], [] -> true
  | [], _::_ | _::_, [] -> false
  | a1::l1, a2::l2 -> eq a1 a2 && equal eq l1 l2
```

Listing 1. A higher-order equality function for lists in OCaml

The second case for `equal` combines the case where the left or the right list are longer and in both cases returns `false`. When compiling to Hazel, which does not support disjunctive pattern-matching cases, these can be copied into two separate pattern-matching cases where the right-hand side is duplicated.

3.2 Using OCaml to Type Hazel

OCaml has a more mature type system than Hazel that supports type inference. At the time of writing, Hazel is still limited to explicit polymorphism à la System F. Using OCaml's type inference, we can rewrite functions in Hazel with type quantification and where that function is used, perform the correct type applications. For example, consider a polymorphic map function in Listing 2[1].

```
let rec map f = function
  | [] -> []
  | x :: xs -> f x :: map f xs

let floats = map float_of_int [1; 2; 3]
```

Listing 2. A recursive `map` function for lists in OCaml

[1]Hazel only supports parametric polymorphism for lists.

OCaml infers the principal type as $\forall\alpha\beta.(\alpha \to \beta) \to \alpha$ `list` $\to \beta$ `list`. In the recursive call to `map` in the function body, OCaml infers that the arguments are fully polymorphic and when creating the list of floating point numbers it unifies α with `int` and β with `float`. Our compiler reuses this type information to generate an explicitly polymorphic version of the `map` function in Hazel.

```
let map :
 forall a -> forall b -> (a -> b) -> [a] -> [b] =
 typfun a -> typfun b -> fun f -> fun xs ->
 case xs
    | [] => []
    | x :: xs => f (x) :: map@<a>@<b>(f)(xs)
 end in
let floats =
  map@<Int>@<Float>(float_of_int)([1, 2, 3])
in
```

Listing 3. An explicitly polymorphic `map` function for lists in Hazel derived from Listing 2

Type quantifications are introduced in the type annotation with `forall` and type abstractions are introduced with `typfun`. The `@<Int>` syntax is used for type application.

4 Use Cases

With this work, we hoped to accelerate the creation of Hazel's ecosystem by reusing existing code from OCaml's standard libary and aid programming language research by providing derived corpora of programs from other languages. These corpora could be used in research like "Total Type Error Localization and Recovery with Holes" or as compiler unit tests [3, 14].

4.1 A Derived Corpus of Programs

Our primary goal with this work was to derive a corpus of programs to help better understand how features of Hazel might help new programmers. In their paper "Dynamic witnesses for static type errors" Sedeil et al. create and use a dataset of ill-typed OCaml programs from their UC San Diego undergraduate programming languages course [9, 10].

We took the same dataset and derived ill-typed Hazel programs. Whilst these do not reflect what a user might write (Hazel's structured editor might help or hinder novice programmers), we do get ill-typed programs for free and can use them to examine how Hazel's type errors perform.

Consider the example in Listing 4, an ill-typed program for summing a list of integers.

Using the ill-typed `sumList` from Listing 4 we can derive an ill-typed Hazel program. This is shown in figure 1.

There are multiple errors being indicated by the Hazel editor. First, the case-statement is not exhaustive. Second, the integer plus operator (+) is inconsistent with the return

```
let rec sumList xs =
  match xs with
  | [] -> []
  | h1::h2::t -> h1 + (h2 sumList t)
```

Listing 4. An ill-typed OCaml program to sum an integer list.

```
let sumList :
  forall a -> [Int] -> [a] =
  typfun a -> fun xs -> case xs
    | [] => []
    | h1 :: h2 :: t => h1 + h2(sumList)(t)
  end in
?
```

Figure 1. An ill-typed Hazel program derived from Listing 4 shown in the Hazel online editor.

type of the `sumList` (a list). And third, `h2` has been given the integer type by the type constraint, its use as a function is marked as incorrect.

It is possible to derive programs without using any of OCaml's typechecking. These programs will make use of Hazel's gradual type system instantiating many holes where a type is unknown. The same example from Listing 4 when derived without any type information is shown in figure 2. In this case, with less type information, the error is only about the pattern-matching not being exhaustive.

```
let sumList = fun xs -> case xs
  | [] => []
  | h1 :: h2 :: t => h1 + h2(sumList)(t)
end in
?
```

Figure 2. An ill-typed Hazel program derived from Listing 4 shown in the Hazel online editor without any type annotations.

It is important to note that in this example, the ill-typed program has gone wrong in the first case too, returning a list where the user would want to return a zero. The type that OCaml derived and that `hazel_of_ocaml` added to the Hazel program is working against the user.

All of this is useful information for developing better type error handling, for use as compiler unit tests and to build better UIs to help novice programmers. This suggests another source of ill-typed programs, the OCaml compiler's extensive testsuite, which has comprehensive coverage for many edge cases that would be suitable for a Hazel corpus of typed holes.

5 Future Work

This work has showed promise and aided in undergraduate research into better type error debugging in Hazel. There are plenty of avenues for further development.

Large scale compilation. Compiling larger projects from OCaml to Hazel will require more knowledge about the contexts in which a portion of code is being typed. Generating Hazel code across OCaml compilation units (modules) would likely require reusing existing tooling from the OCaml ecosystem like Merlin and `ocamldep`.

Bidirectional translation. There have been adjacent efforts to compile a subset of Hazel to OCaml.[2] Continuing this work alongside our `hazel_of_ocaml` tool will create a powerful, bidirectional translation between a subset of OCaml and a subset of Hazel. OCaml developers could benefit from Hazel's gradual type system and typed holes when developing their programs, whilst Hazel developers gain access to an industry-tested compiler toolchain that supports multiple operating systems and architectures.

Supporting more OCaml. We took 65 functions from `list.ml` in OCaml's standard library and compiled them to Hazel revealing more corners of the language to support such as: handling external modules like `Option`, working around OCaml's built-in, structural, polymorphic comparison operators perhaps by making functions that use them higher-order and supporting more desugaring such as OCaml's `when` syntax for conditional pattern-matching.

Using typed holes. So far we have only used Hazel's typed holes to aid in the development of `hazel_of_ocaml`. To build a more comprehensive corpus of Hazel programs we would need to explore typed holes as an actual language feature rather than as an escape hatch. To this end future work could translate OCaml programs that use assertions as a means to write partial programs or translate subsets of other languages with full support for typed holes such as Agda and Idris [2, 6].

The `hazel_of_ocaml` compiler is available under a liberal open source license from https://github.com/patricoferris/hazel_of_ocaml. There are plans to upstream `hazel_of_ocaml` to the Hazel codebase.

References

[1] Michael D Adams, Eric Griffis, Thomas J Porter, Sundara Vishnu Satish, Eric Zhao, and Cyrus Omar. 2025. Grove: A Bidirectionally Typed Collaborative Structure Editor Calculus. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 2176–2204.

[2] EDWIN BRADY. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. doi:10.1017/S095679681300018X

[3] Hazel Developers. 2025. *Hazel.* https://github.com/hazelgrove/hazel

[2] https://github.com/hazelgrove/hazel/tree/transpile

[4] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, KC Sivaramakrishnan, and Jérôme Vouillon. 2025. The OCaml system release 5.3. https://ocaml.org/manual/5.3/ocaml-5.3-refman.pdf

[5] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-driven QuickChecking of compilers. *Proc. ACM Program. Lang.* 1, ICFP, Article 15 (Aug. 2017), 23 pages. doi:10.1145/3110259

[6] Ulf Norell. 2009. Dependently typed programming in Agda. In *Proceedings of the 4th international workshop on Types in language design and implementation*. 1–2.

[7] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling typed holes with live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 511–525.

[8] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A Hammer. 2019. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32.

[9] Eric L Seidel and Ranjit Jhala. 2017. A Collection of Novice Interactions with the OCaml Top-Level System. doi:10.5281/zenodo.806814

[10] Eric L Seidel, Ranjit Jhala, and Westley Weimer. 2016. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 228–242.

[11] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proceedings of the ACM on Programming Languages* 4, ICFP (aug 2020), 1–30. doi:10.1145/3408995

[12] Peter Van Roy and Seif Haridi. 2004. *Concepts, Techniques, and Models of Computer Programming* (1st ed.). The MIT Press.

[13] Yongwei Yuan, Scott Guest, Eric Griffis, Hannah Potter, David Moon, and Cyrus Omar. 2023. Live pattern matching with typed holes. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 609–635.

[14] Eric Zhao, Raef Maroof, Anand Dukkipati, Andrew Blinn, Zhiyi Pan, and Cyrus Omar. 2024. Total type error localization and recovery with holes. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 2041–2068.