Generating Hazel Programs from Ill-typed OCaml Programs

Patrick Ferris and Anil Madhavapeddy

Sunday 12th October, 2025

University of Cambridge

The Problem

The Problem

New languages *do not* come with a large set of programs written in that language!

Background

Language designers juggle [2, 3]:

- Encoding semantics
- Providing backwards compatibility
- Keeping historic familiarity
- Dealing with foreign function interfaces
- Designing useful error messages
- Ensuring usability for novice and expert programmers

All of these require a corpus of both correct and incorrect programs.

Background

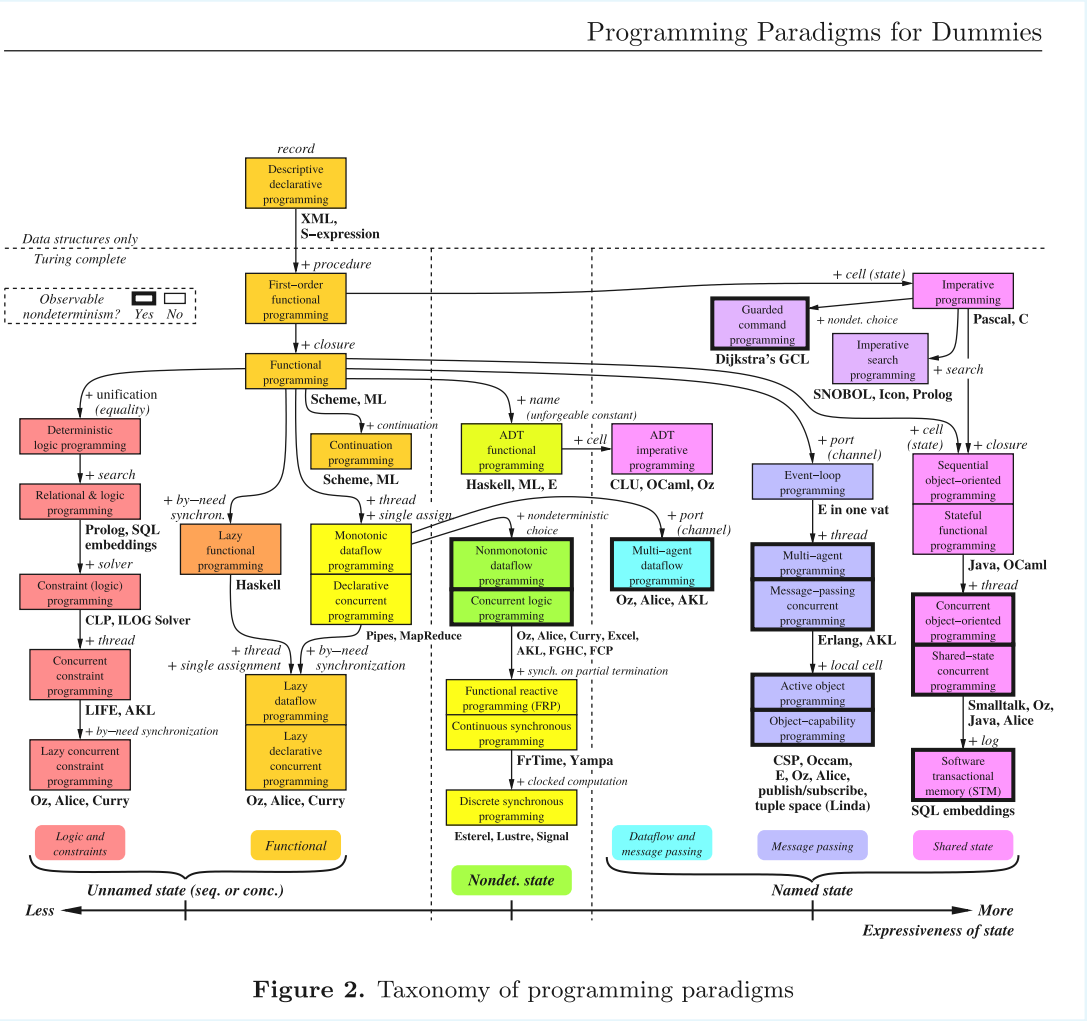


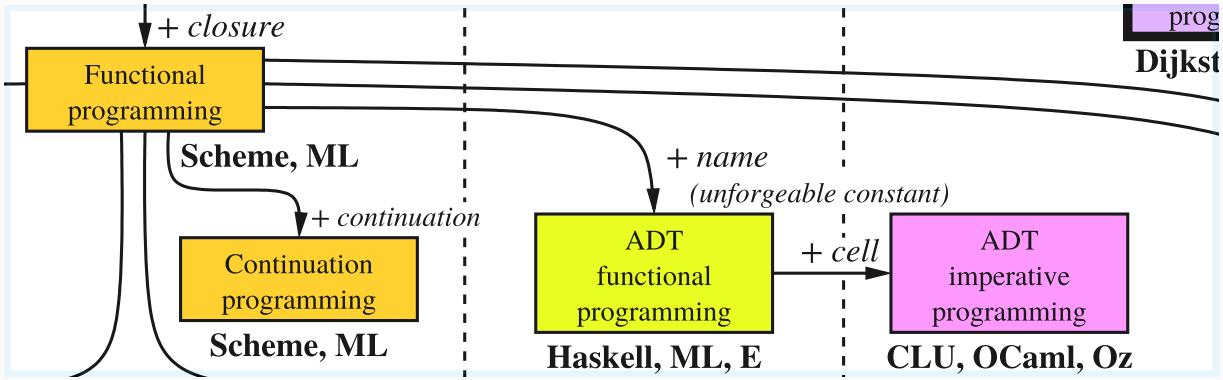
Figure 2. Taxonomy of programming paradigms

Maturer languages that share the same design and lineage can help.

(*From "Programming Paradigms for Dummies" by Peter Van Roy* [5]).

Hazel & OCaml

Hazel



"Hazel is a live functional programming environment that is able to typecheck, manipulate, and even run incomplete programs, i.e. programs with holes."[4]

- Hazel implements an ML-like type system that is *gradual*.
- Editor and type-system are tightly co-designed.
- Incomplete programs are completely allowed, in fact programs that might not typecheck in OCaml may evaluate to a full value in Hazel!

A Taste of Hazel: An ML

```
(* OCaml integer tree *)
type itree =
  | Leaf
  | Branch of int * itree * itree

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

```
# Hazel integer tree
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
```

A Taste of Hazel: Typed Holes

```
let length : [Int] -> Int = fun lst -> case lst
  | [] => 0
  | x :: xs => 1 + length(xs)
end in
let sum : [Int] -> Int = fun lst -> case lst
  | [] => 0
  | x :: xs => x + sum(xs)
end in
length([1, ?, ?, ?]) + sum([1, ?])

# 4 + (1 + (? : Int + 0))
```

Decomposable Type Highlighting for Bidirectional Type and Cast Systems

We explore how to provide programmers with an interactive interface for explaining the process by which static types and dynamic casts are derived, with the goal of improving the debugging of static and dynamic type error.

```
let add = fun x -> fun y -> x + y in
add("hello")(2)
```

```
≡ "hello": Int + 2
```

To be presented at HATRA 2025 by Max Carroll

OCaml Programs

Correct OCaml programs are not hard to come by: the standard library, 4500+ packages, research tools, an extensive compiler testsuite etc.

Incorrect, and in particular, ill-typed programs exist too. In “Dynamic witnesses for static type errors” Seidel et al. use their dataset of “novice interactions” with OCaml as a corpus to evaluate their algorithms. [6, 7]

Hazel of OCaml

Transforming Syntax

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

- Syntactic changes (e.g. `->` becomes `=>`)
- Desugaring (e.g. disjunctive pattern-matches)
- Optional module removal (e.g. `List.map` becomes `list_map`)

Reusing Typed Holes as Placeholders

```
(* hazel_of_ocaml sourcecode, transforming expressions *)
let rec of_expression (e : Parsetree.expression) : AST.exp =
  match e.pexp_desc with
  | Pexp_constant constant -> of_constant constant
  | Pexp_ifthenelse (e1, e2, Some e3) ->
      AST.If (of_expression e1, of_expression e2, of_expression e3)
  (* ... *)
  | _ -> EmptyHole
```

📝 Source code

Automatic Explicit Polymorphism (1/2)

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

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

- `map` is given a principal type: $\forall\alpha.\forall\beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$
- recursive `map` application keeps α and β
- `map` used with `float_of_int` unifies α with `int` and β with `float`

Automatic Explicit Polymorphism (2/2)

```
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
```

- Reuse OCaml's typing to add the type abstractions and fill the type applications.

Use Cases

Ill-typed OCaml Code

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

```
| h1::h2::t -> h1 + (h2 sumList t)
       Error: This expression has type int
       This is not a function; it cannot be applied.
```

Ill-typed Hazel Code

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

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

Type Highlighting?

```
let concat : [[Int]] -> [Int] = fun x -> case x
    | [] => []
    | x :: xs => x :: concat(xs)
end in
concat([[1], [2], [3]])
```

```
≡ [[1]: Int, [2]: Int, [3]: Int]
```

See Max's talk at HATRA 2025

Experiences

Experiences: OCaml

- ✓ State of the art language server as an easy type checking API (see Merlin [1])
- ✓ Strong ecosystem of AST-based transformations (ocaml-ppx/ppxlib) [8]
- ✓ Fairly popular in industry and research
- ✓ Complicated features that are seldom used[1]
- × Module system adds complexity

[1] At least in novice or portable code.

Experiences: Hazel

- ✓ Typed holes are *fantastic* placeholders!
- ✓ Hazel is written in OCaml[1]
- × Only web-based editor and evaluation
- × Lots of features not yet upstreamed (modules, records, implicit polymorphism)

[1] Actually in Reason, a JS-like syntax for OCaml

Takeaways and Questions

- This approach benefits greatly because OCaml and Hazel are very similar, how similar do the language have to be?
  - Static vs. Live Programming?
  - Static vs. Gradual?
- In terms of bootstrapping a corpus of ill-typed programs for Hazel, this was mostly a success
- Having flexibility in the target language (typed holes + gradual typing) was useful
- Having good types and APIs for parsetrees made the code quite simple
- Friendly APIs for typecheckers

Bibliography

[1] Frédéric Bour, Thomas Refis, and Gabriel Scherer. 2018. Merlin: a language server for OCaml (experience report). *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–15.

[2] Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2021. PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design. *ACM Trans. Comput.-Hum. Interact.* 28, 4 (July 2021). https://doi.org/10.1145/3452379

[3] Linda McIver and Damian Conway. 1996. Seven deadly sins of introductory programming language design. In *Proceedings 1996 International Conference Software Engineering: Education and Practice*, 1996. 309–316.

[4] 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.

[5] Peter Van Roy and others. 2009. Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music* 104, (2009), 616–621.

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

[7] 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*, 2016. 228–242.

[8] Leo White. 2013. Extension points for OCaml. In *OCaml Users and Developers Workshop*, 2013.