Programmable Patterns for OCaml

Oghenevwogaga Ebresafe
gaga@cs.toronto.edu
University of Toronto

Kiarash Sotoudeh
kiarash.sotoudeh@mail.utoronto.ca
University of Toronto

Ningning Xie
ningningxie@cs.toronto.edu
University of Toronto

Abstract

Pattern matching is a hallmark feature of OCaml, but it remains in fundamental tension with data abstraction. Since patterns expose concrete constructors, a module cannot support pattern matching over an abstract type without also revealing its representation. We propose programmable patterns, a language design that brings abstraction to OCaml pattern matching. Our design allows modules to expose pattern interfaces independently of their concrete implementations, preserving data abstraction while retaining the familiar structure of pattern matching. We implement programmable patterns as a desugaring pass in the OCaml compiler.

1 Motivation

Pattern matching is beloved by OCaml programmers, and for good reason: it makes case analysis concise, readable, and hard to get wrong. However, pattern matching is fundamentally at odds with data abstraction. Traditional pattern matching requires exposing concrete data constructors, while good software engineering practice demands hiding implementation details behind abstract interfaces. This creates an unfortunate dilemma: library authors must choose between providing clean abstractions or enabling pattern matching for their users. For example, consider the following OCaml program:

```
module JoinList : sig
  type 'a seq
  val empty : 'a seq
  val cons : 'a * 'a seq -> 'a seq
  val append : 'a seq * 'a seq -> 'a seq
end = struct
  type 'a seq = Nil | One of 'a
       | App of 'a seq * 'a seq
  let empty = Nil
  let cons (x, xs) = append (One x, xs)
  let rec append = function
    | (xs, Nil) -> xs
    | (Nil, ys) -> ys
    | (xs, ys) -> App (xs, ys)
end
```

In this example, the abstract interface prevents users from scrutinizing the constructors of `seq` via pattern matching, forcing reliance on interface functions (e.g., `cons` and `append`). While using the interface functions preserves the invariant that `Nil` never appears as a child of `App`, it eliminates the expressiveness that pattern matching provides. Conversely, exposing the constructors in the interface enables pattern matching but breaks the `JoinList` abstraction boundary. Moreover, this abstraction problem is closely related to the infamous PPX versioning problem[1], which has led to breakage in PPX tooling. The core issue is that `ppxlib` manipulates programs using the OCaml compiler's internal syntax tree constructors, which are not a stable abstraction boundary and are therefore prone to change. A similar situation arises in Rocq plugin development in OCaml[2].

The OCaml community has long recognized the lack of abstraction mechanisms in the pattern matching machinery. This is reflected in efforts such as `ppx_view`[3], `ppx_viewpattern`[4], and Gabriel Scherer's decade-old proposal for similar abstractions[5]. Nevertheless, OCaml still lacks a principled solution to the abstraction problem for pattern matching. In this respect, OCaml lags behind peer languages that support some form of pattern matching abstraction, such as F# [Syme et al. 2007] and Haskell [Pickering et al. 2016].

2 Programmable patterns

In this work, we propose programmable patterns for OCaml: a language extension that lets module interfaces expose patterns without revealing the constructors that implement them. For instance, the `JoinList` module can be refactored to expose the pattern abstractions `Empty` and `Cons`, as shown below:

```
module JoinList : sig
  type 'a seq
  pattern Cons: 'a * 'a seq -> 'a seq
  pattern Empty: 'a seq
  val append: 'a seq * 'a seq -> 'a seq
end = struct
  type 'a seq = Nil | One of 'a | App of 'a seq * 'a seq

  pattern Empty = function
    | Nil -> match
    | (One _) -> mismatch
    | (App (_, _)) -> mismatch

  pattern Cons = function
    | Nil -> mismatch
    | One x -> match (x, Nil)
    | App (Cons (x, xs), ys) -> match (x, (App (xs, ys)))
    | App (Empty, Cons (y, ys)) -> match (y, ys)
    | App (Empty, Empty) -> mismatch

  let rec append = function
```

[1] https://blog.janestreet.com/an-solution-to-the-ppx-versioning-problem/
[2] https://sympa.inria.fr/sympa/arc/coq-club/2024-07/msg00017.html
[3] https://github.com/ocaml-ppx/ppx_view
[4] https://github.com/sim642/ppx_viewpattern
[5] https://cambium.inria.fr/blog/pattern-synonyms-as-expressions

```
    | (xs , Nil) -> xs
    | (Nil , ys) -> ys
    | (xs , ys) -> App (xs , ys)
end
```

Crucially, these pattern definitions hide the concrete representation of `seq`: clients match on `Empty` and `Cons` without ever observing the constructors `Nil`, `One`, and `App`, thereby preserving abstraction. Notably, the patterns here are themselves recursive (for instance, `Cons` refers to both `Cons` and `Empty`), but we restrict this recursion to be well-founded, so that pattern matching is guaranteed to terminate (see §2.2).

2.1 Compilation

Programmable patterns compile away to vanilla OCaml; no runtime support is required. Each pattern declaration becomes an ordinary function returning an `option`, and a module exposes these functions in its signature in place of the patterns themselves. For example, the `JoinList` module is compiled to a module with the following signature:

```
module JoinList : sig
 type 'a seq
 val match_Empty: 'a seq -> unit option
 val match_Cons: 'a seq -> ('a * 'a seq) option
 val append: 'a seq * 'a seq -> 'a seq
end
```

Use sites are then rewritten to call these functions. Consider a `map` written against the programmable patterns `Cons` and `Empty`:

```
let rec map = fun xs f -> match xs with
    | Cons (x, xs') -> cons (f x, map f xs')
    | Empty -> empty
```

It is compiled to a function of the same type that defers to the match functions of the compiled `JoinList` module:

```
let rec map  = fun xs f ->
    match JoinList.match_Empty xs, JoinList.match_Cons xs
      with
    | Some (), _ -> JoinList.empty
    | _, Some (x, xs') -> JoinList.cons (f x, map xs' f)
```

2.2 Exhaustiveness checking

Existing systems take different positions on exhaustiveness checking. F# active patterns distinguish between total active patterns, whose cases are defined together and are therefore exhaustive by construction, and partial active patterns, which may fail to match and hence cannot in general be checked for exhaustiveness[6]. This design provides a useful abstraction mechanism, but the presence of arbitrary computation in patterns, together with the separation between total and partial active patterns, limits what the exhaustiveness checker can guarantee. GHC, by contrast, provides `COMPLETE` pragmas for pattern synonyms, allowing programmers to tell the compiler that a given set of patterns is exhaustive. However, these declarations are trusted by the compiler, and therefore do not themselves provide strong guarantees.

Our design combines the flexibility of GHC's approach with the static guarantees of total active patterns. Users may declare that a group of patterns is exhaustive for a given data type. For example, the `JoinList` signature could be written as follows:

```
module JoinList : sig
  type 'a seq =
    with pattern Empty | Cons
  pattern Empty : 'a seq
  pattern Cons : 'a * 'a seq -> 'a seq
end
```

This declaration states that `Empty` and `Cons` together match values of type `seq` exhaustively. Unlike GHC's `COMPLETE` pragmas, however, we do not treat such declarations as trusted axioms. Instead, we interpret them as proof obligations: the compiler statically verifies that the declared patterns are genuinely exhaustive for the underlying type. In GHC or F#, such a verifier would not be feasible, because their patterns can perform arbitrary computation. Our design avoids this difficulty by restricting pattern definitions to well-founded recursion.

3 Status and future plans

We have developed a prototype compiler that lowers programmable patterns away into vanilla OCaml. We plan to integrate the exhaustiveness checking mechanism described above, and study the interaction of programmable patterns with first-class modules and GADTs. We also intend to validate the design through case studies and discussion from the community[7].

References

Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern synonyms. In *Proceedings of the 9th International Symposium on Haskell* (Nara, Japan) *(Haskell 2016)*. Association for Computing Machinery, New York, NY, USA, 80–91. doi:10.1145/2976002.2976013

Don Syme, Gregory Neverov, and James Margetson. 2007. Extensible pattern matching via a lightweight language extension. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 29–40. doi:10.1145/1291151.1291159

[6] https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/active-patterns

[7] https://github.com/ocaml/ocaml/issues/14627