

Programmable Patterns for OCaml

Oghenevwogaga
Ebresafe
University of Toronto

Kiarash
Sotoudeh
University of Toronto

Ningning
Xie
University of Toronto



ML Family Workshop 2026 • Paris, France

Pattern matching makes data easy to take apart

```
type rope =  
  | Str of string * int * int  
  | App of rope * rope * int * int
```

Definition of a Rope
data structure

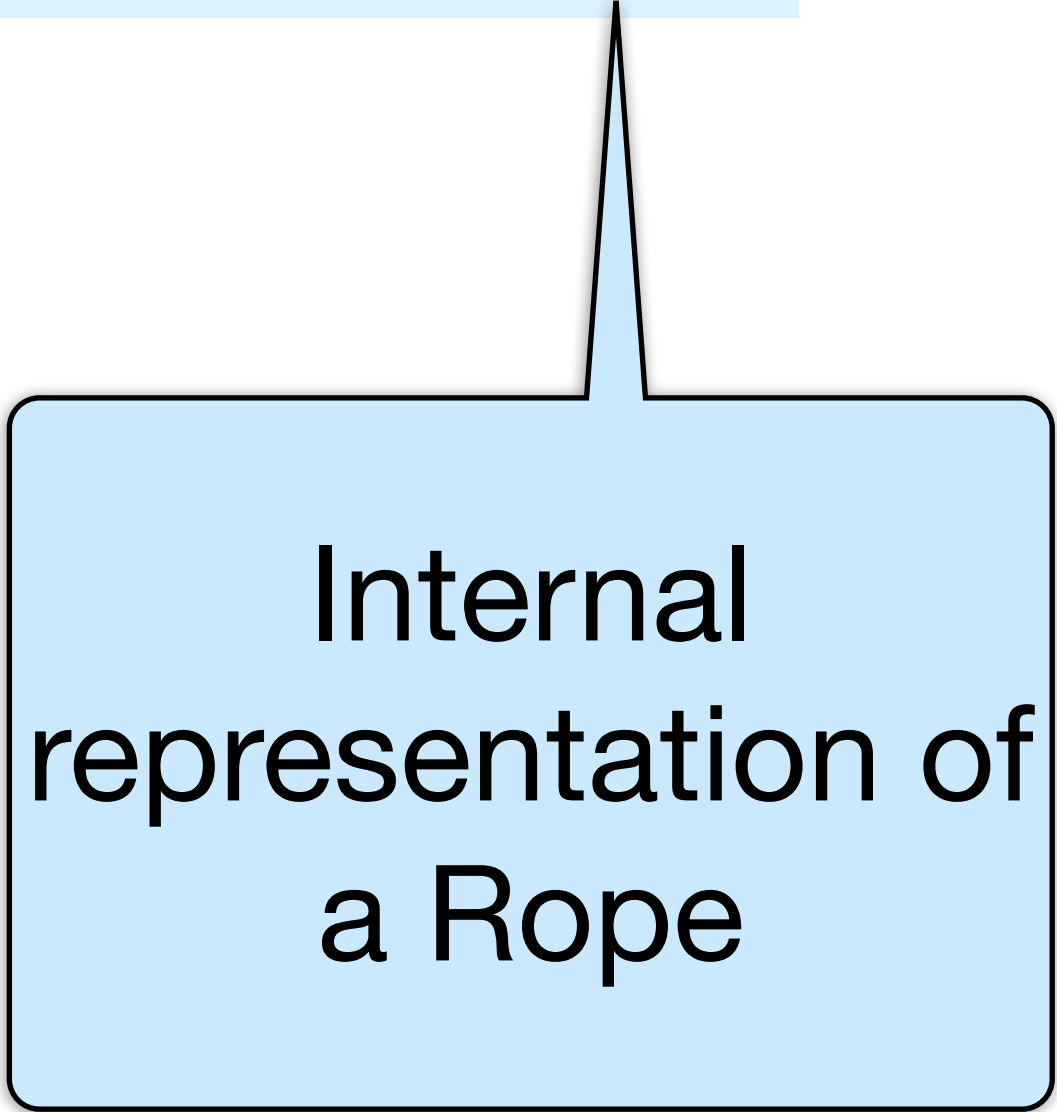
```
let split_on_char c t =  
  let rec go acc = function  
    | Str (s, ofs, len) →  
      leaf_pieces c s ofs len acc  
    | App (t1, t2, _, _) →  
      go (go t2 acc) t1  
  in go [empty] t
```

Scrutinization of the
Rope data structure

Module interfaces let us hide the representation

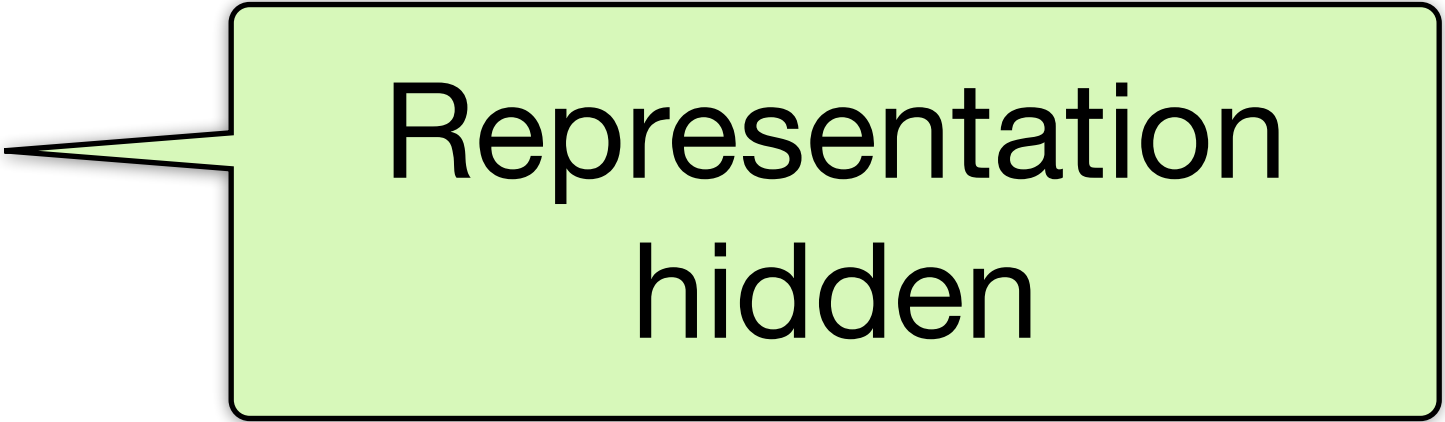
```
module Rope = struct
  type t =
    | Str of string * int * int
    | App of t * t * int * int

  let get = ...
  let append = ...
  let sub = ...
end
```



```
module Rope : sig
  type t

  val get : t → int → char
  val append : t → t → t
  val sub : t → int → int → t
  ...
end
```



Pattern matching is at odds with abstraction

```
module Rope : sig  
  type t
```

```
...  
end
```

Doesn't expose
constructors for pattern
matching

```
let split_on_char c (t: Rope.t) =  
  let rec go acc = function  
    | Str (s, ofs, len) →  
      leaf_pieces c s ofs len acc  
    | App (t1, t2, -, -) →  
      go (go t2 acc) t1  
  in go [empty] t
```

This pattern
matching isn't
possible

Problem #1

One representation, several useful views

```
type rope =  
  | Str of string * int * int  
  | App of rope * rope * int * int
```

An efficient string representation

```
type rope =  
  | Empty  
  | Cons of char * rope
```

A functional-programmer-friendly string representation

Problem #2

What we want: matching that respects the interface

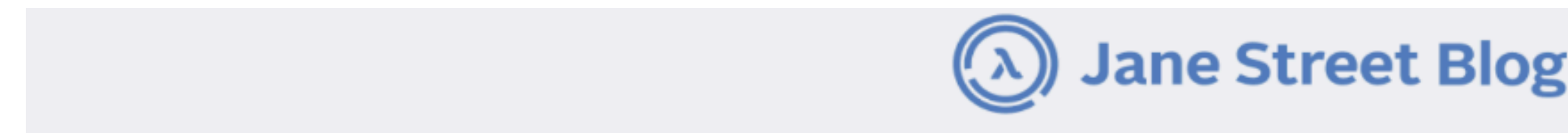
```
module Rope : sig
  type t =
    | Str of string * int * int
    | App of t * t * int * int
  ...
end
```

We want to see
Rope.t as a list of
characters

```
module Rope : sig
  type t
  ...
end
```

We want to be able to pattern
match on Rope.t but still keep
it abstract

This bites real OCaml code: the PPX versioning problem



A solution to the ppx versioning problem

NOV 08, 2016 | 11 MIN READ



By: [Jeremie Dimino](#)

2016

[ANN] Ppxlib: Support for future compilers

■ Ecosystem ppx, compiler, ppxlib



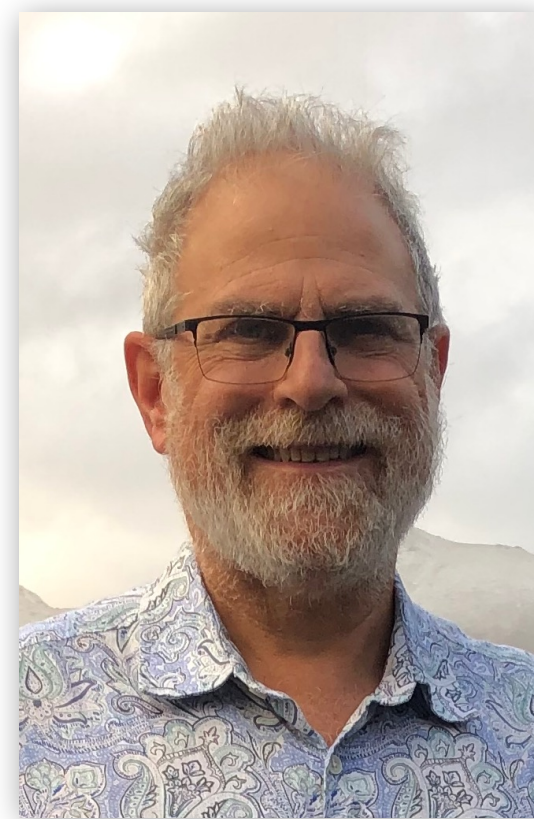
NathanReb

2 Oct 2025

Handling future AST changes in ppxlib

2025

Wadler solved this in 1987



Views:

A way for pattern matching to
cohabit with data abstraction

Philip Wadler

Programming Research Group, Oxford University, UK
and Programming Methodology Group, Chalmers University, Sweden

POPL 1987

We already have a known solution!

Wadler solved this in 1987



Views:
A way for pattern matching to
cohabit with data abstraction

Philip Wadler
Programming Research Group, Oxford University, UK
and Programming Methodology Group, Chalmers University, Sweden

```
module Rope = struct
  type t =
    | Str of string * int * int
    | App of t * t * int * int
end
```

```
module Rope = struct
  type t
  with viewtype =
    | Empty
    | Cons of char * t
end
```

The idea: what you match on $\neq$ how it's represented

So why doesn't OCaml have views?

OCaml programmers value the following things:

Exhaustiveness
checking

Interoperability with
OCaml patterns

Good runtime & compile
time performance

Programmable patterns: an attempt to implement
view patterns in a way that aligns with the values of
OCaml

Declaring programmable patterns

```
module Rope = struct
  type t =
    | Str of string * int * int
    | App of t * t * int * int
end
```

Rope.t internal
representation
remains abstract

```
module Rope : sig
  type t
  pattern Empty: t
  pattern Cons: char * t → t
end
```

Pattern
signatures in
the interface

Programming with programmable patterns

pattern Cons: $\text{char} * t \rightarrow t$

Extraction payload

Scrutinee

```
let rec starts_with prefix r =  
  match prefix, r with  
  | Empty, _ → true  
  | Cons _, Empty → false  
  | Cons (c, p'), Cons (d, r') when c = d → starts_with p' r'  
  | Cons _, Cons _ → false
```

Defining programmable patterns

```
module Rope : sig  
  pattern Empty: t  
end
```

```
module Rope = struct  
  pattern Empty = function  
    | Str (_, _, 0)      → match  
    | App (_, _, 0, _)  → match  
    | Str (_, _, _)     → mismatch  
    | App (_, _, _, _)  → mismatch  
end
```

Scrutinizing the
input via pattern
matching

Match when
the length is
zero

Don't match
otherwise

Ropes with programmable patterns

```
module Rope : sig
  pattern Cons: char * t → t
end
```

```
module Rope = struct
  pattern Cons = function
    | Str (_, _, 0) → mismatch
    | Str (s, off, len) →
        match (s.[off], Str (s, off + 1, len - 1))
    | App (_, _, 0, _) → mismatch
    | App (Cons (c, l'), r, _, _) → match (c, app l' r)
    | App (Empty, Cons (c, r'), _, _) → match (c, r')
    | App (Empty, Empty, _, _) → mismatch
end
```

Match carries
a payload

Recursively
defined

Mixed views: multiple views of Ropes

Split view
safely exposes
tree structure

```
module Rope : sig
  type t
  ...
  pattern Split : t * t → t
  pattern Snoc   : t * char → t
end
```

Snoc allows
backward
traversal

```
let rec has_close r =
  match r with
  | Empty → false
  | Split (Snoc (_, '*'), Cons ('/', _)) → true
  | Split (l, r') → has_close l || has_close r'
  | Cons ('*', Cons ('/', _)) → true
  | Cons (_, rest) → has_close rest
```

A list, a tree, and back-to-front — in a
single match

Well-formedness for pattern definitions

The function itself
should be total

Payload should
be pure

Must be the
shape of the
scrutinee

```
module Rope = struct
  pattern Cons = function
  | Str (_, _, 0) → mismatch
  | Str (s, off, len) →
    match (s.[off], Str (s, off + 1, len - 1))
  | App (_, _, 0, _) → mismatch
  | App (Cons (c, l'), r, _, _) → match (c, app l' r)
  | App (Empty, Cons (c, r'), _, _) → match (c, r')
  | App (Empty, Empty, _, _) → mismatch
end
```

Recursion must
be well-founded

Exhaustiveness checking & separate compilation

How do we check that this pattern usage is exhaustive?

Module A

```
let rec starts_with prefix r =  
  match prefix, r with  
  | Empty, _ → ...  
  | Cons _, Empty → ...  
  | Cons (c, p'), Cons (d, r') when c = d → ...  
  | Cons _, Cons _ → ...
```

Exhaustiveness checking & separate compilation

How do we check that this pattern usage is exhaustive?

Module A

```
let rec starts_with prefix r =  
  match prefix, r with  
  | Empty, _ → ...  
  | Cons _, Empty → ...  
  | Cons (c, p'), Cons (d, r') when c = d → ...  
  | Cons _, Cons _ → ...
```

We could look at the implementation of the pattern

Module B

```
pattern Cons = function  
  | Str (_, _, 0) → mismatch  
  | Str (s, off, len) → ...  
  | App (_, _, 0, _) → ...  
  | App (Cons (c, l'), r, _, _) → ...  
  | App (Empty, Cons (c, r'), _, _) → ...  
  | App (Empty, Empty, _, _) → ...
```

Exhaustiveness checking & separate compilation



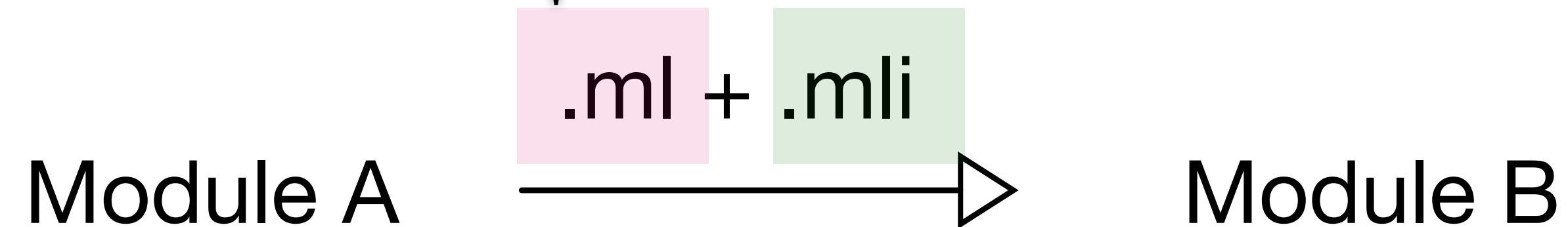
Separate compilation works 

Exhaustiveness checking & separate compilation



Separate compilation works ✓

Have to look at the
implementation



Separate compilation breaks ⚠

Exhaustiveness checking with separate compilation

```
module Rope : sig
  type t
  with pattern Empty | Cons
  pattern Empty: t
  pattern Cons: char * t → t
end
```

Tell the compiler to check that Empty/Cons are exhaustive

```
module Rope = struct
  type t = ...
  pattern Empty = ...
  pattern Cons = ...
end
```

Empty/Cons are checked to be exhaustive

```
let rec starts_with prefix r =
  match prefix, r with
  | Empty, _ → true
  | Cons _, Empty → false
  | Cons (c, p'), Cons (d, r') when c = d → starts_with p' r'
  | Cons _, Cons _ → false
```

Usage of Empty/Cons will be marked exhaustive

Exhaustiveness annotation inference

```
module Rope : sig
  type t
  with ...
  pattern Empty: t
  pattern Cons: char * t → t
  pattern Split : t * t → t
  pattern Snoc   : t * char → t
end
```

Open question: is it possible to infer the exhaustiveness annotation for all exposed patterns?

```
let rec has_close r =
  match r with
  | Empty → false
  | Split (Snoc (_, '*'), Cons ('/', _)) → true
  | Split (l, r') → has_close l || has_close r'
  | Cons ('*', Cons ('/', _)) → true
  | Cons (_, rest) → has_close rest
```

Compiling programmable patterns

```
module Rope : sig
  type t
  pattern Empty: t
  pattern Cons: char → t → t
end
```

Programmable
patterns

Our OCaml compiler fork



```
module Rope : sig
  type t
  val match_Empty: t → unit option
  val match_Cons: t → (char * t) option
end
```

Vanilla OCaml

Compiling programmable pattern usage

```
let rec starts_with prefix r =  
  match prefix, r with  
  | Empty, _ → true  
  | Cons _, Empty → false  
  | Cons (c, p'), Cons (d, r') when c = d → ...  
  | Cons _, Cons _ → false
```



```
let rec starts_with prefix r =  
  match Rope.match_Empty prefix, Rope.match_Cons prefix,  
        Rope.match_Empty r,      Rope.match_Cons r with  
  | Some (), _, _, _ → true  
  | _, Some _, Some (), _ → false  
  | _, Some (c, p'), _, Some (d, r') when c = d → starts_with p' r'  
  | _, Some _, _, Some _ → false  
  | _, _, _, _ → assert false
```

Compiling programmable pattern usage

```
let rec starts_with prefix r =  
  match prefix, r with  
  | Empty, _ → true  
  | Cons _, Empty → false  
  | Cons (c, p'), Cons (d, r') when c = d → ...  
  | Cons _, Cons _ → false
```



Open question: a compilation approach for mixed views that avoids allocation?

```
let rec starts_with prefix r =  
  match Rope.match_Empty prefix, Rope.match_Cons prefix,  
        Rope.match_Empty r,      Rope.match_Cons r with  
  | Some (), _, _, _ → true  
  | _, Some _, Some (), _ → false  
  | _, Some (c, p'), _, Some (d, r') when c = d → starts_with p' r'  
  | _, Some _, _, Some _ → false  
  | _, _, _, _ → assert false
```

Prior designs



- Characterized the problem in the setting of OCaml
- Focused on a simple design, left out exhaustiveness checking, but says it's very important for OCaml
- Supports bidirectional patterns

Pattern synonyms, arbitrary patterns as expressions

January 25, 2014 By Gabriel Scherer Last updated on 2014/01/27

Gabriel Scherer's 2014 design

https://github.com/ocaml-ppx/ppx_view

Specific to Ppxlib's
problems

https://github.com/sim642/ppx_viewpattern

More general, inspired by
Haskell view patterns

Work in progress

- Exhaustiveness annotation inference
- Study runtime & compile time overhead and ways to improve it
- Study feature interaction with GADTs and first-class modules

Programmable Patterns for OCaml

Oghenevwogaga
Ebresafe
University of Toronto

Kiarash
Sotoudeh
University of Toronto

Ningning
Xie
University of Toronto



ML Family Workshop 2026 • Paris, France

