BEYOND THE BASICS OF LSP ADVANCED IDE SERVICES FOR OCAML

Xavier Van de Woestyne - @vdwxv - xvw.lol

BEYOND THE BASICS OF LSP ADVANCED IDE SERVICES FOR OCAML

Xavier Van de Woestyne - @vdwxv - xvw.lol

Tarides: Making Critical Systems Better

We help developers and companies build **robust**, **secure**, **high-performance** applications whilst maintaining crucial **reliability**.

BEYOND THE BASICS OF LSP
ADVANCED IDE SERVICES
FOR OCAML

And more ...

DUNE

OCaml

compiler
ecosystem
platform/tooling

SpaceOS

Xavier Van de Woestyne - @vdwxv - xvw.lol

Tarides: Making Critical Systems Better

We help developers and companies build **robust**, **secure**, **high-performance** applications whilst maintaining crucial **reliability**.

BEYOND THE BASICS OF LSP ADVANCED IDE SERVICES FOR OCAML

Xavier Van de Woestyne - @vdwxv - xvw.lol

And more ...

DUNE

OCaml

compiler
ecosystem
platform/tooling

SpaceOS

Tarides: Making Critical Systems Better

We help developers and companies build **robust**, **secure**, **high-performance** applications whilst maintaining crucial **reliability**.

Working in the Editor Team (on IDE)

*An adventure of **Merlin** and **OCaml-LSP-Server***

BEYOND THE BASICS OF LSP ADVANCED IDE SERVICES FOR OCAML

And more ...

DUNE

OCaml

compiler ecosystem platform/tooling

SpaceOS

Xavier Van de Woestyne - @vdwxv - xvw.lol

***Tarides**: Making Critical Systems Better*

We help developers and companies build **robust**, **secure**, **high-performance** applications whilst maintaining crucial **reliability**.

*Working in **the Editor Team** (on IDE)*

*An adventure of **Merlin** and **OCaml-LSP-Server***

BEYOND THE BASICS OF LSP
ADVANCED IDE SERVICES FOR OCAML

*An ML language, a **strict Functional, Imperative**, with a powerful **type system** (ADTs, GADTs and row polymorphism), **Type inference**, Advanced **module system**, **OOP** with structural subtyping and **user defined effect** (as core language feature)*

And more ...

DUNE

OCaml

compiler ecosystem platform/tooling

SpaceOS

Xavier Van de Woestyne - @vdwxv - xvw.lol

***Tarides**: Making Critical Systems Better*

We help developers and companies build **robust**, **secure**, **high-performance** applications whilst maintaining crucial **reliability**.

*Working in **the Editor Team** (on IDE)*

sorry for the **bazar** with my slides,
they're displaying my speaker notes!

An adventure of **Merlin** and **OCaml-LSP-Server**

BEYOND THE BASICS OF LSP ADVANCED IDE SERVICES FOR OCAML

And more ...

DUNE

OCaml

compiler
ecosystem
platform/tooling

SpaceOS

An ML language, a **strict Functional**, **Imperative**, with a powerful **type system** (ADTs, GADTs and row polymorphism), **Type inference**, Advanced **module system**, **OOP** with structural subtyping and **user defined effect** (as core language feature)

Xavier Van de Woestyne - @vdwxv - xvw.lol

Tarides: Making Critical Systems Better

We help developers and companies build **robust**, **secure**, **high-performance** applications whilst maintaining crucial **reliability**.

Working in
the Editor Team (on IDE)

PLAN

What is a Language Server
A very naive editor Timeline

About Merlin
Basics workflow and features

About Lsp
Capabilities/Features and migration strategy

Implementation details about Some Advanced Features
Including a proof that Lambda Calculus can be concretely used as a functional programmer

Pro/Cons Lsp
and about VSCode integration

Conclusion and future work

BUT FIRST, WHY?

Discussions with other functional programming language users about their IDEs and their **expectations**, and also **present** our work :)

PLAN

About Merlin
Basics workflow and features

About Lsp
Capabilities/Features and migration strategy

Implementation details about Some Advanced Features
Including a proof that Lambda Calculus can be concretely used as a functional programmer

Pro/Cons Lsp
and about VSCode integration

Conclusion and future work

What is a Language Server
A very naive editor Timeline

And maybe find how to **solve some issues** with you

WHAT A GOOD IDE SHOULD BRING

Good support for code editing (hilighting/folding)

WHAT A GOOD IDE SHOULD BRING

WHAT A GOOD IDE SHOULD BRING

Good support for code editing (hilighting/folding)

Improve productivity (completions/tools/refactoring)

WHAT A GOOD IDE SHOULD BRING

Good support for code editing (hilighting/folding)

Improve productivity (completions/tools/refactoring)

Smart and quick feedbacks (diagnosis, hints)

WHAT A GOOD IDE SHOULD BRING

Good support for code editing (hilighting/folding)

Improve productivity (completions/tools/refactoring)

Smart and quick feedbacks (diagnosis, hints)

Tools orchestration and project management

WHAT A GOOD IDE SHOULD BRING

Good support for code editing (hilighting/folding)

Improve productivity (completions/tools/refactoring)

Smart and quick feedbacks (diagnosis, hints)

Tools orchestration and project management

Project discovery (Code Navigation)

WHAT A GOOD IDE SHOULD BRING

Good support for code editing (hilighting/folding)

Improve productivity (completions/tools/refactoring)

Smart and quick feedbacks (diagnosis, hints)

Tools orchestration and project management

Project discovery (Code Navigation)

Return valid information and mutations

NAIVE EDITOR TIMELINE

Configurable
TextBased
Editors

ACME

NAIVE EDITOR TIMELINE

Configurable
TextBased
Editors

NAIVE EDITOR TIMELINE

ACME

Fully specialized
IDE

NAIVE EDITOR TIMELINE

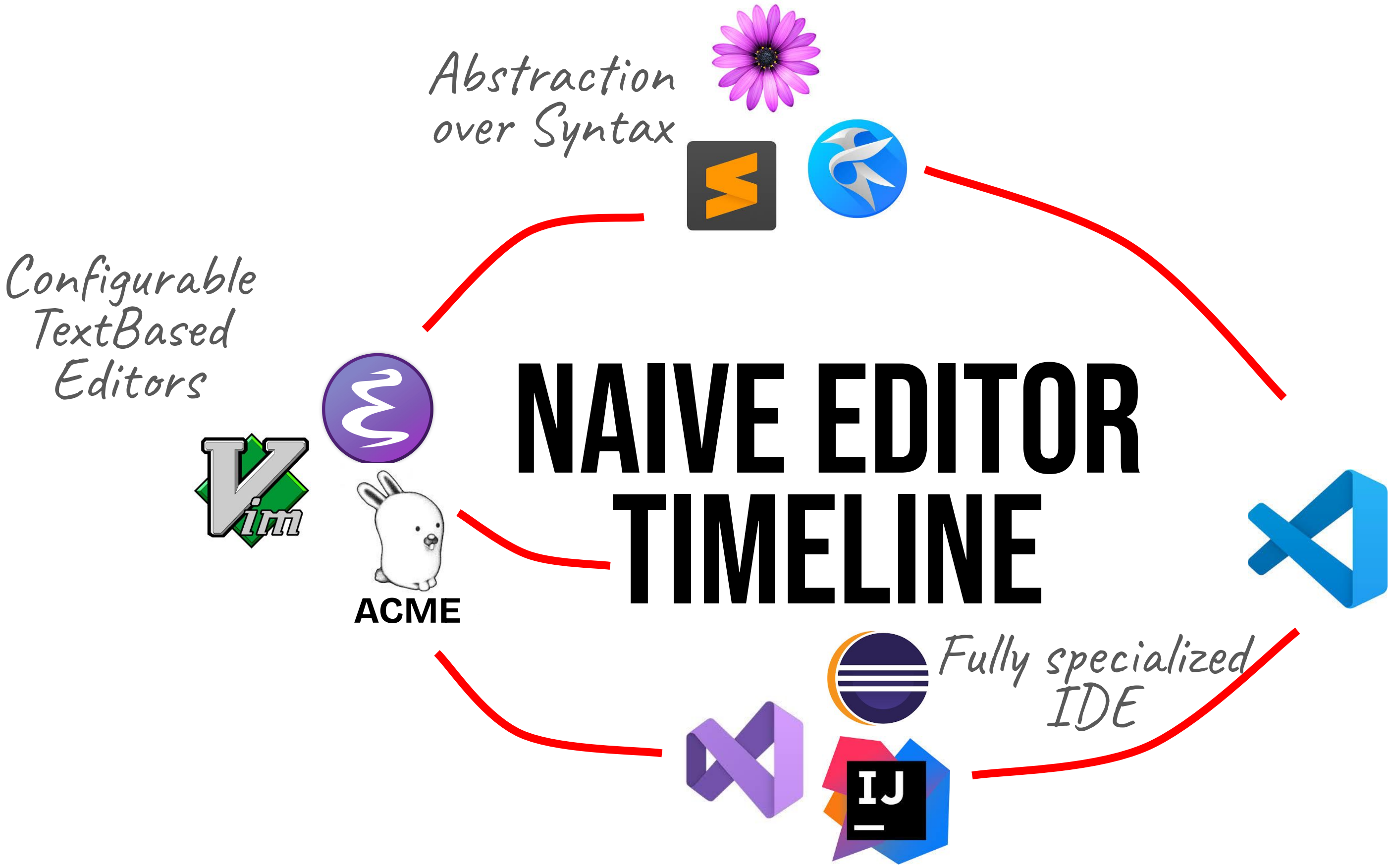
Configurable TextBased Editors

ACME

Abstraction over Syntax

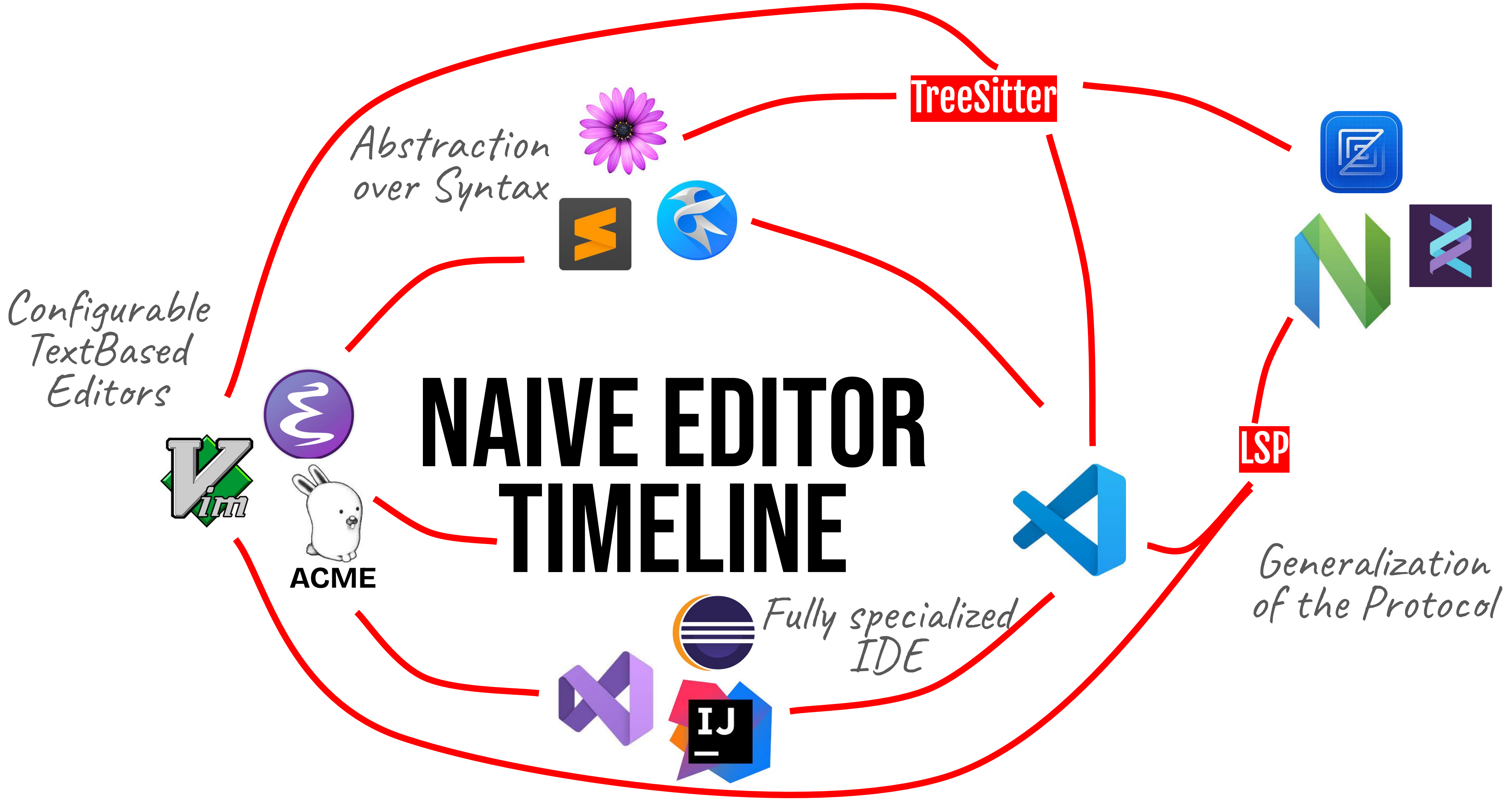
Fully specialized IDE

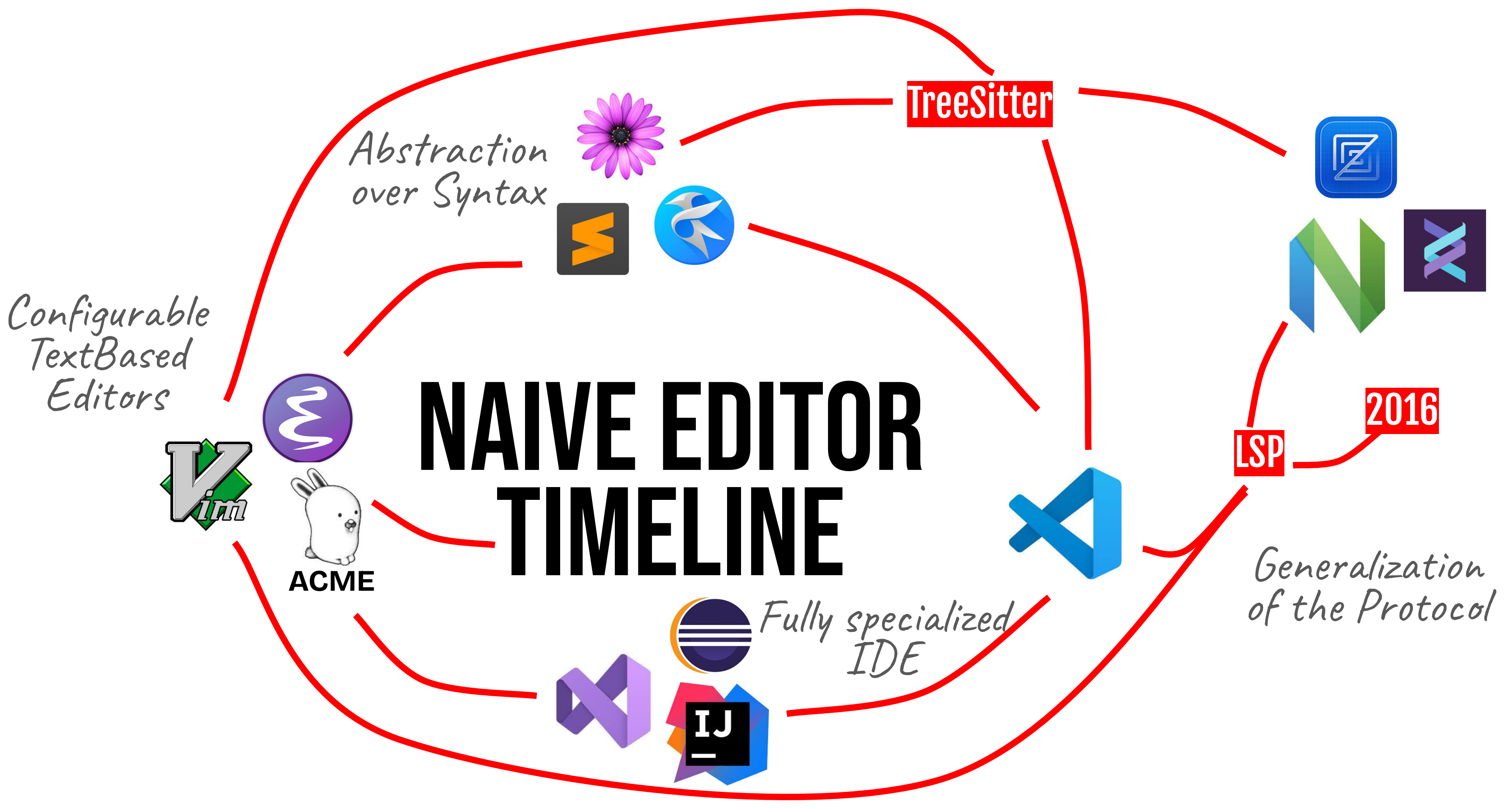
NAIVE EDITOR TIMELINE

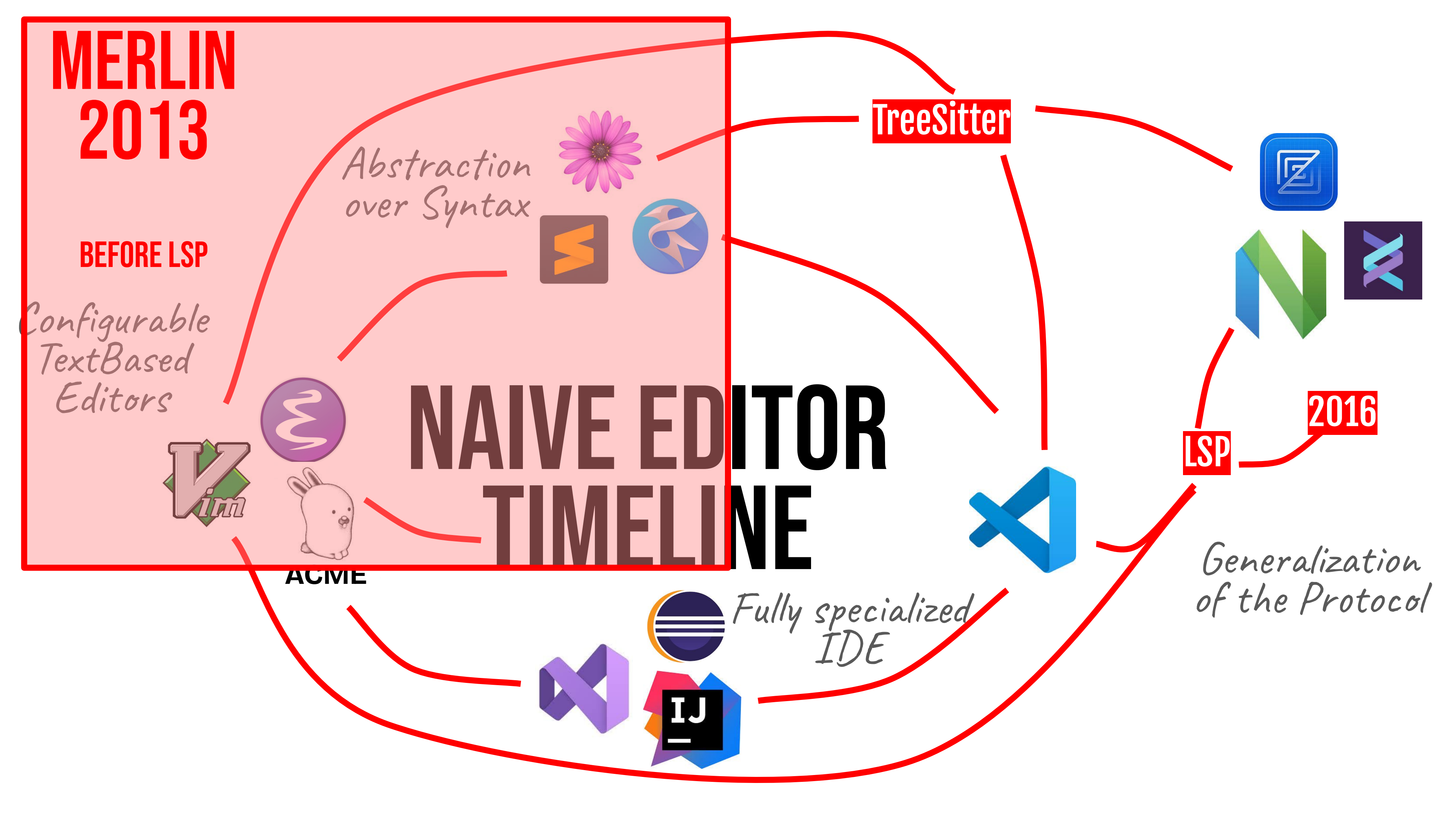



NAIVE EDITOR TIMELINE
Configurable TextBased Editors
ACME
Abstraction over Syntax
TreeSitter
LSP
Generalization of the Protocol
Fully specialized IDE
IJ


NAIVE EDITOR TIMELINE
Configurable TextBased Editors
ACME
Abstraction over Syntax
TreeSitter
LSP
Generalization of the Protocol
Fully specialized IDE
IJ


NAIVE EDITOR TIMELINE
Configurable TextBased Editors
ACME
Abstraction over Syntax
TreeSitter
LSP
Generalization of the Protocol
Fully specialized IDE
IJ


NAIVE EDITOR TIMELINE
Configurable TextBased Editors
Abstraction over Syntax
TreeSitter
ACME
Fully specialized IDE
IJ
LSP
2016
Generalization of the Protocol


MERLIN
2013
BEFORE LSP
Configurable TextBased Editors
Abstraction over Syntax
TreeSitter
NAIVE EDITOR
TIMELINE
ACME
Fully specialized IDE
LSP
2016
Generalization of the Protocol
IJ

SO WHAT IS A LANGUAGE SERVER?

A daemon that receives text buffer
and user queries

SO WHAT IS A LANGUAGE SERVER?

GLOBAL ARCHITECTURE

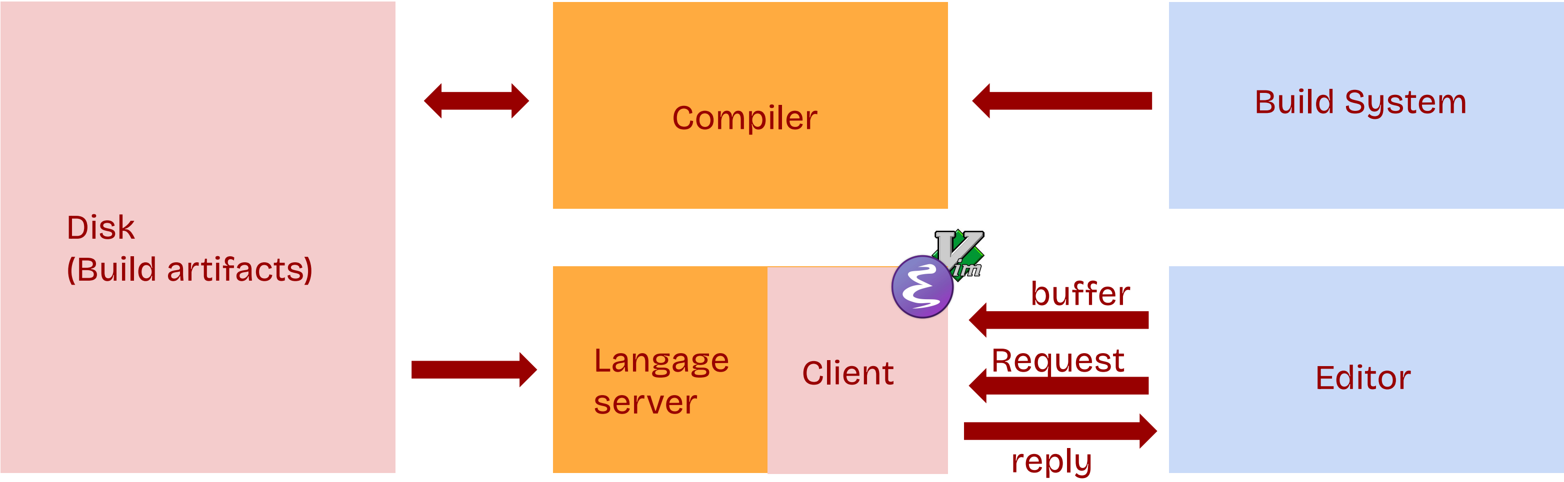
Documented in

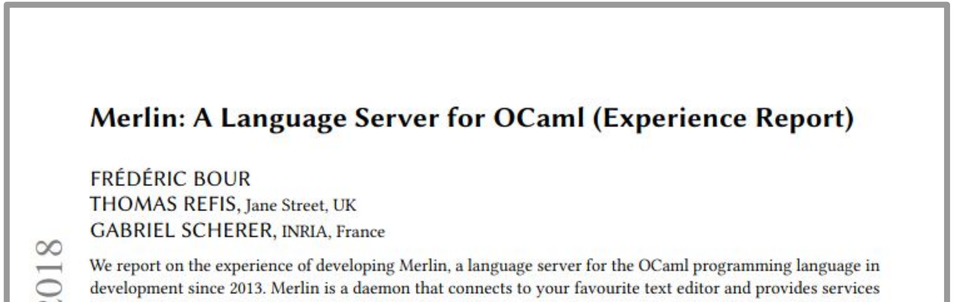
Merlin: A Language Server for OCaml (Experience Report)

FRÉDÉRIC BOUR
THOMAS REFIS, Jane Street, UK
GABRIEL SCHERER, INRIA, France

2018

We report on the experience of developing Merlin, a language server for the OCaml programming language in development since 2013. Merlin is a daemon that connects to your favourite text editor and provides services

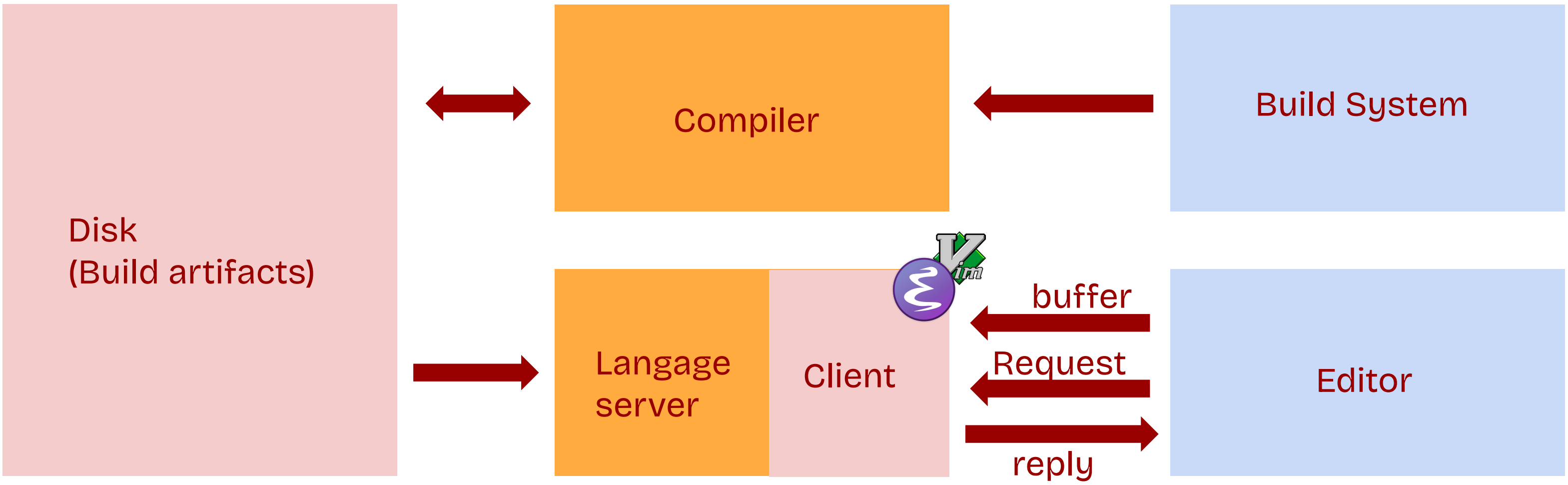




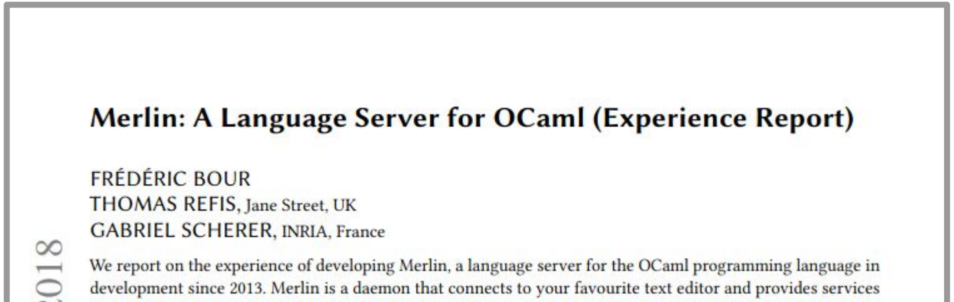
Merlin: A Language Server for OCaml (Experience Report)

FRÉDÉRIC BOUR
THOMAS REFIS, Jane Street, UK
GABRIEL SCHERER, INRIA, France

We report on the experience of developing Merlin, a language server for the OCaml programming language in development since 2013. Merlin is a daemon that connects to your favourite text editor and provides services



Include an
Incremental and Partial
language frontend
(Parser+ Typechecker)

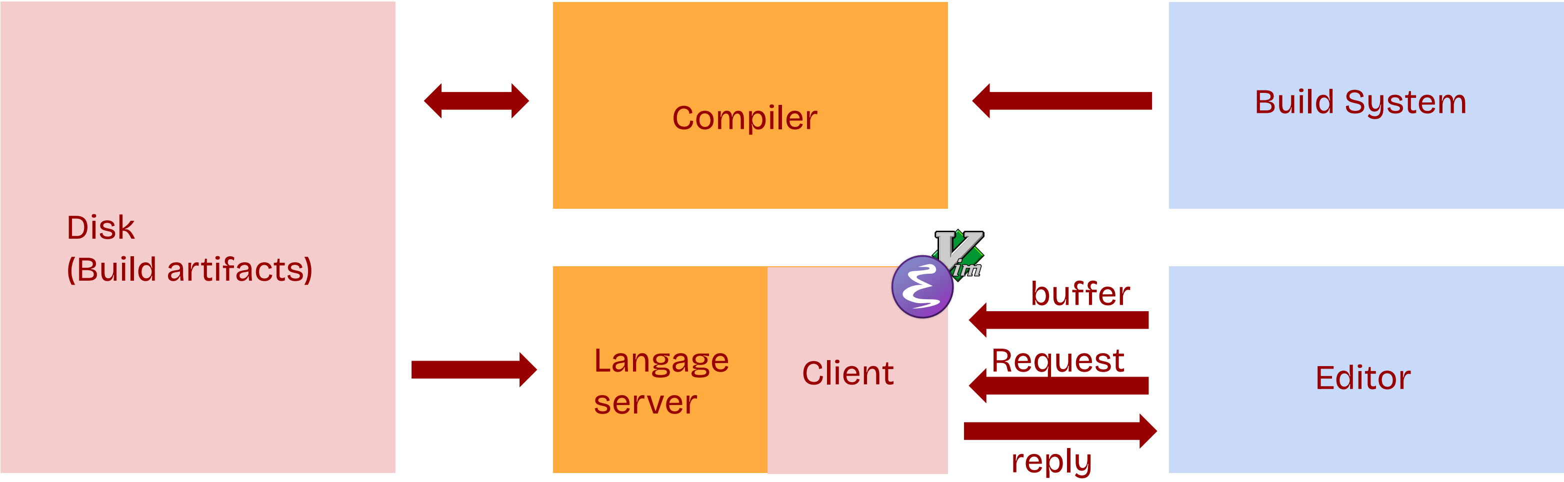


Merlin: A Language Server for OCaml (Experience Report)

FRÉDÉRIC BOUR
THOMAS REFIS, Jane Street, UK
GABRIEL SCHERER, INRIA, France

We report on the experience of developing Merlin, a language server for the OCaml programming language in development since 2013. Merlin is a daemon that connects to your favourite text editor and provides services

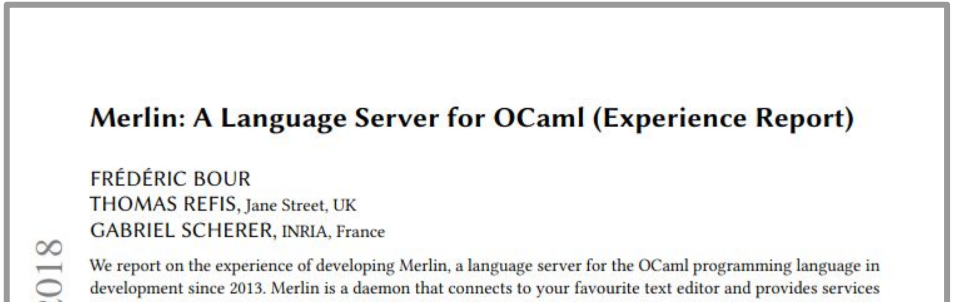
2018



Include an **Incremental and Partial** language frontend (Parser+ Typechecker)

deal with **missing** or **wrong** part

only recomputed **changed part**

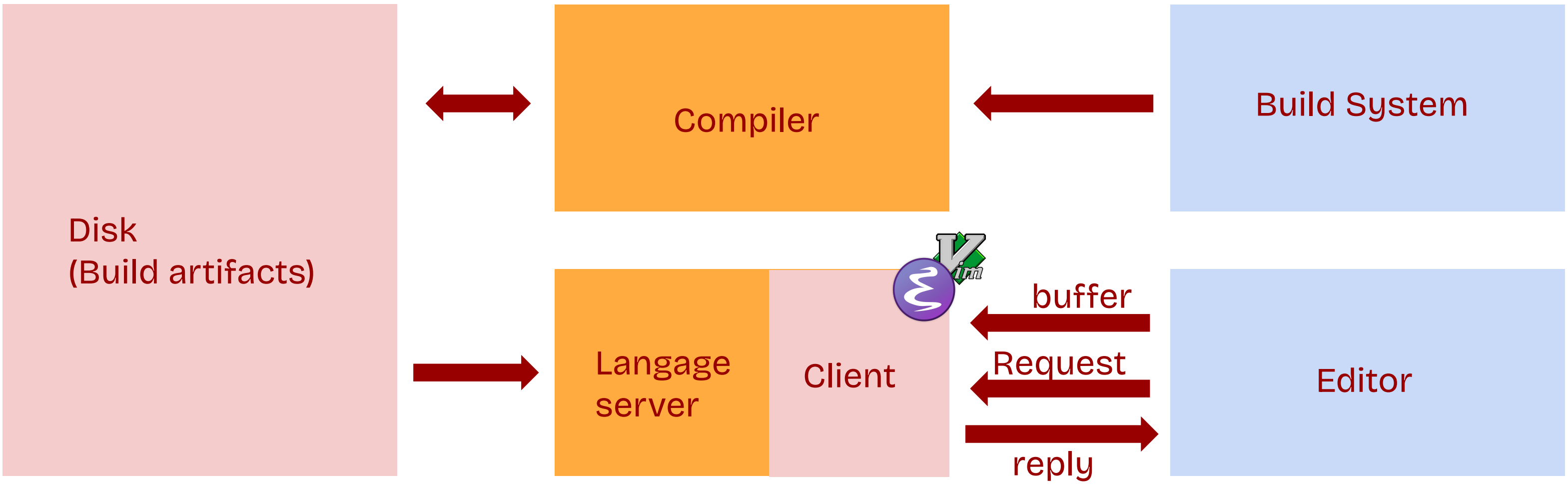


Merlin: A Language Server for OCaml (Experience Report)

FRÉDÉRIC BOUR
THOMAS REFIS, Jane Street, UK
GABRIEL SCHERER, INRIA, France

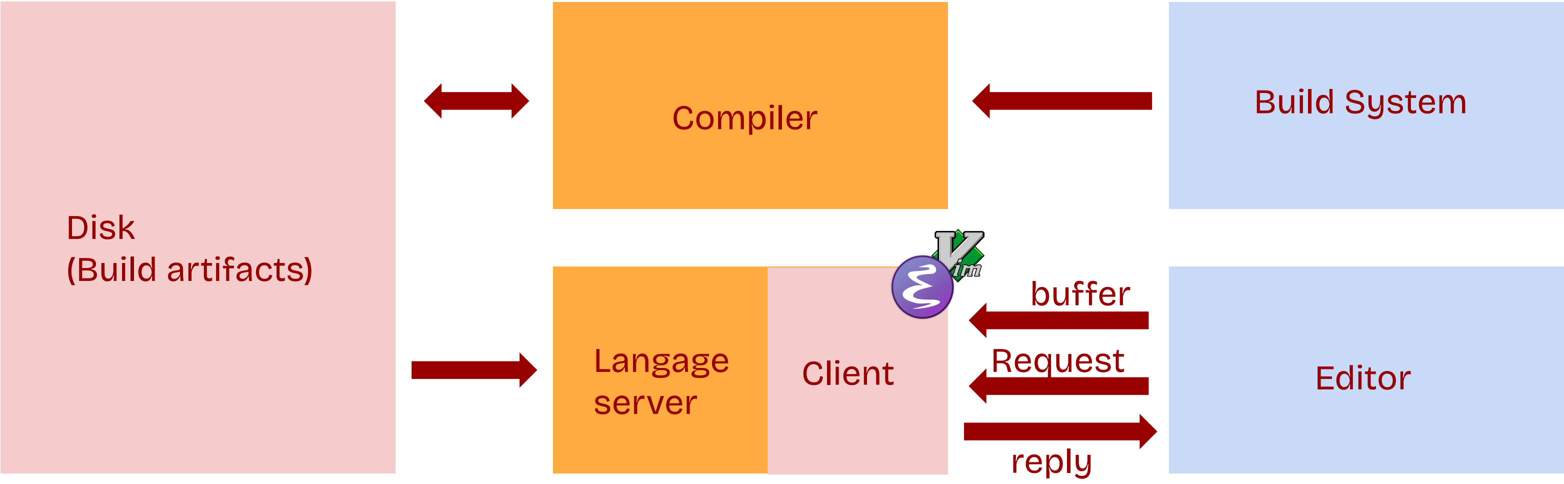
2018

We report on the experience of developing Merlin, a language server for the OCaml programming language in development since 2013. Merlin is a daemon that connects to your favourite text editor and provides services



vendoring and **adapting** the existing OCaml toolchain

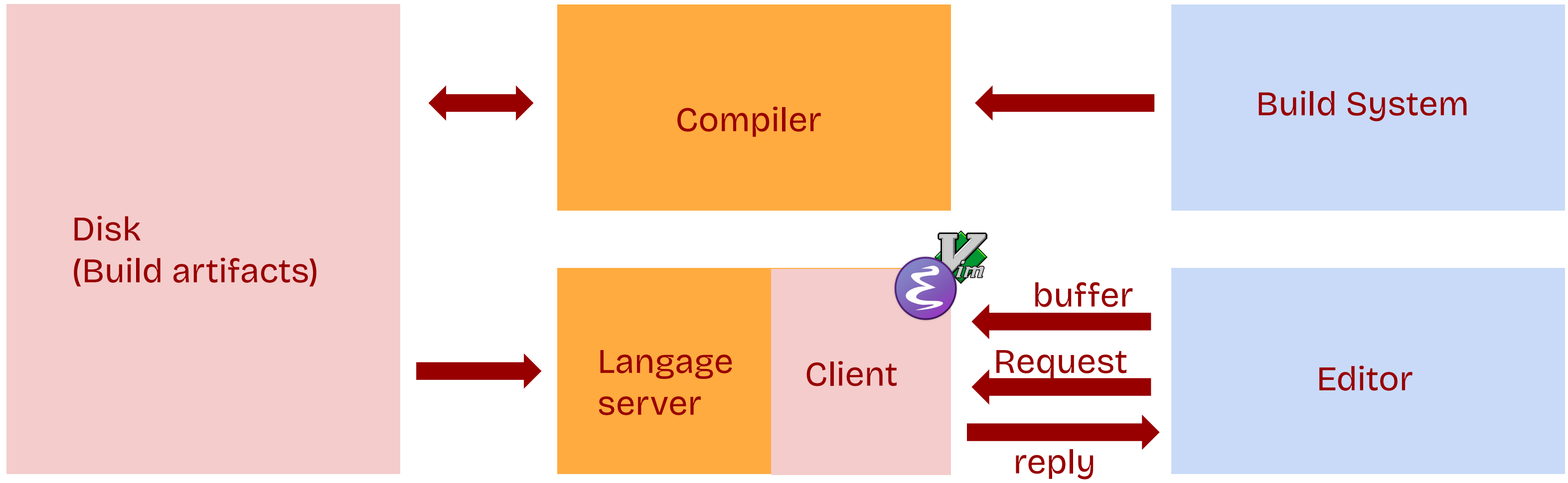
- making everything **pure** or **backtrackable**
- pure **lexer**
- pure **parser**
- **parsing error recovery** (faking data when it is missing)
- typechecking already has backtracking



purity unlock **memoization** for incrementality

vendoring and **adapting** the existing OCaml toolchain

- making everything **pure** or **backtrackable**
- pure **lexer**
- pure **parser**
- **parsing error recovery** (faking data when it is missing)
- typechecking already has backtracking

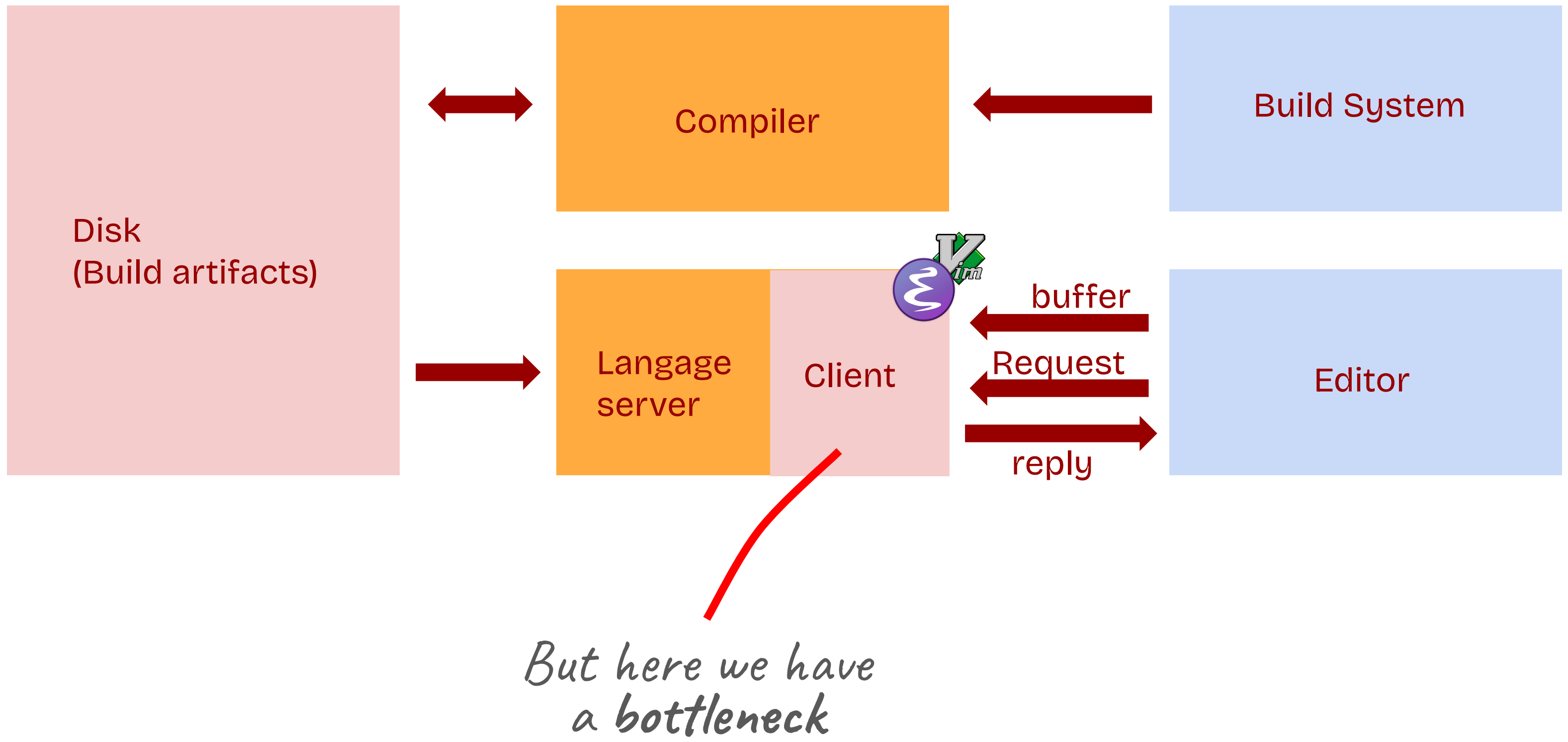


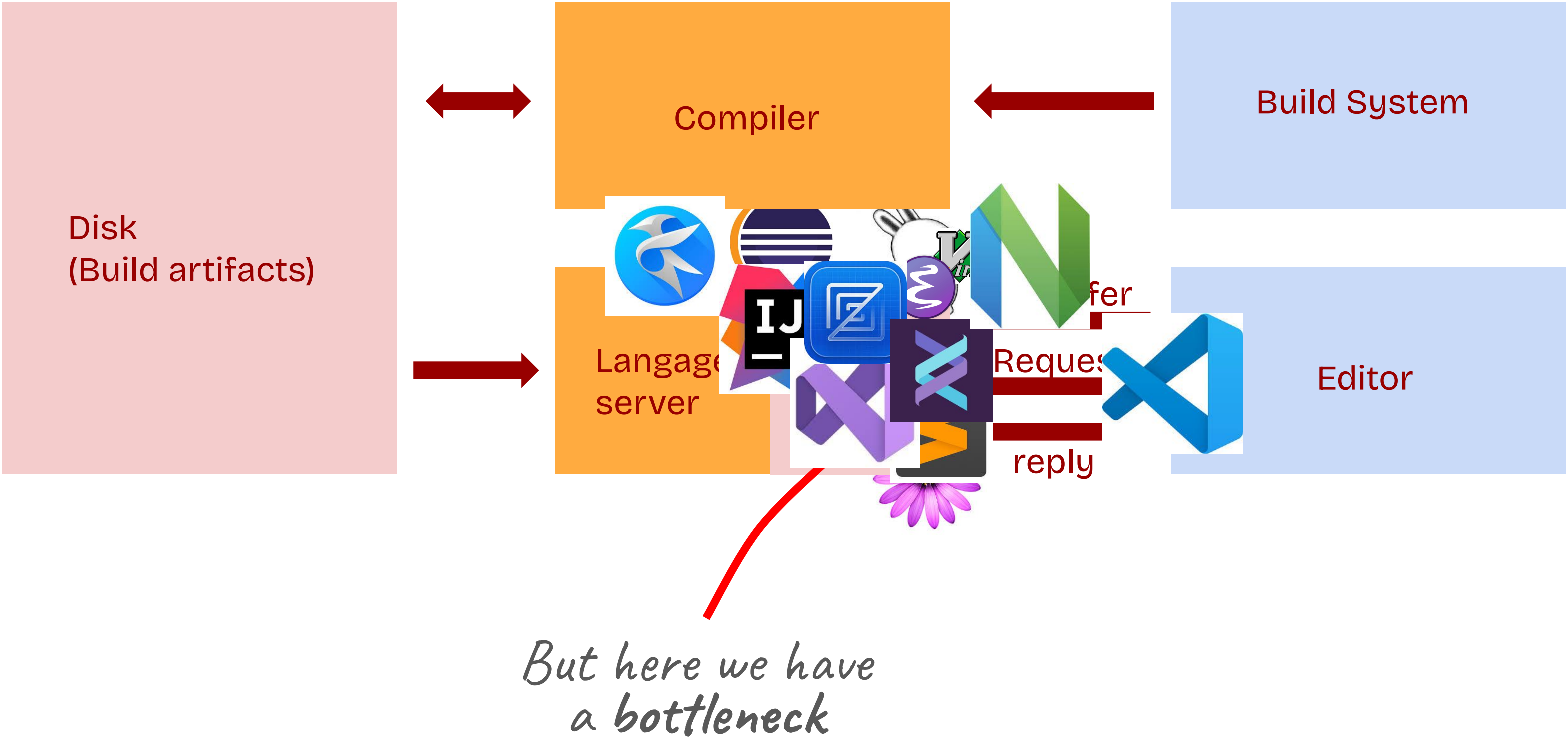
purity unlock **memoization** for incrementality

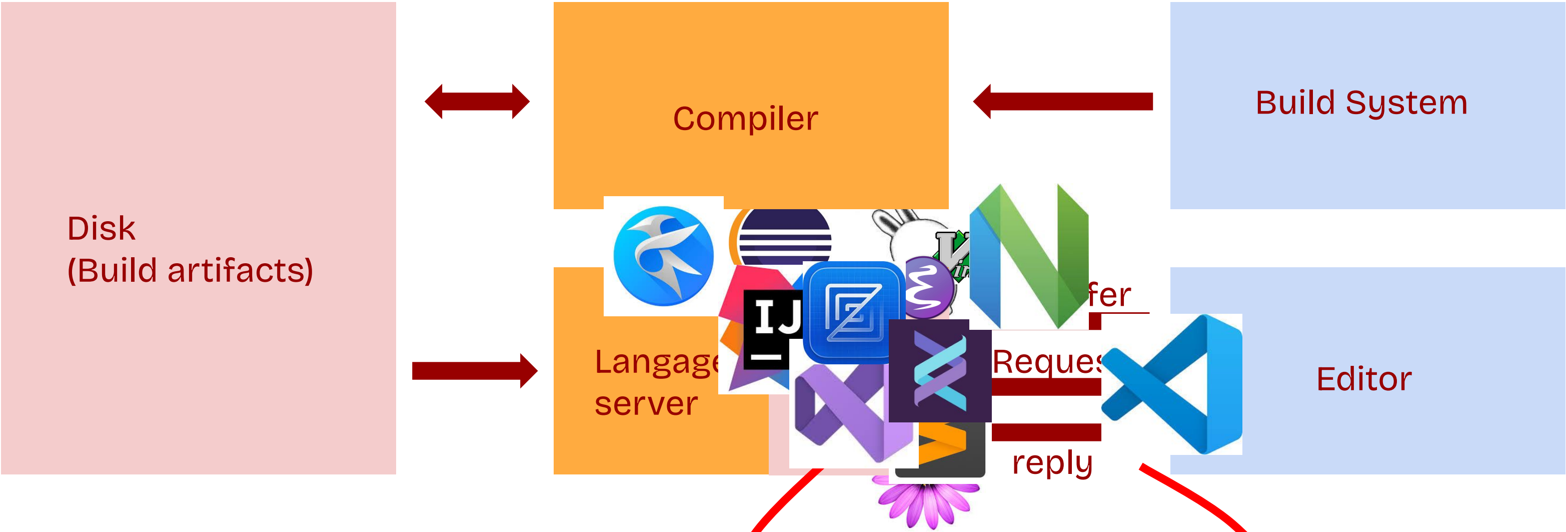
The paper also describes cool applications of static information about grammar (**error recovery, error messages, multi grammar**, **local GLR parser emulation**)

vendoring and **adapting** the existing OCaml toolchain

- making everything **pure** or **backtrackable**
- pure **lexer**
- pure **parser**
- **parsing error recovery** (faking data when it is missing)
- typechecking already has backtracking

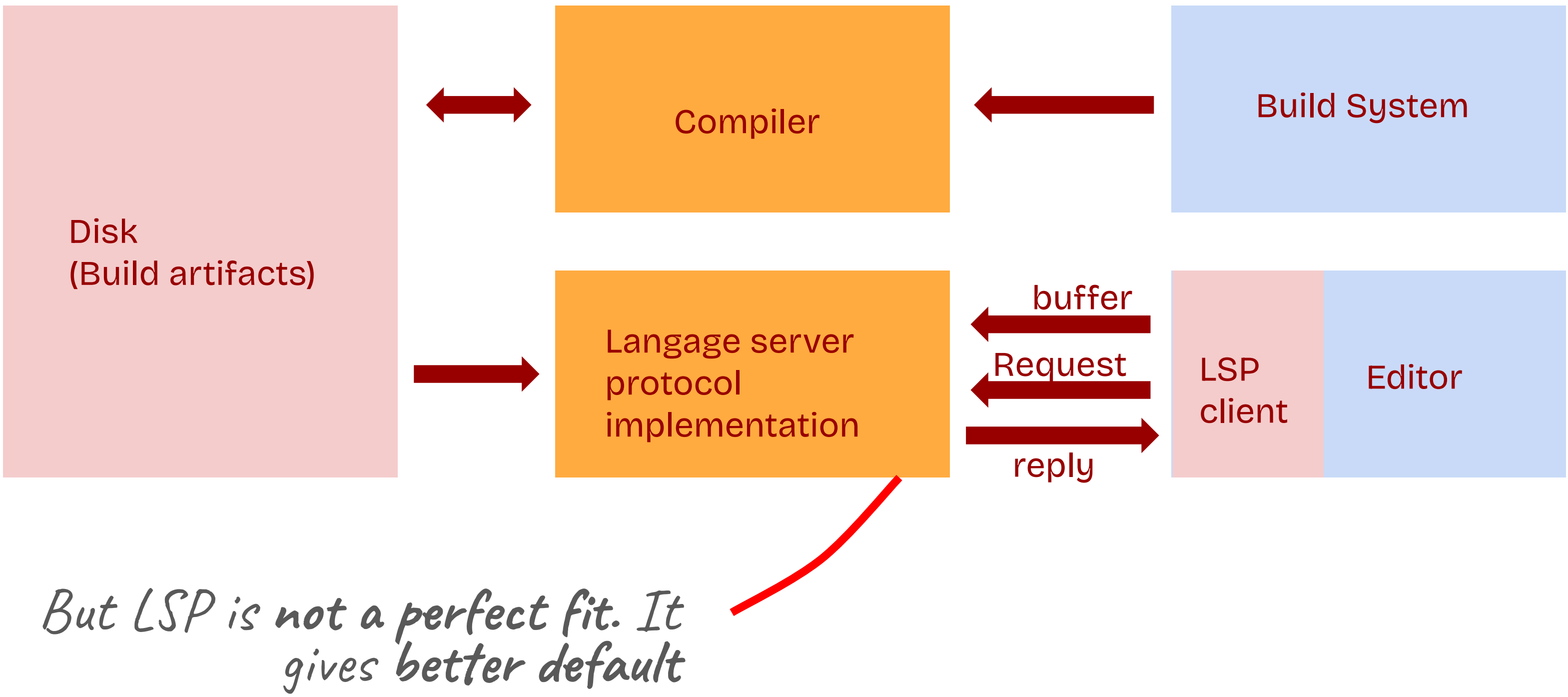

Compiler
Build System
Disk
(Build artifacts)
Langage
server
Client
buffer
Request
reply
Editor
But here we have
a bottleneck


Compiler
Build System
Disk
(Build artifacts)
Langage
server
fer
Reques
reply
Editor
But here we have
a bottleneck



But here we have
a **bottleneck**

this is why LSP
since a lot of editors **are**
shipped with an LSP Client


Compiler
Build System
Disk
(Build artifacts)
Langage server
protocol
implementation
buffer
Request
reply
LSP
client
Editor
But LSP is not a perfect fit. It
gives better default

easy to extend
Syntax highlighting
by default

Syntax Highlighting
and code folding by
default

Hard to extend
**Complicated
protocol**

ACME

LSP

Syntax Highlighting, **auto-complete**, **jump-to-definition**, **hints** and **hovers**, **project manipulation**, **advanced search** (and a proper parallel client/server + capabilities notion)

and more feature but that assumes class-based and statement based languages.

easy to extend Syntax highlighting by default

Syntax Highlighting and code folding by default

Hard to extend ***Complicated protocol***

ACME

Let's observe some feature that diverges from the ones supported ***out of the box***

AN HUGE MIGRATION FROM MERLIN SERVER TO LSP

AN HUGE MIGRATION FROM MERLIN SERVER TO LSP

*Let's observe some feature that diverges from the ones supported **out of the box***

via **Code Action**

Contextual triggerable action on the document

via **Custom Request**

Lets the client implementing the reaction

AN HUGE MIGRATION FROM MERLIN SERVER TO LSP

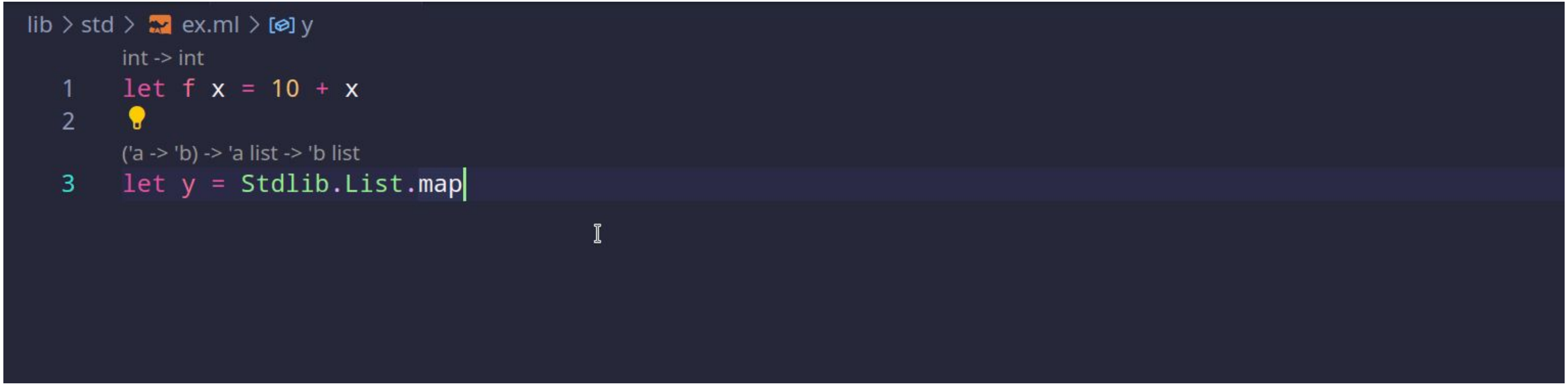
Let's observe some feature that diverges from the ones supported ***out of the box***

via ***Code Action***
Contextual triggerable action on the document

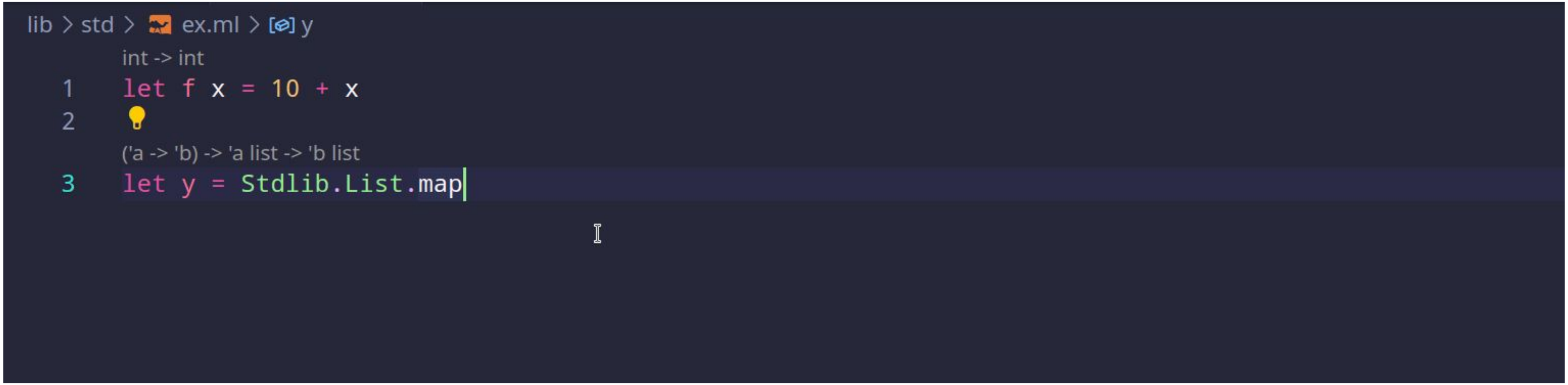
via ***Custom Request***
Lets the client implementing the reaction

that need ***SPECIFIC IMPLEMENTATION***
The comeback of our potential ***Bottleneck***

TYPES AND DOCUMENTATION



TYPES AND DOCUMENTATION



TYPES AND DOCUMENTATION

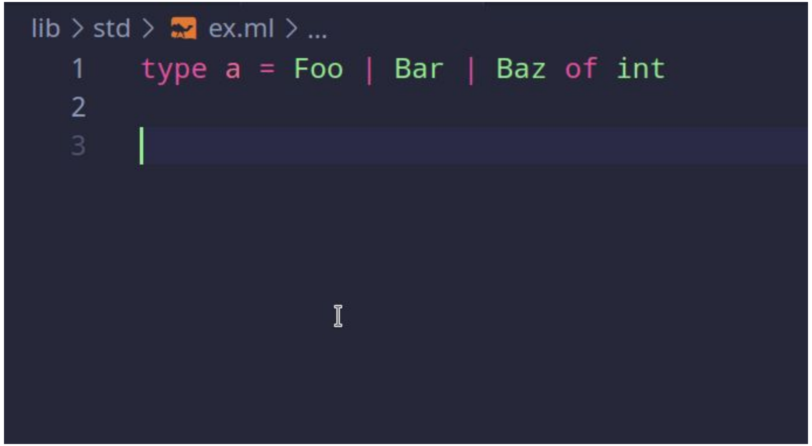
*Available via an **Hover Prodiver** and **Inlay Hints***

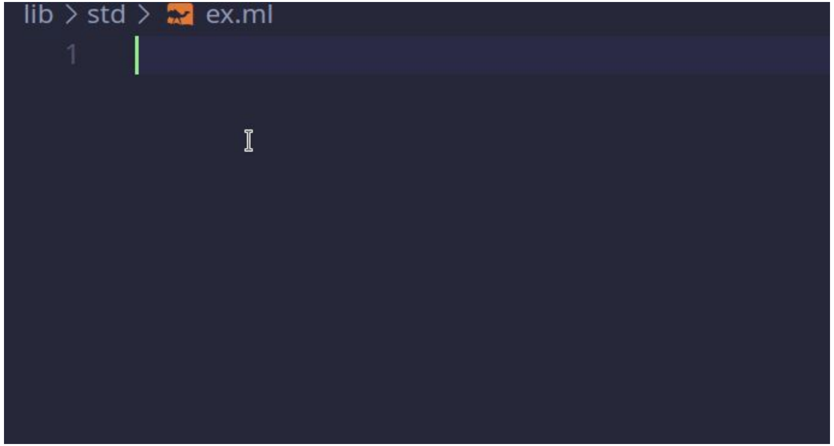
CASE ANALYSIS
AND CONSTRUCT EXPRESSION

lib > std > ex.ml > ...

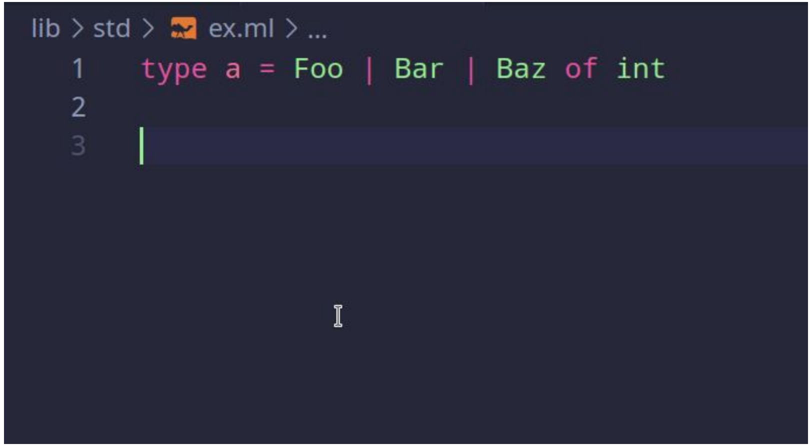
```
type a = Foo | Bar | Baz of int

```

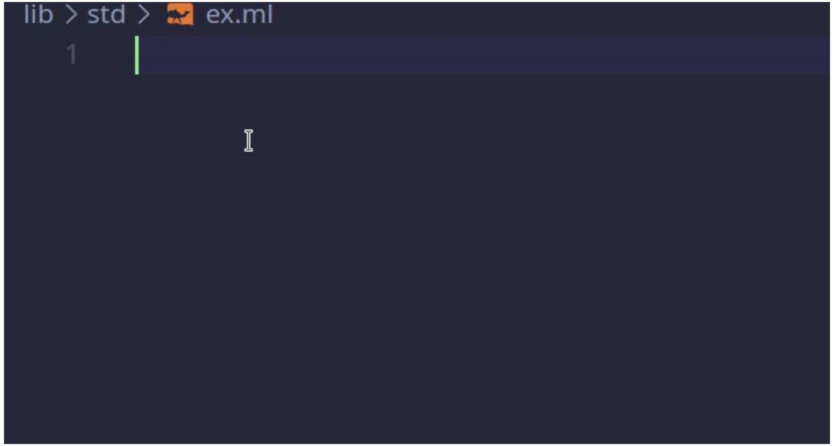
CASE ANALYSIS AND CONSTRUCT EXPRESSION





CASE ANALYSIS AND CONSTRUCT EXPRESSION





Available via ***Code Action*** *and* ***Completion***

CASE ANALYSIS AND CONSTRUCT EXPRESSION

OPEN REFACTORING

lib > std > ex.ml > ...

```
module Foo = struct
  let a = 10
  let b =11
  let c = 12

end


let x = Foo.a + Foo.b + Foo.c
```

OPEN REFACTORING

lib > std > ex.ml > ...

```
module Foo = struct
    int
    let a = 10
    int
    let b =11
    int
    let c = 12

end


int
let x = Foo.a + Foo.b + Foo.c
```

Available via a ***Code Action***

OPEN REFACTORING

SOURCE NAVIGATION

- Jump to the **prev** or **next** phrase (toplevel-definition)
- Switch from implementation to interface and vice-versa
- Jump to **fun/let/module/match**

SOURCE NAVIGATION

SOURCE NAVIGATION

- Jump to the **prev** or **next** phrase (toplevel-definition) — Available via a **Code Action**
- Switch from implementation to interface and vice-versa — Available via a **Code Action**
- Jump to **fun/let/module/match** — Available via a **Code Action**

- Jump to the **prev** or **next** phrase (toplevel-definition)
- Switch from implementation to interface and vice-versa
- Jump to **fun/let/module/match**

Available via a ***Code Action***

Available via a ***Code Action***

Available via a ***Code Action***

BUT

Highly pollutes the 'code-action' menu

No nesting/grouping in the protocol!

SOURCE NAVIGATION

- Jump to the **prev** or **next** phrase (toplevel-definition)

*Available via a **Code Action***

- Switch from implementation to interface and vice-versa

*Available via a **Code Action***

- Jump to **fun/let/module/match**

*Available via a **Code Action***

BUT

Highly pollutes the 'code-action' menu

No nesting/grouping in the protocol!

*Moving to a **Custom Request***

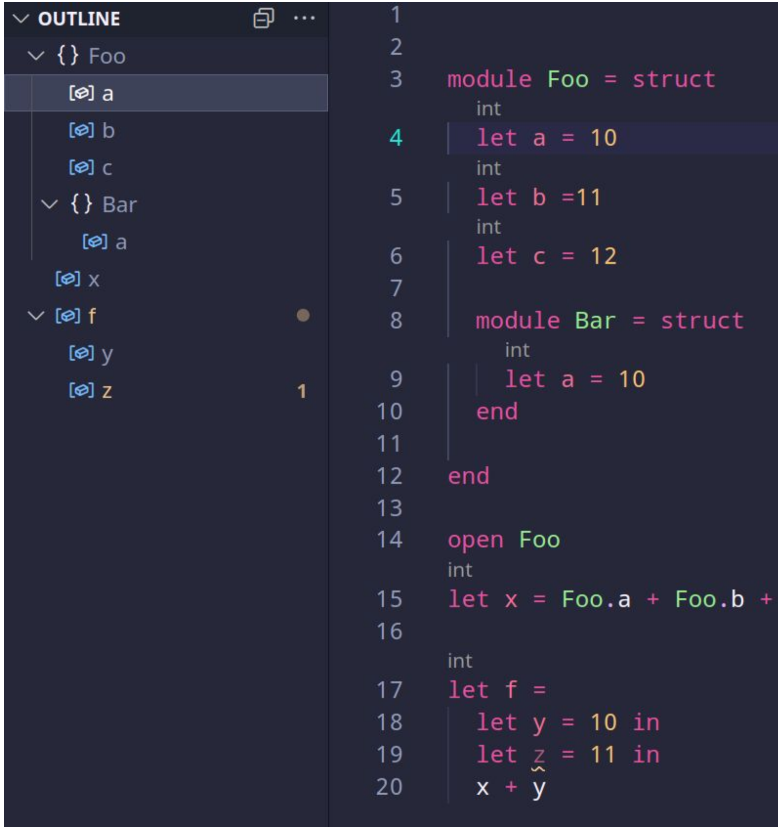
SOURCE NAVIGATION

OUTLINES AND CODE STRUCTURE

Works well for outlines but not for document navigation

OUTLINES AND CODE STRUCTURE

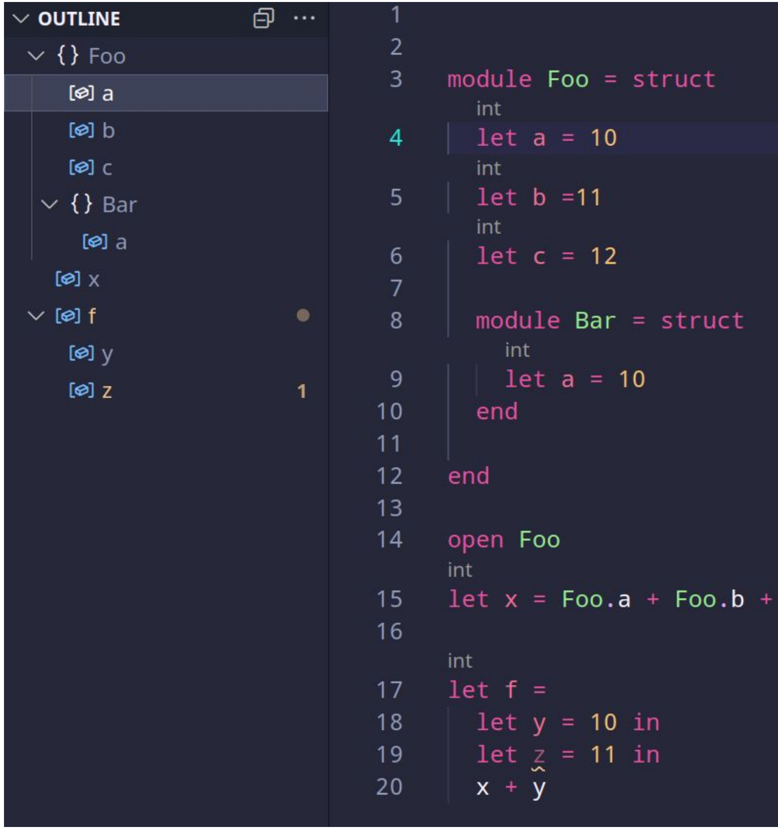
Works well for outlines but not for document navigation



OUTLINES AND CODE STRUCTURE

Assumes that **all languages are TypeScript-like** (in Outline Kind)

Works well for outlines but not for document navigation



OUTLINES AND CODE STRUCTURE

TYPE-ENCLOSING

TYPE-ENCLOSING

one of the **main feature** of
Merlin

```
type t =
 { result : int
 ; job_done : bool
 }
let f x =
 let init_value = x + 1 in
 let z =
   Stdlib.List.fold_left
     (fun acc x -> acc + int_of_string x)
     init_value
     [ "1"; "2"; "3"; "4" ]
 in
 { result = z; job_done = true }
let result = f 10
```

TYPE-ENCLOSING

one of the main feature of Merlin

```ocaml
type t =
 { result : int
 ; job_done : bool
 }
let f x =
 let init_value = x + 1 in
 let z =
   Stdlib.List.fold_left
     (fun acc x -> acc + int_of_string x)
     init_value
     [ "1"; "2"; "3"; "4" ]
 in
 { result = z; job_done = true }
let result = f 10
```

The "result" has type *t*

TYPE-ENCLOSING

one of the **main feature** of Merlin

```
type t =
 { result : int
 ; job_done : bool
 }
let f x =
 let init_value = x + 1 in
 let z =
   Stdlib.List.fold_left
     (fun acc x -> acc + int_of_string x)
     init_value
     [ "1"; "2"; "3"; "4" ]
 in
 { result = z; job_done = true }
let result = f 10
```

but outside of the module it hard to guess what is really **t**

The "result" has type **t**

TYPE-ENCLOSING

one of the **main feature** of Merlin

```ocaml
type t =
 { result : int
 ; job_done : bool
 }
let f x =
 let init_value = x + 1 in
 let z =
   Stdlib.List.fold_left
     (fun acc x -> acc + int_of_string x)
     init_value
     [ "1"; "2"; "3"; "4" ]
 in
 { result = z; job_done = true }
let result = f 10
```

so we want to **increase** the verbosity (in Emacs + Merlin we can re-call the feature to get the following definition)

but outside of the module it hard to guess what is really **t**

The "result" has type **t**

TYPE-ENCLOSING

one of the **main feature** of Merlin

```
type t =
 { result : int
 ; job_done : bool
 }
let f x =
 let init_value = x + 1 in
 let z =
   Stdlib.List.fold_left
     (fun acc x -> acc + int_of_string x)
     init_value
     [ "1"; "2"; "3"; "4" ]
 in
 { result = z; job_done = true }
let result = f 10
```

An other useful feature is to **grow** and **shrink** the observable enclosing.

TYPE-ENCLOSING

one of the **main feature** of Merlin

```ocaml
type t =
 { result : int
 ; job_done : bool
 }
let f x =
 let init_value = x + 1 in
 let z =
   Stdlib.List.fold_left
     (fun acc x -> acc + int_of_string x)
     init_value
     [ "1"; "2"; "3"; "4" ]
 in
 { result = z; job_done = true }
let result = f 10
```

An other useful feature is to **grow** and **shrink** the observable enclosing.

('acc -> 'b -> 'acc) -> 'acc -> 'b list -> 'acc

TYPE-ENCLOSING

one of the **main feature** of Merlin

```ocaml
type t =
 { result : int
 ; job_done : bool
 }
let f x =
 let init_value = x + 1 in
 let z =
   Stdlib.List.fold_left
     (fun acc x -> acc + int_of_string x)
     init_value
     [ "1"; "2"; "3"; "4" ]
 in
 { result = z; job_done = true }
let result = f 10
```

An other useful feature is to **grow** and **shrink** the observable enclosing.

[int -> string -> int] -> int -> string list -> int

TYPE-ENCLOSING

one of the **main feature** of Merlin

```ocaml
type t =
 { result : int
 ; job_done : bool
 }
let f x =
 let init_value = x + 1 in
 let z =
   Stdlib.List.fold_left
     (fun acc x -> acc + int_of_string x)
     init_value
     [ "1"; "2"; "3"; "4" ]
 in
 { result = z; job_done = true }
let result = f 10
```

int

An other useful feature is to **grow** and **shrink** the observable enclosing.

TYPE-ENCLOSING

one of the **main feature** of Merlin

```
type t =
 { result : int
 ; job_done : bool
 }
let f x =
 let init_value = x + 1 in
 let z =
   Stdlib.List.fold_left
     (fun acc x -> acc + int_of_string x)
     init_value
     [ "1"; "2"; "3"; "4" ]
 in
 { result = z; job_done = true }
let result = f 10
```

An other useful feature is to ***grow*** *and* ***shrink*** *the observable enclosing.*

t : { result: int; job_done: bool }

TYPE-ENCLOSING

one of the ***main feature*** *of Merlin*

```
type t =
 { result : int
 ; job_done : bool
 }
let f x =
 let init_value = x + 1 in
 let z =
   Stdlib.List.fold_left
     (fun acc x -> acc + int_of_string x)
     init_value
     [ "1"; "2"; "3"; "4" ]
 in
 { result = z; job_done = true }
let result = f 10
```

An other useful feature is to **grow** and **shrink** the observable enclosing.

t : { result: int; job_done: bool }

TYPE-ENCLOSING

one of the **main feature** of Merlin

```ocaml
type t =
  { result : int
  ; job_done : bool
  }
let f x =
  let init_value = x + 1 in
  let z =
    Stdlib.List.fold_left
      (fun acc x -> acc + int_of_string x)
      init_value
      [ "1"; "2"; "3"; "4" ]
  in
  { result = z; job_done = true }
let result = f 10
```

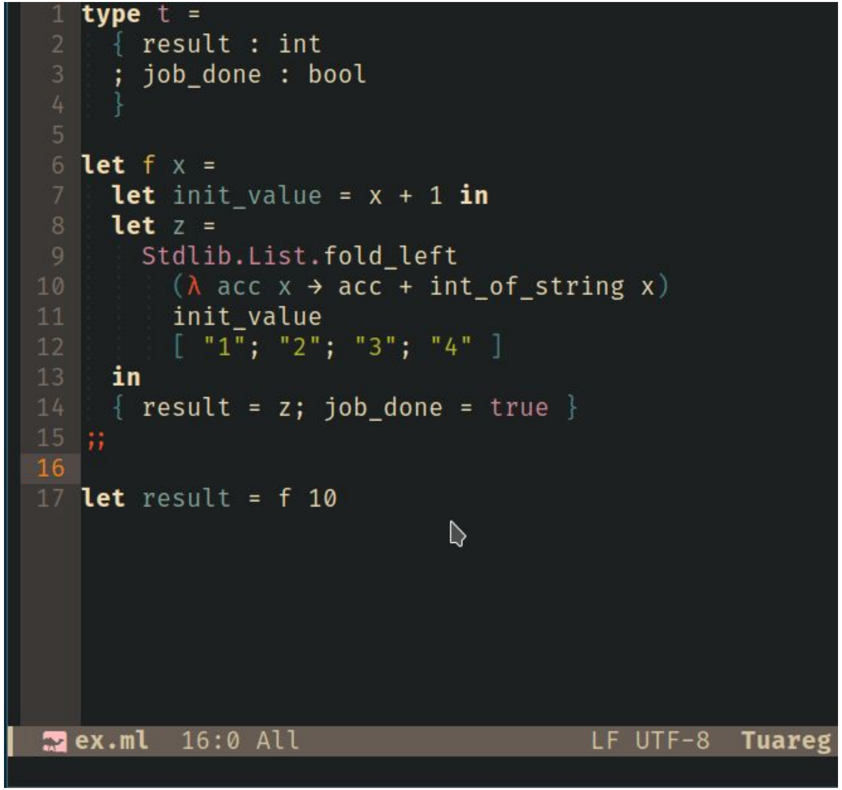
An other useful feature is to **grow and shrink** the observable enclosing.

int -> t

TYPE-ENCLOSING

one of the **main feature** of Merlin

```ocaml
type t =
 { result : int
 ; job_done : bool
 }
let f x =
 let init_value = x + 1 in
 let z =
   Stdlib.List.fold_left
     (fun acc x -> acc + int_of_string x)
     init_value
     [ "1"; "2"; "3"; "4" ]
 in
 { result = z; job_done = true }
let result = f 10
```



TYPE-ENCLOSING

one of the main feature of Merlin

Custom Request +
Stateful management on
the client-side

```ocaml
type t =
  { result : int
  ; job_done : bool
  }

let f x =
  let init_value = x + 1 in
  let z =
    Stdlib.List.fold_left
      (λ acc x → acc + int_of_string x)
      init_value
      [ "1"; "2"; "3"; "4" ]
  in
  { result = z; job_done = true }
;;

let result = f 10
```

ex.ml 16:0 All LF UTF-8 Tuareg

TYPE-ENCLOSING

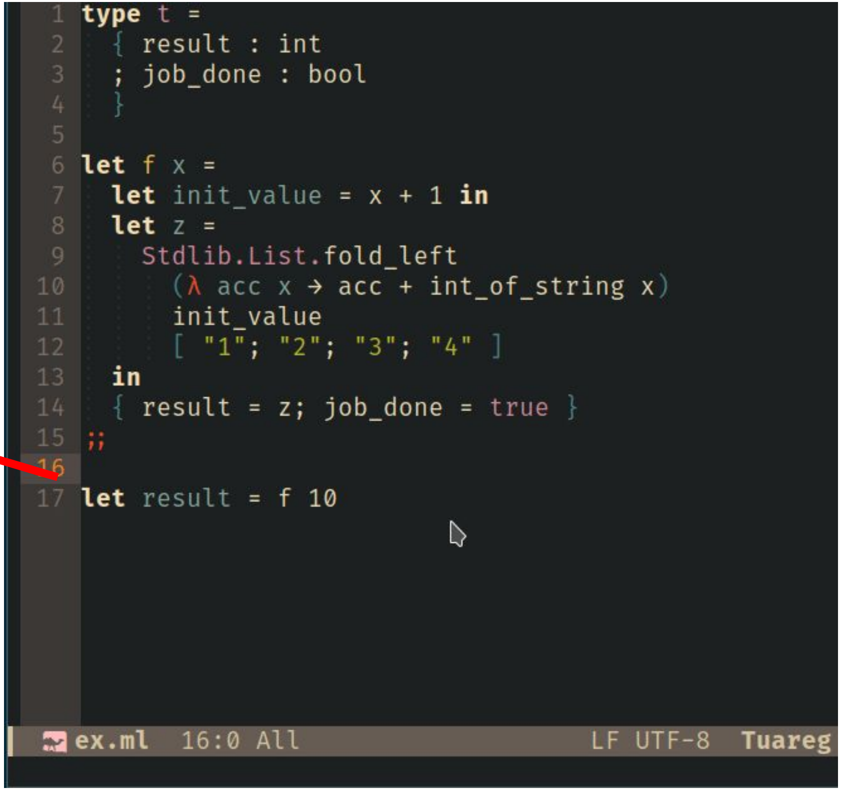
one of the main feature of Merlin

TYPE-ENCLOSING

one of the ***main feature*** *of Merlin*

Custom Request + Stateful management on the client-side

Can't really hook the ***Hover Provider***



LSP MAKES A LOT OF THINGS SIMPLER

LSP MAKES A LOT OF THINGS SIMPLER

*but we still need dedicated clients to handle **custom requests***

LSP MAKES A LOT OF THINGS SIMPLER

*but we still need dedicated clients to handle **custom requests***

*Implementation of every dedicated requests on LSP side + a **tunneling request***

for client independence. **Already used by NeoVim**

Start providing **canonical implementation** for Vim, Emacs and VSCode

LSP MAKES A LOT OF THINGS SIMPLER

but we still need dedicated clients to handle **custom requests**

Implementation of every dedicated requests on LSP side + a **tunneling request**

for client independence. **Already used by NeoVim**

the UI of VSCode is **surprisingly hard** to extend properly.

Start providing **canonical implementation** for Vim, Emacs and VSCode

LSP MAKES A LOT OF THINGS SIMPLER

but we still need dedicated clients to handle **custom requests**

Implementation of every dedicated requests on LSP side + a **tunneling request**

for client independence. **Already used by NeoVim**

LSP MAKES A LOT OF THINGS SIMPLER

the UI of VSCode is **surprisingly hard** to extend properly.

Start providing **canonical implementation** for Vim, Emacs and VSCode

recentlu released!
OCaml-eglot

but we still need dedicated clients to handle **custom requests**

Implementation of every dedicated requests on LSP side + a **tunneling request**

for client independence. **Already used by NeoVim**

IMPLEMENTATION DETAILS

Where the fun begins

PROJECT-WIDE OCCURRENCES

return every usage of the selected identifier
across all of the project's source files

PROJECT-WIDE OCCURRENCES

return every usage of the selected identifier across all of the project's source files

Hard to achieve in presence of powerful module system and separate compilation

PROJECT-WIDE OCCURRENCES

return every usage of the selected identifier across all of the project's source files

Hard to achieve in presence of powerful module system and separate compilation

PROJECT-WIDE OCCURRENCES

Module Shapes for Modern Tooling

Ulysse Gérard, Thomas Refis, and Leo White

The ability to look up the definition of a variable is an essential feature of modern programming tooling. Beyond the simple code browsing action of jumping to that definition, it is a preliminary for more advanced tasks like fetching documentation or refactoring. The operation of finding a definition requires deep knowledge of a language's semantics to prevent finding erroneous positions in the presence of overlapping names, shadowed values, or complex features like module systems with includes and functor applications.

While imprecise results are tolerable for an *interactive* "jump to definition" use case, where the user can immediately assess the relevance of the definition the tool shows him

the source location of a declaration is therefore as simple as a lookup in the typing environment.

When trying to find the location of a definition however, there's no help to be had from the compiler. So a natural strategy that tools (e.g. merlin, rotor) can (and do) resort to is to *walk up* the typed AST, looking for the definition. On the example above, to find the definition of `N.x` we would first look for the module `N`, and then inspect its structure to find the definition of `x`.

We put an emphasis on walking "up", because, due to shadowing, the order in which the tree is visited matters. Consider a slight variation of the previous example where

OCaml's module system supports **aliases**, **includes**, and **(higher-order) functors**.

All of these make

finding any definition more complicated

EXAMPLE

LET'S FIND THE DEFINITION OF M.X

```
module type S = sig
 val x : int
end

module Identity (X : S) : S = X

module Simple = struct
 let x = 3
 let y = true
end
module Apply (Id : functor (_ : S) -> S)
             (X : S) = struct
 include Id (X)
end

module Alias = Simple
module M = Apply (Identity) (Alias)

let y = M.x
```

EXAMPLE

LET'S FIND THE DEFINITION OF M.X

```
module type S = sig
 val x : int
end

module Identity (X : S) : S = X

module Simple = struct
 let x = 3
 let y = true
end
module Apply (Id : functor (_ : S) -> S)
             (X : S) = struct
 include Id (X)
end

module Alias = Simple
module M = Apply (Identity) (Alias)

let y = M.x
```

- **M** is the result of applying **Apply**

EXAMPLE

LET'S FIND THE DEFINITION OF M.X

```
module type S = sig
 val x : int
end

module Identity (X : S) : S = X

module Simple = struct
 let x = 3
 let y = true
end
module Apply (Id : functor (_ : S) -> S)
             (X : S) = struct
 include Id (X)
end

module Alias = Simple
module M = Apply (Identity) (Alias)

let y = M.x
```

- **M** is the result of applying **Apply**
- Look up of the **Apply functor**

EXAMPLE

LET'S FIND THE DEFINITION OF M.X

```
module type S = sig
 val x : int
end


module Identity (X : S) : S = X

module Simple = struct
 let x = 3
 let y = true
end
module Apply (Id : functor (_ : S) -> S)
             (X : S) = struct
 include Id (X)
end

module Alias = Simple
module M = Apply (Identity) (Alias)

let y = M.x
```

- **M** is the result of applying **Apply**
- Look up of the **Apply functor**
- **x** Come frome the application of **Id**

EXAMPLE

LET'S FIND THE DEFINITION OF M.X

```
module type S = sig
 val x : int
end


module Identity (X : S) : S = X

module Simple = struct
 let x = 3
 let y = true
end
module Apply (Id : functor (_ : S) -> S)
             (X : S) = struct
 include Id (X)
end

module Alias = Simple
module M = Apply (Identity) (Alias)

let y = M.x
```

- **M** is the result of applying **Apply**
- Look up of the **Apply functor**
- **x** Come frome the application of **Id**
- **Id** is a parameter

EXAMPLE

LET'S FIND THE DEFINITION OF M.X

```
module type S = sig
 val x : int
end


module Identity (X : S) : S = X

module Simple = struct
 let x = 3
 let y = true
end
module Apply (Id : functor (_ : S) -> S)
             (X : S) = struct
 include Id (X)
end


module Alias = Simple
module M = Apply (Identity) (Alias)

let y = M.x
```

- **M** is the result of applying **Apply**
- Look up of the **Apply functor**
- **x** Come frome the application of **Id**
- **Id** is a parameter
- Let's back to the application to inspect it

EXAMPLE

LET'S FIND THE DEFINITION OF M.X

```
module type S = sig
 val x : int
end

module Identity (X : S) : S = X

module Simple = struct
 let x = 3
 let y = true
end
module Apply (Id : functor (_ : S) -> S)
             (X : S) = struct
 include Id (X)
end

module Alias = Simple
module M = Apply (Identity) (Alias)

let y = M.x
```

- **M** is the result of applying **Apply**
- Look up of the **Apply functor**
- **x** Come frome the application of **Id**
- **Id** is a parameter
- Let's back to the application to inspect it
- The parameter is the functor **Identity**

EXAMPLE

LET'S FIND THE DEFINITION OF M.X

```
module type S = sig
 val x : int
end

module Identity (X : S) : S = X

module Simple = struct
 let x = 3
 let y = true
end
module Apply (Id : functor (_ : S) -> S)
             (X : S) = struct
 include Id (X)
end

module Alias = Simple
module M = Apply (Identity) (Alias)

let y = M.x
```

- **M** is the result of applying **Apply**
- Look up of the **Apply functor**
- **x** Come frome the application of **Id**
- **Id** is a parameter
- Let's back to the application to inspect it
- The parameter is the functor **Identity**
- **x** come from the **X** argument of the functor

EXAMPLE

LET'S FIND THE DEFINITION OF M.X

```
module type S = sig
 val x : int
end


module Identity (X : S) : S = X

module Simple = struct
 let x = 3
 let y = true
end
module Apply (Id : functor (_ : S) -> S)
             (X : S) = struct
 include Id (X)
end

module Alias = Simple
module M = Apply (Identity) (Alias)

let y = M.x
```

- **M** is the result of applying **Apply**
- Look up of the **Apply functor**
- **x** Come frome the application of **Id**
- **Id** is a parameter
- Let's back to the application to inspect it
- The parameter is the functor **Identity**
- **x** come from the **X** argument of the functor
- Let's back to the application to inspect the second parameter

EXAMPLE

LET'S FIND THE DEFINITION OF M.X

```
module type S = sig
 val x : int
end


module Identity (X : S) : S = X

module Simple = struct
 let x = 3
 let y = true
end
module Apply (Id : functor (_ : S) -> S)
             (X : S) = struct
 include Id (X)
end

module Alias = Simple
module M = Apply (Identity) (Alias)

let y = M.x
```

- **M** is the result of applying **Apply**
- Look up of the **Apply functor**
- **x** Come frome the application of **Id**
- **Id** is a parameter
- Let's back to the application to inspect it
- The parameter is the functor **Identity**
- **x** come from the **X** argument of the functor
- Let's back to the application to inspect the second parameter
- It is the **Alias** module

EXAMPLE

LET'S FIND THE DEFINITION OF M.X

```
module type S = sig
 val x : int
end

module Identity (X : S) : S = X

module Simple = struct
 let x = 3
 let y = true
end
module Apply (Id : functor (_ : S) -> S)
             (X : S) = struct
 include Id (X)
end

module Alias = Simple
module M = Apply (Identity) (Alias)

let y = M.x
```

- **M** is the result of applying **Apply**
- Look up of the **Apply functor**
- **x** Come frome the application of **Id**
- **Id** is a parameter
- Let's back to the application to inspect it
- The parameter is the functor **Identity**
- **x** come from the **X** argument of the functor
- Let's back to the application to inspect the second parameter
- It is the **Alias** module
- Which is an *alias* (hehe) for **Simple**

EXAMPLE

LET'S FIND THE DEFINITION OF M.X

```
module type S = sig
 val x : int
end

module Identity (X : S) : S = X

module Simple = struct
 let x = 3
 let y = true
end
module Apply (Id : functor (_ : S) -> S)
             (X : S) = struct
 include Id (X)
end

module Alias = Simple
module M = Apply (Identity) (Alias)

let y = M.x
```

- **M** is the result of applying **Apply**
- Look up of the **Apply functor**
- **x** Come frome the application of **Id**
- **Id** is a parameter
- Let's back to the application to inspect it
- The parameter is the functor **Identity**
- **x** come from the **X** argument of the functor
- Let's back to the application to inspect the second parameter
- It is the **Alias** module
- Which is an *alias* (hehe) for **Simple**
- **We finally find our definition**

```
module type S = sig
 val x : int
end


module Identity (X : S) : S = X


module Simple = struct
 let x = 3
 let y = true
end
module Apply (Id : functor (_ : S) -> S)
             (X : S) = struct
 include Id (X)
end


module Alias = Simple
module M = Apply (Identity) (Alias)


let y = M.x
```

EXAMPLE

LET'S FIND THE DEFINITION OF M.X

- **M** is the result of applying **Apply**
- Look up of the **Apply functor**
- **x** Come frome the application of **Id**
- **Id** is a parameter
- Let's back to the application to inspect it
- The parameter is the functor **Identity**
- **x** come from the **X** argument of the functor
- Let's back to the application to inspect the second parameter
- It is the **Alias** module
- Which is an *alias* (hehe) for **Simple**
- **We finally find our definition**

Shapes are a new build artifcact that store that kind of path in the form of a small **typed lambda-calculus with products** associated with UID

A SIMPLIFIED REPRESENTATION

```
type t = desc * source_loc
and item = string * namespace (* val, type, module... *)
and var = ident
and desc =
| Leaf
| Var of var
| Abs of var * t
| App of t * t
| Struct of (item, t) Map.t
| Proj of t * item
```

A SIMPLIFIED REPRESENTATION

```
type t = desc * source_loc
and item = string * namespace (* val, type, module... *)
and var = ident
and desc =
| Leaf
| Var of var
| Abs of var * t
| App of t * t
| Struct of (item, t) Map.t
| Proj of t * item
```

REPRESENTING OUR MODULE M

```
module M = Apply (Identity) (Alias)
```

A SIMPLIFIED REPRESENTATION

```
type t = desc * source_loc
and item = string * namespace (* val, type, module... *)
and var = ident
and desc =
| Leaf
| Var of var
| Abs of var * t
| App of t * t
| Struct of (item, t) Map.t
| Proj of t * item
```

REPRESENTING OUR MODULE M

```
module M = Apply (Identity) (Alias)

App (
   App (
     Abs ("Id", Abs("X", App(Var "Id", Var "X"))), 7:0
   , Abs ("X", Var "X"), 3:0),
   Struct { ("x", value) -> Leaf, 5:23
            ("y", value) -> Leaf, 5:33 }), 11:0
```

A SIMPLIFIED REPRESENTATION

```
type t = desc * source_loc
and item = string * namespace (* val, type, module... *)
and var = ident
and desc =
| Leaf
| Var of var
| Abs of var * t
| App of t * t
| Struct of (item, t) Map.t
| Proj of t * item
```

REPRESENTING OUR MODULE M

```
module M = Apply (Identity) (Alias)

App (
   App (
     Abs ("Id", Abs("X", App(Var "Id", Var "X"))), 7:0
   , Abs ("X", Var "X"), 3:0),
   Struct { ("x", value) -> Leaf, 5:23
            ("y", value) -> Leaf, 5:33 }), 11:0
```

this calculus is **implicitly typed**
since its terms, which we call shapes, **are derived from OCaml's module terms which are typed-checked by the compiler.**

This implies that shapes **have a normal form in this calculus equipped with the usual reduction rules:**

$$\text{App}[\text{Abs}[x, \text{body}], \text{arg}] \ \beta > \text{body}[x \leftarrow \text{arg}]$$

$$\text{Proj}\ [[\ \text{Struct}\ \varphi\ ,\ \_\ ], e\ ]\ \pi > \varphi\ [e]$$

REDUCTION

Tricky to solve in presence of
separate compilation

REDUCTION

Tricky to solve in presence of
separate compilation

REDUCTION

An OCaml use case for strong call-by-need reduction

Gabriel Scherer (Partout, INRIA, France)
Nathanaëlle Courant (Cambium, INRIA, France)

2022

Shapes

The compiler artifact produces build artifacts that include, in particular, the "typed tree" of each source file. This is a good representation to use for programming tools (IDEs, code analyzers, etc.), but it is sometimes too complex. Consider the following OCaml program:

```
module Origin = struct let x = 1 end
module Second = struct let x = 2 let y = 2 end

module F(X) = struct
  include X
  include (Second : sig val y : int end)
end
```

A very smart idea using **Strong Call By Need Reduction**
usually useful for proof assistant

Tricky to solve in presence of
separate compilation

REDUCTION

> An OCaml use case for strong call-by-need reduction
>
> Gabriel Scherer (Partout, INRIA, France)
> Nathanaëlle Courant (Cambium, INRIA, France)
>
> 2022
>
> **Shapes**
>
> The compiler artifact produces build artifacts that include, in particular, the “typed tree” of each source file. This is a good representation to use for programming tools (IDEs, code analyzers, etc.), but it is sometimes too complex. Consider the following OCaml program:

```
module Origin = struct let x = 1 end
module Second = struct let x = 2 let y = 2 end

module F(X) = struct
  include X
  include (Second : sig val y : int end)
end
```

Everything is **more complicated** in presence of a sophisticated **module language** and **separate compilation**

A very smart idea using ***Strong Call By Need Reduction***

usually useful for proof assistant

Tricky to solve in presence of ***separate compilation***

REDUCTION

An OCaml use case for strong call-by-need reduction

Gabriel Scherer (Partout, INRIA, France)
Nathanaëlle Courant (Cambium, INRIA, France)

2022

Shapes

The compiler artifact produces build artifacts that include, in particular, the "typed tree" of each source file. This is a good representation to use for programming tools (IDEs, code analyzers, etc.), but it is sometimes too complex. Consider the following OCaml program:

```
module Origin = struct let x = 1 end
module Second = struct let x = 2 let y = 2 end

module F(X) = struct
  include X
  include (Second : sig val y : int end)
end
```

Next step
Project Wide Renaming

Everything is **more complicated** in presence of a sophisticated **module language** and **separate compilation**

A very smart idea using ***Strong Call By Need Reduction***
usually useful for proof assistant

Tricky to solve in presence of
separate compilation

REDUCTION

An OCaml use case for strong call-by-need reduction

Gabriel Scherer (Partout, INRIA, France)
Nathanaëlle Courant (Cambium, INRIA, France)

2022

Shapes

The compiler artifact produces build artifacts that include, in particular, the "typed tree" of each source file. This is a good representation to use for programming tools (IDEs, code analyzers, etc.), but it is sometimes too complex. Consider the following OCaml program:

```
module Origin = struct let x = 1 end
module Second = struct let x = 2 let y = 2 end

module F(X) = struct
  include X
  include (Second : sig val y : int end)
end
```

Next step

Project Wide Renaming

Available on last version

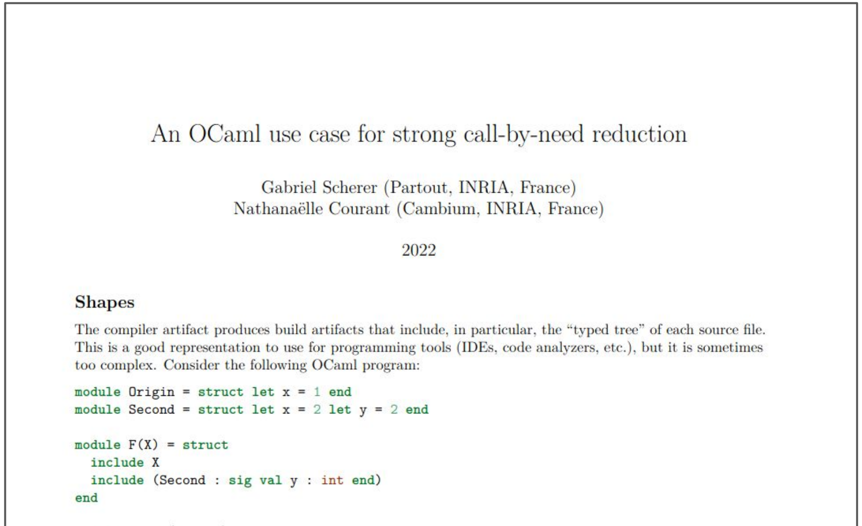
Everything is **more complicated** in presence of a sophisticated **module language** and **separate compilation**

*A very smart idea using **Strong Call By Need Reduction***

usually useful for proof assistant

*Tricky to solve in presence of **separate compilation***

REDUCTION



An OCaml use case for strong call-by-need reduction

Gabriel Scherer (Partout, INRIA, France)
Nathanaëlle Courant (Cambium, INRIA, France)

2022

Shapes

The compiler artifact produces build artifacts that include, in particular, the "typed tree" of each source file. This is a good representation to use for programming tools (IDEs, code analyzers, etc.), but it is sometimes too complex. Consider the following OCaml program:

```
module Origin = struct let x = 1 end
module Second = struct let x = 2 let y = 2 end

module F(X) = struct
  include X
  include (Second : sig val y : int end)
end
```

SEARCH BY TYPES

SEARCH BY TYPES

Discoverving a new code base can ***be complicated***

SEARCH BY TYPES

understanding architecture

Discoverving a new code base can **be complicated**

finding function and modules

SEARCH BY TYPES

We can use:
find-occurences
jump to definition

understanding architecture

Discoverving a new code base can **be complicated**

finding function and modules

SEARCH BY TYPES

We can use:
find-occurences
jump to definition

understanding architecture

Discoverving a new code base can
be complicated

finding function and modules

ocaml.org + manual

SEARCH BY TYPES

Discoverving a new code base can **be complicated**

understanding architecture

We can use:
find-occurences
jump to definition

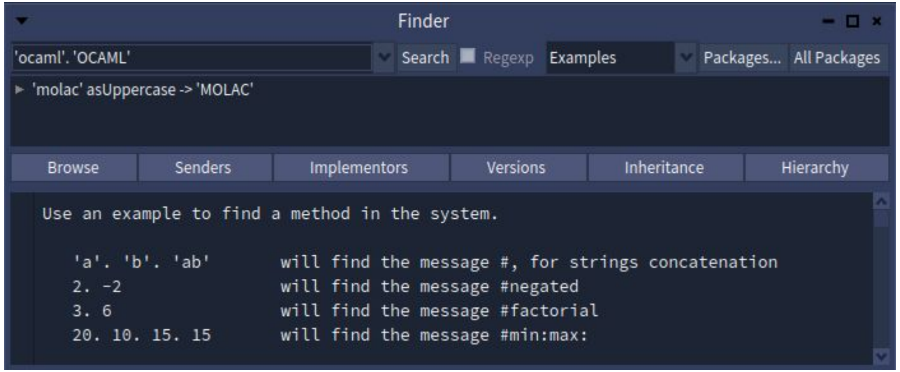
finding function and modules

???

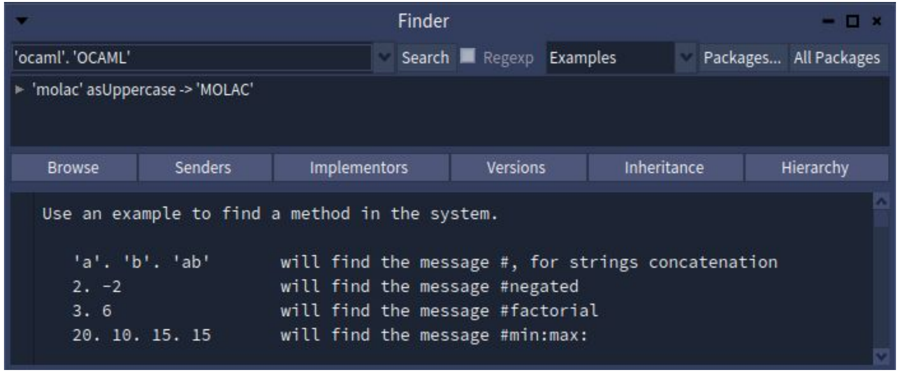
ocaml.org + manual

HOW TO FIND FUNCTION IN AN EXISTING CODEBASE

Finding by usage/example like in Pharo



HOW TO FIND FUNCTION IN AN EXISTING CODEBASE

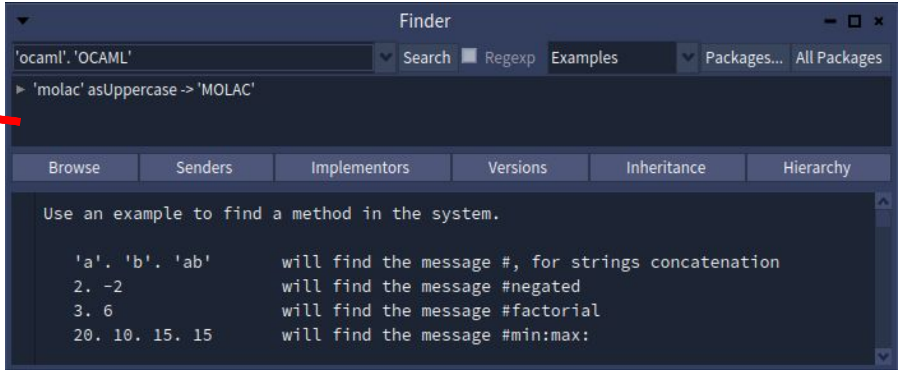


Finding by usage/example like in Pharo

HOW TO FIND FUNCTION IN AN EXISTING CODEBASE

Finding by types like with Hoogle (Haskell)

Hard to implement at the editor level

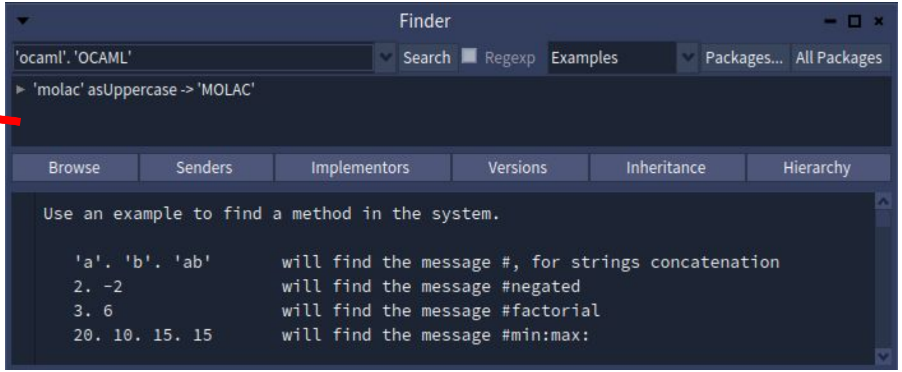


Finding by usage/example like in Pharo

HOW TO FIND FUNCTION IN AN EXISTING CODEBASE

Finding by types like with Hoogle (Haskell)

Hard to implement at the editor level



Finding by usage/example like in Pharo

HOW TO FIND FUNCTION IN AN EXISTING CODEBASE

Finding by types like with Hoogle (Haskell)

In Type we trust !!!
(and it is a very good specification tool)

A VERY DESIRED FEATURE!

SINCE 2015

- ocaml-hoogle
- ocamlscope
- ocamlscope2
- ocp-index

Type-directed API search #459

Open **ghost** opened this issue on Oct 29, 2015 · 7 comments

ghost commented on Oct 29, 2015 ...

This is a feature request.

It would be really nice if Merlin could perform Coq-like type-directed searches, similar to what `SearchPattern` does in Coq.

The main use case I would have for this feature would be to query "which functions can produce a type M.t"? Or "which functions use this type M.t"? All that in the context of a large code base where the user needs a value of type `C.t` and he knows that function `f: A.t -> B.t -> C.t` exists, but then he wonders how to produce a `B.t` in order to give it to `f`, and then he finds out after much effort that a function `g: D.t -> B.t` exists, but then he has to find out how to obtain a `D.t`

and the not well documented search by polarity in Merlin

WHAT IS POLARITY SEARCH?

Every function can be declared as

type [-'a, +'b] t = 'a -> 'b

WHAT IS **POLARITY SEARCH**?

Every function can be declared as

```
type (-'a, +'b) t = 'a -> 'b
```

a is ***contravariant*** *b is* ***covariant***

WHAT IS POLARITY SEARCH?

Every function can be declared as
type [-'a, +'b] t = 'a -> 'b

a is **contravariant** b is **covariant**

WHAT IS POLARITY SEARCH?

flagging variances allows us to define a very small distance computation

Every function can be declared as
type [-'a, +'b] t = 'a -> 'b

a is **contravariant** b is **covariant**

WHAT IS POLARITY SEARCH?

```
string -> int  -string +int
float -> int    +int -float
int -> int -> int    -int -int +int
```

minus for contravariant
plus for covariant

flagging variances allows us to
define a very small distance
computation

WHAT IS POLARITY SEARCH?

Every function can be declared as
type [-'a, +'b] t = 'a -> 'b

a is **contravariant**   b is **covariant**

string -> int option   -string +option

no support for parametric polymorphism

string -> int   -string +int
float -> int   +int -float
int -> int -> int   -int -int +int

minus for contravariant
plus for covariant

flagging variances allows us to define a very small distance computation

WHAT IS POLARITY SEARCH?

Hard to use but a **proof that Merlin can fold definition**

string -> int option   -string +option

no support for parametric polymorphism

string -> int   -string +int
float -> int   +int -float
int -> int -> int   -int -int +int

minus for contravariant
plus for covariant

Every function can be declared as
type [-'a, +'b] t = 'a -> 'b

a is **contravariant**   b is **covariant**

Flagging variances allows us to define a very small distance computation

DURING THE TIME

DURING THE TIME

Type indexing in OCaml: search and find functions in a large ecosystem

Gabriel Radanne, Inria CASH/LIP
Laure Gonnord – Grenoble INP/LCIS & LIP/CASH

2021-2022

1 Context

Sometimes, we need a function so deeply that we have to go out and search for it. How do we find it ? Sometimes, we have a precise idea of the desired type : "this function has at least 2 parameters, a *bike* and a *date* and returns a boolean value". We will then search at some precise places (the directory containing the `Bike` module, for

DURING THE TIME

Sherlodoc

'a list → ('a → 'b) → 'b list

Results for : 'a list → ('a → 'b) → 'b list

ocaml 5.1.0
`val` **`Stdlib.List.map`** `: ('a → 'b) → 'a list → 'b list`

`map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`.

ocaml 5.1.0
`val` **`Stdlib.List.rev_map`** `: ('a → 'b) → 'a list → 'b list`

`rev_map f l` gives the same result as `rev (map f l)`, but is more efficient.

ocaml 5.1.0
`val` **`Stdlib.ListLabels.map`** `: f:('a → 'b) → 'a list → 'b list`

`map ~f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`.

Dowsing

Type indexing in OCaml: search and find functions in a large ecosystem

Gabriel Radanne, Inria CASH/LIP
Laure Gonnord – Grenoble INP/LCIS & LIP/CASH

2021-2022

1 Context

Sometimes, we need a function so deeply that we have to go out and search for it. How do we find it ? Sometimes, we have a precise idea of the desired type : "this function has at least 2 parameters, a *bike* and a *date* and returns a boolean value". We will then search at some precise places (the directory containing the `Bike` module, for

DURING THE TIME

Perform real unification, extremely precise

Sherlodoc

'a list → ('a → 'b) → 'b list

Results for : 'a list → ('a → 'b) → 'b list

ocaml 5.1.0
val **Stdlib.List.map** : ('a → 'b) → 'a list → 'b list

map f [a1; ... ; an] applies function f to a1, ... , an, and builds the list [f a1; ... ; f an] with the results returned by f.

ocaml 5.1.0
val **Stdlib.List.rev_map** : ('a → 'b) → 'a list → 'b list

rev_map f l gives the same result as rev (map f l), but is more efficient.

ocaml 5.1.0
val **Stdlib.ListLabels.map** : f:('a → 'b) → 'a list → 'b list

map ~f [a1; ... ; an] applies function f to a1, ... , an, and builds the list [f a1; ... ; f an] with the results returned by f.

Dowsing

Type indexing in OCaml: search and find functions in a large ecosystem

Gabriel Radanne, Inria CASH/LIP
Laure Gonnord – Grenoble INP/LCIS & LIP/CASH

2021-2022

1 Context

Sometimes, we need a function so deeply that we have to go out and search for it. How do we find it ? Sometimes, we have a precise idea of the desired type : "this function has at least 2 parameters, a *bike* and a *date* and returns a boolean value". We will then search at some precise places (the directory containing the Bike module, for

DURING THE TIME

Compute syntactic score between type signatures

Sherlodoc

'a list → ('a → 'b) → 'b list

Results for : 'a list → ('a → 'b) → 'b list

ocaml 5.1.0
`val Stdlib.List.map : ('a → 'b) → 'a list → 'b list`

`map f [a1; ...; an]` applies function f to a1, ..., an, and builds the list `[f a1; ...; f an]` with the results returned by f.

ocaml 5.1.0
`val Stdlib.List.rev_map : ('a → 'b) → 'a list → 'b list`

`rev_map f l` gives the same result as `rev (map f l)`, but is more efficient.

ocaml 5.1.0
`val Stdlib.ListLabels.map : f:('a → 'b) → 'a list → 'b list`

`map ~f [a1; ...; an]` applies function f to a1, ..., an, and builds the list `[f a1; ...; f an]` with the results returned by f.

Perform real unification, extremely precise

Dowsing

Type indexing in OCaml: search and find functions in a large ecosystem

Gabriel Radanne, Inria CASH/LIP
Laure Gonnord – Grenoble INP/LCIS & LIP/CASH

2021-2022

1 Context

Sometimes, we need a function so deeply that we have to go out and search for it. How do we find it ? Sometimes, we have a precise idea of the desired type : "this function has at least 2 parameters, a *bike* and a *date* and returns a boolean value". We will then search at some precise places (the directory containing the Bike module, for

easier to integrate (and maybe more efficient for discoveries, rather than finding the perfect-fit function)

Compute syntactic score between type signatures

DURING THE TIME

Perform real unification, extremely precise

Sherlodoc

```
'a list → ('a → 'b) → 'b list
```

Results for : 'a list → ('a → 'b) → 'b list

ocaml 5.1.0

`val Stdlib.List.map : ('a → 'b) → 'a list → 'b list`

`map f [a1; ...; an]` applies function f to a1, ..., an, and builds the list `[f a1; ...; f an]` with the results returned by f.

ocaml 5.1.0

`val Stdlib.List.rev_map : ('a → 'b) → 'a list → 'b list`

`rev_map f l` gives the same result as `rev (map f l)`, but is more efficient.

ocaml 5.1.0

`val Stdlib.ListLabels.map : f:('a → 'b) → 'a list → 'b list`

`map ~f [a1; ...; an]` applies function f to a1, ..., an, and builds the list `[f a1; ...; f an]` with the results returned by f.

Dowsing

LIP

Type indexing in OCaml: search and find functions in a large ecosystem

Gabriel RADANNE, Inria CASH/LIP
Laure Gonnord – Grenoble INP/LCIS & LIP/CASH

2021-2022

1 Context

Sometimes, we need a function so deeply that we have to go out and search for it. How do we find it ? Sometimes, we have a precise idea of the desired type : "this function has at least 2 parameters, a *bike* and a *date* and returns a boolean value". We will then search at some precise places (the directory containing the Bike module, for

because the complicated part is about indexation the full OCaml list of available packages

easier to integrate (and maybe more efficient for discoveries, rather than finding the perfect-fit function)

Compute syntactic score between type signatures

DURING THE TIME

Perform real unification, extremely precise

Sherlodoc

Dowsing

```
'a list → ('a → 'b) → 'b list
```

Results for : 'a list → ('a → 'b) → 'b list

ocaml 5.1.0
`val Stdlib.List.map : ('a → 'b) → 'a list → 'b list`

`map f [a1; ... ; an]` applies function `f` to `a1, ... , an`, and builds the list `[f a1; ... ; f an]` with the results returned by `f`.

ocaml 5.1.0
`val Stdlib.List.rev_map : ('a → 'b) → 'a list → 'b list`

`rev_map f l` gives the same result as `rev (map f l)`, but is more efficient.

ocaml 5.1.0
`val Stdlib.ListLabels.map : f:('a → 'b) → 'a list → 'b list`

`map ~f [a1; ... ; an]` applies function `f` to `a1, ... , an`, and builds the list `[f a1; ... ; f an]` with the results returned by `f`.

Type indexing in OCaml: search and find functions in a large ecosystem

Gabriel Radanne, Inria CASH/LIP
Laure Gonnord – Grenoble INP/LCIS & LIP/CASH

2021-2022

1 Context

Sometimes, we need a function so deeply that we have to go out and search for it. How do we find it? Sometimes, we have a precise idea of the desired type : "this function has at least 2 parameters, a *bike* and a *date* and returns a boolean value". We will then search at some precise places (the directory containing the Bike module, for

SHERLODOC INTEGRATION INSIDE MERLIN

without indexation part
and more precise type parameters representation

SHERLODOC INTEGRATION INSIDE MERLIN

without indexation part
and more precise type parameters representation

SHERLODOC INTEGRATION INSIDE MERLIN

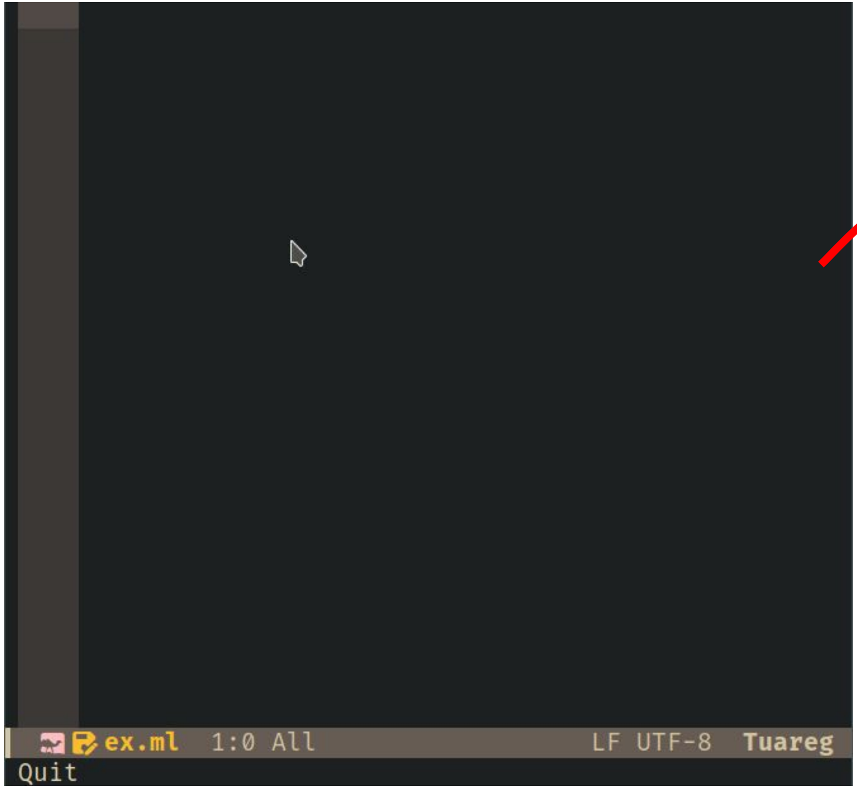
- We give a **standard representation** for a query and **OCaml types**
- We normalize parameters types (making 'a -> 'b isomorphic to 'c -> 'd)
- We create a list of path and computing distances with specific heuristics to every "cases" (ie, Damareau levensthein distance for Type constructors, and relaxed distance between a * b -> c and a -> b -> c, to capture more isomorphism)
- We use a stable-marriage algorithm on the matrix (for input parameters) to find the best-scored path
- And we have a score !

without indexation part
and more precise type parameters representation

SHERLODOC INTEGRATION INSIDE MERLIN

- We give a **standard representation** for a query and **OCaml types**
- We normalize parameters types (making 'a -> 'b isomorphic to 'c -> 'd)
- We create a list of path and computing distances with specific heuristics to every "cases" (ie, Damareau levensthein distance for Type constructors, and relaxed distance between a * b -> c and a -> b -> c, to capture more isomorphism)
- We use a stable-marriage algorithm on the matrix (for input parameters) to find the best-scored path
- And we have a score !

And adding some DX tool (constructible, doc etc)



future improvement:

- Better heuristics for tycon
- Support for modules, objects, labelled arguments and polymorphic variant (modulo isomorphism)
- Taking account of user-feedback

TO CONCLUDE!

Working with IDE is fun!

TO CONCLUDE!

Working with IDE is fun!

TO CONCLUDE!

LSP is a good default
but still need work at the client level

Working with IDE is fun!

TO CONCLUDE!

LSP is a good default
but still need work at the client level

We definitely should make issues on VSCode and LSP
to relax some part of the protocol

TO CONCLUDE!

Working with IDE is fun!

We are working on
- Improve performances
- Maintenance and improvement
- LSP canonical client for Vim

LSP is a good default
but still need work at the client level

We definitely should make issues on VSCode and LSP
to relax some part of the protocol

TO CONCLUDE!

Very open to contribution,
feedbacks and REX

We are working on
- Improve performances
- Maintenance and improvement
- LSP canonical client forVim

Working with IDE is fun!

LSP is a good default
but still need work at the client level

We definitely should make issues on VSCode and LSP
to relax some part of the protocol

TO CONCLUDE!

Very open to contribution,
feedbacks and REX

We are working on
- Improve performances
- Maintenance and improvement
- LSP canonical client for Vim

Dreams: refactoring engine based on
beta-reduction and more interactive
features

Working with IDE is fun!

LSP is a good default
but still need work at the client level

We definitely should make issues on VSCode and LSP
to relax some part of the protocol

TO CONCLUDE!

Very open to contribution, feedbacks and REX

Dreams: refactoring engine based on beta-reduction and more interactive features

We are working on

- Improve performances
- Maintenance and improvement
- LSP canonical client for Vim

Working with IDE is fun!

Upstreaming some part of Merlin inside the OCaml Compiler

LSP is a good default but still need work at the client level

We definitely should make issues on VSCode and LSP to relax some part of the protocol

TO CONCLUDE!

Working with IDE is fun!

Dreams: refactoring engine based on beta-reduction and more interactive features

We have an intern to bootstrap it!

Upstreaming some part of Merlin inside the OCaml Compiler

Very open to contribution, feedbacks and REX

We are working on

- Improve performances
- Maintenance and improvement
- LSP canonical client for Vim

LSP is a good default but still need work at the client level

We definitely should make issues on VSCode and LSP to relax some part of the protocol

TO CONCLUDE!

Working with IDE is fun!

Dreams: refactoring engine based on beta-reduction and more interactive features

We have an intern to bootstrap it!

Very open to contribution, feedbacks and REX

We are working on
- Improve performances
- Maintenance and improvement
- LSP canonical client for Vim

LSP is a good default but still need work at the client level

We definitely should make issues on VSCode and LSP to relax some part of the protocol

Debug Adapter Protocol?
TreeSitter?

Upstreaming some part of Merlin inside the OCaml Compiler

Questions ?

BEYOND THE BASICS OF LSP ADVANCED IDE SERVICES FOR OCAML

Xavier Van de Woestyne - @vdwxv - xvw.lol