Extending OCaml's `open`

Runhang Li
rli@twitter.com

Jeremy Yallop
jeremy.yallop@cl.cam.ac.uk

`open` v.s. `include`

two differences

```
module M = struct … end

open M
```

Introduce bindings defined in module `M` into the current scope

```
module M = struct … end

include M
```

Introduce bindings defined in module `M`
into the current scope

+

re-exports the bindings from the current scope

```
module M = struct
  let x = 1
end;
```

A.ml

```
open M;
let y = x;
```

A.mli

```
let y : int
```

B.ml

```
include M;
let y = x;
```

B.mli

```
let x : int
let y : int
```

`open`	`include`

`open A.B.C;`	`include A.B.C;`
	`include struct … end`
	`include M(struct … end)`
	`include (T:S);`

`open` only accepts module path

In this talk:

Eliminate the second difference so that both `open` and `include` accept an *arbitrary* module expression

Many useful applications

+

problems solved

Example A: a workaround for type shadowing

- One common programming pattern in OCaml is to define a type `t` in each module
- Problem may arise when there are multiple definitions of `t` in scope, and one refers to another

Example A: a workaround for type shadowing

```
type t1 = A
module M = struct
  type t2 = B of t2 * t1 | C
end
```

Example A: a workaround for type shadowing

```
type t1 = A
module M = struct
  type t2 = B of t2 * t1 | C
end
```

Example A: a workaround for type shadowing

```
type t1 = A
module M = struct
  type t2 = B of t2 * t1 | C
end
```

Problem: `t`$_1$ and `t`$_2$ cannot both be renamed to `t`

Example A: a workaround for type shadowing

```
type t = A
module M = struct
  type t = B of t * t | C
end
```

Problem: t_1 and t_2 cannot both be renamed to `t`

How about `type nonrec`

```
type t = A
module M = struct
  type nonrec t = t
end
```

How about `type nonrec`

```
type t = A
module M = struct
  type nonrec t = t
end
```

How about `type nonrec`

```
type t = A
module M = struct
  type nonrec t =
    B of t * t | C
end
```

`nonrec` makes **all** `t` within definition `t` refer to the single most-recent definition

Solution using open extension

```
type t = A
module M = struct
  open struct type t’=t end
  type t =
    B of t * t’ | C
end
```

Example B: local exception definition

```
include struct
  open struct exception Interrupt end
  let rec loop () = … raise Interrupt
  let rec run =
    match (loop ()) with
    | exception Interrupt ->
        Error "failed"
    | x -> Ok x
end
```

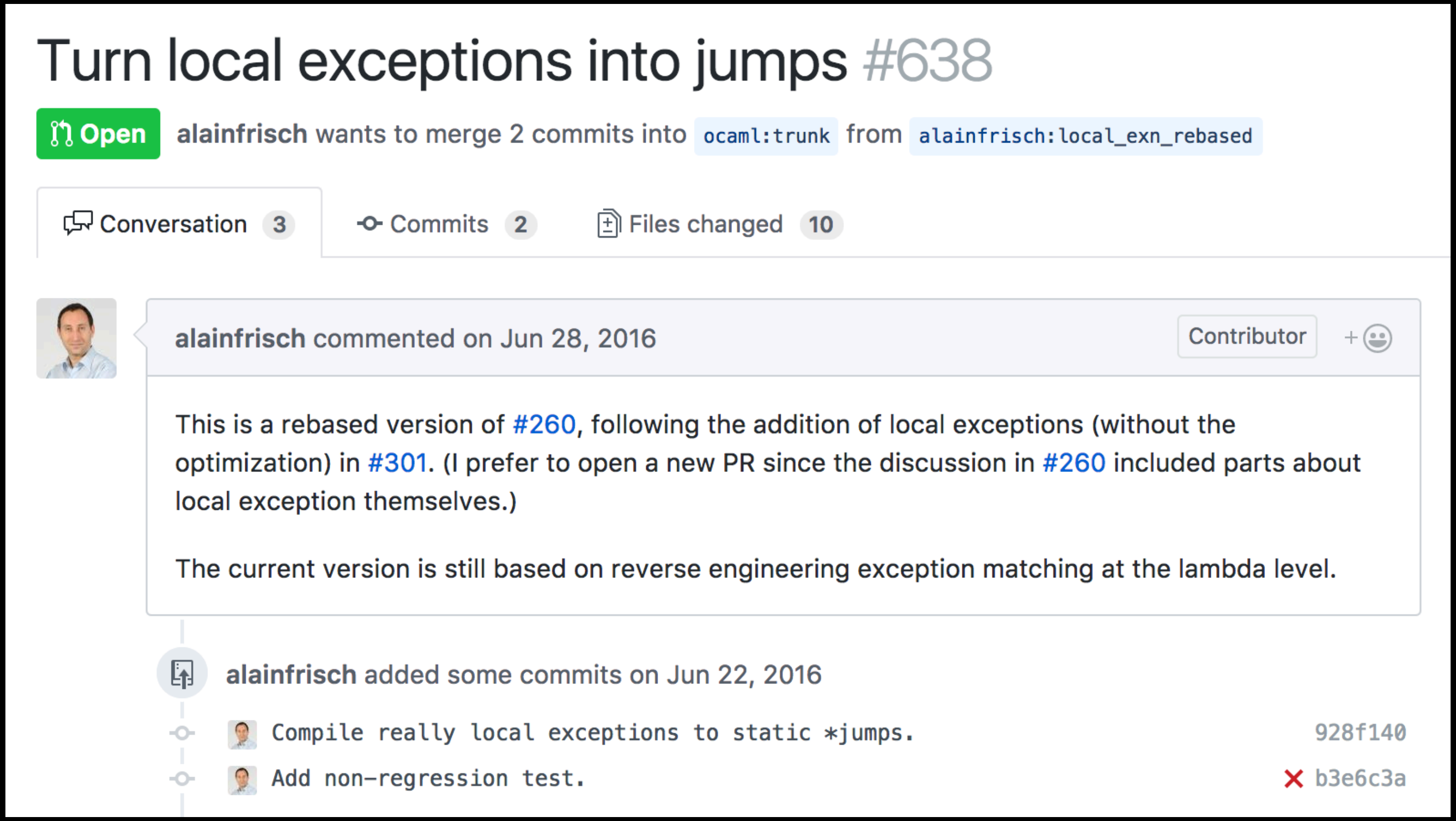
the `Interrupt` exception is only visible within the bindings for loop and run

Example B: local exception definition

- Pass information between a raiser and a handler without the possibility of interception:
  exception is shared secrets

Example B: local exception definition

- Easier to understand control flow, easier to optimize program (in some cases can be compiled to a local-jump: OCaml GitHub PR #638)



Turn local exceptions into jumps #638

Open alainfrisch wants to merge 2 commits into `ocaml:trunk` from `alainfrisch:local_exn_rebased`

alainfrisch commented on Jun 28, 2016

This is a rebased version of #260, following the addition of local exceptions (without the optimization) in #301. (I prefer to open a new PR since the discussion in #260 included parts about local exception themselves.)

The current version is still based on reverse engineering exception matching at the lambda level.

alainfrisch added some commits on Jun 22, 2016

```
Compile really local exceptions to static *jumps.    928f140
Add non-regression test.                           ✗ b3e6c3a
```

Example B: local exception definition

- Also useful when programming using algebraic effect handlers

```
open struct effect Get : int end
```

Example C: locally shared state

```
open struct
  open struct
    let counter = ref 0
  end
  let inc () = incr counter
  let dec () = decr counter
end
```

Example D: restricted `open`

```
open (Option: MONAD)
```

Example D: restricted `open`

```
open (Option: MONAD)
```

Problem: module type ascription is not transparent. Concrete type definitions are hidden.

Example D: restricted `open`

```
open (Option: MONAD)
```

Problem: module type ascription is not transparent. Concrete type definitions are hidden.

You may want to write this instead

```
open (Option : MONAD with type 'a t = 'a option)
```

Extended `open` in module signatures

One useful feature of OCaml compiler:

**passing `-i` flag when compiling
a module to see the inferred signature
of the module**

Unwritable, unprintable signatures?

A.ml	printed sig

`type t = T1`
 	
`module M = struct`
`  type t = T2`
`  let f T1 = T2`
`end`
	`type t = T1`
	
	`module M = struct`
	`  type t = T2`
	`  val f : t -> t`
	`end`

Unwritable, unprintable signatures?

B.ml	printed sig
`type t = T`	`type t = T`
`module M = struct`	`module M = struct`
`  type ’a t = 'a`	`  type 'a t = 'a`
`  let f T = T`	`  val f : t -> t`
`end`	`end`

Unwritable, unprintable signatures?

A.ml	corrected sig

type t = T1
 	
module M = struct
  type t = T2
  let f T1 = T2
end
	type t = T1
	open struct
	  type t’= t
	end
	module M : sig
	  type t = T2
	  val f : t’ -> t
	end

Restriction and design considerations

Dependency elimination

Dependency elimination

```
module F(X: sig type t val x: t end) =
   struct let x = X.x end

module N =
  F(struct type t = T let x = T end);;
```

Dependency elimination

```
module F(X: sig type t val x: t end) =
   struct let x = X.x end

module N =
  F(struct type t = T let x = T end);;
```

```
Error: This functor has type
       functor (X : sig type t val x : t end) -> sig val x : X.t end
       The parameter cannot be eliminated in the result type.
        Please bind the argument to a module identifier.
```

Dependency elimination

```
module F(X: sig type t val x: t end) =
   struct let x = X.x end

module N = F(struct
  type t = T
  let x = T end
)
```

Rejected by type checker!

What will be the type of `x` in `N`?

`X.t?`

But `X` (functor argument) is gone after application!

Dependency elimination

```
module F(X: sig type t val x: t end) =
   struct let x = X.x end

module N = F(struct
  type t = T
  let x = T end
)
```

Rejected by type checker!

Functor argument has been eliminated after application.
It is impossible to give a type for `N.x`.

Dependency elimination

```
include struct
  open struct
    type t = T
  end
  let x = T
end
```

Dependency elimination

```
include struct
  open struct
    type t = T
  end
  let x = T
end
```

Error: The module identifier `M#0` cannot be eliminated from `let x : M#0.t`

Checked using `Mtype.nondep`

(Xavier Leroy. A modular module system. Journal of Functional Programming, 10(3):269–303, 2000.)

Restriction and design considerations

`open` should be a purely static?

Questions?

playground at:

ocamllabs.io/iocamljs/open-struct.html