Extending ReasonML's open

Runhang Li
rli@twitter.com

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

`open` v.s. `include`

two differences

```
module M = {…};

open M;
```

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

```
module M = {…};
```

```
include M;
```

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

+

re-exports the bindings from the current scope

```
module M = {
  let x = 1
};
```

A.re

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

A.rei

```
let y : int
```

B.re

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

B.rei

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

open	include

`open A.B.C;`	`include A.B.C;`
	`include {…}`
	`include M({…})`
	`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

Example: a workaround for type shadowing

- One common programming pattern in ReasonML 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 workaround for type shadowing

```
type t1 = | A;
module M = {
  type t2 = | B t2 t1 | C
};
```

Example: a workaround for type shadowing

```
type t1 = | A;
module M = {
  type t2 = | B t2 t1 | C
};
```

Example: a workaround for type shadowing

```
type t1 = | A;
module M = {
  type t2 = | B t2 t1 | C
};
```

Problem: `t1` and `t2` cannot both be renamed to `t`

Example: a workaround for type shadowing

```
type t = | A;
module M = {
    type t = | B t t | C
};
```

Problem: t_1 and t_2 cannot both be renamed to t

How about `type nonrec`

```
type t  = A
module M = {
   type nonrec t = t
}
```

How about `type nonrec`

```
type t   = A
module M = {
   type nonrec t = t
}
```

How about `type nonrec`

```
type t = | A;
module M = {
  type nonrec t = | B t t | C
};
```

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

Example: local exception definition

- Pass information between a raiser and a handler without the possibility of interception: **exception is shared secrets**
- Easier to understand, easier to optimize (in some cases can be compiled to a local-jump: OCaml GitHub PR #638)

How about `let module`

`let module` allows defining exceptions whose names are visible only within particular expressions

Extended open improves upon this pattern, by limiting visibility to particular declarations.

Example: local exception definition

```
include {
  open {exception Interrupt};
  let rec loop () => … raise Interrupt;
  let rec run =
    switch (loop ()) {
    | exception Interrupt =>
      Error "failed"
    | x => Ok x
    };
};
```

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

Example: locally shared state

```
open {
  open {
    let counter = ref 0 end;
    let inc () = incr counter;
    let dec () = decr counter;
  }
}
```

Extended `open` in module signatures

One useful feature of ReasonML/OCaml compiler:

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

Unwritable, unprintable signatures?

A.re	printed sig

`type t =`
`  | T1;`
 	
`module M = {`
`  type t =`
`  | T2;`
`  let f T1 => T2;`
`};`
	`type t =`
	`  | T1;`
	
	`module M = {`
	`  type t =`
	`  | T2;`
	`  let f : t => t;`
	`};`

Unwritable, unprintable signatures?

B.re	printed sig

`type t =`
`  | T;`
 	
`module M = {`
`  type t 'a = 'a;`
`  let f T => T;`
`};`
	`type t =`
	`  | T;`
	
	`module M = {`
	`  type t 'a = 'a;`
	`  let f : t => t;`
	`};`

Unwritable, unprintable signatures?

A.re	corrected sig

`type t =`
`  | T1;`
 	
`module M = {`
`  type t =`
`  | T2;`
`  let f T1 => T2;`
`};`
	`type t =`
	`  | T1;`
	
	`module M = {`
	`  type t =`
	`  | T2;`
	`  open {type t'=t}`
	`  let f : t => t';`
	`};`

Restriction and design considerations

Dependency elimination

Dependency elimination

```
module F (X: {type t; let x: t;}) => {
  let x = X.x;
};

module N = F {
  type t = | T;
  let x = T;
};
```

Dependency elimination

```
module F (X: {type t; let x: t;}) => {
  let x = X.x;
};
```

```
module N = F {
  type t = | T;
  let x = T;
};
```

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: {type t; let x: t;}) => {
  let x = X.x;
};
```

```
module N = F {
  type t = | T;
  let x = T;
};
```

Rejected by type checker!

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

In other words, we cannot *anchor* type of `x`

Dependency elimination

```
include {open {type t = T; let x = T}}
```

Dependency elimination

```
include {open {type t = T; let x = T}}
```

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

(`M#0` is generated module identifier)

Any questions?

twitter.com/objmagic

(I tweet camels too much…)