Capabilities as First-Class Modules with Separate Compilation

Jam Kabeer Ali Khan
Advisor: Prof. Bruno C. d. S. Oliveira

❑ What are *First-Class* Environments?

- Environments: Mapping of variables and values at *runtime!*
- *First-class? Programmer can create, pass and reify environments at runtime!*

```
@pure module Environments

val env1 = ({"y" = 3} ,, {"x" = 2});  (* Environment merge *)

let {
    x: Int = 1;
    y: Bool = True
} in {
    x - (with env1 in env.x)  (* Environment scoping *)
}
```

❑ What are Capabilities?

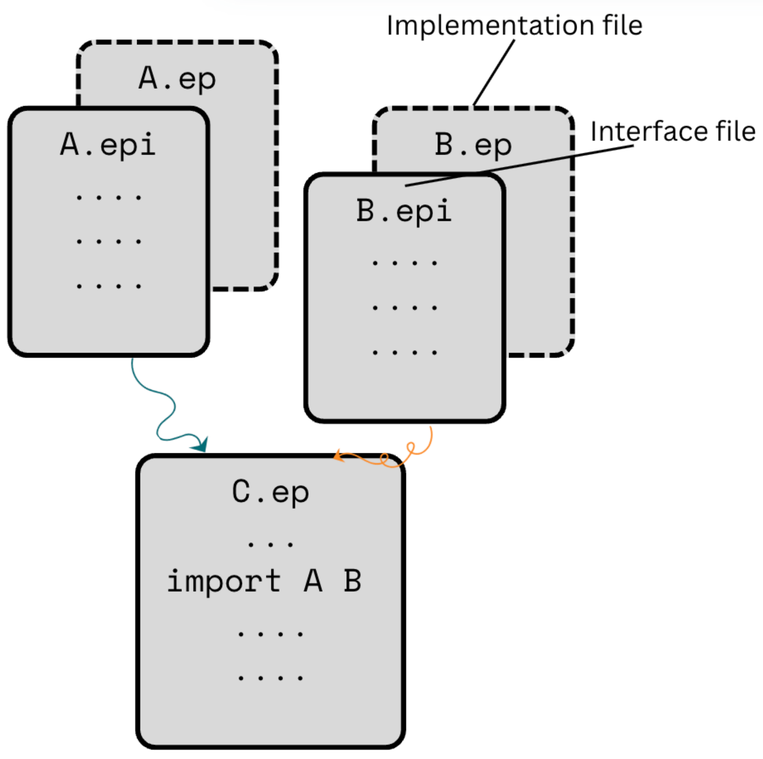
- Used to introduce access control with resources passed as parameters rather than direct imports.

```
@pure module Y
(* import X -- this is not allowed *)
require X;
let ans = X.resource + 1
```

```
@resource module main
import X Y;
open Y(X);

let final = ans
```

❑ What is Separate Compilation?

- OCaml has interface files! C++ has header files.
- Compiling without implementation of imports.

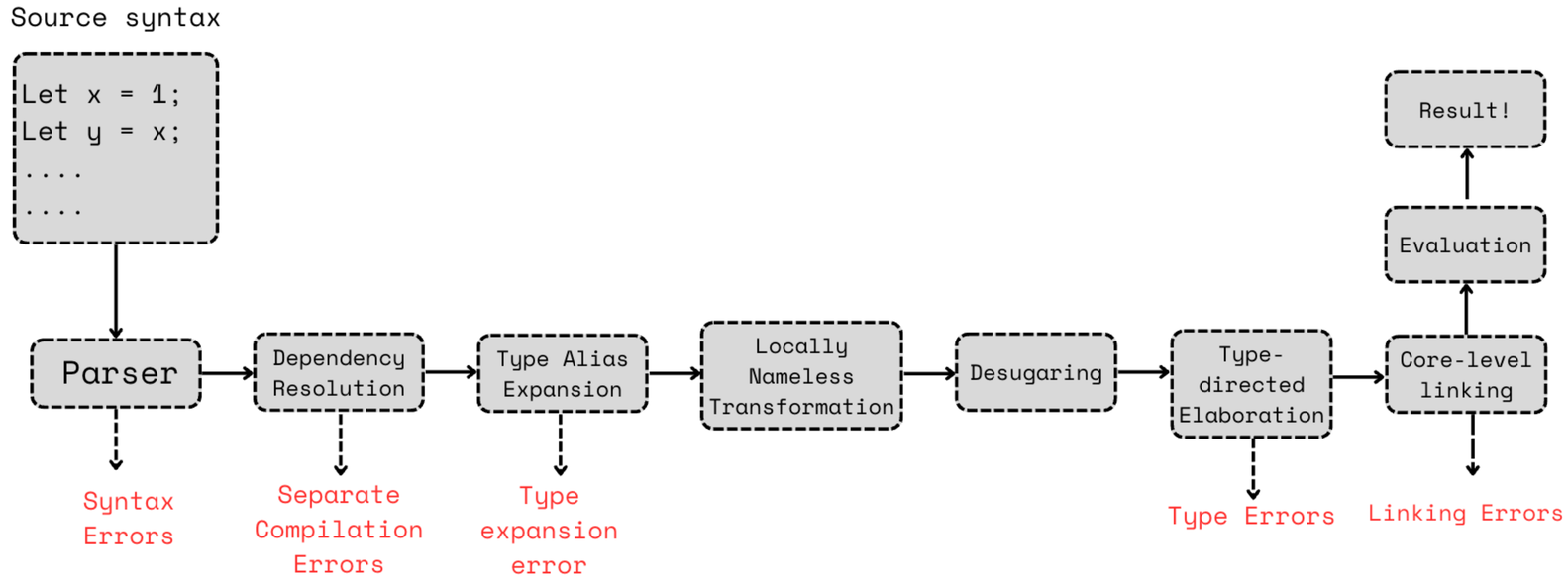



```
*** Implementation File (.ep) ***

@pure module Factorial

function factorial(n: Int, dec : Int -> Int)
: Int {
    if (n == 0)
  then 1
  else { n * factorial(dec(n), dec) }
};

*** Interface File (.epi) ***

@pure interface Factorial

function factorial : Int -> (Int -> Int) ->
Int
```

- Challenge: Capabilities without objects?
- ✓ Solution: *First-class Environments & Sandboxing*

- Challenge: Separate Compilation and Linking directly in the core?
- ✓ Solution: *First-class Environments & Dependent merges!*

---------- ENVCAP ----------



❑ **ENVCAP** Syntax

Fragment	$U ::= \text{program } S\ I\ R\ E\ A$
Authority	$S ::= \text{@pure} \mid \text{@resource}$
Import	$I ::= \cdot \mid \text{import } l : A,\ I$
Requirements	$R ::= \cdot \mid \text{require } l : A,\ R$
Expressions	$E ::= \text{env} \mid E.n \mid i \mid \epsilon \mid \lambda A.\ E \mid \text{with } E_1 \text{ in } E_2$
	$\mid E_1\ E_2 \mid E_1; E_2 \mid E_1, E_2 \mid \{\ell = E\} \mid E.\ell$
	$\mid \text{function } \ell\ A : B\ E \mid \text{struct } A\ E \mid \textit{struct}\ E$
	$\mid E_1 * E_2 \mid \text{functor } \ell\ A : B\ E \mid \text{module } \ell : B\ E$
	$\mid \text{let } x\ E_1 \mid \text{open } E_1\ E_2 \mid E : A$
Types	$A, B, \Gamma ::= \text{Int} \mid \epsilon \mid A \to B \mid \{\ell : A\}$
	$\mid A \,\&\, B \mid \text{Sig}[A, B]$

❑ Language Features

- Arithmetic (`+`, `-`, `..`), Boolean (`&&`, `||`, `..`) and Comparison (`>`, `>=`, `..`) operators.
- Supported Types: `Int`, `Bool`, `String`, `Unit`, `A -> A`, `{l : A}`, `A & A`, `A || A`, `List A`, `Sig[A, B]`
- Type Aliases
- *First-Class Environments*
- *First-Class* Modules and Functors
- Capabilities (`@resource` vs. `@pure` modules)
- Sandboxing
- Conditionals and Switch Statements
- Anonymous and *First-Class* Functions
- `Let` and `Letrec` expressions
- Recursion
- Tuples, Lists and Records
- Algebraic Data Types & Pattern Matching
- Separate Compilation

❑ Type-directed Elaboration **ENVCAP** to λ_E

EL-MODULE
$$\frac{\epsilon \,\&\, A \vdash E : B \rightsquigarrow e}{\Gamma \vdash \textit{struct}\ A\ E : \text{Sig}[A, B] \rightsquigarrow \epsilon \triangleright \lambda|A|.e}$$

EL-MODAPP
$$\frac{\Gamma \vdash E_1 : \text{Sig}[A, B] \rightsquigarrow e_1 \quad \Gamma \vdash E_2 : A \rightsquigarrow e_2}{\Gamma \vdash E_1 * E_2 : B \rightsquigarrow e_1\ e_2}$$

EL-CLOS
$$\frac{\Gamma \vdash E_1 : \Gamma_1 \rightsquigarrow e_1 \quad \Gamma_1 \& A \vdash E_2 : B \rightsquigarrow e_2}{\Gamma \vdash \langle E_1, \lambda A.E_2 \rangle : A \to B \rightsquigarrow e_1 \triangleright \lambda|A|.e_2}$$

EL-NON-DEPENDENT-MERGE
$$\frac{\Gamma \vdash E_1 : A_1 \rightsquigarrow e_1 \quad \Gamma \vdash E_2 : A_2 \rightsquigarrow e_2}{\Gamma \vdash E_1, E_2 : A_1 \,\&\, A_2 \rightsquigarrow (\lambda|\Gamma|.(\underline{0} \triangleright e_1)\ ,\ (\underline{1} \triangleright e_2))\ ?}$$

EL-OPEN
$$\frac{\Gamma \vdash E_1 : \{\ell : A\} \rightsquigarrow e_1 \quad \Gamma \vdash E_1.l : A \rightsquigarrow e_1.l \quad \Gamma \& A \vdash E_2 : B \rightsquigarrow e_2}{\Gamma \vdash \textbf{open}\ E_1\ E_2 : B \rightsquigarrow (\lambda|A|.e_2)\ (e_1.l)}$$

❑ **Rocq** Formalization

Theorem 1 (Type Preservation).

if $\Gamma \vdash E : A \rightsquigarrow e$, *then* $|\Gamma| \vdash e : |A|$.

Theorem 2 (Uniqueness of Type Inference).

if $\Gamma \vdash E : A_1 \rightsquigarrow e_1$ *and* $\Gamma \vdash E : A_2 \rightsquigarrow e_2$, *then* $A_1 \equiv A_2$.

Theorem 3 (Uniqueness of Elaboration).

if $\Gamma \vdash E : A_1 \rightsquigarrow e_1$ *and* $\Gamma \vdash E : A_2 \rightsquigarrow e_2$, *then* $e_1 \equiv e_2$.

❑ Further Work

- Formalization of Separate Compilation & Linking.
- Extension with modular subtyping.
- Formalization of Authority-Safety Proof.

❑ Extended Abstract

❑ GitHub