UNIVERSITY OF HONG KONG

FINAL YEAR PROJECT

Capabilities as *First-Class* Modules with Separate Compilation

Author:
Jam Kabeer Ali KHAN

Supervisor :
Prof. Bruno C. d. S. OLIVEIRA
2nd *Examiner* :
Prof. Chenxiong QIAN

In fulfillment of the requirements
for the degree of
Bachelor of Engineering in Computer Science with Minor in Mathematics

School of Computing & Data Science

April 21, 2025

Abstract of thesis entitled

Capabilities as *First-Class* Modules with Separate Compilation

Submitted by

Jam Kabeer Ali KHAN

for the degree of
Bachelor of Engineering in Computer Science with Minor in Mathematics

at The University of Hong Kong

in April, 2025

We present ENVCAP, a statically typed language that supports capabilities using first-class modules and separate compilation based on environment-based semantics (λ_E). Unlike traditional object-capability models, ENVCAP treats capabilities as parameterized first-class modules implemented using *first-class* environments, enabling fine-grained authority control via `@pure`/`@resource` annotations. It eliminates extra-linguistic linking mechanisms (e.g., linksets) by embedding separate compilation directly in the core calculus, where interfaces desugar to types and fragments elaborate to λ_E expressions. We formalize type-directed elaboration in Coq and implement an interpreter (∼6K LOC Haskell). ENVCAP offers a principled alternative to object-capability systems based on *first-class* environments and introduces the notion of separate compilation based on the unified types and interfaces.

Declaration

I, Jam Kabeer Ali KHAN, declare that this report titled, "Capabilities as *First-Class* Modules with Separate Compilation", which is submitted in fulfillment of the requirements for the Degree of Bachelor of Engineering in Computer Science with Minor in Mathematics, represents my own work except where due acknowledgement have been made.

Signed: _____

Date: April 21, 2025

Acknowledgements

The highlight of my undergraduate journey was discovering formal methods and programming languages research—a field that elegantly bridges mathematics and computer science. Working in this area has been an extraordinary source of intellectual excitement and personal fulfillment. None of this would have been possible without the support of several remarkable individuals.

First and foremost, I am profoundly grateful to my advisor, **Prof. Bruno C. d. S. Oliveira**, whose mentorship has been invaluable. His unwavering support, insightful feedback, and constant encouragement pushed me to strive for excellence. What I cherished most was the open and collaborative environment he fostered—our meetings were always constructive, and I felt empowered to share ideas freely, without hesitation or fear of judgment. His guidance was instrumental in shaping both this project and my growth as a researcher.

I am equally indebted to **Jinhao Tan** for his exceptional dedication. His patient mentorship—through weekly discussions, prompt replies to my late-night questions, and expertly curated references—transformed this project into a deeply enriching learning experience. Without his time, expertise, and generosity, I would not have progressed nearly as far.

To both Prof. Bruno and Jinhao, I extend my deepest gratitude for their ideas, advice, and unwavering support. It has been an absolute privilege to learn from and collaborate with you.

I would also like to thank **Prof. Gilles Barthe** for hosting me at the Max Planck Institute for Security and Privacy at the time of writing this report.

I am incredibly fortunate to have such supportive and inspiring academic mentors. Their guidance has shaped not only this work but also my aspirations for the future.

I am deeply grateful to my girlfriend, **Elynna**; she has provided encouragement and support throughout my research journey, whether it be trying to learn Rocq with me, or giving feedback. I am very fortunate to have you in my life.

Jam Kabeer Ali KHAN
University of Hong Kong
April 21, 2025

Contents

List of Figures

List of Abbreviations

STLC	**S**imply **T**yped **L**ambda **C**alculus
BNF	**B**ackus **N**aur **F**orm
LOC	**L**ines **O**f **C**ode

Chapter 1

Introduction

In programming language implementation, *environments* are widely used instead of *substitution* for efficiency. However, environments are typically a *meta-level construct*, hidden from programmers. *First-class environments* elevate this meta-level notion into the language itself, making it accessible to programmers. This project showcases the utility of first-class environments through a novel implementation of *Capabilities* and *Separate Compilation*.

1.1 Preliminaries

1.1.1 *Simply Typed Lambda Calculus*

The Simply Typed Lambda Calculus (STLC) [21] is foundational to typed functional programming languages and type theory. Its extension of the untyped lambda calculus with type-annotated abstractions enables static type checking. More complex type systems—featuring, for example, subtyping, polymorphism, or dependent types—often build upon STLC. We briefly introduce STLC as it forms the basis for λ_E [26], the core calculus used herein, which notably extends STLC with first-class environments.

Syntax The syntax of STLC consists of *types* and *terms*. Types classify terms, while terms represent computations.

- **Types (T):** Types are typically defined inductively, starting from a set of base types (e.g., $\mathbb{B}$ for booleans, $\mathbb{N}$ for naturals, $\mathbb{S}$ for strings) and including function types.

  $$T ::= \quad B \quad | \quad T_1 \to T_2$$

  Here, B represents a base type, and $T_1 \to T_2$ denotes the type of functions that accepts an argument of type T_1 and produces a result of type T_2.

- **Terms (t):** Terms include variables, lambda abstractions (functions), and applications (function calls).

$$t ::= x \quad | \quad \lambda x : T.t_1 \quad | \quad t_1\, t_2$$

  Here, x is a variable, $\lambda x : T.t_1$ is a function definition binding variable x of type T within the body t_1, and $t_1\, t_2$ represents the application of the function t_1 to the argument t_2.

Typing Rules The typing rules are based on a typing relation defined by a *typing judgement* $\Gamma \vdash t : T$ which asserts that term t has type T under the typing context Γ. The typing context Γ is a mapping from variables to their types, with the form: $\Gamma = x_1 : T_1, \ldots, x_n : T_n$.

$\boxed{\Gamma \vdash e : A}$ **(Typing)**

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}\ \text{T-VAR} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2}\ \text{T-ABS} \qquad \frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\, t_2 : T_{12}}\ \text{T-APP}$$

Rule T-VAR states that a variable has the type T assigned to it in the context. Rule T-ABS states that if the body t has type T_2 under the assumption that x has type T_1, then the abstraction $\lambda x : T_1.t_1$ has the function type $T_1 \to T_2$. Rule T-APP states that if t_1 has a function type $T_1 \to T_2$ and t_2 has the corresponding argument type T_1, then the application $t_1\, t_2$ has the type T_2.

Operational Semantics The operational semantics of the Simply Typed Lambda Calculus can be defined using either small-step or big-step approaches:

- **Small-step semantics** ($e \hookrightarrow e'$) models computation as a sequence of fine-grained reduction steps
- **Big-step semantics** ($e \Downarrow v$) relates terms directly to their final values

$\boxed{e \hookrightarrow e'}$ **(Small-Step Semantics)**

$$\frac{e_1 \hookrightarrow e_1'}{e_1\, e_2 \hookrightarrow e_1'\, e_2}\ \text{E-APP1} \qquad \frac{e_2 \hookrightarrow e_2'}{v\, e_2 \hookrightarrow v\, e_2'}\ \text{E-APP2} \qquad (\lambda x : T.e)\, v \hookrightarrow [x \mapsto v]e\ \text{E-APPABS}$$

The substitution-based semantics of STLC, while theoretically elegant, presents two practical challenges: (1) inefficient term traversal during evaluation, and (2) name capture complicating formal metatheory. Standard implementations

address this through *environment-based semantics*, where runtime variable bindings are tracked externally as a map of variables and values, called *environment*. However, this approach treats environments as meta-level artifacts rather than language entities.

This suggests a natural progression: elevating environments to *first-class status* within the calculus itself. Such an approach would (1) bridge the gap between formal semantics and implementation, (2) eliminate substitution-related name capture problems, and (3) enable direct manipulation of environments by the programmer.

1.1.2 Environment-based semantics: λ_E calculus

Environment-based semantics bridge the gap between formal theory and practical implementation by incorporating environments directly into the calculus, moving beyond substitution models. While building on foundational work such as STLC with explicit environments [10], closures [13], and environment manipulation techniques [5, 1], this work specifically employs the λ_E calculus [26].

λ_E is particularly suitable as it provides a call-by-value, statically typed framework with first-class environments and dependent merges [27], aligning well with the goals of ENVCAP. It serves as the target formalism for defining ENVCAP's semantics via elaboration. We briefly present its syntax, typing rules, and operational semantics below; readers seeking a comprehensive treatment are referred to the original λ_E paper [26].

Syntax

Expressions	$e ::= ? \mid e.n \mid i \mid \epsilon \mid \lambda A.\, e \mid e_1 \rhd e_2 \mid \langle v, \lambda A.e \rangle \mid e_1\ e_2 \mid e_1 \,,\, e_2 \mid \{\ell = e\} \mid e.\ell$
Types	$A, B, \Gamma ::= \mathrm{Int} \mid \epsilon \mid A \to B \mid \{\ell : A\} \mid A \,\&\, B$
Values	$v ::= i \mid \epsilon \mid \langle v, \lambda A.e \rangle \mid v_1 \ , \ v_2 \mid \{\ell = v\}$

Figure 1.1: λ_E syntax.

Types and Contexts. Metavariables A, B, Γ range over types, which also serve as contexts; there is no distinction. Types include integers (`Int`), the unit type ϵ (representing empty environments), function types $A \to B$, single-field record types $\{\ell : A\}$, and intersection types $A \,\&\, B$.

Expressions. Expressions include environment query ?, indexed projection $e.n$ (accessing elements by de Bruijn index, typically used as $?.n$ for variables), labeled selection $e.\ell$, and environment application $e_1 \triangleright e_2$ (evaluating e_2 in environment e_1). Core functional constructs comprise integer literals i, the unit value ϵ, lambda abstractions $\lambda A.e$, function application $e_1 e_2$, and explicit closures $\langle v, \lambda A.e \rangle$ (capturing environment v). Composition constructs are single-field records $\{\ell = e\}$ and dependent merges $e_1 \,,\, e_2$. Multi-field records are formed via merges (e.g., $\{\ell_1 = e_1\} \,,\, \{\ell_2 = e_2\}$).

Values. Values v represent irreducible results, including literals, closures, merged values, and record values, as shown in Figure 1.1.

Typing rules

$\boxed{\ell : A \in B}$ **(Containment)**

$$\text{CTM-RCD}\ \frac{}{\ell : A \in \{\ell : A\}} \qquad \text{CTM-ANDL}\ \frac{\ell : A \in B \quad \ell \notin \text{label(C)}}{\ell : A \in B \,\&\, C} \qquad \text{CTM-ANDR}\ \frac{\ell : A \in C \quad \ell \notin \text{label(B)}}{\ell : A \in B \,\&\, C}$$

$\boxed{\Gamma \vdash e : A}$ **(Typing)**

$$\text{TYP-CTX}\ \frac{}{\Gamma \vdash ? : \Gamma} \qquad \text{TYP-PROJ}\ \frac{\Gamma \vdash e : B \quad lookup(B, n) = A}{\Gamma \vdash e.n : A} \qquad \text{TYP-LIT}\ \frac{}{\Gamma \vdash i : Int} \qquad \text{TYP-TOP}\ \frac{}{\Gamma \vdash \varepsilon : \varepsilon}$$

$$\text{TYP-BOX}\ \frac{\Gamma \vdash e_1 : \Gamma_1 \quad \Gamma_1 \vdash e_2 : B}{\Gamma \vdash e_1 \triangleright e_2 : B} \qquad \text{TYP-MERGE}\ \frac{\Gamma \vdash e_1 : A \quad \Gamma \,\&\, A \vdash e_2 : B}{\Gamma \vdash e_1 \,,\, e_2 : A \,\&\, B} \qquad \text{TYP-LAM}\ \frac{\Gamma \,\&\, A \vdash e : B}{\Gamma \vdash \lambda A.e : A \to B}$$

$$\text{TYP-APP}\ \frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B} \qquad \text{TYP-CLOS}\ \frac{\Gamma \vdash v : \Gamma_1 \quad \Gamma_1 \& A \vdash e : B}{\Gamma \vdash \langle v, \lambda A.e \rangle : A \to B} \qquad \text{TYP-RCD}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash \{l = e\} : \{l : A\}}$$

$$\text{TYP-SEL}\ \frac{\Gamma \vdash e : B \quad l : A \in B}{\Gamma \vdash e.l : A}$$

$$lookup(A \& B, 0) = B$$

$$lookup(A \& B, n+1) = lookup(A, n)$$

Figure 1.2: Typing rules of λ_E

Figure 1.2 presents the typing rules for λ_E, centered around the judgment $\Gamma \vdash e : A$, where contexts Γ are themselves types. Standard rules type literals (TYP-LIT, TYP-TOP), application (TYP-APP), and lambda abstraction (TYP-LAM), with the latter extending the context Γ using intersection ($\Gamma \,\&\, A$) to bind the argument. Environment interaction is typed via rules for querying the current context (TYP-CTX for ?), indexed projection (TYP-PROJ for $e.n$, using an auxiliary 'lookup'), and environment application (TYP-BOX for $e_1 \rhd e_2$, checking e_2 within the context typed by e_1). Composition rules include dependent merge (TYP-MERGE for $e_1 \,,\, e_2$, typing e_2 in the context $\Gamma \,\&\, A$ to allow dependency on e_1), single-field record creation (TYP-RCD for $\{\ell = e\}$), and selection (TYP-SEL for $e.\ell$, relying on the containment judgment $\ell : A \in B$). Explicit closures (TYP-CLOS for $\langle v, \lambda A.e \rangle$) are typed as functions, verifying the body e under the type of the captured environment v extended with the input type A.

Operational semantics

$$\text{BSTEP-CTX}\quad \frac{}{v \vdash ? \Rightarrow v} \qquad \text{BSTEP-PROJ}\quad \frac{v \vdash e \Rightarrow v_1}{v \vdash e.n \Rightarrow lookupv(v_1, n)} \qquad \text{BSTEP-LIT}\quad \frac{}{v \vdash i \Rightarrow i}$$

$$\text{BSTEP-CLOS}\quad \frac{}{v \vdash \langle v_1, \lambda A.e \rangle \Rightarrow \langle v_1, \lambda A.e \rangle} \qquad \text{BSTEP-UNIT}\quad \frac{}{v \vdash \varepsilon \Rightarrow \varepsilon}$$

$$\text{BSTEP-MERGE}\quad \frac{v \vdash e_1 \Rightarrow v_1 \quad v \,,\, v_1 \vdash e_2 \Rightarrow v_2}{v \vdash e_1 \,,\, e_2 \Rightarrow v_1 \,,\, v_2} \qquad \text{BSTEP-BOX}\quad \frac{v \vdash e_1 \Rightarrow v_1 \quad v_1 \vdash e_2 \Rightarrow v_2}{v \vdash e_1 \rhd e_2 \Rightarrow v2}$$

$$\text{BSTEP-APP}\quad \frac{v \vdash e_1 \Rightarrow \langle v_1, \lambda A.e \rangle \quad v \vdash e_2 \Rightarrow v_2 \quad v_1 \,,\, v_2 \vdash e \Rightarrow v'}{v \vdash e_1 e_2 \Rightarrow v'} \qquad \text{BSTEP-LAM}\quad \frac{}{v \vdash \lambda A.e \Rightarrow \langle v_1, \lambda A.e \rangle}$$

$$\text{BSTEP-RCD}\quad \frac{v \vdash e \Rightarrow v_1}{v \vdash \{l = e\} \Rightarrow \{l = v_1\}} \qquad \text{BSTEP-SEL}\quad \frac{v \vdash e \Rightarrow v_1 \quad v_1.l \rightsquigarrow v_2}{v \vdash e.l \Rightarrow v_2}$$

Figure 1.3: Big-step operational semantics

Figure 1.3 defines the big-step operational semantics for λ_E via the judgment $v \vdash e \Rightarrow v'$, indicating expression e evaluates to value v' within the environment v, where environments are themselves values. This semantics computes the final result directly: values evaluate to themselves (BSTEP-LIT, BSTEP-UNIT, BSTEP-CLOS), while lambda abstractions $\lambda A.e$ form closures capturing the current environment v (BSTEP-LAM). Key evaluation steps involve environment manipulation: application (BSTEP-APP) evaluates the function body e within the closure's captured environment v_1 merged with the evaluated argument v_2; environment application (BSTEP-BOX) evaluates e_2 under the environment v_1 obtained from evaluating e_1; and dependent merge (BSTEP-MERGE) evaluates e_2 under the current environment v extended by the value v_1 resulting from e_1. Rules for environment query (BSTEP-CTX), projection (BSTEP-PROJ), record creation (BSTEP-RCD), and selection (BSTEP-SEL) handle direct environment and record access, completing the direct mapping from expressions to their final values.

1.1.3 *First-Class Environments*

In practical implementations, runtime evaluation employs a meta-level *environment*, a mapping from variables to their corresponding values. During execution, variables are dynamically resolved through environment lookups rather than via explicit substitution. This approach contrasts with the theoretical substitution model, where all variable occurrences are statically replaced with their values prior to evaluation.

First-class environments refer to the environments that can be manipulated directly by the programmer. This raises the status of environments from a meta-level construct to the *first-class* entity, enabling greater expressive power. Most programming languages do not support *first-class* environments. There are a few dynamically typed languages, e.g., Symmetric Lisp [8] and a dialect of R [9], which support *first-class* environments, but statically typed languages are rare.

The following example demonstrates *first-class environments* in R [9], showcasing their creation, manipulation, and association with functions:

```
# Create a new environment
my_env <- new.env()

# Add variables to the environment
my_env$x <- 5
```

```
my_env$y <- 10

my_func <- function() print(x)
# Execute the function inside a specific environment
environment(my_func) <- my_env
my_func()  # Prints 5 (accesses x from my_env)
```

This illustrates three key operations:

- **Environment instantiation:** `new.env()` creates a mutable namespace.
- **Variable binding:** Dynamically assigns `x` and `y` within `my_env`.
- **Lexical scoping control:** Links `my_func` to `my_env`, forcing variable resolution within the designated environment.

First-class environments enable simplifying the implementation of complex features, e.g. Capabilities and Separate Compilation. This work demonstrates the utility of the implementation of Capabilities and Separate Compilation based on the environment-based semantics.

1.1.4 *Capabilities*

Capabilities [6] provide a mechanism to introduce authority control within the programming language. This enables programmers to restrict a certain part of code from freely accessing other parts of code. Instead, any restricted code is explicitly passed via parameters to a module instead of direct import.

Capabilies are commonly implemented based on the object-capability model [23, 12]. In this model, the capability is modeled as an object that is explicitly passed via parameters to other modules that require it. This provides a mechanism for authority control and, consequently, introduces access control at the programming language level.

However, recent work on λ_E [26] proposes using *first-class* environments as an alternative to objects. This idea forms the core foundation of ENVCAP, where capabilities are implemented as *first-class* modules built upon *first-class* environments.

1.1.5 *Separate Compilation*

Separate compilation is a fundamental technique for enabling scalable software development by allowing modules to be compiled independently using only interface (*signature*) information. This approach offers several key advantages:

- **Decoupled Development**: Modules can be type-checked and compiled without access to their dependencies' implementations
- **Interface Stability**: Changes to a module's implementation that preserve its interface do not require recompilation of dependent modules
- **Incremental Builds**: Reduces compilation time by reusing previously compiled artifacts

Modern programming languages implement separate compilation through either:

- **Explicit Interfaces**: Where programmers manually specify module interfaces (e.g., C++ header files, OCaml `.mli` files)
- **Implicit Interfaces**: Where the compiler automatically determines interfaces (e.g., Haskell's type inference system) [20]

Example: Separate Compilation in OCaml

OCaml provides robust support for separate compilation through its module system, using *signatures* (interfaces) and *structures* (implementations). Consider a stack module:

```
(* Abstract type declaration *)
type 'a t

(* Operations *)
val empty : 'a t
val push : 'a -> 'a t -> 'a t
val pop : 'a t -> ('a * 'a t) option
```

Listing 1.1: Stack interface (stack.mli)

The implementation in `stack.ml` provides the concrete details:

```
type 'a t = 'a list

let empty = []

let push x s = x :: s

let pop = function
  | [] -> None
  | x :: xs -> Some (x, xs)
```

Listing 1.2: Stack implementation (stack.ml)

Client modules can then use the stack through its interface:

```
let s = Stack.(empty |> push 1 |> push 2)

let () =
  match Stack.pop s with
  | Some (x, _) -> print_int x
  | None -> print_string "Empty stack"
```

Listing 1.3: Client module (client.ml)

The key benefits demonstrated in this example are:

- The client module compiles against `stack.mli` without needing access to `stack.ml`
- The stack's implementation can be changed (e.g., to use arrays instead of lists) without requiring recompilation of client modules

This approach is particularly powerful in large codebases where compilation time and modularity are critical concerns. The OCaml compiler (`ocamlc`) handles the separate compilation process automatically, generating `.cmi` files for interfaces and `.cmo` files for implementations.

The foundational work on separate compilation was given by Cardelli[2] introduces the concept of *linksets* to formalize the process of linking. In this framework, the overall structure of separate compilation is as follows: Cardelli proposes a module system called *bindings*, which corresponds to the source language in our setting. These bindings are compiled into linksets, where the actual process of separate compilation takes place. However, *linksets* are extralinguistic structures and hence, can aid into the complexity of the implementation. An alternate approach towards separate compilation is possible with unified types and interfaces based on the environment-based semantics of λ_E, as proposed in [27].

1.2 Contributions and Overview

This work presents the design and implementation of EnvCap, a programming language that leverages *first-class environments* to support capabilities and separate compilation. The language elaborates into the λ_E core calculus through a type-directed transformation. The primary contributions of this work are:

1. The EnvCap programming language, featuring 1) First-class environments, 2) Capabilities as *first-class* modules and 3) Separate compilation with unified types and interfaces.

2. A type-directed elaboration from core ENVCAP to the λ_E calculus
3. A novel approach to separate compilation and linking using unified interface types
4. A formalization of the elaboration process in Rocq [28] and a reference implementation in Haskell [16].

Chapter 2

ENVCAP Programming Language: Syntax, Features & Case Study

This chapter presents a comprehensive exploration of the ENVCAP programming language, examining its syntactic structure, key features, and underlying design philosophy. We systematically analyze the language's architecture, emphasizing deliberate design choices that distinguish it from existing paradigms. The discussion begins with an overview of ENVCAP's core syntax and semantics, followed by an in-depth examination of its distinctive features, such as *first-class* environments, *first-class* modules, *capabilities*, *algebraic data types*, *pattern matching*, *separate compilation*, etc. To demonstrate the language's practical utility, we conclude with a case study implementing a sorting library, showcasing ENVCAP's utility to implement real-world use cases.

2.1 Design

ENVCAP is a statically-typed functional language that introduces three key innovations: 1) first-class environments, 2) capability-based first-class modules, and 3) separate compilation with unified types and interfaces.

Building on OCaml's module system [15] and Wyvern's capabilities [12], it unifies these features while preserving λ_E's expressiveness. The design objectives are:

1. Pure functional semantics
2. Capability-based authority control
3. First-class module manipulation
4. Separate compilation

5. Robust dependency management
6. Full λ_E expressiveness

Program	P	::= `@pure module` *name* I R `Statement`
		| `@resource module` *name* I R `Statement`
Import	I	::= $\cdot$ | `import` l; I
Requirements	R	::= $\cdot$ | `require` l; R
Statements	Ss	::= S | S; Ss
Statement	S	::= `function` *name* $(x_1 : A_1, ..., x_n : A_n) : B$ $\{E\}$ $(*$ `Function` $*)$
		| `module` *name* $: B$ { `Statements` } $(*$ `Simple module` $*)$
		| `functor` *name* $(x_1 : A_1, ..., x_n : A_n) : B$ { `Statements` } $(*$ `Functor` $*)$
		| `val` *name* $=$ `Term` $(*$ `Binding` $*)$
		| `type` *name* $=$ `Type` $(*$ `Type alias` $*)$
		| `Term`
Term	E	::= `env` | `unit` | `Bool` | `String` | $E_1 E_2$ | i | var | $\{l = E\}$ | $E.l$ | $E.n$
		| `if` E_1 `then` E_2 `else` E_3 | E_1 `;` E_2 | `with` E_1 `in` E_2
		| `let`$\{x_1 : T_1 = E_1, ..., x_n : T_n = E_n\}$ `in` E
		| `letrec`$\{x_1 : T_1 = E_1, ..., x_n : T_n = E_n\}$ `in` E
		| `module struct`$\{Ss\}$ | `module struct`$(x_1 : A_1, ..., x_n : A_n)\{Ss\}$
		| $\lambda(x_1 : A_1, ..., x_n : A_n).E$ | $(E_1, ..., E_n)$ | $(E_1 \, , \ldots , E_n)$
		| E_1 op E_2 | `match` E `of` $\overline{\texttt{case}_i \ (name \ x_1 \ \ldots \ x_{n_i}) \Rightarrow \{E_i\}}^{i \in \{1,\ldots,m\}}$
		| `match` E `of case` $[\,] \ \Rightarrow \ E_1$
		`case` $(x : xs) \ \Rightarrow \ E_2$
		| $\{name \ x_1 \ldots x_{n_i}\}$ *as* T $(*$ *Tagging* $*)$
		| $[E_1, ..., E_n] < T >$ $(*$*List with Tagging*$)$
Operations	op	::= `+` | `-` | `*` | `/` | `%` | `==` | `!=` | `<=` | `<` | `>=` | `>` | `&&` | `||`
Types	A, B	::= Int | $String$ | $Bool$ | $unit$ | $A \to B$ | $[A]$ | $\{l : A\}$ | $Sig[A, B]$
		| $\bigcup_{i=1}^{m}$ name $A_1 \ldots A_{n_i}$ | $A \& B$

Figure 2.1: ENVCAP Syntax

2.2 Syntax & Features

The BNF grammar for the relevant syntax of the ENVCAP programming language is shown in the (Figure-4.1).

Basic. ENVCAP provides a comprehensive set of operators following standard precedence rules:

- **Arithmetic**: $+, -, *, /, \%$ with standard integer semantics
- **Comparison**: $==, !=, <, <=, >, >=$ returning Boolean values
- **Logical**: $\&\&, ||, !$

```
@pure module BasicOperators

let x = ((1 + 2) / (3 + 4) == (5 + 5) % 2);  (* Boolean expression
    *)
let y = 3 * 1 == 10;                          (* false *)
let z = 1 > 3                                  (* false *)
```

Purely Functional. Consistent with its purely functional design, ENVCAP enforces single-assignment semantics for all bindings:

```
@pure module Bindings

let x = 10;
let x = 20  (* Compilation error: rebinding prohibited *)
```

Control flow. The language provides two complementary conditional constructs:

1. `if-then-else`

```
@pure module Conditional1

let x = 10;
let z = if (x >= 0) then x else -1 * x
```

2. `switch`, which is simply deusgared into nested conditionals during compilation.

```
@pure module Conditional2

let x = 10;
let y = switch x of
            | == 10       => { True }
```

```
            | == 1         => { False }
            | otherwise    => { False }

(* Above is same as below*)
let z =
    if (x == 10)
        then True
        else {  if (x == 1) then False else False  }
```

Functions: *first-class* & anonymous. Functions are *first-class* citizens in ENVCAP, these can be passed as parameters to other functions.

```
function temp(f: Int -> Int, n: Int) : Int {
    f(n)
};
let result1 = temp(factorial, 3);              (* 6 *)
let result2 = temp(\(n:Int) => {n + 1}, 10)    (* 11 *)
```

Above example demonstrates the *first-class* nature of functions and it shows the anonymous lambda functions support in the ENVCAP.

Let expressions. ENVCAP enables support for the `let` and `letrec` expressions, where the latter supports recursive definitions for bindings.

```
@pure module Let1

let x = letrec {factorial : Int -> Int =
                        \(n:Int) => { if (n == 0)
                                      then { 1 }
                                      else { n * factorial(n - 1) } };
                result : Int = factorial(10)}
                in { result }
```

Recursion. ENVCAP supports recursion through:

- Implicit recursion in named function declarations
- Explicit `letrec` bindings as shown in the previous example

```
@pure module Recursion

function factorial(n: Int) : Int {
    if (n == 0)
        then 1
        else n * factorial(n - 1)  (* Recursive call *)
}
```

First-class Environments. ENVCAP's static typing of first-class environments enables:

- Runtime scope manipulation
- Secure sandboxing
- Explicit environment passing via `env` and `with`

```
@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 *)
}
```

The example demonstrates:

- Environment construction using merge operator (,,).
- Environment reification using the `env` operator.
- Running computation under a specific environment `temp` using `with` construct.

First-Class Modules ENVCAP's design of *first-class modules* draws inspiration from OCaml but extends it by supporting both non-parameterized modules and *first-class functors* (parameterized modules). Key benefits include:

1. **Dynamic Composition**: Enables runtime module selection and composition, essential for ENVCAP's capability-based security and first-class environments.
2. **Type-Safe Abstraction**: Functors preserve static typing while allowing modular reuse, aligning with ENVCAP's goals of separate compilation and interface consistency.

Example: First-Class Module Usage First, we define an interface type `OUT`:

```
@pure module FirstClassModuleExample

type OUT = {y : Int, f : Int -> Int};
```

A higher-order function `getFunction` accepts a module of type `Sig[Int, OUT]`, instantiates it, and returns its embedded function:

```
function getFunction(m: Sig[Int, OUT]) : Int -> Int {
    (\(applied: OUT) => applied.f)(m(20))
};
```

Two modules implement `OUT` with distinct behaviors:

1. `moduleOne`: Computes the sum of integers from `0` to `n`.

```
module moduleOne (x: Int) : OUT {
    val y = x + 1;
    function f(n: Int) : Int {
        if (n == 0) then 0 else n + f(n - 1)
    }
};
```

2. Anonymous functor (`moduleTwo`): Computes the factorial of `n`.

```
let moduleTwo = struct (x: Int) {
    val y = x + 1;
    function f(n: Int) : Int {
        if (n == 0) then 1 else n * f(n - 1)
    }
};
```

Both modules are interoperable with `getFunction`, demonstrating *first-class* flexibility:

```
let sumResult = let {f = getFunction(moduleOne)} in f(10);  (* 55
    *)
let facResult = let {f = getFunction(moduleTwo)} in f(10);  (*
    3628800 *)
```

This illustrates how *first-class modules* enables polymorphic code reuse while maintaining type safety.

Algebraic Data Types ENVCAP supports *algebraic data types* (ADTs), enhancing modularity through compositional type definitions. ADTs enable:

- Type-safe data composition via sum (variants) and product (tuples) types
- Pattern matching for expressive control flow

```
@resource module Area

type Shape  = Circle Int
            | Rectangle Int Int;
```

```
5
6  let circ = {Circle 10} as Shape;
7  let rect = {Rectangle 3 5} as Rectangle;
8
9  function area(object: Shape): Int {
10     match object of
11         case (Circle r)         => { (22 / 7) * r * r }
12         case (Rectangle w h)    => { w * h }
13 };
14
15 let circArea = area(circ);
16 let rectArea = area(rect)
```

This system statically enables functions to take input of multiple possible types while guaranteeing type-safety as shown in the example above.

Lists ENVCAP provides built-in list support with explicit typing and pattern matching. The type annotation requirement (e.g., `<Int>`) simplifies the type system by eliminating ambiguity for empty lists.

Key features:

- **Typed construction**: `[`$v_1, \ldots, v_n$`]<T>` creates a list of type `[T]`
- **Pattern matching**: Exhaustive case analysis
  - `[]` for empty lists
  - `(x:xs)` for head/rest decomposition

```
1  @pure module ListExample
2
3  let nums = [1, 2, 3, 4]<Int>;  (* Explicitly typed *)
4
5  function isEmpty(ls: [Int]): Bool {
6      match ls with
7      case []     => True
8      case (x:xs) => False
9  };
10
11 let test = isEmpty(nums)  (* => False *)
```

The example demonstrates basic list operations through the `isEmpty` function, which uses pattern matching to distinguish empty and non-empty lists.

Capabilities as *first-class* modules. ENVCAP enforces authority control, crucial for safety guarantees, by implementing capabilities using its *first-class* module system. Inspired by Wyvern, ENVCAP classifies modules as either `@resource` or `@pure`. While `@resource` modules may freely import any module, `@pure` modules are restricted: they cannot directly import `@resource` modules. Instead, if a `@pure` module requires access to resources provided by a `@resource` module, the latter must be explicitly passed as a parameter—effectively acting as a capability. This access control is enforced statically during elaboration. The following example demonstrates this mechanism.

```
@resource module X

let resource = 10
```

The snippet above defines `module X`, a simple `@resource` module containing a resource binding.

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

`Module Y` is declared as `@pure` and requires `module X`. A direct import is disallowed because `X` is a `@resource` module. Access requires `X` to be supplied as an argument by another module.

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

let final = ans
```

The `@resource module main` imports both `X` and `Y`. It then grants `Y` access to `X` by passing `X` as an argument within the `open` construct. The `open Y(X)` expression instantiates `Y` with the capability `X` and brings `Y`'s resulting bindings (like `ans`) into the current scope.

Separate compilation & interfaces. ENVCAP enables separate compilation with interface files that are inspired by OCaml. We demonstrate the separate compilation with a simple example. Assume we have a module `Utils` that has not been implemented yet, but it needs to be used by another module `Main` for compilation. In order to let `Main` separately compile, we can define the interface file for `Utils`.

```
(* Utils.epi *)
@resource interface Utils

val func : Int -> Int
```

Using the interface file `Utils.epi`, `Main.epi` can be compiled now.

```
(* Main.ep *)
@resource module Main

import Utils;
open Utils;

let x = func(10)
```

There are more features, including dependency resolution and capability-safety, during separate compilation that are reflected in Chapter 3 and Chapter 4.

2.3 Case study: Sorting & Lists Library

To demonstrate ENVCAP's expressiveness, we implement a sorting algorithm with supporting list operations (220 LOC).

Utility library for integer lists. We begin by introducing fundamental list utilities required for the subsequent algorithms:

```
(* Listutils.ep file *)
@resource module Listutils

function isEmpty(ls : [Int]) : Bool {
    match ls of
        case []          => { True }
        case (x:xs)      => { False }
};

function head(ls: [Int]) : Int {
    match ls of
        case []          => { 0 }
        case (x : xs)    => { x }
};

function rest(ls: [Int]) : [Int] {
    match ls of
        case []          => { []<Int> }
        case (x : xs)    => { xs }
```

```
};
```

These operations provide three essential capabilities:

- `isEmpty`: Determines list emptiness through pattern matching
- `head`: Safely extracts the first element (defaulting to 0 for empty lists)
- `rest`: Returns the tail while preserving type information

Now, we can define further utilities.

```
function length(ls: [Int]) : [Int] {
    match ls of
        case []          => { 0 }
        case (x:xs)      => { 1 + lengthLsInt(xs) }
};

function append(ls: [Int], v: Int) : [Int] {
    match ls of
        case []          => { [v]<Int> }
        case (x : xs)    => { append(xs, v) }
};

function reverse(ls: [Int]) : [Int] {
    match ls of
        case []          => { []<Int> }
        case (x: xs)     => {
            let { ys : [Int] = reverse(xs) } in { append(ys, x) }
        }
};
```

Now, we have functions to determine the length of an integer list, append an element to the end of a list and reverse a list.

Furthermore, we need utilities to access the last or nth element.

```
function last(ls: [Int]) : Int {
    match ls of
        case []          => { 0 }
        case (x : xs)    =>
        {   match xs of
                case []          => { x }
                case (y: ys)     => { last(xs) }
        }
};

function nth(n: Int, ls: [Int]) : Int {
    match ls of
        case []              => { 0 }
```

```
        case (x : xs)       => {
            if (n == 0) then x else nth(n - 1, xs)
        }
};
```

Furthermore, utilies are needed for manipulation of lists, including adding to the front, concatenating two lists, take n elements from the list and drop n elements from the list.

```
let addFront = \(n: Int, ls: [Int]) => { reverse(append(reverse(ls)
    , n)) };

function concat(lsx: [Int], lsy: [Int]) : [Int] {
    match lsx of
        case []     => { lsy }
        case (x:xs) => { addFront(x, concat(xs, lsy)) }
};

function take(n: Int, ls: [Int]) : [Int] {
    if (n == 0)
        then []<Int>
        else match ls of
                case []     => { []<Int> }
                case (x:xs) => { concat([x]<Int>, take(n - 1, xs))
    }
};

let drop = \(n: Int, ls: [Int]) => { reverse(take(length(ls) - n,
    reverse(n))) };
```

In functional programming, there are common functions, e.g. map, filter, foldl, foldr, etc, that are useful for lists. We define few of the common utilities that apply functions on the elements of the list below:

```
function map(f: Int -> Int, ls: [Int]) : [Int] {
    match ls of
        case []      => { []<Int> }
        case (x:xs)  => { concat([f(x)]<Int>, map(f, xs)) }
};

function filter(pred: Int -> Int, ls: [Int]) : [Int] {
    match ls of
        case []     => { []<Int> }
        case (x:xs) => {
            if      (pred(x) == True)
            then    add(x, filter(pred, xs))
            else    filter(pred, xs)
        }
```

```
};

function foldl(f: Int -> Int -> Int, init: Int, ls: [Int]) : Int {
        match ls of
            case []         => { init }
            case (x:xs)     => { foldl(f, f(init, x), xs) }
};

function foldr(f: Int -> Int -> Int, init: Int, ls: [Int]) : Int {
        match ls of
            case []         => { init }
            case (x:xs)     => { f(x, foldr(f, init, xs)) }
};
```

At this point, our library has several useful utility functions. We further implement more basic utility functions that built upon the utilities defined so far.

```
function any(pred: Int -> Bool, ls: [Int]) : Bool {
    match ls of
        case []         => { False }
        case (x:xs)     => { (pred(x) == True) || any(pred, xs) }
};

let all = \(pred: Int -> Bool, ls: [Int]) => {
    foldl(\(x:Int, acc: Bool) => { x && acc}, True, ls)
};

let contains = \(elem: Int, ls: [Int]) => {
    foldl(\(x: Int, acc: Bool) => { (elem == x) || acc }, False, ls
    )
};

function repeat(elem: Int, n: Int) : [Int] {
    if (n == 0)
        then []<Int>
        else addFront(elem, repeat(elem, n - 1))
};

function range(start: Int, end: Int) : [Int] {
    if (start > end)
        then []<Int>
        else addFront(start, range(start + 1, end))
}
```

Now, we need to define *Listutils.epi* interface file for our list utilities library, in order to enable separate compilation for file that import this library.

```
(* Listutils.epi *)
@resource interface Listutils

function isEmpty(ls : [Int]) : Bool;

function head(ls: [Int]) : Int;

function rest(ls: [Int]) : [Int];

function length(ls: [Int]) : [Int];

function append(ls: [Int], v: Int) : [Int];

function reverse(ls: [Int]) : [Int];

function last(ls: [Int]) : Int;

function nth(n: Int, ls: [Int]) : Int;

val addFront : Int -> [Int] -> [Int];

function concat(lsx: [Int], lsy: [Int]) : [Int];

function take(n: Int, ls: [Int]) : [Int];

val drop    : Int -> [Int] -> [Int];

function map(f: Int -> Int, ls: [Int]) : [Int];

function filter(pred: Int -> Int, ls: [Int]) : [Int];

function foldl(f: Int -> Int -> Int, init: Int, ls: [Int]) : Int;

function foldr(f: Int -> Int -> Int, init: Int, ls: [Int]) : Int;

function any(pred: Int -> Bool, ls: [Int]) : Bool;

val all      : (Int -> Bool) -> [Int] -> Bool;

val contains : Int -> [Int] -> Bool;

function repeat(elem: Int, n: Int) : [Int];

function range(start: Int, end: Int) : [Int]
```

This interface file provides the necessary typing information for the other implementation files to separate compile.

Now, our integer lists utility library is complete. However, this code is only useful for a list of integers. This could have been made for general with polymorphism, but that will be beyond our scope. However, utilizing type aliases, we can enable re-using this code to some extent.

```
(* Listutils.ep file *)
@resource module Listutils
type Ty = Int;
...
```

If we utilize `Ty` instead of the actual type Int throughout the codebase, then to make this code usable for a list of booleans, or other types, we only need to change the `Ty` at the top.

Finally, we can utilize the integer list library to implement classical functional algorithm: Quicksort.

```
(* Quicksort.ep *)
@pure module QuickSort

require Listutils;

function quickSort(ls: [Int]) : [Int] {
    match ls of
        case []          => { []<Int> }
        case (pivot:xs) => {
            let {
                smaller: [Int] = filter(\(x: Int) => { x <= pivot
    }, xs);
                larger: [Int] = filter(\(x: Int) => { x > pivot },
    xs)
            } in {
                concat(quickSort(smaller), concat([pivot]<Int>,
    quickSort(larger)))
            }
        }
}
```

Below is the interface file for `Quicksort` module:

```
(* Quicksort.epi *)

@pure interface QuickSort

require Listutils;

function quickSort(ls: [Int]) : [Int]
```

Since `Listutils` is a `@resource` module, it is not allowed to be imported directly in the `@pure` Quicksort module. Therefore, we need another module that has the authority to import both modules.

```
(* main.ep *)
@resource module Main

import QuickSort Listutils;

open QuickSort(Listutils);

val example = [4, 3, 2, 1]<Int>;

val sortedQ = quickSort(example)
(* result: [1, 2, 3, 4] *)
```

This `Main` module has the authority to import both `Quicksort` and `Listutils` modules, and then, it instantiates and opens the module `QuickSort` by passing the `Listutils` into it.

This concludes the implementation of this case study in ENVCAP, reflecting upon the expressiveness and complexity of this programming language.

Chapter 3

Elaboration of ENVCAP $\rightsquigarrow \lambda_E$

The semantics of ENVCAP are formally defined through a type-directed elaboration into λ_E. This elaboration serves a crucial role: it demonstrates the practical utility of *first-class* environments by systematically translating ENVCAP's features into λ_E, a foundational calculus explicitly designed with first-class environment support.

3.1 Typed-directed elaboration ENVCAP $\rightsquigarrow \lambda_E$

The type-directed elaboration translates ENVCAP terms to their corresponding λ_E counterparts while preserving types throughout the translation. This process inherently type-checks the source ENVCAP terms and closely resembles compilation. The translation is formally defined by the judgment:

$$\Gamma \vdash E : A \rightsquigarrow e$$

This judgment reads: "In the typing context Γ, the ENVCAP expression E has type A and elaborates to the λ_E expression e."

$$\begin{aligned}
|Int| &= Int \\
|\epsilon| &= \epsilon \\
|A \to B| &= |A| \to |B| \\
|A \,\&\, B| &= |A| \,\&\, |B| \\
|\{\ell : A\}| &= \{\ell : |A|\} \\
|\mathbf{Sig}[A, B]| &= |A| \to |B|
\end{aligned}$$

Figure 3.1: Type translation: ENVCAP type to λ_E

$\boxed{\ell : A \in B}$ **(Containment)**

CTM-RCD
$$\frac{}{\ell : A \in \{\ell : A\}}$$

CTM-ANDL
$$\frac{\ell : A \in B \quad \ell \notin \text{label(C)}}{\ell : A \in B \,\&\, C}$$

CTM-ANDR
$$\frac{\ell : A \in C \quad \ell \notin \text{label(B)}}{\ell : A \in B \,\&\, C}$$

$\boxed{\Gamma \vdash E : A \rightsquigarrow e}$ **(Typing)**

EL-CTX
$$\frac{}{\Gamma \vdash \texttt{env} : \Gamma \rightsquigarrow \ ?}$$

EL-PROJ
$$\frac{\Gamma \vdash E : B \rightsquigarrow e \quad lookup(B, n) = A}{\Gamma \vdash E.n : A \rightsquigarrow e.n}$$

EL-LIT
$$\frac{}{\Gamma \vdash i : Int \rightsquigarrow i}$$

EL-TOP
$$\frac{}{\Gamma \vdash unit : \varepsilon \rightsquigarrow \epsilon}$$

EL-BOX
$$\frac{\Gamma \vdash E_1 : \Gamma_1 \rightsquigarrow e_1 \quad \Gamma_1 \vdash E_2 : A \rightsquigarrow e_2}{\Gamma \vdash E_1 \triangleright E_2 : A \rightsquigarrow e_1 \triangleright e_2}$$

EL-MERGE
$$\frac{\Gamma \vdash E_1 : A \rightsquigarrow e_1 \quad \Gamma \,\&\, A \vdash E_2 : B \rightsquigarrow e_2}{\Gamma \vdash E_1 \ , \ E_2 : A \,\&\, B \rightsquigarrow e_1 \,, e_2}$$

EL-LAM
$$\frac{\Gamma \& A \vdash E : B \rightsquigarrow e}{\Gamma \vdash \lambda A.E : A \to B \rightsquigarrow \lambda|A|.e}$$

EL-APP
$$\frac{\Gamma \vdash E_1 : A \to B \rightsquigarrow e_1 \quad \Gamma \vdash E_2 : A \rightsquigarrow e_2}{\Gamma \vdash E_1 \ E_2 \to B \rightsquigarrow e_1 \ e_2}$$

EL-RCD
$$\frac{\Gamma \vdash E : A \rightsquigarrow \text{e}}{\Gamma \vdash \{l = E\} : \{l : A\} \rightsquigarrow \{l = e\}}$$

EL-SEL
$$\frac{\Gamma \vdash E : B \rightsquigarrow \text{e} \quad l : A \in B}{\Gamma \vdash E.l : A \rightsquigarrow e.l}$$

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

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

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

Figure 3.2: Elaboration $\rightsquigarrow \lambda_E$

Figure 3.2 presents the key elaboration rules. We highlight some notable rules:

- **EL-CTX**: This rule types the `env` keyword with the current typing context Γ. It elaborates `env` to the environment query operator '?' in λ_E, providing access to the runtime environment.
- **EL-LAM**: Standard lambda abstraction is translated by elaborating the function body E under an extended context (Γ & A) and wrapping the result e in a λ_E abstraction. Crucially, the ENVCAP type A is translated to its λ_E counterpart $|A|$ using the type translation function defined in Figure 3.1.
- **EL-MODULE**: An ENVCAP anonymous module ($\texttt{struct}\ A\ \{E\}$) is elaborated into a sandboxed λ_E abstraction $\epsilon \triangleright \lambda|A|.e$. The explicit empty environment ϵ ensures the module body E is evaluated in isolation, receiving only its declared input of type A. The resulting term has the signature type $\mathbf{Sig}[A, B]$, translated to $|A| \to |B|$ in λ_E.
- **EL-MODAPP**: Module application $E_1 * E_2$ is translated to standard function application $e_1\ e_2$ in λ_E, presupposing E_1 elaborates to a function corresponding to a module signature.
- **EL-MERGE**: This rule directly maps ENVCAP's dependent merge $E_1 \,,\, E_2$ to λ_E's dependent merge $e_1 \,,\, e_2$. The typing reflects the dependency: E_2 is typed in a context extended by the type of E_1.
- **EL-NON-DEPENDENT-MERGE**: In contrast to the dependent merge, ENVCAP's non-dependent pairing E_1, E_2 (forming a record type A_1 & A_2) requires careful encoding in λ_E to prevent unintended dependency. The elaboration uses λ_E's merge operator , but structures the translation (using context capture '?', abstraction, and explicit projections $\underline{0}\triangleright, \underline{1}\triangleright$) to ensure that the elaborated term e_2 is evaluated in a context independent of e_1.
- **EL-OPEN**: This rule models the common `open` construct (similar to OCaml's), allowing the fields of a record E_1 (specifically field ℓ of type A) to be brought into the scope of E_2. The elaboration achieves this by selecting the field $e_1.l$ and passing it as an argument to a function derived from the elaborated body e_2, effectively $(\lambda|A|.e_2)\ (e_1.l)$.
- **EL-CLOS**: Explicit closure creation $\langle E_1, \lambda A.E_2 \rangle$ in ENVCAP translates directly to λ_E's closure formation $e_1 \triangleright \lambda|A|.e_2$, capturing the environment e_1 with the function.

- The remaining rules like **EL-PROJ**, **EL-LIT**, **EL-TOP**, **EL-BOX**, **EL-APP**, **EL-RCD**, and **EL-SEL** follow standard patterns for projection, literals, unit, environment scoping (boxing), application, record creation, and record selection, respectively, translating straightforwardly to their λ_E equivalents while respecting the type translation $|\cdot|$.

3.2 Meta-theory

Following is the meta-theory formalized related to the Elaboration from ENVCAP $\rightsquigarrow$ λ_E.

Lemma 1 (Unique type translation)**.** *The type translation function is unique.*

$$if\ |E| = E'\ and\ |E| = E''\ ,\ then\ E' \equiv E''.$$

Proof. Formalized in Rocq. □

Lemma 2 (Type preserving lookup)**.** *If lookup performed on a type A at an index n corresponds to a type B in the ENVCAP, then performing type lookup on $|A|$ on the same index n will lead to $|B|$.*

$$if\ lookup(A, n)\ =\ B,\ then\ lookup(|A|, n)\ =\ |B|.$$

Proof. Formalized in Rocq. □

Lemma 3 (Type safe label existence)**.** *If ℓ exists as a label in the ENVCAP type A, then ℓ exists as a label in the λ_E type $|A|$.*

$$\ell \in label(A)\ \iff\ \ell \in label(|A|).$$

Proof. Formalized in Rocq. □

Lemma 4 (Type safe label non-existence)**.** *If ℓ is not a label in the ENVCAP type A, then ℓ is not a label in the λ_E type $|A|$.*

$$if\ \neg(\ \ell \in label(A)),\ then\ \neg(\ell \in label(|A|)).$$

Proof. Formalized in Rocq. □

Lemma 5 (Type safe containment)**.** *If ℓ is contained in the ENVCAP type B with type A, then ℓ is contained in the λ_E type $|B|$ with type $|A|$.*

$$if\ \ell : A \in B,\ then\ \ \ell : |A| \in |B|.$$

Proof. Formalized in Rocq. □

Theorem 1 (Type Preservation)**.** *Elaboration of ENVCAP term E with type A under the typing context Γ leads to λ_E term e with type $|A|$ under the typing context $|\Gamma|$.*

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

Proof. Formalized in Rocq. □

Lemma 6 (Uniqueness of indexed lookup)**.** *Indexed lookup on an expression E in the ENVCAP is unique.*

$$if\ lookup(E, n) = A_1\ and\ lookup(E, n) = A_2,\ then\ A_1 = A_2.$$

Proof. Formalized in Rocq. □

Lemma 7 (Uniqueness of record lookup)**.** *Record lookup on a type B in the ENVCAP is unique. if $\ell : A_1 \in B$ and $\ell : A_2 \in B$, then $A_1 = A_2$.*

Proof. Formalized in Rocq. □

Theorem 2 (Uniqueness of Type Inference)**.** *Type of an expression E in the ENVCAP is unique.*

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

Proof. Formalized in Rocq. □

Theorem 3 (Uniqueness of Elaboration)**.** *Elaboration of ENVCAP to a λ_E term is unique.*

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

Proof. Formalized in Rocq. □

Formalization of the meta-theory serves as rigorous evidence to the approach of this work in demonstrating the utility of *first-class* environments by utilizing λ_E and its environment-based semantics.

3.3 Elaboration of fragments

In the Figure 3.3, we present the elaboration rules of the *fragments* that have not been formalized and therefore, are not part of the meta-theory. However, these rules are part of the implementation.

Definition 1 (Set of capabilities)**.** $\forall$ *Imports I, caps(I) is the set of authority annotations (@pure or @resource) of the imports in the interface files of imports.*

Definition 2 (Interface to type)**.** $\forall$ *Interfaces X, typ(X) gives the type desugared from the interface X.*

EL-PURE

$$\frac{@resource \notin caps(I) \quad \epsilon \vdash \lambda typ(I).(\lambda typ(R).E) : typ(I) \to typ(H) \rightsquigarrow \lambda|typ(I)|.(\lambda|typ(R)|.e)}{\Gamma \vdash \textbf{\textit{fragment name}}\ @pure\ I\ R\ E\ H : typ(I) \to typ(H) \rightsquigarrow \{name = \epsilon \rhd \lambda|typ(I)|.(\lambda|typ(R)|.e)\}}$$

EL-RESOURCE

$$\frac{\epsilon \vdash \lambda typ(I).(\lambda typ(R).E) : typ(I) \to typ(H) \rightsquigarrow \lambda|typ(I)|.(\lambda|typ(R)|.e)}{\Gamma \vdash \textbf{\textit{fragment name}}\ @resource\ I\ R\ E\ H : typ(I) \to typ(H) \rightsquigarrow \{name = \epsilon \rhd \lambda|typ(I)|.(\lambda|typ(R)|.e)\}}$$

Figure 3.3: Elaboration of fragments $\rightsquigarrow \lambda_E$

The rule EL-RESOURCE and EL-PURE are utilized to ensure capability safety and separate compilation. The main difference between the two rules is $@resource \notin caps(I)$ predicate which ensures that *@pure* modules do not import any *@resource* modules, using the **Definition 1**. Furthermore, elaboration rules utilize **Definition 2** in order to extract the types directly from the interface files.

Chapter 4

Implementation

The ENVCAP implementation employs a type-directed elaboration process, translating source programs into an extended λ_E calculus. This process serves as a form of compilation, targeting λ_E rather than a lower-level language like assembly. The overall implementation consists of approximately 6,000 LOCs. Section 4.1 first describes the necessary extensions to λ_E, followed by Section 4.2 which details the architecture of the implementation.

4.1 λ_E extensions

To serve as a suitable target for ENVCAP's elaboration, the core λ_E calculus requires extensions to its syntax, typing, and semantics. This section details the syntactic additions. The corresponding extensions to the typing rules and operational semantics are also given.

4.1.1 Syntax.

Expressions	$e ::= \cdots \mid S \mid B \mid fix\ A.\ e \mid if\ e_1\ e_2\ e_3 \mid e\langle A \rangle \mid match\ e\ \overline{constructor_i\ \cdots \Rightarrow e_i}^{i \in \{1 \cdots n\}}$
	$\mid nil\ A \mid cons\ e_1\ e_2 \mid lmatch\ e_1\ e_2\ e_3 \mid e_1\ bop\ e_2 \mid !\ e$
Types	$A, B, \Gamma ::= \cdots \mid Bool \mid String \mid A \cup B \mid [A]$
Values	$v ::= ... \mid S \mid B \mid cons\ v_1\ v_2 \mid nil\ A \mid v\langle A \rangle$
Operators	$binop ::= + \mid - \mid * \mid / \mid \% \mid == \mid != \mid < \mid <= \mid > \mid >= \mid \&\& \mid \mid\mid$

Figure 4.1: Extended λ_E syntax.

Expression Extensions. The expression language is augmented with primitive string (S) and boolean (B) literals. Recursion is supported via the fixed-point combinator $fix\ A.e$ over type A. Conditional logic employs standard

$if\ e_1\ e_2\ e_3$ expressions. To handle algebraic data types (variants), the syntax includes tagged values $e\langle A\rangle$ (likely representing injection into a sum type component designated by A) and a corresponding *match* construct for pattern matching and discrimination based on constructors. List manipulation primitives comprise the typed empty list $nil\ A$, the list constructor $cons\ e_1\ e_2$, and list-specific pattern matching $lmatch\ e_1\ e_2\ e_3$ (presumably for head/tail decomposition). Standard binary operators ($e_1\ binop\ e_2$) and boolean negation ($!\ e$) are also included.

Type Extensions. Correspondingly, the type system incorporates primitive types *Bool* and *String*, union types $A \cup B$ to represent sums/variants, and list types $[A]$ for homogeneous lists of elements with type A.

Value Extensions. The set of values is naturally extended to include the new primitive literals (S, B), constructed list values ($cons\ v_1\ v_2$), the typed empty list value ($nil\ A$), and tagged values ($v\langle A\rangle$) representing injections into sum types.

4.1.2 Typing rules

$\boxed{\Gamma \vdash e : A}$ **(Typing)**

$$\text{TYP-STRING}\ \frac{}{\Gamma \vdash S : String} \qquad \text{TYP-BOOL}\ \frac{}{\Gamma \vdash B : Bool} \qquad \text{TYP-FIX}\ \frac{\Gamma \& A \vdash e : A}{\Gamma \vdash fix A.e : A}$$

$$\text{TYP-IF}\ \frac{\Gamma \vdash e_1 : Bool \quad \Gamma \vdash e_2 : A \quad \Gamma \vdash e_3 : A}{\Gamma \vdash if\ e_1\ e_2\ e_3 : A} \qquad \text{TYP-TAGGING}\ \frac{\Gamma \vdash e : B \quad B \in A}{\Gamma \vdash e\langle A \rangle : A}$$

$$\text{TYP-MATCH}\ \frac{\Gamma \vdash e : A \quad \overline{constructor_i \in A}^{\,i \in \{1,\cdots,n\}} \quad \Gamma \& A \vdash e_i : B\ \forall i \in 1, \cdots, n}{\Gamma \vdash match\ e\ \overline{constructor_i\ \cdots \Rightarrow e_i}^{\,i \in \{1 \cdots n\}} : B}$$

$$\text{TYP-NIL}\ \frac{}{\Gamma \vdash nil\ A : [A]} \qquad \text{TYP-CONS}\ \frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : [A]}{\Gamma \vdash cons\ e_1\ e_2 : [A]}$$

$$\text{TYP-LMATCH}\ \frac{\Gamma \vdash e_1 : [A] \quad \Gamma \& [A] \vdash e_2 : B \quad \Gamma \& [A] \vdash e_3 : B}{\Gamma \vdash lmatch\ e_1\ e_2\ e_3\ : B}$$

Figure 4.2: Typing rules of λ_E extensions

Figure 4.2 extends the λ_E typing judgment $\Gamma \vdash e : A$. Rules for primitives (TYP-STRING, TYP-BOOL) and conditionals (TYP-IF) are standard. The fixpoint combinator (TYP-FIX) uses context extension similar to λ-abstraction. Algebraic data type rules involve type containment checks (TYP-TAGGING) and validation against sum types (TYP-MATCH), while list operations (TYP-NIL, TYP-CONS, TYP-LMATCH) enforce element type homogeneity. Rules for standard binary operators are omitted for brevity.

4.1.3 Operational semantics

$$\text{BSTEP-STRING}\quad \frac{}{v \vdash S \Rightarrow S} \qquad \text{BSTEP-BOOL}\quad \frac{}{v \vdash B \Rightarrow B} \qquad \text{BSTEP-FIX}\quad \frac{}{v \vdash fixA.e \Rightarrow \langle v, fixA.e\rangle}$$

$$\text{BSTEP-APPFIX}\quad \frac{v \vdash e_1 \Rightarrow \langle v_1, fixA.e\rangle \quad v \vdash e_2 \Rightarrow v_2 \quad v_1 \,,\, \langle v_1, fixA.e\rangle \,,\, v_2 \vdash e \Rightarrow v'}{v \vdash e_1 e_2 \Rightarrow v'}$$

$$\text{BSTEP-IFTRUE}\quad \frac{v \vdash e_1 \Rightarrow True \quad v \vdash e_2 \Rightarrow v_2}{v \vdash if\ e_1\ e_2\ e_3 \Rightarrow v_2} \qquad \text{BSTEP-IFFALSE}\quad \frac{v \vdash e_1 \Rightarrow False \quad v \vdash e_3 \Rightarrow v_3}{v \vdash if\ e_1\ e_2\ e_3 \Rightarrow v_3}$$

$$\text{BSTEP-TAGGING}\quad \frac{v \vdash e \Rightarrow v'}{v \vdash e\langle A\rangle \Rightarrow v'\langle A\rangle}$$

$$\text{BSTEP-MATCH}\quad \frac{v \vdash e \Rightarrow v'\langle A\rangle \quad match\ v'\ on\ constructors \quad v \,,\, v' \vdash e_j \Rightarrow v_j}{v \vdash match\ e\ \overline{constructor_i\ \cdots \Rightarrow e_i}^{\,i\in\{1\cdots n\}} \Rightarrow v_j}$$

$$\text{BSTEP-NIL}\quad \frac{}{v \vdash nil\ A \Rightarrow nil\ A} \qquad \text{BSTEP-CONS}\quad \frac{v \vdash e_1 \Rightarrow v_1 \quad v \vdash e_2 \Rightarrow v_2}{v \vdash cons\ e_1\ e_2 \Rightarrow cons\ v_1\ v_2}$$

$$\text{BSTEP-LMATCHNIL}\quad \frac{v \vdash e_1 \Rightarrow nilA \quad v \,,\, nilA \vdash e_2 \Rightarrow v_2}{v \vdash lmatch\ e_1\ e_2\ e_3\ \Rightarrow v_2} \qquad \text{BSTEP-LMATCHNIL}\quad \frac{v \vdash e_1 \Rightarrow consv'v'' \quad v \,,\, v' \,,\, v'' \vdash e_3 \Rightarrow v_3}{v \vdash lmatch\ e_1\ e_2\ e_3\ \Rightarrow v_3}$$

Figure 4.3: Big-step operational semantics for λ_E extensions

Figure 4.3 extends the big-step semantics judgment $v \vdash e \Rightarrow v'$. Primitives evaluate to themselves (BSTEP-STRING, BSTEP-BOOL). Recursion (BSTEP-FIX, BSTEP-APPFIX) uses a closure-like mechanism, unrolling the definition upon application. Conditionals (BSTEP-IFTRUE/FALSE) branch normally. Tagging (BSTEP-TAGGING) preserves values, while pattern matching (BSTEP-MATCH, BSTEP-LMATCHNIL/CONS) evaluates the subject, deconstructs it, and executes the appropriate branch within an extended environment.

4.2 Interpreter

Figure 4.4 illustrates the architecture of the ENVCAP implementation[1]. This multi-pass process resembles a traditional compilation pipeline, translating ENVCAP source code into the target calculus λ_E rather than into machine code or assembly. Haskell is used as the implementation language, selected primarily for its robust support for algebraic data types and pattern matching—features particularly well-suited for managing the successive Abstract Syntax Tree (AST) transformations required. The complete implementation consists of approximately 6,000 lines of Haskell code.

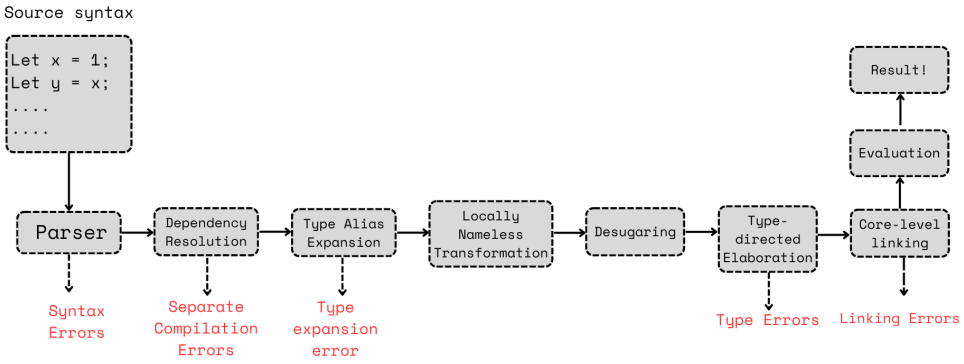

Figure 4.4: Architecture of ENVCAP Interpreter

Parsing. The initial parsing stage used a handwritten recursive-descent parser built with the Parsec combinator library in Haskell. While this approach offered fine-grained error reporting, it proved cumbersome for managing evolving grammars and lacked automated ambiguity detection. To support rapid prototyping and accommodate growing language complexity, the implementation transitioned to the Happy parser generator. This shift significantly streamlined experimentation with both syntax and language structure.

Module Dependency Resolution. ENVCAP prohibits cyclic module dependencies among both imported and required modules. To enforce this restriction, the system constructs a module dependency graph (illustrated in Figure 4.5)

[1]While often referred to as an interpreter, the multi-stage process culminating in evaluation bears resemblance to a compiler pipeline.

and performs a topological sort to detect cycles and establish a valid compilation order. Kahn's algorithm [10] is employed for this task.

Kahn's algorithm, a variant of Breadth-First Search (BFS), identifies cycles and computes a linear ordering of modules consistent with their dependencies. This topological order dictates the sequence in which module fragments are composed using ENVCAP's *dependent merges*. The type checking context for each fragment is thereby constructed based on the types provided by its predecessors in the established order, ensuring dependencies are resolved sequentially.

A critical constraint, depicted in Figure 4.5, is that `@pure` modules may only depend on other `@pure` modules; dependencies on `@resource` modules are disallowed. This restriction is fundamental to ENVCAP's capability system, preventing `@pure` code from gaining uncontrolled access to sensitive operations encapsulated within `@resource` modules (which function as capabilities). By enforcing this rule alongside the topological ordering, the system maintains separation and controlled delegation of authority. Once the modules are ordered and structured via dependent merges, the type-directed elaboration process performs separate type checking and compilation for each fragment.

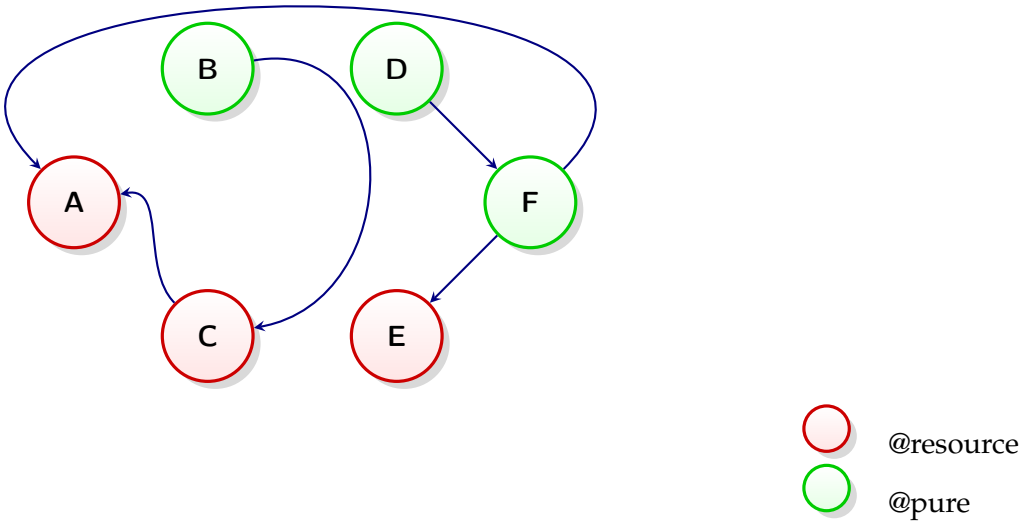

Figure 4.5: Dependency Graph showing resource and pure modules.

Type Alias Expansion. ENVCAP allows users to define type aliases, also known as type synonyms, to assign alternative names to existing types. For example:

```
1 @resource module Example
```

```
2
3 type NewType = Int; (* Defines NewType as an alias for Int *)
4
5 function inc(x : NewType) : NewType { x + 1 }
```

The current implementation handles type aliases through a direct expansion strategy for simplicity. It involves a dedicated pass over the Abstract Syntax Tree (AST) where type alias declarations are identified. Subsequently, another pass substitutes all occurrences of the defined alias with its corresponding base type throughout the remaining AST. While effective, alternative approaches, such as maintaining a symbol table for type aliases to enable on-demand lookup during type checking or elaboration, could offer potentially more efficient or integrated solutions.

Locally Nameless Representation. To prevent name capture during the elaboration to λ_E, ENVCAP employs a locally nameless representation [3] for variable binding. In this approach, variables bound by abstractions, such as function parameters, are systematically represented using de Bruijn indices, while free variables refer to bindings available in the lexical environment. This distinction is crucial because direct translation of ENVCAP code that involves shadowing could otherwise lead to ambiguities or type errors in λ_E.

Consider the following ENVCAP fragment:

```
1 @pure module nameless
2
3 let x = 1; (* Outer binding *)
4
5 function inc(x : Int) : Int { x + 1 }
6 (* Parameter 'x' shadows outer 'x' *)
```

The locally nameless transformation distinguishes the parameter ‘x‘ from the outer ‘let‘-bound ‘x‘. The parameter, being the most recently bound variable within ‘inc‘, is mapped to de Bruijn index 0, accessed via environment projection:

```
1 @pure module nameless
2
3 let x = 1;
4
5 function inc(x : Int) : Int { env.0 + 1 }
6 (* Parameter 'x' becomes index 0 *)
```

This transformation ensures that the reference within ‘inc‘ correctly points to the function parameter, resolving the potential ambiguity and allowing for a

sound, type-preserving elaboration into λ_E. Consequently, programmers can utilize familiar shadowing patterns in ENVCAP without compromising the formal correctness of the target λ_E representation.

Desugaring. Before the type-directed elaboration, the interpreter removes syntactic sugar from ENVCAP, transforming it into the core ENVCAP. This simplification enables the type-directed elaboration, described in Chapter 3, to be minimalistic and concise. This approach has the advantage of keeping the implementation's elaboration as close as possible to the formalized elaboration. However, its main drawback is that it can lead to imprecise type errors, as many syntactic constructs are eliminated prior to type checking. Like most design choices, it has its pros and cons. In our case, since ENVCAP is a research prototype, it is better to keep formal elaboration minimalistic.

Elaboration. This phase performs the type-directed translation from core ENVCAP to λ_E, as formally detailed in Chapter 3. The elaboration process itself serves as the primary static analysis stage for ENVCAP, performing type checking and ensuring type preservation. Consequently, user-facing type errors originate at this source level, abstracting away the underlying λ_E target, assuming the soundness of the elaboration. While the formalization guarantees that well-typed ENVCAP elaborates to well-typed λ_E (Type Preservation), the implementation incorporates an additional type check on the generated λ_E code (core-level). This theoretically redundant check proved practically valuable during development as a sanity check, helping to verify the correctness of the elaboration logic itself. Any inconsistencies between source-level typeability and core-level typeability would indicate a flaw in the elaboration implementation.

Linking. For projects structured using separate compilation, a distinct linking phase follows the elaboration of individual modules. This stage utilizes the previously determined topological sort order to sequentially compose the elaborated λ_E expressions corresponding to each module. The composition is achieved using λ_E's dependent merge operator ($,_9$), ensuring that dependencies between modules are correctly established in the final linked term. The outcome is a single λ_E expression representing the fully linked program.

Evaluation. The final phase executes the resulting λ_E expression (either from direct elaboration of a single module or after the linking phase). Evaluation proceeds according to the big-step operational semantics defined for λ_E ultimately yielding the program's final value.

Chapter 5

Epilogue

This project is based upon ideas generated by many related works. We further discuss some related work in Section 5.1, conclusion and future work in Section 5.2.

5.1 Related Work

First-class environments have been supported by different languages, e.g. Symmetric Lisp [8], dialect of R [9], theScheme [17]. However, most of the work is not based on the environment-based semantics and static typing, which not only enable the notion of *first-class* environments naturally, but provide static type-checking guarantees. In comparison to λ_E, E_i[27] calculus also enables static guarantees with *first-class* environments. However, E_i calculus supports subtyping as well which could be utilized to enable modular subtyping in the separate compilation to restrict exports in the modules.

The notion of capabilities was initially introduced to secure operating system resources [7] and later, introduced in the programming languages. The widely adopted model of capabilities is based on objects [19], implemented by Wyvern [12], E [18] and W7 [22]. Furthermore, there is a reference-capability model pioneered by Pony Lang [24]. In ENVCAP, we introduce the notion of capabilities based on the *first-class* environments, instead of objects. Furthermore, most of the work does not has mechanical formalization.

Separate compilation was formalized based on the framework of *linksets* provided by Cardelli [2]. This made the basis for adding separate compilation to the Standard ML [25]. However, *linksets* remain extra-linguistic features and there are multiple works that try to avoid this. One such example is that the separate compilation for DatalogIR [11], whereby the linksets or other external

mechanisms are avoided. However, the approach utilized to extend the separate compilation to the DatalogIR remains specific to the concrete language. The idea of unified types and interfaces was introduced in the E_i calculus [27], whereby interfaces are simply desugared into types and implementations are essentially abstractions over the types. In ENVCAP, we implement this idea and furthermore, introduce linking rules in λ_E without utilizing any extra-linguistic constructs, e.g. *linksets*.

5.2 Conclusion & Future Work

ENVCAP has demonstrated the utility of *first-class* environments with capabilities as *first-class* modules and separate compilation based on the environment-based semantics. However, the formalization remains restricted to a subset of the ENVCAP. One possible future direction is to extend the separate compilation mechanism and prove the authority safety [12] property. Currently, due to the increased complexity of the ENVCAP interpreter than originally expected, there needs better support for extensive testing, but unit testing may not suffice to provide strong guarantees for the implementation. Therefore, writing extensive property-based testing using QuickCheck [4] can be a potential future direction. In addition, type-directed elaboration guarantees type preservation, but it does not guarantees preserving the semantics. One potential way to mitigate this and provide a stronger form of elaboration is to provide operational semantics to ENVCAP and perform not only type-preserving, but also, semantics-preserving elaboration, akin to Compcert [14].

The implementation of ENVCAP is publicly available in the GitHub and remains under active development beyond the scope of this document: GitHub.

Bibliography

[1] M. Biernacka and O. Danvy. "A concrete framework for environment machines". In: *ACM Trans. Comput. Logic* 9.1 (Dec. 2007), 6–es. ISSN: 1529-3785. DOI: 10.1145/1297658.1297664. URL: https://doi.org/10.1145/1297658.1297664.

[2] L. Cardelli. "Program fragments, linking, and modularization". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. Paris, France: Association for Computing Machinery, 1997, pp. 266–277. ISBN: 0897918533. DOI: 10.1145/263699.263735. URL: https://doi.org/10.1145/263699.263735.

[3] A. Charguéraud. "The Locally Nameless Representation". In: *Journal of Automated Reasoning* 49.3 (Dec. 2012), pp. 363–408. DOI: 10.1007/s10817-011-9225-2.

[4] K. Claessen and J. Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *SIGPLAN Not.* 35.9 (Sept. 2000), pp. 268–279. ISSN: 0362-1340. DOI: 10.1145/357766.351266. URL: https://doi.org/10.1145/357766.351266.

[5] P.-L. Curien. "An Abstract Framework for Environment Machines". In: *Theor. Comput. Sci.* 82 (1991), pp. 389–402. URL: https://api.semanticscholar.org/CorpusID:41601809.

[6] J. B. Dennis and E. C. Van Horn. "Programming semantics for multiprogrammed computations". In: *Commun. ACM* 9.3 (Mar. 1966), pp. 143–155. ISSN: 0001-0782. DOI: 10.1145/365230.365252. URL: https://doi.org/10.1145/365230.365252.

[7] J. B. Dennis and E. C. Van Horn. "Programming semantics for multiprogrammed computations". In: *Commun. ACM* 9.3 (Mar. 1966), pp. 143–155. ISSN: 0001-0782. DOI: 10.1145/365230.365252. URL: https://doi.org/10.1145/365230.365252.

[8] D. Gelernter, S. Jagannathan, and T. London. "Environments as first class objects". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. Munich, West Germany: Association for Computing Machinery, 1987, pp. 98–110. ISBN: 0897912152.

DOI: 10.1145/41625.41634. URL: https://doi.org/10.1145/41625.41634.

[9] A. Goel and J. Vitek. "First-class environments in R". In: *Proceedings of the 17th ACM SIGPLAN International Symposium on Dynamic Languages*. DLS 2021. Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 12–22. ISBN: 9781450391054. DOI: 10.1145/3486602.3486768. URL: https://doi.org/10.1145/3486602.3486768.

[10] A. B. Kahn. "Topological sorting of large networks". In: *Commun. ACM* 5.11 (Nov. 1962), pp. 558–562. ISSN: 0001-0782. DOI: 10.1145/368996.369025. URL: https://doi.org/10.1145/368996.369025.

[11] D. Klopp, A. Pacak, and S. Erdweg. "Separate Compilation and Partial Linking: Modules for Datalog IR". In: *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE '24. Pasadena, CA, USA: Association for Computing Machinery, 2024, pp. 94–106. ISBN: 9798400712111. DOI: 10.1145/3689484.3690737. URL: https://doi.org/10.1145/3689484.3690737.

[12] D. Kurilova, A. Potanin, and J. Aldrich. "Wyvern: Impacting Software Security via Programming Language Design". In: *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU '14. Portland, Oregon, USA: Association for Computing Machinery, 2014, pp. 57–58. ISBN: 9781450322775. DOI: 10.1145/2688204.2688216. URL: https://doi.org/10.1145/2688204.2688216.

[13] P. J. Landin. "The Mechanical Evaluation of Expressions". In: *Comput. J.* 6 (1964), pp. 308–320. URL: https://api.semanticscholar.org/CorpusID:5845983.

[14] X. Leroy. "Formal verification of a realistic compiler". In: *Commun. ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814.

[15] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system release 4.10*. Feb. 2020. URL: https://caml.inria.fr/pub/docs/manual-ocaml/.

[16] S. Marlow, ed. *Haskell 2010 – Language Report*. 2010. URL: www.haskell.org/onlinereport/haskell2010/.

[17] J. Miller and G. Rozas. "Free variables and first-class environments". In: *Lisp and Symbolic Computation* 4.1-2 (1991), pp. 107–141. DOI: 10.1007/BF01813016. URL: https://doi.org/10.1007/BF01813016.

[18] M. S. Miller. *The E Language*. URL: http://erights.org/elang/.

[19] M. S. Miller. "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control". PhD thesis. Baltimore, Maryland, USA: Johns Hopkins University, May 2006.

[20] B. C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN: 0262162288.

[21] B. C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091.

[22] J. A. Rees. *A Security Kernel Based on the Lambda-Calculus*. Tech. rep. USA, 1996.

[23] J. S. Shapiro and M. S. Miller. "Robust composition: towards a unified approach to access control and concurrency control". In: 2006. URL: https://api.semanticscholar.org/CorpusID:61931826.

[24] G. Steed. "A Principled Design of Capabilities in Pony". Master's thesis. London, UK: Imperial College London, 2016.

[25] D. Swasey, T. Murphy, K. Crary, and R. Harper. "A separate compilation extension to standard ML". In: *Proceedings of the 2006 Workshop on ML*. ML '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 32–42. ISBN: 1595934839. DOI: 10.1145/1159876.1159883. URL: https://doi.org/10.1145/1159876.1159883.

[26] J. Tan and B. C. d. S. Oliveira. "A Case for First-Class Environments". In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024). DOI: 10.1145/3689800. URL: https://doi.org/10.1145/3689800.

[27] J. Tan and B. C. d. S. Oliveira. "Dependent Merges and First-Class Environments". In: *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Ed. by K. Ali and G. Salvaneschi. Vol. 263. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 34:1–34:32. ISBN: 978-3-95977-281-5. DOI: 10.4230/LIPIcs.ECOOP.2023.34. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.34.

[28] The Coq Development Team. *The Coq Reference Manual – Release 8.19.0*. https://coq.inria.fr/doc/V8.19.0/refman. 2024.