Interim Report

Capabilities as First-Class Modules

Jam Kabeer Ali Khan
3035918749

Supervisor: Bruno C. d. S. Oliveira

The University of Hong Kong
Department of Computer Science
BEng Computer Science

January 26, 2024

Abstract

The growing complexity of programming languages and codebases necessitates a certain extent of access control within the code. This project involves the development of the ENVCAP programming language, which models *capabilities* as *first-class modules*, addressing the challenges of access control in software systems with additional expressiveness and flexibility of *first-class* environments. ENVCAP is designed and implemented with a type-directed elaboration to the λ_E. Currently, it supports essential programming constructs, e.g. conditional statements, recursion, and first-class functions. In summary, this report provides an overview of the project and discusses the progress.

Contents

1 Introduction

The implementation of programming languages often relies on environments and closures, which can diverge from the theoretical foundations of lambda calculus, primarily based on a substitution model. This divergence creates a growing gap between theory and practice, complicating the reasoning about the semantics of programming languages.

To address this, a call-by-value statically typed calculus, denoted as λ_E, has been proposed by Tan and Oliveira [1]. This calculus introduces the concept of *first-class environments* and employs de Bruijn indices with environment lookups to circumvent the binding issues typically associated with conventional substitution models.

This project leverages λ_E as the foundational calculus for the design and implementation of the ENVCAP programming language. It aims to showcase the expressiveness of the λ_E calculus by modeling capabilities as *first-class modules*.

1.1 Capabilities

In many programming languages, the absence of robust access control mechanisms often leads to significant security vulnerabilities, particularly as codebases expand. Unrestricted access to critical resources, such as file systems or network interfaces, can result in insecure or poorly designed code that exposes sensitive components to exploitation. A promising approach to addressing this issue is the integration of capabilities into programming languages. Capabilities provide fine-grained access control to system resources and modules, ensuring that each component operates with the minimal privileges necessary. This aligns with the Principle of Least Authority [2], which restricts access to only what is necessary for functionality. For example, module A can interact with module B or access the file system only if it possesses the appropriate capabilities, thereby minimizing unnecessary exposure to sensitive resources. Several programming languages use capabilities as an access-control mechanism. Existing research has focused on languages that incorporate capabilities, such as Wyvern [3] and Pony [4].

Traditionally, capabilities are modeled as objects that encapsulate both data and behavior, making the implementation complex and tedious. However, recent advancements in programming language theory, such as the λ_E calculus, propose an alternative: first-class modules as a means of modeling capabilities [1]. This approach not only enhances security with access control but also increases the expressiveness of the programming language.

1.2 *First-Class* Modules

The term ”first-class” refers to entities that can be dynamically created, passed as arguments, returned from functions, and manipulated at runtime. This concept enhances modularity, expressiveness, and flexibility in programming, enabling developers to design systems with fine-grained control over resource access and behavior.

Many modern programming languages support first-class classes [5], including Python [6], JavaScript [7], and Racket [8]. These languages allow classes to be treated as first-class entities, meaning they can be created dynamically, passed as arguments, or returned from functions. This capability is particularly useful for implementing advanced features such as mixins, traits, and dynamic dispatch.

To illustrate the concept of first-class modules, consider the following example in OCaml [9]:

```
(* Define a module type *)
module type Show = sig
  val show : int
end;;

(* Creating a module  *)
module Three : Show = struct let show = 3 end;;

(* Unpacking a module and storing it is a first-class expression *)
let three = (module Three : Show);;

(* A list with module three and an anonymous module *)
let modules = [three; (module struct let show = 4 end)];;

(* Now, we take the anonymous module and unpack it into a normal
    module *)
module Four = (val (List.nth modules 1) : Show);;

(* Result: 4 *)
Four.show
```

A module type **Show** is defined, and an ordinary module **Three** is created of type *Show*. The module can be packed into a first-class module and stored in a list with an anonymous module. This demonstrates the expressiveness of *first-class modules* and showcases the dynamic creation and manipulation of modules at runtime.

1.3 Outline and contributions

This project explores the utility of first-class environments by modeling capabilities as *first-class* modules. We introduce a new programming language, ENVCAP, whose operational semantics are defined via type-directed elaboration into the λ_E calculus. The key contributions of this interim report are as follows:

- **Extension of the λ_E Calculus**: λ_E is extended with additional types, recursion, control structures, and algebraic data types. This report provides extended big-step operational semantics and bidirectional typing rules.
- **Design of ENVCAP**: The syntax and module system of ENVCAP are presented, along with its type-directed elaboration rules into the λ_E calculus. This design enables the modeling of capabilities as first-class modules.
- **Implementation in Haskell**: An interpreter implementation of ENVCAP is provided in Haskell, validated through unit- and property-based testing. Case studies demonstrating the implementation of factorial and Fibonacci are included to illustrate the language's features and capabilities.

2 Methodology

This section outlines the systematic approach taken in the design, implementation, and testing of the ENVCAP programming language. The methodology consists of two primary components: Design and Implementation (Section 2.1), and Testing and Correctness (Section 2.2).

2.1 Design and Implementation

The design of ENVCAP's syntax is guided by two key features: *first-class modules* and *expressiveness*. A programming language must be expressive enough to allow developers to achieve their desired functionality, while also ensuring that the module system demonstrates the research goal of this projecty. This programming language is mainly a research prototype and hence, the primary design goal is to show case utility of *first-class environments and modules*.

There are two primary approaches to implementing a programming language: using an interpreter or a compiler. For this project, an interpreter was chosen over a compiler. Interpreters are better suited for prototyping new languages, as they avoid the additional complexity of code generation required by compilers. Since ENVCAP is a prototype language, an interpreter is the most suitable choice. The structure of an interpreter is illustrated in Figure 1.

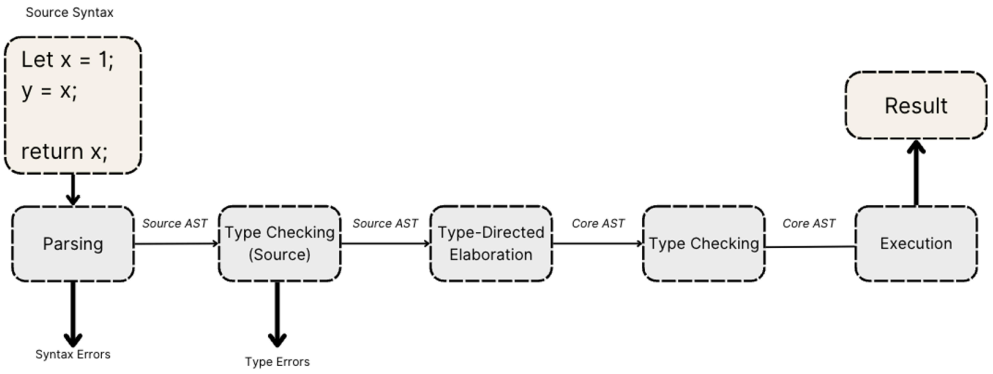

Figure 1: Structure of an interpreter

An interpreter processes code written in ENVCAP syntax through several stages. Initially, the code is parsed by a Parser, producing a Surface AST. If the programmer fails to adhere to ENVCAP's syntax rules, syntax errors are generated during this parsing phase. The AST serves as a data structure that represents the

code's structure. Subsequently, the Surface AST undergoes elaboration, translating it into a Core AST. This Core AST is then subjected to type checking, ensuring compliance with the typing rules established in the λ_E calculus. Type errors are raised if any violations occur. Upon successful type checking, the Core AST is executed according to the simplification rules of Core λ_E calculus, and the results are returned to the programmer.

The programming language used for the implementation of this project is Haskell. Haskell is a purely functional language, making it an excellent choice for research projects where correctness and reasoning are crucial. Unlike many other languages, Haskell has no side effects, and code is written as pure functions, where a given set of inputs always produces the same output. In addition, Haskell offers features like pattern-matching, which makes it well-suited for implementing compilers and interpreters.

The parser for the interpreter is written using a combination of the parser combinator library *Parsec* [10] and a parser generator *Happy*[11]. Utilizing a parser generator helps to define the syntax and grammar in a non-ambiguous manner. *Happy* generates a parser based on the EBNF grammar form and then, once the design is finalized, the specific feature is added to the hand-written parser with better error handling and exceptions using the *Parsec* library.

2.2 Testing and Correctness

Given the emphasis on correctness in this project, multiple levels of testing were employed to ensure its reliability. While implementations typically rely on unit testing, where programmers manually write test cases, this method may not encompass all possible edge cases. As such, property-based testing [12] is also utilized. This technique tests the properties of the code through the automatic generation of hundreds of random test cases, scrutinizing whether the code adheres to a specific property for each test case. Any deviation results in the production of a counter-example for that specific property. The random generation of test cases significantly enhances the quality of the implementation by covering a broad spectrum of code behavior, unlike unit testing, which is restricted by the programmer's ability to anticipate potential edge cases.

Initial testing is conducted on the code using the unit testing approach to ensure comprehensive coverage. This method allows for the early detection of basic errors, an advantage given the potentially time-consuming nature of property-based testing. Hspec [13] is employed to write unit tests, each of which focuses on a specific aspect of the codebase's functionality. The aim is to create sufficient unit tests to guarantee complete coverage, thus effectively testing the entire codebase.

module	Top Level Definitions			Alternatives			Expressions		
	%	covered / total		%	covered / total		%	covered / total	
module EnvCap-0.1.0.0-inplace/LambdaE.BigStep	100%	4/4		84%	22/26		82%	88/107	
module EnvCap-0.1.0.0-inplace/LambdaE.Syntax	52%	11/21		100%	5/5		100%	13/13	
module EnvCap-0.1.0.0-inplace/LambdaE.TypeChecker	83%	5/6		53%	21/39		49%	79/160	
Program Coverage Total	64%	20/31		68%	48/70		64%	180/280	

Figure 2: Coverage report generated by Hspec

As demonstrated in Figure 2, Hspec generates a coverage report detailing the percentage of code tested with unit tests. Complete coverage will display 100% for all modules in the coverage report. In addition, Figure 3 highlights any untested code to guide further unit testing. Upon achieving complete code coverage, property-based testing is undertaken using another Haskell package, Quickcheck [14].

```
17  -- containment
18  -- checks if no duplicate labels exist
19  containment :: Typ -> Typ -> Bool
20  containment (TRecord l tA) (TRecord label typ ) =
21      l == label && tA == typ
22  containment (TRecord l tA) (TAnd tB tC)         =
23      (containment (TRecord l tA) tB && not (isLabel l tC)) ||
24      (containment (TRecord l tA) tC && not (isLabel l tB))
25  containment _ _                                 = False
```

Figure 3: Highlighted areas indicate untested code segments.

3 Progress

Below is an overview of the progress made so far. The design of the language is almost complete, except the Module system is in the process of design and implementation. Firstly, we go through the extensions to the core calculus followed by an overview of the design and syntax of ENVCAP and some programs implemented in it.

3.1 λ_E extensions

λ_E is an environment-based calculus. Essentially, it has the environments and closures, that are commonly used in programming language implementation. The syntax, bi-directional typing rules, and semantics of λ_E are provided in Appendix A, and further details can be found in the paper *A Case for First-Class Environments* [1]. To achieve the usability of an intermediate programming language, the core calculus needs to be extended with basic constructs, e.g. recursion and conditionals, and some advanced ones, e.g. algebraic data types.

3.1.1 Base Data Types

λ_E currently only supports the Int type. The calculus is further extended with the $Bool$ and $String$ to support booleans and strings. Below are the typing rules and semantics for the extension.

TYP-BOOL	TYP-STRING	BSTEP-BOOL	BSTEP-STR
$\dfrac{}{\Gamma \vdash b : \text{Bool}}$	$\dfrac{}{\Gamma \vdash s : \text{String}}$	$\dfrac{}{v \vdash b \Rightarrow \text{b}}$	$\dfrac{}{v \vdash s \Rightarrow s}$

3.1.2 General Recursion

Recursion is a fundamental feature in programming languages, enabling functions to call themselves and solve problems by breaking them into smaller subproblems. To support recursion at the source level, the core calculus must provide a mechanism for self-reference, such as the *fixpoint operator* (Fix). The Fix operator computes the fixed point of a function f, satisfying $f(\text{Fix}(f)) = \text{Fix}(f)$, and allows recursive definitions without explicit self-reference. For example, a recursive factorial function can be defined as fact $= \text{Fix}(\lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times f(n-1))$. While alternatives like the *Y-combinator* exist, Fix is often preferred for its simplicity and explicitness, making it a practical choice for implementing recursion in core calculi.

BSTEP-FIX

$$\frac{v \vdash \lambda A.e_1 \Rightarrow \langle v_1, \lambda A.e_1 \rangle}{v \vdash \text{Fix } \lambda A.e_1 \Rightarrow \langle v_1 \text{ , } \langle v_1, \text{Fix } \lambda A.e_1 \rangle, \text{Fix } \lambda A.e_1 \rangle}$$

BSTEP-FixApp

$$\frac{v \vdash (\text{Fix } e_1) \Rightarrow \langle v_1, \text{Fix } \lambda A.e \rangle \quad v \vdash e_2 \Rightarrow v_2 \quad v_1 \text{ , } v_2 \vdash e \Rightarrow v'}{v \vdash (\text{Fix } e_1)\ e_2 \Rightarrow v'}$$

In λ_E calculus, de Bruijn indices are utilised. Hence, BSTEP-FIX loads the fix-points closure into the environment of the resulting closure so that the function is capable of referring to itself in a locally nameless style. Below is the typing rule for the Fix:

TYP-FIX

$$\frac{\Gamma \& A \to A \vdash e : B}{\Gamma \vdash \text{Fix } \lambda A.e : B}$$

3.1.3 Let expression and conditionals

Conditionals and `let` expressions are fundamental constructs in programming languages. In this project, the `if` conditional is directly included in the core calculus. While Church encodings could be used to represent conditionals, adding `if` directly simplifies the design and allows focus on the core language features. Similarly, `let` expressions could be desugared into function applications, but they are included explicitly for clarity. These simplifications can be explored later for a more minimalistic core calculus, but for now, both constructs are added directly.

BSTEP-IF_True

$$\frac{v \vdash e_1 \Rightarrow \text{True} \quad v \vdash e_2 \Rightarrow v_2}{v \vdash \text{if } e_1\ e_2\ e_3 \Rightarrow v_2}$$

BSTEP-IF_False

$$\frac{v \vdash e_1 \Rightarrow \text{False} \quad v \vdash e_3 \Rightarrow v_3}{v \vdash \text{if } e_1\ e_2\ e_3 \Rightarrow v_3}$$

BSTEP-LET

$$\frac{v \vdash e_1 \Rightarrow v_1 \quad v \text{ , } v_1 \vdash e_2 \Rightarrow v_2}{v \vdash \text{let } e_1\ e_2 \Rightarrow v_2}$$

Below are the typing rules:

TYP-LET

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \text{ , } A \vdash e_2 : B}{\Gamma \vdash \text{let } e_1\ e_2 : B}$$

TYP-IF

$$\frac{\Gamma \vdash e_1 \Rightarrow \text{Bool} \quad \Gamma \vdash e_2 \Rightarrow A \quad \Gamma \vdash e_3 \Rightarrow A}{\Gamma \vdash \text{if } e_1\ e_2\ e_3 \Rightarrow A}$$

3.1.4 Built-in Lists

Lists in λ_E are implemented as a built-in type with two constructs:

- `nil T`: An empty list of type `T`.
- `cons e1 e2`: A list with head `e1` (type `T`) and tail `e2` (type `List T`).

The `lcase` construct enables pattern matching on lists:

- **Empty List (`nil T`)**: Executes an expression for the empty case.
- **Non-empty List (`cons v1 v2`)**: Executes an expression with access to the head (`v1`) and tail (`v2`).

Typing Rules

TYP-Nil

$$\frac{}{\Gamma \vdash \text{nil}\ T : \text{List}\ T}$$

TYP-Cons

$$\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : \text{List}\ T}{\Gamma \vdash \text{cons}\ e_1\ e_2 : \text{List}\ T}$$

TYP-LCASE

$$\frac{\Gamma \vdash e_1 : \text{List}\ T_1 \quad \Gamma \vdash e_2 : T_2 \quad \Gamma\text{ , }T_1\text{ , List}\ T_1 \vdash e_3 : T_2}{\Gamma \vdash \text{lcase}\ e_1\ e_2\ e_3 : T_2}$$

Big-Step Semantics

BSTEP-Nil

$$\frac{}{v \vdash \text{nil}\ T \Rightarrow \text{nil}\ T}$$

BSTEP-Cons

$$\frac{v \vdash e_1 \Rightarrow v_1 \quad v \vdash e_2 \Rightarrow v_2}{v \vdash \text{cons}\ e_1\ e_2 \Rightarrow \text{cons}\ v_1\ v_2}$$

BSTEP-LCASE_Nil

$$\frac{v \vdash e_1 \Rightarrow \text{nil}\ T \quad v \vdash e_2 \Rightarrow v_2}{v \vdash \text{lcase}\ e_1\ e_2\ e_3 \Rightarrow v_2}$$

BSTEP-LCASE_Cons

$$\frac{v \vdash e_1 \Rightarrow \text{cons}\ v_1\ v_2 \quad v\text{ , }v_1\text{ , }v_2 \vdash e_3 \Rightarrow v_3}{v \vdash \text{lcase}\ e_1\ e_2\ e_3 \Rightarrow v_3}$$

3.1.5 ADTs: Sums and Pairs

Algebraic Data Types (ADTs) in λ_E include sums (disjoint unions) and pairs (product types). Pairs allow grouping two values into a single entity, while sums represent disjoint unions, enabling values of different types to coexist. The semantics for these constructs are defined as follows:

$$\text{BSTEP-PAIR}\ \frac{v \vdash e_1 \Rightarrow v_1 \quad v \vdash e_2 \Rightarrow v_2}{v \vdash (e_1, e_2) \Rightarrow (v_1, v_2)} \qquad \text{BSTEP-Fst}\ \frac{v \vdash e \Rightarrow (v_1, v_2)}{v \vdash \mathsf{Fst}\ e \Rightarrow v_1} \qquad \text{BSTEP-Snd}\ \frac{v \vdash e \Rightarrow (v_1, v_2)}{v \vdash \mathsf{Snd}\ e \Rightarrow v_2}$$

$$\text{BSTEP-Inl}\ \frac{v \vdash e \Rightarrow v_1}{v \vdash \mathsf{inl}\ T\ e \Rightarrow \mathsf{inl}\ T\ v_1} \qquad \text{BSTEP-Inr}\ \frac{v \vdash e \Rightarrow v_1}{v \vdash \mathsf{inr}\ T\ e \Rightarrow \mathsf{inr}\ T\ v_1}$$

$$\text{BSTEP-CASEInl}\ \frac{v \vdash e_1 \Rightarrow \mathsf{inl}\ T\ v_1 \quad v\ ,\ v_1 \vdash e_2 \Rightarrow v_2}{v \vdash \mathsf{case}\ e_1\ e_2\ e_3 \Rightarrow v_2} \qquad \text{BSTEP-CASEInr}\ \frac{v \vdash e_1 \Rightarrow \mathsf{inr}\ T\ v_1 \quad v\ ,\ v_1 \vdash e_3 \Rightarrow v_3}{v \vdash \mathsf{case}\ e_1\ e_2\ e_3 \Rightarrow v_3}$$

Pairs are constructed using (e_1, e_2), with `Fst` and `Snd` for projections. These constructs will also be simplified with the current $e.n$ operator supported in the core calculus as the work on elaboration from ENVCAP to core progresses. Sums are constructed using `inl T e` and `inr T e`, with `case` for pattern matching. For example, $(1, \text{True})$ creates a pair, and `case (inl Int 5) (...) (...)` matches on the left variant of a sum. These constructs provide a flexible and type-safe way to model complex data structures in λ_E.

Sums and pairs in λ_E provide a flexible and type-safe way to model complex data structures, enabling expressive programming patterns.

3.1.6 Arithmetic, Boolean, and Comparison Operators

Basic operations are added to the core level and these are mainly supported by the corresponding operations in the Haskell itself. The typing rules and semantics are similar to those of the application.

3.2 ENVCAP Programming Language

The implementation structure is mainly divided into a minimalistic core calculus and the source language. The ENVCAP syntax is parsed into a source-level AST which then goes through the process of desugaring and elaboration. For example, de Bruijn indices are taken care of at the source level and then, the expression is translated to the core expression where it gets type-checked before execution. Eventually, once the design of ENVCAP is finalized, the typing rules will also be defined at the source level and then, type-directed elaboration will be performed to the core calculus. If the source level is well-typed, then the core expression must also be well-typed and both must have the same type. Below is a brief overview of the Syntax and Design of ENVCAP is briefly reviewed followed by sample programs that can be fully executed in the interpreter currently.

Basic Constructs

Literals

- Integers: `1`, `42`, `-5`
- Booleans: `True`, `False`

Variables

- Variable names: `x`, `y`, `myVar`
- Example: `x`, `myVar`

Arithmetic Operations

Operators

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Modulus: `%`

Examples

```
1 + 2
3 * 4
10 / 2
5 % 2
```

Comparison Operations

Operators

- Greater than or equal: `>=`
- Greater than: `>`
- Equal: `==`
- Not equal: `!=`
- Less than: `<`
- Less than or equal: `<=`

Examples

```
1 < 2
2 <= 3
3 > 2
4 >= 1
1 == 1
2 != 3
```

Boolean Operations

Operators

- Logical AND: `&&`
- Logical OR: `||`

Examples

```
True && False
False  True
```

Control Flow

If-Then-Else

```
if (condition) then { statements } else { statements }
```

Example:

```
if (x > 0) then { x } else { 0 }
```

Functions

Function Definition

```
function name(param1: Type, ... , param2: Type) { statements }
```

Example:

```
function add(x: Int, y: Int) { x + y }
```

Lambda Functions

```
\(param1: Type, param2: Type) => { statements }
```

Example:

```
def add = \(x: Int, y: Int) => { x + y }
```

Bindings

Binding

```
def {binding-name} = {expression}
```

Example:

```
def x = 10
```

The variable bindings can not be updated as the ENVCAP does not support variable updates. Hence, the language is purely functional rather than imperative (even if the syntax looks similar to that of an imperative language).

```
def x = \(x:Int) => {x + 1}
```

Note: This is a valid program since the implementation adopts a locally nameless representation.

Context

Context Query

This operator returns the current environment in the execution and allows the programmer to manipulate it and hence, *first-class environments* are supported.

```
?
```

Example:

```
def x = 10; def y = 5; def k = ?
```

This code will store the environment in k.

Statements

Multiple Statements

Multiple statements are connected via the dependent merge `,` operator.

```
statement1; statement2
```

Example:

```
def x = 1; def y = 2
```

Types

Basic Types

- `Int`: Integer type.
- `Bool`: Boolean type.
- `String`: String type.

Function Types

```
Type1 -> Type2
```

Example:

```
Int -> Int
```

Record Types

```
{ fieldName: Type }
```

Example:

```
{ name: String, age: Int }
```

List Types

```
[Type]
```

Example:

```
[Int]
```

Records

Record Construction

```
{ fieldName = value }
```

Example:

```
{ name = "Alice", age = 30 }
```

Field Access

```
record.label
```

Example:

```
def x = 15;
def y = 20;
def z = ?.x
-- z gets value 15
```

Lists

List Construction

```
[element1, ..., element2]
```

Example:

```
[1, 2, 3]
```

Pattern matching on lists

```
[element1, ..., element2]
```

Example:

```
function reverse (ls: [Int]) {
    match ls of
        []      =>  { [] }
        (x:xs)  => { reverse (xs) ++ [x] }
};
reverse([1, 2, 3])
-- result is [3, 2, 1]
```

Tuples

Tuple construction

```
(item1, item2, item3, ..., itemN)
```

Example:

```
(false, 10, true, "hello", 1001)
```

Tuple is simply a generalization of the products. However, the construct can be potentially modeled with intersections and merges.

Precedence and Associativity

Operator Precedence (from highest to lowest)

1. `*`, `/`, `%`
2. `+`, `-`
3. `>=`, `>`, `==`, `!=`, `<`, `<=`
4. `&&`
5. `||`
6. `=`, `=>`

Associativity

- Left-associative: `+`, `-`, `*`, `/`, `%`, `&&`, `||`
- Right-associative: `=`, `=>`

Sample Programs

Below programs can be parsed, elaborated, type-checked and executed completely in the current ENVCAP interpreter.

Fibonacci

The Fibonacci sequence is computed using recursion:

```
function inc(i : Int) { i + 1 };

function fibonacci(n: Int) {
    if (n == 0  n == 1) then {
        1
    } else {
        fibonacci(n - 1) + fibonacci(n - 2)
    }
};

fibonacci(inc(10))
```

Factorial

The factorial of a number is computed using recursion and a lambda function:

```
def dec = \(n : Int) => { n - 1 };

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

factorial(5)
```

First-Class Functions

ENVCAP supports first-class functions, allowing functions to be passed as arguments and returned as values:

```
function square(double : Int -> Int, n : Int) {
    double(n)
};

function fancy(n : Int) {
    \(x : Int) => { x + square(\(n: Int) => { n * 2 }, 10) }
};

def temp = fancy(10);
temp(20)
```

Another example of first-class functions:

```
((\(call : Int -> Int, n : Int) => { call(n) })(\(n : Int) => { n +
    10 }))(10)
```

Conditionals

Conditionals are used to implement a prime-checking function:

```
def n = 11;

function isPrime(i : Int) {
    if (n < 2)
        then { 0 }
        else {
            if ((i * i) > n)
                then { 1 }
                else {
                    if (n % i == 0)
                        then {
                            0
                        } else {
                            isPrime(i + 1)
                        }
                }
            }
};

isPrime(11) == 1
```

4 Further plan

The original project schedule is outlined in Table 1. The core implementation, including extensions, has been completed. Currently, work is underway on the design of the module system for ENVCAP. The implementation of the interpreter is expected to be finished by February 2025, followed by rigorous testing with Property-Based and Unit tests by March 2025, and extensive documentation by April 2025. The project is on track with progress; the design of the module system is nearly complete, and the implementation of the module system is the next step. Additionally, tooling support for the interpreter, including REPL, Pretty Printing, and a step-wise debugger, will be added.

Milestones	**Expected Completion Date**
Literature Review	September, 2024
Core Implementation	October, 2024
Extension of Core λ_E	November, 2024
Design of ENVCAP Syntax	December, 2024
Interpreter for ENVCAP	January - February, 2025
Testing and Tools	March, 2025
Extensive Documentation	April, 2025

Table 1: Schedule

However, this project is a research prototype and there are a few research problems that this research aims to solve. Below are the further goals that are beyond the scope of FYP.

- Formalization of the type-directed elaboration from the ENVCAP to λ_E using the Rocq Theorem Prover and proving theorems on type-safety and uniqueness of the elaboration.
- Add separate compilation/type checking to the design of ENVCAP using the dependent merges and intersection types. Formalize the meta-theory of the linking process in Rocq.
- Simplify λ_E with generalized box constructs and formalize in Rocq Theorem Prover.
- Extend λ_E with subtyping.
- Implement a simple online interactive playground for ENVCAP with syntax highlighting.

- Create a simple VS code extension for ENVCAP syntax highlighting.

Even though the above goals are beyond the scope of FYP, the final report may contain the results if any of the above goals are achieved.

References

[1] J. Tan and B. C. d. S. Oliveira, "A case for first-class environments," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, p. 30, Oct. 2024. doi: 10.1145/3689800. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3689800.

[2] D. Melicher, Y. Shi, A. Potanin, and J. Aldrich, "A capability-based module system for authority control," in *31st European Conference on Object-Oriented Programming (ECOOP) June 19-23, Barcelona, Spain*, P. Müller, Ed., ser. LIPIcs, vol. 74, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 20:1–20:27. doi: 10.4230/LIPICS.ECOOP.2017.20. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2017.20.

[3] ——, "A Capability-Based Module System for Authority Control," in *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, P. Müller, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 74, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, 20:1–20:27, isbn: 978-3-95977-035-4. doi: 10.4230/LIPIcs.ECOOP.2017.20. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2017.20.

[4] G. Steed and S. Drossopoulou, "A principled design of capabilities in pony," 2016. [Online]. Available: https://www.ponylang.io/media/papers/a_prinicipled_design_of_capabilities_in_pony.pdf.

[5] M. Flatt, R. B. Findler, and M. Felleisen, "Scheme with classes, mixins, and traits," in *Proceedings of the 4th Asian Conference on Programming Languages and Systems*, ser. APLAS'06, Sydney, Australia: Springer-Verlag, 2006, pp. 270–289, isbn: 3540489371. doi: 10.1007/11924661_17. [Online]. Available: https://doi.org/10.1007/11924661_17.

[6] Python Software Foundation, *Python*, Accessed: 2023-10-30. [Online]. Available: https://www.python.org/.

[7] JavaScript, *Javascript*, Accessed: 2023-10-30. [Online]. Available: https://www.javascript.com/.

[8] Racket, *Racket*, Accessed: 2023-10-30. [Online]. Available: https://racket-lang.org/.

[9] OCaml, *Ocaml*, Accessed: 2023-10-30. [Online]. Available: https://ocaml.org/.

[10] *Parsec: Monadic parser combinators*, Hackage, Version 3.1.18.0, 2025. [Online]. Available: https://hackage.haskell.org/package/parsec.

[11] *Happy: The happy parser generator for haskell*, GitHub repository, 2025. [Online]. Available: https://github.com/haskell/happy.

[12] G. Fink and M. Bishop, "Property-based testing: A new approach to testing for assurance," *SIGSOFT Softw. Eng. Notes*, vol. 22, no. 4, pp. 74–80, Jul. 1997, issn: 0163-5948. doi: 10.1145/263244.263267. [Online]. Available: https://doi.org/10.1145/263244.263267.

[13] *Hspec: A testing framework for haskell*, Version 2.11.10, 2024. [Online]. Available: https://hackage.haskell.org/package/hspec.

[14] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," *SIGPLAN Not.*, vol. 35, no. 9, pp. 268–279, Sep. 2000, issn: 0362-1340. doi: 10.1145/357766.351266. [Online]. Available: https://doi.org/10.1145/357766.351266.

A Appendix A

A.1 Syntax of λ_E

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

A.2 Bi-directional Typing rules of λ_E

$\Rightarrow$ infer
$\Leftarrow$ check

$\boxed{\ell : A \in B}$ **(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
$$\frac{\ell : A \in C \quad \ell \notin \text{label(B)}}{\ell : A \in B \,\&\, C}$$

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

TYP-CTX
$$\frac{}{\Gamma \vdash ? \Rightarrow \Gamma}$$

TYP-PROJ
$$\frac{\Gamma \vdash e \Rightarrow B \quad lookup(B, n) = A}{\Gamma \vdash e.n \Rightarrow A}$$

TYP-LIT
$$\frac{}{\Gamma \vdash i \Rightarrow Int}$$

TYP-TOP
$$\frac{}{\Gamma \vdash \varepsilon \Rightarrow \varepsilon}$$

TYP-Chk
$$\frac{\Gamma \vdash e \Rightarrow A \quad A = B}{\Gamma \vdash e \Leftarrow B}$$

TYP-BOX
$$\frac{\Gamma \vdash e_1 \Rightarrow \Gamma_1 \quad \Gamma_1 \vdash e_2 \Rightarrow B}{\Gamma \vdash e_1 \triangleright e_2 \Rightarrow B}$$

TYP-MERGE
$$\frac{\Gamma \vdash e_1 \Rightarrow A \quad \Gamma \,\&\, A \vdash e_2 \Rightarrow B}{\Gamma \vdash e_1 \,,\, e_2 \Rightarrow A \,\&\, B}$$

TYP-LAM
$$\frac{\Gamma \,\&\, A \vdash e : B}{\Gamma \vdash \lambda A.e : A \to B}$$

TYP-APP
$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \quad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\, e_2 \Rightarrow B}$$

TYP-CLOS
$$\frac{\Gamma \vdash v \Rightarrow \Gamma_1 \quad \Gamma_1 \& A \vdash e \Rightarrow B}{\Gamma \vdash \langle v, \lambda A.e\rangle \Rightarrow A \to B}$$

TYP-RCD
$$\frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash \{l = e\} \Rightarrow \{l : A\}}$$

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

A.3 Big-step operational semantics

BSTEP-CTX
$$\frac{}{v \vdash ? \Rightarrow v}$$

BSTEP-PROJ
$$\frac{v \vdash e \Rightarrow v_1}{v \vdash e.n \Rightarrow lookupv(v_1, n)}$$

BSTEP-LIT
$$\frac{}{v \vdash i \Rightarrow i}$$

BSTEP-CLOS
$$\frac{}{v \vdash \langle v_1, \lambda A.e\rangle \Rightarrow \langle v_1, \lambda A.e\rangle}$$

BSTEP-UNIT
$$\frac{}{v \vdash \varepsilon \Rightarrow \varepsilon}$$

BSTEP-MERGE
$$\frac{v \vdash e_1 \Rightarrow v_1 \quad v \;,\; v_1 \vdash e_2 \Rightarrow v_2}{v \vdash e_1 \,,\, e_2 \Rightarrow v_1 \;,\; v_2}$$

BSTEP-BOX
$$\frac{v \vdash e_1 \Rightarrow v_1 \quad v_1 \vdash e_2 \Rightarrow v_2}{v \vdash e_1 \triangleright e_2 \Rightarrow v2}$$

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

BSTEP-LAM
$$\frac{}{v \vdash \lambda A.e \Rightarrow \langle v_1, \lambda A.e\rangle}$$

BSTEP-RCD
$$\frac{v \vdash e \Rightarrow v_1}{v \vdash \{l = e\} \Rightarrow \{l = v_1\}}$$

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