Capabilities as First-Class Modules with Separate Compilation

Jam Kabeer Ali Khan
jamkhan@connect.hku.hk
The University of Hong Kong
Hong Kong SAR, China

ABSTRACT

We present **ENVCAP**, a statically typed programming language based on environment-based semantics that supports ***first-class* environments**, **capabilities**, and **separate compilation**. By utilizing the environment-based semantics of λ_E [15], ENVCAP models capabilities [5] as *first-class* modules [15], as an alternative to the object-capability model [12], and enables separate compilation without *extra-linguistic* structures such as *linksets* [2].

1 PROBLEM & MOTIVATION

Programming language implementations often use environments—maps of bindings and values—for efficiency, while theoretical calculi rely on substitution, weakening the correctness guarantees of implementations. Environment-based semantics [4, 8, 14, 15] address this gap by incorporating environments into formal calculi, enabling *first-class* environments—environments as values that can be manipulated. This approach provides a foundation for addressing two key problems in language design and implementation: (1) capabilities are commonly modeled as objects [5], which introduces complexity due to object-oriented calculi, making reasoning about and implementing capabilities challenging; and (2) traditional approaches to separate compilation rely on external structures, such as *linksets* [2], which are not part of the core language.

ENVCAP is the first programming language to address these problems by providing two key solutions based on *first-class* environments:

(1) **Capabilities as *First-Class* Modules**: ENVCAP models capabilities as *first-class* modules [15] using *first-class* environments, offering a simpler alternative to the object-capability model [12].
(2) **Separate Compilation**: ENVCAP enables separate compilation entirely within the core λ_E language, eliminating the need for *extra-linguistic* structures such as linksets.

2 BACKGROUND & RELATED WORK

The λ calculus relies on substitution-based semantics, where beta-reduction replaces terms during application. This approach is inefficient in practice and challenging to formalize due to issues like name capture. Environment-based semantics address these limitations by integrating environments—maps of bindings and values—directly into the formalization, aligning with implementation practices. The λ_E [15] and E_i [14] calculi formalize this approach, unifying expressions and environments to enable *first-class* environments and introducing the box construct $e_1 \rhd e_2$, which evaluates e_2 under environment e_1, allowing for modeling capabilities as *first-class* modules.

Capabilities [5] enforce access control, commonly implemented using the object-capability model in programming languages such as Newspeak [1] and Wyvern [7, 11]. As their name implies, capabilities are primarily designed to restrict access to resources. When modeled as first-class modules, capabilities can be passed as arguments, effectively granting access to specific resources. Tan and Oliveira [15] proposed sandboxed *first-class* modules as an alternative, where sandboxed modules restrict access to the global environment by requiring all dependencies to be explicitly passed as arguments, allowing to model capabilities. In addition, most capability-based languages lack formalization for separate compilation, which is essential to maintain authority control.

Separate compilation enables programs to type-check and compile based on interfaces rather than implementations. Cardelli [2] introduced *linksets* as a formal framework for separate compilation, later adapted by Standard ML [13]. In this framework, *linksets* provides a structure to link code components post-compilation, formalized through a simple module system called *bindings*. However, *linksets* are extra-linguistic—they are not part of the core language and lack concrete syntax, complicating implementation.

A simpler and more generalized solution could leverage core language constructs, where program fragments act as abstractions over interfaces, and interfaces are unified with types [14]. For instance, if component A is imported by component B, it can be represented as $\lambda(\text{interface } A).B$. Using first-class environments in λ_E, ENVCAP enables separate compilation within the core language, eliminating the need for extra-linguistic mechanisms. Additionally, Cardelli's framework lacks support for first-class modules. This work extends Cardelli's framework by replacing *linksets* with constructs from the λ_E calculus, offering a more expressive and streamlined solution.

3 APPROACH & UNIQUENESS

The semantics of ENVCAP are derived from λ_E via elaboration and syntax design is inspired by OCaml [9], Wyvern [7, 11], and λ_E.

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

Figure 1: Core ENVCAP syntax.

3.1 Syntax & Design

Figure 1 presents the core syntactic constructs of ENVCAP and we discuss the most relevant constructs. A fragment (`program` $S\ \ I\ \ R\ \ E\ \ A$) represents an implementation paired with an interface A, where interfaces desugar into types. First-class modules include `struct` $A\ \ E$ (taking A as input) and `struct` E (syntactic sugar for E). The construct `module` $\ell : B\ \ E$ desugars into a record $\{\ell = E : B\}$, while `functor` $\ell\ \ A : B\ \ E$ desugars into $\{\ell = \texttt{struct}\ \ A\ \ (E : B)\}$. Module application ($E_1 * E_2$) requires E_1 to have type `Sig[`A`,` B`]`. The `open` $E_1\ \ E_2$ construct requires E_1 to be of record type and loads its contents into the current context via elaboration. The reification operator `env` retrieves the current environment. Sequences $E_1; E_2$ elaborate to dependent merges ($e_1 \,{,}\, e_2$) in λ_E, which is helpful to model a sequence of declarations, e.g. $\{x = 1\} \,{,}\, \{y = x + x\}$, allowing E_2 to depend on E_1. Pairs (E_1, E_2) are non-dependent and generalized to support *tuples*, but elaborate to $e_1 \,{,}\, e_2$ in λ_E. The construct `function` $\ell\ \ A : B\ \ E$ elaborates into `fix` $A \to B.E$, enabling recursion.

3.2 *First-class* Environments & Modules

We illustrate *first-class* environments and modules with an example. First, we define a @pure fragment with $UTIL$ and $MATH$ interfaces:

```
@pure module Example1
interface UTIL {val diff : Int};
interface MATH {val fact : Int -> Int};
```

Next, we define a functor (a parameterized module) that desugars into a labeled record with an anonymous *first-class* module (*struct A E*):

```
functor math (util: UTIL) : MATH {
    (* loads contents into environment, e.g. diff *)
    open util;
    function fact(n: Int): Int {
        if (n == 0) then 1 else n * fact(n - diff)
    }
};
```

Finally, we use the *with* E_1 E_2 construct to evaluate E_2 in environment E_1. We create a new environment containing the current environment (extracted via the *env* operator), the result of applying the *math* functor to an anonymous module of type $UTIL$, and a new binding for x. The computation proceeds under this new environment, computing factorials for both the old and new values of x:

```
let x = 5;
with
({prevEnv = env}; math(struct {let diff = 1}); {x  = 6})
in { let resultOld = fact(prevEnv.x);  (* 120 *)
     let resultNew = fact(x)           (* 720 *)}
```

This example demonstrates ENVCAP's support for *first-class* modules and environments, showcasing its expressive power.

3.3 Capabilities as *First-Class* Modules

Capabilities in ENVCAP are modeled using **sandboxed first-class modules**, which enforce controlled access by requiring all resources to be explicitly passed as parameters, and **authority annotations** (`@pure` and `@resource`) inspired by Wyvern [11]. The modules are sandboxed via elaboration to $\epsilon \rhd \lambda A.e$ in λ_E, so computation runs under *empty* (ϵ) environment and hence, resources are only passed via parameters. A `@resource` fragment can import any fragment, while a `@pure` fragment can only import other `@pure` fragments. If a `@pure` fragment requires functionality from a `@resource` fragment, it must declare it as a requirement rather than an import, ensuring explicit capability propagation. ENVCAP supports the use of interface files in requirements, so interfaces of required fragments can be utilized for separate compilation. Only valid imports are linked and the validity of imports with authority annotations is checked during elaboration to λ_E, maintaining authority control.

We present an example of how capabilities are propagated in ENVCAP, consisting of a pure fragment B and a resource fragment A.

```
@pure module B
require (U: System.Utils);
let mapList =
    U.Map(\(x:Int) => {x + 1}, [1, 2, 3])
```

Fragment B is a `@pure` module requiring `System.Utils` as a capability. Since `System.Utils` is a resource, it cannot be imported directly. Instead, it is passed as a parameter, ensuring explicit capability propagation.

```
@resource module A
import System.Utils B;

let result = B(System.Utils).mapList
```

Fragment A, a `@resource` module, imports `System.Utils` and instantiates B with it. The `mapList` function is extracted and assigned to the result. Here, `System.Utils` acts as a capability: A has the authority to access and pass it to B, which requires this capability.

3.4 Separate Compilation with Unified Types and Interfaces

We enable separate compilation by unifying interfaces and types, where an implementation file is treated as an abstraction over the interfaces of its imports at the core level of λ_E. The linking mechanism is also defined at the core level, eliminating the need for external structures.

3.4.1 Example. To illustrate this, we adapt one of Cardelli's examples [2] in ENVCAP. For clarity, we present an example within a single file.

```
@pure module Example3
interface N { val x : Int };
module n : N {
    let x = 3
}
```

`n` is a module that implements `interface N`. Next, we define a `functor m` that implements `interface M` and takes an implementation of `N` as input; this is similar to an import at the `fragment` level.

```
interface M {
    val f : Int -> Int;
    val m : Int
};
functor m (n: N) : M {
    open n;
    let f = \(y: Int) => y + x;
    let m = f(x)
```

```
}
```

In this example, the functor m represents a program fragment that imports the module n with the interface N. In ENVCAP, interfaces are treated as types, so $interface\ N$ and $interface\ M$ desugar into record types $\{N : \{x : Int\}\}$ and $\{M : \{f : Int \to Int\}\&\{m : Int\}\}$, respectively.

3.4.2 Compilation: ENVCAP $\rightsquigarrow \lambda_E$. In our setting, the elaboration from ENVCAP to λ_E is analogous to Cardelli's compilation of bindings to linksets. Modules and functors elaborate into records and boxed abstractions. In this example, specifically:

$$module\ n : \{n : \{x : Int\}\} \rightsquigarrow \{n = \{x = 3\} : \{x : Int\}\}$$

$$functor\ m : \{m : sig[\{n : \{x : Int\}\}, \{f : Int \to Int\}\&\{m : Int\}]\}$$

$$\rightsquigarrow \{m = \epsilon \rhd \lambda\{n : \{x : Int\}.(f = .., m = (?.f)(?.n.x)) : type..\}$$

For simplicity, we omit the elaboration of the open (given in Figure 2) statement, which essentially loads the contents of module n ($\{x = 3\}$) into the environment.

3.4.3 Linking. Once elaborated to λ_E, the expressions can be linked. To ensure correct linking order, we use Kahn's [6] algorithm for topological sorting to determine module dependencies and linking order. It enables the detection of cyclic dependencies that are not allowed in ENVCAP for both imports and requirements for the sake of simplicity. The linking process proceeds as follows.

(1) **Initial State:** $L \equiv \epsilon$.
(2) **Link n:** $L \hookrightarrow L' \equiv \epsilon \,;\, \{n = \{x = 3\} : \{x : \text{Int}\}\}$.
(3) **Link m:** $L' \hookrightarrow L' \,;\, \{m = \epsilon \,;\, \{n = \{x = 3\}\} \rhd (f = .. \,;\, m = (?.f)(?.n.x)) : \{f : \text{Int} \to \text{Int}\}\&\{m : \text{Int}\}\}$.

When no further linking steps are possible ($L \not\hookrightarrow$), the resulting structure, with dependent merges ($;$), contains fully linked fragments that can be executed individually.

4 RESULTS & CONTRIBUTIONS

The main results achieved are the following: 1) Implementation of the ENVCAP interpreter in `Haskell`[10]; 2) Formalization of type-directed elaboration $ENVCAP \rightsquigarrow \lambda_E$ in Rocq [16].

4.1 Implementation

The ENVCAP interpreter, implemented in ~5000 lines of Haskell, supports recursion, algebraic datatypes, and other key features. Its architecture comprises parsing, locally nameless transformation, desugaring, elaboration, and execution. The locally nameless representation [3] simplifies implementation by avoiding ambiguous environment lookups, which λ_E forbids. Furthermore, the interpreter produces `.epc` files containing λ_E expressions, enabling independent linking and execution, ensuring flexibility by decoupling from specific code generation.

4.2 Meta-theory

Type-directed elaboration translates *core* ENVCAP to λ_E expressions while preserving type consistency. Currently, the proof excludes the compilation of fragments of the form program $\langle S, I, R, E, A\rangle$; the key elaboration rules are shown in Figure 2.

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

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

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

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

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

where $|.|$ is simply translation of ENVCAP types to λ_E types.

Figure 2: Elaboration: $\Gamma \vdash ENVCAP : A \rightsquigarrow \lambda_E$

The theorems formalized in Rocq are the following:

THEOREM 1 (TYPE PRESERVATION).

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

THEOREM 2 (UNIQUENESS OF TYPE INFERENCE).

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

THEOREM 3 (UNIQUENESS OF ELABORATION).

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

Significance. ENVCAP presents *first-class* modules as an alternative to objects and enables separate compilation without external constructs, such as linksets. This simplifies the implementation of programming languages by providing simpler alternatives using *first-class* environments and environment-based semantics.

5 ONGOING AND FUTURE WORK

The ENVCAP project aims to formalize separate compilation in ROCQ, based on Cardelli's framework. Inspired by WYVERN, it enforces authority safety for capability guarantees. To enhance modularity, it introduces subtyping support, enabling export restrictions via interface files.

ACKNOWLEDGEMENTS

We thank Bruno C. d. S. Oliveira and Jinhao Tan for their foundational work on first-class environments and supervision.

REFERENCES

[1] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules as Objects in Newspeak. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 405–428. https://doi.org/10.1007/978-3-642-14107-2_20

[2] Luca Cardelli. 1997. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 266–277. https://doi.org/10.1145/263699.263735

[3] Arthur Charguéraud. 2012. The Locally Nameless Representation. *Journal of Automated Reasoning* 49, 3 (2012), 363–408. https://doi.org/10.1007/s10817-011-9225-2

[4] Pierre-Louis Curien. 1991. An Abstract Framework for Environment Machines. *Theor. Comput. Sci.* 82 (1991), 389–402. https://api.semanticscholar.org/CorpusID:41601809

[5] Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (March 1966), 143–155. https://doi.org/10.1145/365230.365252

[6] A. B. Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (Nov. 1962), 558–562. https://doi.org/10.1145/368996.369025

[7] Darya Kurilova, Alex Potanin, and Jonathan Aldrich. 2014. Wyvern: Impacting Software Security via Programming Language Design. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools* (Portland, Oregon, USA) *(PLATEAU '14)*. Association for Computing Machinery, New York, NY, USA, 57–58. https://doi.org/10.1145/2688204.2688216

[8] Peter J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6 (1964), 308–320. https://api.semanticscholar.org/CorpusID:5845983

[9] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2020. *The OCaml system release 4.10.* https://caml.inria.fr/pub/docs/manual-ocaml/

[10] Simon Marlow (Ed.). 2010. *Haskell 2010 – Language Report.* www.haskell.org/onlinereport/haskell2010/

[11] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-Based Module System for Authority Control. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 20:1–20:27. https://doi.org/10.4230/LIPIcs.ECOOP.2017.20

[12] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph.D. Dissertation. Johns Hopkins University, Baltimore, Maryland, USA.

[13] David Swasey, Tom Murphy, Karl Crary, and Robert Harper. 2006. A separate compilation extension to standard ML. In *Proceedings of the 2006 Workshop on ML* (Portland, Oregon, USA) *(ML '06)*. Association for Computing Machinery, New York, NY, USA, 32–42. https://doi.org/10.1145/1159876.1159883

[14] Jinhao Tan and Bruno C. d. S. Oliveira. 2023. Dependent Merges and First-Class Environments. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 34:1–34:32. https://doi.org/10.4230/LIPIcs.ECOOP.2023.34

[15] Jinhao Tan and Bruno C. d. S. Oliveira. 2024. A Case for First-Class Environments. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 360 (Oct. 2024), 30 pages. https://doi.org/10.1145/3689800

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