Project Plan

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

Contents

1 Introduction **2**

2 Background **2**
2.1 First-Class Environments 2
2.2 Capabilities 2
2.3 Case Study: *Wyvern* [3] 2

3 Objectives **3**

4 Methodology **3**
4.1 Literature Review 3
4.2 Implementation 4
4.3 Testing 4

5 Project Schedule **5**

References **6**

1 Introduction

Most common programming languages do not restrict access to system resources across different sections of the codebase. An increase in codebase with no authority division between its components can lead to security vulnerabilities as insecure code can exploit system resources even if it does not require access to specific system resources to execute its tasks. For instance, a section of code adhering to weaker security standards can compromise the entire system by exploiting resources such as the file system or network. *Capabilities* provide a mechanism to restrict access to system resources, thereby adhering to the *Principle of Least Authority* [4]. While capabilities are traditionally modeled as objects, recent advancements in λ_E calculus propose using first-class environments as an alternative representation for capabilities [10]. This approach aims to bridge the gap between the implementation and theoretical aspects of programming languages, enhancing reasoning about code. This research project will design and implement a programming language that incorporates capabilities as first-class modules and supports separate compilation, utilizing λ_E as the core calculus. In this detailed plan, we will go through the background, objectives, methodology, and timeline.

2 Background

2.1 First-Class Environments

Environments are used for the implementation of the programming languages to maintain scope and bindings etc. These can be implemented as a dictionary or other relevant data structures. *First-Class* refers to values/constructs in a programming language that can be created and manipulated during the runtime. *First-Class Environments* [2] can be modified and mainipulated during the run-time. First-class environments are adopted by a few dynamically typed languages such as R [1]. Due to dynamic typing, these languages are prone to runtime type errors, and therefore, it creates a need for statically typed semantics with first-class environments. A call-by-value statically typed calculus called λ_E [10] allows for reasoning about first-class environments and provides static typing.

2.2 Capabilities

Capabilities, as the name suggests, allow programs the capability to access other programs. Suppose we have a module that allows access to a specific system resource, then we can allow only specific code to access this module by providing it a *Capability*. In addition, *Capabilities* can help write secure concurrent code as it can restrict access to specific system resources and therefore, avoid data races or deadlocks. *Types* can be specified for capabilities and hence, allowing to avoid non-restricted access before run-time. Typically, capabilities are modeled as Objects which can be substituted by *First-Class Environments* [10].

2.3 Case Study: *Wyvern* [3]

Wyvern has first-class modules and treats modules as capabilities. If module A wants a reference to module B, it has to have the capability for module A. There are *resource*

and *pure* modules where all the modules that encapsulate system resources, use other resource modules, or contain mutable state are considered *resource* modules, and the rest are *pure* modules. *Resource* modules are security-critical modules and are written with higher levels of security. *Resource* modules can only be accessed by *resource* modules, but any module can access a *pure* module. *Wyvern* utilizes a *Threat* model which requires secure interfaces to access *resource* modules and therefore, creates a distinction between trusted and untrusted codebase. [3]

3 Objectives

This project aims to demonstrate the use of first-class environments as an alternative to objects for modeling capabilities by design and implementation of a programming language with capabilities as first class modules. The design of the programming language will be followed by an interpreter or compiler implementation to realize the design. Design and implementation are divided into Surface and Core, where Surface will be the programming language used by the user and it will be translated into the core calculus to keep it close to the theoretical foundation. This translation from Surface to Core will be specified as an elaboration. Further, support for separate compilation will be added to the language. Here are the main tasks involved in this project:

- Implementation of the core calculus λ_E.
- Extension of the Core λ_E with If-Else, Recursive functions, etc.
- Utilize modular type checking to add support for separate compilation.
- Design the surface programming language with capabilities.
- Specify elaboration from the Surface to the Core.
- Implement the surface language and its translation to the core.
- Extensive documentation for the implementation.
- Unit and Property-Based testing of the implementation.

4 Methodology

4.1 Literature Review

As part of the methodology, a comprehensive literature review will be conducted to establish a solid foundation for the project. This review will encompass an in-depth examination of existing research related to programming languages that incorporate capabilities, e.g. Pony [8] and Wyvern [3], and calculi with environment-based semantics, e.g. E_i [9] and λ_E [10]. In addition, theoretical foundations of programming languages will be studied from *Software Foundations* Vol I [7] & II [6] and *Types and Programming Languages* [5] to support research literature comprehension.

4.2 Implementation

The programming language used for implementation is **Haskell**. Core Calculus λ_E will be implemented first which will also require the use of bidirectional typing to implement the type checker for the core. To support the separate compilation for the surface language later, modular type checking will also be part of the core. Then, surface language will be designed followed by the specification of elaboration from the surface to the core. Next, we will implement the parser for the surface and its elaboration to the core. The surface will be parsed into an Abstract Syntax Tree which will be further translated into an equivalent representation in the core calculus before execution as shown in Figure 1.

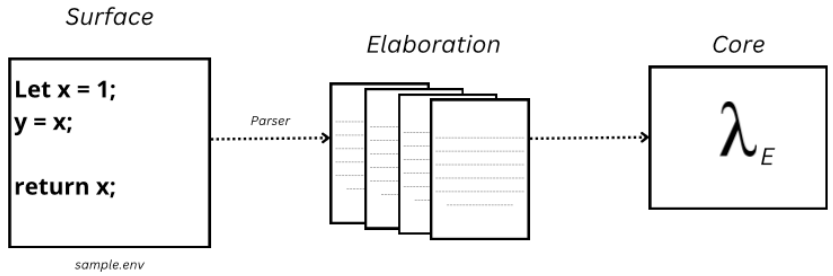

Figure 1

4.3 Testing

Unit and Property-based testing will be performed to ensure the correctness of the implementation. For Unit testing, tests will be written to aim for full coverage. However, we want our implementation to have strong guarantees, and therefore, utilization of property-based testing with *QuickCheck* framework in Haskell will allow us to uncover edge cases as it generates randomized test cases to test for properties of the code. This will allow the gap between the implementation and theoretical foundation to be minimal and therefore, utilizing formal verification with **Coq**, if needed, will be more pleasant. Furthermore, GitHub will be used for version control. Testing will be performed in parallel to the implementation.

5 Project Schedule

Table 1: Milestones and Deadlines

Milestone	**Timeline**
Literature Review	August to September
Project Proposal and Web Page Setup	September
Core Calculus Implementation	October to November
Design of Surface Programming Language	November to December
Interim Report and Presentation	January
Implementation of Surface	January to March
Further Improvements	March to April
Final Report and Presentation	April

References

[1] Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. R melts brains: an ir for first-class environments and lazy effectful arguments. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2019, page 55–66, New York, NY, USA, 2019. Association for Computing Machinery.

[2] 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, page 98–110, New York, NY, USA, 1987. Association for Computing Machinery.

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

[4] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. A capability-based module system for authority control. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 20:1–20:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[5] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[6] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. Electronic textbook, 2024. Version 6.7, http://softwarefoundations.cis.upenn.edu.

[7] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*, volume 1 of *Software Foundations*. Electronic textbook, 2024. Version 6.7, http://softwarefoundations.cis.upenn.edu.

[8] George Steed and Sophia Drossopoulou. A principled design of capabilities in pony. *URL: https://www. ponylang. io/media/papers/a_ priniciple d_ design_ of_ capabilities_ in_ pony. pdf*, 2016.

[9] Jinhao Tan and Bruno C. d. S. Oliveira. Dependent Merges and First-Class Environments. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:32, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[10] Jinhao Tan and Bruno C. d. S. Oliveira. A case for first-class environments (artifact), August 2024.