Extending Abstract Effects with Bounds and Algebraic Handlers

Anlun Xu

November 2020

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Submitted in partial fulfillment of the requirements
for the degree of Master of Science.

Keywords: Effect Systems

For my dog

Abstract

Effect systems have been a subject of active research for nearly four decades, with the most notable practical example being checked exceptions in programming languages such as Java. While many exception systems support abstraction, aggregation, and hierarchy (e.g., via class declaration and subclassing mechanisms), it is rare to see such expressive power in more generic effect systems. We designed an effect system around the idea of protecting system resources and incorporated our effect system into the Wyvern programming language. Similar to type members, a Wyvern object can have effect members that can abstract lower-level effects, allow for aggregation, and have both lower and upper bounds, providing for a granular effect hierarchy. We argue that Wyvern's effects capture the right balance of expressiveness and power from the programming language design perspective. We present a full formalization of our effect-system design, show that it allows reasoning about authority and attenuation. Our approach is evaluated through a security-related case study.

Acknowledgments

My advisor is cool.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An effect system can be used to reason about the side effects of code, such as reads and writes to memory, exceptions, and I/O operations. Java's checked exceptions is a simple effect system that has found widespread use, and interest is growing in effect systems for reasoning about security [**?**], memory effects [**?**], and concurrency [**? ? ?**].

Requirements for a scalable effect system Unfortunately, effect systems have not been widely adopted, other than checked exceptions in Java, a feature that is widely viewed as problematic [**?**]. The root of the problem is that existing effect systems do not provide adequate support for scaling to programs that are larger and have complex structure. Any adequate solution must support *effect abstraction*, *effect composition*, and *path-dependent effects*. Furthermore, effects should be an inherent part of the type system, instead of being encoding of other type abstractions such as monad.

Abstraction is key to achieving scale in general, and a principal form of abstraction is abstract types [**?**], a modern form of which appears as abstract type members in Scala [**?**]. Analogously to type abstraction, we define *effect abstraction* as the ability to define higher-level effects in terms of lower-level effects, and potentially to *hide* that definition from clients of an abstraction. In order to integrate with modularity mechanisms, and by analogy to type members, we define effects using *effect members* of modules or objects. For example, a file .Read effect could

abstract a lower-level system.FFI effect. Then clients of a file should be able to reason about side effects in terms of file reads and writes, not in terms of the low-level calls that are made to the foreign function interface (FFI). In large-scale systems, abstraction should be *composable*. For example, a database component might abstract file .Read further, exposing it as a higher-level db.Query effect to clients. Clients of the database should be oblivious to whether db.Query is implemented in terms of a file .Read effect or a network.Access effect (in the case that the backend is a remote database).

Effect polymorphism is a form of parametric polymorphism that allows functions or types to be implemented generically for handling computations with different effects [**?**]. In systems at a larger scale, there are various possible effects, and each program component may cause different effects. With effect polymorphism, we can write general code that handles objects with different effects, thereby reducing the amount of replicated code. In practice, we have found that to make effects work well with modules, it is essential to extend effect polymorphism by assigning bounds to effect parameters. We therefore introduce *bounded abstract effects*, which allows programmers to define upper and lower bounds both on abstract effects and on polymorphic effect parameters.

We also leverage *path-dependent effects*, i.e., effects whose definitions depend on an object. This adds expressiveness; for example, if we have two File objects, x and y, we can distinguish effects on one file from effects on the other: the effects x.Read and y.Read are distinct. Path-dependent effects are particularly important in the context of modules, where two different modules may implement the same abstract effect in different ways. For example, it may be important to distinguish db1.Query from db2.Query if db1 is an interface to a database stored in the local file system whereas db2 is a database accessed over the network.

Effects should be an inherent part of the type system, instead of being encoded as a type abstraction. The reason is twofold: although effect checking may be implemented in terms of monad, the safety of the effect system is compromised. The effectful code can bypass the effect-checking process if the programmer use the effectful code in a context which is not encapsulated

by a monad. The other reason is that monad could be unintuitive to use for programmers outside of the functional programming community, while a type system enhanced by effect types doesn't require prior knowledge of monad and is straightforward to use.

Design of the effect system in Wyvern This paper presents a novel and scalable effect-system design that supports effect abstraction and composition. The abstraction facility of our effect-system is inspired by type members in languages such as Scala. Just as Scala objects may define type members, in our effect calculus, any object may define one or more *effect members*. An effect member defines a new effect in terms of the lower-level effects that are used to implement it. The set of lower-level effects may be empty in the base case or may include low-level effects that are hard-coded in the system. Type ascription can enable information hiding by concealing the definition of an effect member from the containing object's clients. In addition to completely concealing the definition of an effect, our calculus provides bounded abstraction, which exposes upper or lower bounds of the definition of an effect, while still hiding the definition of it.

Just as Scala's type members can be used to encode parametric polymorphism over types, our effect members double as a way to provide effect polymorphism. Bounded effect polymorphism is also provided in our system, because abstract effect members can be bounded by upper or lower bounds. We follow numerous prior Scala formalisms in including polymorphism via this encoding rather than explicitly; this keeps the formal system simpler without giving up expressive power.

Finally, because effect members are defined on objects, our effects are *generative*, even dynamically. This yields great expressivity: each object created at runtime defines a new effect for each effect member in that object so that, for example, we can separately track effects on different File objects, statically distinguishing the effects on one object from the effects on another.

Evaluation and Security Applications. A promising area of application for effects is software security. For example, in the setting of mobile code, [**?**] proposed that effects could be used to ensure that any untrusted code we download can only access the system resources it needs to do its tasks, thus following the principle of least privilege [**?**]. We are not aware of prior work that

explores this idea in depth.

In order to evaluate our design for effect abstraction, we have incorporated it into an effect system that tracks the use of system resources such as the file system, network, and keyboard. Our effect system is intended to help developers reason about which source code modules use these resources. Through the use of abstraction, we can "lift" low-level resources such as the file system into higher-level resources such as a logging facility or a database and enable application code to reason in terms of effects on those higher-level resources when appropriate. In fact, even the use of resources such as the file system is scaffolded as an abstraction on top of a primitive system.FFI effect that our system attaches to uses of the language's foreign function interface. A set of illustrative examples demonstrates the benefits of abstraction for effect aggregation, as well as for information hiding and software evolution. Finally, we show how our effect system allows us to reason about the *authority* [**?**] of code, i.e., what effects a component can have, as well as the *attenuation* of that authority.

Our effect system is implemented in the context of Wyvern, a programming language designed for highly productive development of secure software systems. In this paper, we give several concrete examples of how our effect-system design can be used in software production, all of which are functional Wyvern code that runs in the Wyvern regression test suite.

Outline and Contributions. The next section introduces a running example, after which we describe the main contributions of our paper:

- The design of a novel effect system fulfilling the requirements above. Our system is the first to bring together effect abstraction and composition with the effect member construct. Ours is also the first system to provide the programmer with a general form of bounded effect polymorphism and bounded effect abstraction, supporting upper and lower bounds that are other arbitrary effects. (Section **??**);
- The application of our effect system to a number of forms of security reasoning, illustrating its expressiveness and making the benefits described above concrete (Section **??**);

- A precise, formal description of our effect system, and proof of its soundness. Our formal system shows how to generalize and enrich earlier work on path-dependent effects by leveraging the type theory of DOT (Section **??**);
- A formalization of authority using effects, and of authority attenuation (Section **??**);
- A feasibility demonstration, via the implementation of our approach in the Wyvern programming language (Section **??**).

The last sections in the paper discuss related work and conclude.

Chapter 2

Conclusion

Bibliography

[1] Edwin Brady. Programming and Reasoning with Algebraic Effects and Dependent Types. In *International Conference on Functional Programming*, 2013. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500581. URL http://doi.acm.org/10.1145/2500365.2500581.