SPLASH: U: Approximating Polymorphic Effects with Capabilities

Justin Lubin
University of Chicago
justinlubin@uchicago.edu

Abstract

Unforgeable tokens known as capabilities provide a flexible mechanism to control access to sensitive system resources, but reasoning about what parts of a codebase can and do use these capabilities is often imprecise. Effect systems are one solution that can improve the precision of capability-based reasoning, but their verbosity has proven to be a usability concern. To remove the burden of full effect annotation, we propose *quantification lifting*, an automatic semantics-preserving type transformation to handle the mixing of effect-annotated and effect-unannotated code in a capability-safe language with mutable state and effect polymorphism. Because quantification lifting operates on types rather than expressions, it may be used even when the effect-unannotated code is unavailable ahead of time, such as in the cases of dynamic loading and previously compiled code. To evaluate the practicality of quantification lifting, we implemented this transformation for use in the Wyvern programming language, and, in doing so, demonstrate how it allows for secure and ergonomic mixing of effect-unannotated code with effect-annotated code in a realistic capability-safe programming language.

1 Introduction

Programs must often deal with sensitive system resources such as file systems, networks, and databases. An unavoidable concern when working with programs that do interact with these resources is managing exactly how these resources are accessed. In particular, given a large program, limiting the access of these resources to a small subset of the codebase allows for easier security verification by experts and less code surface area for malicious agents to launch an attack upon.

One solution to limiting the access to these resources is the notion of a *capability*, that is, an unforgeable token (such as an object) that gives particular parts of the code access to sensitive resources. Capability-safe languages guarantee that only code explicitly given access to sensitive resources is able to do so [7]. However, capabilities alone do not provide a granular method of formally reasoning about resource access in a codebase; security analysts still need to inspect module signatures manually to ensure that no capabilities are leaked unintentionally.

A popular approach to automated, granular reasoning about incurred actions in a program is the inclusion of an *effect system*, an extension to the type system that tracks precisely what "effects" each function or method can incur. Effect systems have been used to formalize capability-based reasoning [10, 6, 2], but an important usability concern with many effect systems (such as Java's checked exceptions) is the requirement that all effectful code be fully annotated, including third-party plugins, high-level libraries, and other less safety-critical components [3]. Recent work by Craig et al. begins to address this problem; they introduce a special "import" construct for a capability-safe lambda calculus that allows safe mixing of effect-annotated and effect-unannotated code [1].

However, their lambda calculus does not include two important features: mutable state and effect polymorphism. Taken individually, these two commonplace features do not present a problem to Craig et al.'s theory, but together result in an unfortunate all-or-nothing loss of either effect safety or effect precision, as demonstrated in Section 2. In this paper, we present *quantification lifting*, a type-level transformation to handle these two features together, and prove that it is maximally precise subject to safety constraints. As an evaluation of the practicality of quantification lifting, we also present an implementation of this transformation for Wyvern, an object-oriented, capability-safe language [5, 8].

2 The Problem

Consider the scenario of building an extensible text editor that loads user-written plugins at runtime. As programmers, we want the core of the editor that deals with the filesystem to be fully effect-annotated so that we can be sure not to leak any sensitive resources. However, we do not want to place the burden of effect annotation on the plugin authors: they should be able to write code that works without having to worry about low-level details such as precisely which method incurs which effect. Furthermore, we do not want to restrict plugins so that they cannot incur *any* effect, as there are legitimate reasons why a plugin may want to do so—logging to the console is one such example.

Given these goals, consider the following snippet of Wyvern code, which demonstrates the fundamental problem with mutability and effect polymorphism:

```
resource type Logger
  effect log
  def append(s : String) : {log} Unit

module def reversePlugin(name : String)
  var logger : Logger = pureLogger()
  def setLogger(newLogger : Logger) : Unit
    logger = newLogger
  def run(s : String) : String
    val t = s.reverse()
    val message = s + " -> " + t
    logger.append(name + ": " + message)
    t
```

In this example, we have an effect-unannotated ML-style module functor (function returning a module) `reversePlugin` that uses an annotated type `Logger` that declares an abstract (polymorphic) effect `log` and a method `append` that incurs this effect.[1] The problem here is that any code that uses `reversePlugin` will not know what concrete effect the mutable variable `logger` will incur because it could be set to any implementation of the type `Logger`. Indeed, because the `run` method incurs the polymorphic effect `logger.log`, no effect-annotated code can safely call the `run` method unless its effect annotation included every possible effect that could be assigned to `logger.log`. At best, this effect bound would be the union of every single assignment to `logger` in the program's entire AST—an untenable solution. This example demonstrates that when effect polymorphism is mixed with mutable state, it is difficult to pin down an effect bound that is both valid and useful, that is to say, a bound that is both safe and precise.

3 Solution

Our solution is to introduce an automatic transformation to lift effect polymorphism from inside the body of the ML-style module functor to the module functor itself, collapsing each of the universal effect quantifications in the functor body into a single universally quantified effect E (which must be bounded as described at the end of this section to ensure effect safety); this effect then serves as the effect bound for the methods in the module. The programmer will write the same unannotated code as in Section 2, but the compiler will automatically transform its type signatures as follows:

```
module def reversePlugin[effect E](name : String)
  var logger : Logger[E] = pureLogger()
  def setLogger(newLogger : Logger[E]) : {E} Unit
    logger = newLogger
  def run(s : String) : {E} String
    val t = s.reverse()
    val message = s + " -> " + t
    logger.append(name + ": " + message)
    t
```

This transformation operates at the type level, which ensures that it can be performed even if the unannotated code is unavailable ahead of time, such as in the cases of dynamic loading and previously compiled code. Now, whenever this plugin is used in effect-annotated code, the effect E is fixed for all calls to any of its methods, so the annotated code knows exactly what effect bound it must have on

[1] An effect in Wyvern is said to be *abstract* if its definition is left up to the modules that implement the type. For example, we may have modules `fileLogger` and `databaseLogger` that each are of type `Logger` but with effects relating to the filesystem and the database, respectively. In short: abstract effect members allow for polymorphic effects.

any methods that use this plugin: E. Annotated code can now use the unannotated code as follows:

```
1  val logger1 = fileLogger(...)
2  val logger2 = databaseLogger(...)
3  val plugin = reversePlugin[{logger1.log}]("archive")
4  plugin.setLogger(logger1)
5  // plugin.setLogger(logger2) ← not allowed!
```

The effect parameters to `reversePlugin` on line 3 act as a permission system for the interface between the annotated and unannotated code. Architecturally, this ensures that the annotated code is both fully aware of the effects that the unannotated code can incur and does not accidentally give the unannotated code the ability to incur any other effects.

As a concrete example of the result of the type transformation, the type of the variable `plugin` in the above example is equivalent to the following type, `MyPlugin`:

```
resource type MyPlugin
  def setLogger(newLogger : Logger') : {logger1.log} Unit
  def run(s : String) : {logger1.log} String

resource type Logger'
  effect log = {logger1.log}
  def append(contents : String) : {log} Unit
```

These transformed types are precisely why line 5 of the previous code snippet is not allowed.

One caveat to the transformation remains: we must ensure that the universal effect quantification is bounded properly. Crucially, we let the upper bound on the quantification be the intersection of the effects that make each import valid according to the semantics of the import statement given by Craig et al. in [1]; it is in precisely this way that we build on and extend their work. Their work proves that this is the maximal set of effects that we can safely allow an unannotated module to incur; any additional effects could potentially violate the type system. The lower bound on the quantification is much simpler to compute: it is the union of the set of concrete effects present in the imported capabilities. In other words, any effect that is passed into the module functor must include any concrete effect that is referenced by the functor itself. Because of capability safety, all capabilities that the unannotated code uses must be explicitly passed to it, so these bounds are effect-safe, as proven in Section 4.

4 Theory

We now present the quantification lifting transformation more formally, presupposing the existence of standard static and dynamic language semantics for an object-oriented, capability-safe language (as in [5]). We further assume that effects are erased at runtime, or, at least, are uninspectable from effect-unannotated code at runtime. This assumption ensures that quantification lifting will always preserve runtime semantics, because it modifies only effect declarations for effect-unannotated code.

First, we present some definitions. Let $\mathcal{T}$ be the universe of types and $\mathcal{E}$ be the universe of effects. Given a type $\tau \in \mathcal{T}$, let $L_\tau \subseteq \mathcal{E}$ be the union of the set of concrete effects referenced by τ and $U_\tau \subseteq \mathcal{E}$ be the intersection of the set of effects that make each import used by τ valid according to Craig et al.'s import semantics (so that L_τ and U_τ are the lower and upper bounds, respectively, described in Section 3). Finally, for $\tau \in \mathcal{T}$ and $\varepsilon \subseteq \mathcal{E}$, let $(\tau)_\varepsilon$ be τ with its declarations transformed as described in Section 3; that is, with every method annotated as having the effect set ε, and every polymorphic effect instantiated to ε. With this notation, quantification lifting is precisely the type transformation sending a module functor type $\tau_1 \longrightarrow \tau_2$ to the effect-polymorphic module functor type $\forall \varepsilon \in \mathcal{E}\,(L_{\tau_1 \to \tau_2} \subseteq \varepsilon \subseteq U_{\tau_1 \to \tau_2})\,.\,\tau_1 \longrightarrow (\tau_2)_\varepsilon$.

We are now ready to present the main theorems of this paper. For the following two theorems, suppose F is an effect-unannotated module functor and $F : \tau_1 \longrightarrow \tau_2$ is a valid typing judgement.

Theorem 1 (Safety). *The typing judgement* $F : \forall \varepsilon \in \mathcal{E}\,(L_{\tau_1 \to \tau_2} \subseteq \varepsilon \subseteq U_{\tau_1 \to \tau_2})\,.\,\tau_1 \longrightarrow (\tau_2)_\varepsilon$ *is valid.*

Proof (sketch). By capability safety, the module returned by F cannot access any capability other than what is given to it. After the transformation, all references to a particular capability will be replaced with references to that specific capability but with the updated effect ε. As nothing else about the capability changed (and effects are erased at runtime), the method must still be capability-safe because ε is bounded above by $U_{\tau_1 \to \tau_2}$. No other types are changed, so the typing judgement must still be valid. □

Theorem 2 (Precision)**.** *Suppose f_1 and f_2 are method signatures in τ_2. Let E_1 and E_2 be any two valid effect sets for f_1 and f_2 respectively. Then $E_1 = E_2$, and, moreover, $L_{\tau_1 \to \tau_2} \subseteq E_1, E_2$.*

Proof (sketch). As τ_2 has no effect annotations, f_1 may call f_2, incurring the effect set E_2, so E_2 must be a subset of E_1. Similarly, f_2 may call f_1, so E_1 must be a subset of E_2. Thus, $E_1 = E_2$. Moreover, $L_{\tau_1 \to \tau_2}$ is precisely the set of concrete effects incurred by the methods of τ_2 and nothing else, so E_1 and E_2 must both contain $L_{\tau_1 \to \tau_2}$. □

From the first theorem, we see that quantification lifting preserves type safety, and, from the second theorem, we see that no valid assignment of effects to the methods of τ_2 can do any better than quantification lifting. In other words, quantification lifting is maximally precise subject to safety.

5 Implementation and Evaluation

The quantification lifting transformation has been fully implemented in the Wyvern compiler (all of which is freely available at `https://github.com/wyvernlang/wyvern`) except for the inference of upper and lower bounds for the effect quantification. Work is currently being done by the Wyvern team to implement the import bound inference algorithm, which amounts to an implementation of the work of Craig et al. in [1] to compute the upper bound for the effect quantification and a simple union to compute the lower bound for the effect quantification. Nevertheless, the transformation works as-is without the bound checks, and there is example code available on GitHub that successfully and ergonomically uses the transformation as described, serving as an evaluation of its practicality.

6 Future Work

We would like to run a user study of our implementation that includes a questionnaire to observe in what ways, if any, the mental model of users (programmers) previously unfamiliar with the quantification lifting transformation diverges from the actual transformation that the compiler applies. More broadly, a user study would also help answer whether or not an accurate mental model of the transformation is even needed at all to use the facilities that it provides ergonomically. Ideally, the code required to use quantification lifting from a user's perspective will be intuitive enough to write without much understanding of the theory at all, as in Section 3.

7 Limitations

The biggest limitation of our approach is that, compared to full effect annotation, quantification lifting decreases effect precision in the effect-unannotated code by combining all the universal quantifications into a single one; in some cases, fully precise effect sets may actually be necessary. For example, if a programmer wants one method to incur just the effect set E_1 and a second method to incur just the effect set E_2, then quantification lifting would actually annotate the entire module (and every method in it) with the less precise effect set $E_1 \cup E_2$. We must have this limitation because there are no effect signatures in the effect-unannotated code to forbid any method from calling any other method and incurring its effects (see Theorem 2). As such, quantification lifting requires that code relying on this level of precision be effect-annotated. In the example given in Section 2, such precision was not critical for `reversePlugin` to function properly. On the other hand, providing a more precise effect set for the library type `Logger` was desirable, so, accordingly, it was fully effect-annotated. This decision can be made on a module-by-module basis, but not more granularly than that in our current formulation.

Another limitation is the requirement that effects be erased at runtime, or, at least, that effects be uninspectable from effect-unannotated code. If runtime effect inspection is absolutely required or desired, then simply forbidding effect-unannotated code from inspecting effects is a trade-off we view as fair: if the programmer cares enough to manually inspect the effects from the code, then it seems reasonable to require that the code be effect-annotated.

8 Related Work

The Koka language has full effect inference but does not support user-defined object-based effects (e.g., effect members defined on objects) [4]. Furthermore, our approach differs from effect inference in that we do not require the implementation of the unannotated code, but rather only its type. This ensures that we can support cases such as dynamic loading and compiled third-party code for which the source is impossible to obtain.

More generally, the interaction between mutability and inference has been a long-standing challenge. Difficulties of this nature have been noted before in the effect inference for Koka [4] and type inference for SML [9].

9 Conclusion

Effect systems can serve as a formalized reasoning system for capabilities, but often suffer from being overly verbose in that every effectful method must be annotated. Allowing effect-unannotated code to mix safely with effect-annotated code enables programmers to reap the benefits of a rich effect system in a capability-safe language without getting bogged down in the overhead of effect annotations. Our approach extends previous work on this topic in the context of capability-safe languages to include mutability and effect polymorphism—two important aspects of object-oriented design in a language with an effect system—by applying *quantification lifting* to the effect-unannotated code, a type transformation that can be performed even when the code is unavailable ahead of time.

Acknowledgements

This project was supervised by Jonathan Aldrich and Darya Melicher at Carnegie Mellon University (USA) and Alex Potanin at the Victoria University of Wellington (NZ). It was funded by a National Science Foundation Research Experiences for Undergraduates grant.

References

[1] Aaron Craig, Alex Potanin, Lindsay Groves, and Jonathan Aldrich. Capabilities: Effects for Free. In *Proceedings of the 20th International Conference on Formal Engineering Methods*, ICFEM'18, 2018.

[2] Dominique Devriese, Lars Birkedal, and Frank Piessens. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 147–162, March 2016.

[3] Joseph R. Kiniry. Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application. In Christophe Dony, Jørgen Lindskov Knudsen, Alexander Romanovsky, and Anand Tripathi, editors, *Advanced Topics in Exception Handling Techniques*, pages 288–300. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[4] Daan Leijen. Koka: Programming with Row Polymorphic Effect Types. In Paul Levy and Neel Krishnaswami, editors, Proceedings 5th Workshop on *Mathematically Structured Functional Programming,* Grenoble, France, 12 April 2014, volume 153 of *Electronic Proceedings in Theoretical Computer Science*, pages 100–126. Open Publishing Association, 2014.

[5] 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 fuer Informatik.

[6] Darya Melicher, Yangqingwei Shi, Valerie Zhao, Alex Potanin, and Jonathan Aldrich. Using Object Capabilities and Effects to Build an Authority-safe Module System: Poster. In *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security*, HoTSoS'18, pages 29:1–29:1, New York, NY, USA, 2018. ACM.

[7] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006.

[8] Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: A Simple, Typed, and Pure Object-oriented Language. In *Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI '13, pages 9–16, New York, NY, USA, 2013. ACM.

[9] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.

[10] Valerie Zhao. Abstracting Resource Effects. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2017, pages 48–50, New York, NY, USA, 2017. ACM.