Approximating Polymorphic Effects with Capabilities

Justin Lubin[1], Darya Melicher[2], Alex Potanin[3], Jonathan Aldrich[2]

[1]University of Chicago, USA [2]Carnegie Mellon University, USA [3]Victoria University of Wellington, NZ

Motivation

Effect systems can formalize capability-based reasoning, but their verbosity has proven to be a usability concern. To remove the burden of full effect annotation, we propose a method to handle mixing effect-annotated code with effect-unannotated code in a capability-safe language with mutable state and effect polymorphism.

Background

Capability-safe languages guarantee that only code explicitly given access to sensitive resources is able to do so [5], but capabilities alone do not provide a method of formally reasoning about resource access in a codebase.

Effect systems can formalize capability-based reasoning [2, 4, 6], but an important usability concern is the requirement that all effectful code be fully annotated, including third-party plugins, high-level libraries, and other less safety-critical components [3].

Craig et al. introduced semantics for a special **"import" construct** for a capability-safe lambda calculus that allows safe mixing of annotated code with unannotated code [1], but it does not handle mutable state nor effect polymorphism.

The Problem

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

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

How will annotated code use **`reversePlugin`**? Because of effect polymorphism and mutability, the concrete effect in **`logger`** could be anything! At best, the effect bound in the annotated code would be the union of effects in every single assignment to **`logger`** in the entire program.

Import bound inferencing

→

Quantification lifting

Proposed Solution

```
resource type Logger[effect E]
  def append(contents : String) : {E} Unit


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

Our solution is to lift effect polymorphism from inside the ML-style module functor to the module functor itself, collapsing each of the universal effect quantifications into a single quantified effect **`E`**, which then serves as the effect bound for all the methods in the module.

Usage

```
import fileLogger, databaseLogger, reversePlugin
val logger1 = fileLogger(...)
val logger2 = databaseLogger(...)
val plugin = reversePlugin[logger1.log]("archive")
def main() : {logger1.log} Unit
  plugin.setLogger(logger1)
  // plugin.setLogger(logger2) <-- not allowed!
```

The effect parameters act as a permission system for the interface between the annotated and unannotated code.

Transformed Type

```
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
```

The polymorphic code has become monomorphized, so the annotated code knows exactly what the effect bound is.

Implementation

Import bound inferencer

Uses capability safety and Craig et al.'s import semantics to compute a lower and upper bound on the set of valid effects that can be passed into the unannotated code to ensure that it remains effect-safe.

Quantification lifter

Takes an unannotated module functor of type $\tau_1 \to \tau_2$ and transforms it into a functor of type $\forall \varepsilon\ (\mathrm{L} \subseteq \varepsilon \subseteq \mathrm{U})\ .\ \tau_1 \to (\tau_2)_\varepsilon$, where L and U are the bounds from the import bound inferencer and $(\tau_2)_\varepsilon$ is τ_2 with its declarations modified with ε.

References

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

[2] Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. 147–162. https: //doi.org/10.1109/EuroSP.2016.22

[3] Joseph R. Kiniry. 2006. *Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application.* Springer Berlin Heidelberg, Berlin, Heidelberg, 288–300. https://doi.org/10.1007/11818502_16

[4] Darya Melicher, Yangqingwei Shi, Valerie Zhao, Alex Potanin, and Jonathan Aldrich. 2018. 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)*. ACM, New York, NY, USA, Article 29, 1 page. https://doi.org/10.1145/3190619.3191691

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

[6] Valerie Zhao. 2017. 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).* ACM, New York, NY, USA, 48–50. https://doi.org/10.1145/3135932.3135946