Approximating Polymorphic Effects with Capabilities

Justin Lubin¹, Darya Melicher², Alex Potanin³, Jonathan Aldrich²
¹University of Chicago, USA  ²Carnegie Mellon University, USA  ³Victoria 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

How will annotated code use reversePlugin? Because of effect polymorphism and mutability, the concrete effect in the entire program.

Usage

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

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

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 $r_1 \to \tau$, and transforms it into a functor of a type $\forall \epsilon (L \subseteq \epsilon U) \cdot r_1 \to (\epsilon \tau)$, where $L$ and $U$ are the bounds from the import bound inferencer and $\epsilon \tau$ is $\epsilon$ with its declarations modified with $\epsilon$.

Proposed Solution

```
resource type Logger[Effect E]
def append(contents : String) : (E \ Unit)
def setLogger(newLogger : Logger[E]) : (E \ Unit)
def run(s : String) : (E \ String)
var 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.

Transformed Type

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

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

References