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

Goal

Allow ***secure*** and ***ergonomic*** mixing of effect-unannotated code with effect-annotated code in a ***realistic*** capability-safe programming language.

Background

1. Object Capabilities
2. Effect Systems
3. Capability-Safe Import Semantics

1. Object Capabilities

Capabilities
Unforgeable objects that give particular parts of the code access to sensitive resources

Capability-safe language
A language in which the only way to access sensitive resources is via capabilities

```
module def logger(myFile : File)
  ...

module def main(platform : Platform)
  val myFile = file(platform)
  val myLogger = logger(myFile)
  ...
```

2. Effect Systems

Effect system
Annotations on methods describing effects they can incur

Capability-based effect system
Way of formally reasoning about capabilities (*awesome!*)

Downside: verbosity

3. Capability-Safe Import Semantics

Prior work *(Craig et al.)*
Import semantics for capability-safe lambda calculus

Limitation
Does not handle mutable state nor effect polymorphism

Our goal
Scale up to a more realistic programming language

The Problem

Effect polymorphism ***and*** mutability

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
```

Question: *How will annotated code use **`reversePlugin`**?*

Effect polymorphism + mutability ⇒ **`log`** effect could be *anything*!

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
```

Question: *How will annotated code use **`reversePlugin`**?*

Effect polymorphism + mutability ⇒ **`log`** effect could be *anything*!

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
```

Question: *How will annotated code use* ***`reversePlugin`****?*

Effect polymorphism + mutability ⇒ **`log`** effect could be *anything*!

Solution

Quantification lifting

Quantification Lifting: Idea

```
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
```

```
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
```

- ***Lift*** effect polymorphism from inside ML-style module functor to the functor itself
- Collapse each universal effect quantification into single quantified effect `E`
  - Serves as effect bound for all methods in module

Quantification Lifting: Idea

```
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
```

```
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
```

- ***Lift*** effect polymorphism from inside ML-style module functor to the functor itself
- Collapse each universal effect quantification into single quantified effect `E`
  - Serves as effect bound for all methods in module

Quantification Lifting: 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!
```

```
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
```

Quantification Lifting: Type-Level Transformation

Benefit

Don’t need code ahead of time, only type signature

- Dynamic loading (plugins)
- Compiled code
- Third-party libraries

Drawback

Over-approximation of possibly-incurred effects

Related Work

Effect inference

- Operates on *expressions*
- Gives exact bound on effects that can be incurred

Algebraic effects

- Has a different goal
- We use the effect system to formally/statically reason about capabilities

Conclusion

- ***Capabilities*** are good way of managing non-transitive access to system resources
- ***Effect systems*** can formalize capability-based reasoning, but can be verbose
- Craig et al.'s ***import semantics*** work great for lambda calculus
- ***Quantification lifting*** handles tricky interaction between effect polymorphism and mutable state

Thanks to Darya Melicher, Alex Potanin, Jonathan Aldrich, CMU, *and the* NSF!

Thank you!

```
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
```

```
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
```

```
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!
```

```
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
```

Extra Slides

Quantification Lifting: Import Bounds

```
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
```

```
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
```

- ***Something to be careful about:*** bounds on new universally-quantified polymorphism
  - *Upper bound:* Craig et al. import semantics
  - *Lower bound:* Capability-safety

Quantification Lifting: Type-Level Transformation

Before: $\tau_1 \rightarrow \tau_2$

After: $\forall\, \varepsilon\, (L \subseteq \varepsilon \subseteq U)\, .\, \tau_1 \rightarrow (\tau_2)_{\varepsilon}$