Practical Affine Types and Typestate-Oriented Programming

Philipp Haller
KTH Royal Institute of Technology
Stockholm, Sweden

Dagstuhl Seminar 17051
Theory and Applications of Behavioural Types
Schloss Dagstuhl, Germany
January 30th, 2017
Practical Affine Types

• Original motivation:
  *Scala cannot ensure concurrency safety for library-based concurrency abstractions*

• Result (OOPSLA ‘16):
  A design for affine types that enables expressing isolation of concurrent processes in a shared mutable heap

• Goal:
  Generalize type system to *typestate* system

This also applies to Java, C++, and virtually all widely-used programming languages
Why a New Design for Affine Types?

- A lot of progress on related type systems: linear types, static capabilities, uniqueness types, ownership types, region inference, etc.

- Challenges:
  - Sound and robust integration with advanced type system features
  - Adoption on large scale
    - Key: reuse of existing code

Example: local type inference
A New Design for Affine Types

• Enter LaCasa: **Affine types for Scala**
  
  • Objects of affine type have unique, statically-tracked owner
  
  • Ownership is transferable
  
  • LaCasa combines two concepts:
    • Encapsulated boxes
    • Access permissions
Boxes and Access Permissions

- Box = polymorphic heap cell with type member C that uniquely identifies permission required for access
  
  ```
  Box[T] { type C }
  ```

- Permission = stack-local value controlling access to uniquely identified box
  
  ```
  CanAccess { type C }
  ```
Matching Boxes and Permissions

• Type members and path-dependent types match up boxes and permissions

• Example:

```scala
def method[T](box: Box[T])
(p: CanAccess { type C = box.C }): Unit
```
Creating Boxes and Permissions

```scala
sealed trait Packed[+T] {
  val box: Box[T]
  val access: CanAccess { type C = box.C }
}

mkBox[Message] { packed =>
  val access = packed.access
  val box = packed.box
}
```
Accessing Boxes

• Boxes are encapsulated
• Boxes must be opened for access

```scala
mkBox[Message] { packed =>
  val access = packed.access
  val box = packed.box

  box.open({ msg =>
    msg.arr = Array(1, 2, 3, 4)
  })(access)
}
```
Accessing Boxes

- Boxes are encapsulated
- Boxes must be opened for access

```scala
mkBox[Message] { packed =>
  implicit val access = packed.access
  val box = packed.box

  box.open { msg =>
    msg.arr = Array(1, 2, 3, 4)
  }
}
```

Requires implicit access permission

Lewis et al. Implicit parameters: dynamic scoping with static types. POPL '00
Oliveira et al. The implicit calculus: a new foundation for generic programming. PLDI '12
Consuming Permissions

Example: transferring a box from one actor to another consumes its access permission

```scala
mkBox[Message] { packed =>
  implicit val access = packed.access
  val box = packed.box
  ...
  someActor.send(box)

  // illegal to access `box` here!
}
```

How to enforce this?
Permissions and Continuations

- Make implicit permission unavailable in continuation of permission-consuming call
- Scala’s type system is flow-insensitive => use continuation passing
- Restrict continuation to exclude consumed permission
Continuation-Passing Style

```scala
mkBox[Message] { packed =>
  implicit val access = packed.access
  val box = packed.box

  ...

  someActor.send(box) {
    // make `access` unavailable
    ...
  }
}
```
Restricting Continuations

- Continuation disallows the type of access
- Leverage *spores* [1]

``` scala
def send(msg: Box[T])
  (cont: NullarySpore[Unit] {
    type Excluded = CanAccess { type C = msg.C }
  })
  (implicit p: CanAccess { type C = msg.C }): Nothing
```

Encapsulation

Problem: not all types safe to transfer!

```scala
class Message {
  var arr: Array[Int] = _
  def leak(): Unit = {
    SomeObject.fld = arr
  }
}

object SomeObject {
  var fld: Array[Int] = _
}
```
Encapsulation

- Ensuring absence of data races (“concurrency safety”) requires restricting types put into boxes
- Insight: leverage object capability discipline [2]:
  - Methods only access parameters and this
  - Methods only instantiate object-capability safe classes
  - Types of fields are object-capability safe


* simplified
Object Capabilities in Scala

• How common is object-capability safe code in Scala?

• Empirical study of over 75,000 SLOC of open-source Scala code:

<table>
<thead>
<tr>
<th>Project</th>
<th>Version</th>
<th>SLOC</th>
<th>GitHub stats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala stdlib</td>
<td>2.11.7</td>
<td>33,107</td>
<td>⭐5,795 🆕257</td>
</tr>
<tr>
<td>Signal/Collect</td>
<td>8.0.6</td>
<td>10,159</td>
<td>⭐123 🆕11</td>
</tr>
<tr>
<td>GeoTrellis</td>
<td>0.10.0-RC2</td>
<td>35,351</td>
<td>⭐400 🆕38</td>
</tr>
<tr>
<td>-engine</td>
<td></td>
<td>3,868</td>
<td></td>
</tr>
<tr>
<td>-raster</td>
<td></td>
<td>22,291</td>
<td></td>
</tr>
<tr>
<td>-spark</td>
<td></td>
<td>9,192</td>
<td></td>
</tr>
</tbody>
</table>
Object Capabilities in Scala

Results of empirical study:

<table>
<thead>
<tr>
<th>Project</th>
<th>#classes/traits</th>
<th>#ocap (%)</th>
<th>#dir. insec. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala stdlib</td>
<td>1,505</td>
<td>644 (43%)</td>
<td>212/861 (25%)</td>
</tr>
<tr>
<td>Signal/Collect</td>
<td>236</td>
<td>159 (67%)</td>
<td>60/77 (78%)</td>
</tr>
<tr>
<td>GeoTrellis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-engine</td>
<td>190</td>
<td>40 (21%)</td>
<td>124/150 (83%)</td>
</tr>
<tr>
<td>-raster</td>
<td>670</td>
<td>233 (35%)</td>
<td>325/437 (74%)</td>
</tr>
<tr>
<td>-spark</td>
<td>326</td>
<td>101 (31%)</td>
<td>167/225 (74%)</td>
</tr>
<tr>
<td>Total</td>
<td>2,927</td>
<td>1,177 (40%)</td>
<td>888/1,750 (51%)</td>
</tr>
</tbody>
</table>
LaCasa: Summary

- **Type-based notion of the object capability discipline is possible and beneficial**

- **Object capabilities + path-dependent types + stack locality enable affine types**

- **Code reuse with minimal effort**
  - **Valid components of affine types conform to the object capability discipline**
  - **Binary check whether a class is reusable unchanged**

Haller and Loiko:
*LaCasa: Lightweight Affinity and Object Capabilities in Scala.* OOPSLA ’16

Code: [https://github.com/phaller/lacasa](https://github.com/phaller/lacasa)
In the Paper

• **Implementation:**
  - Compiler plugin for Scala 2.11.x and integration with actors
  - Enforcement of continuation-passing style

• **Formalization:** object-oriented core languages
  - CLC$^1$: type-based notion of object capabilities
  - CLC$^2$: uniqueness via flow-insensitive permissions
  - CLC$^3$: concurrent extension

• **Soundness proof**
  - *Isolation theorem* for processes with shared heap

*Formal model in Coq*
Towards Typestate

Idea:

- Refine permission types: typestate as type member
- Change of typestate = permission replacement

```rust
trait Alive // typestate root

trait CanAccess {
    type C
    type State <: Alive
}
```
sealed trait Packed[+T] {
  val box: Box[T]
  val access: CanAccess { type C = box.C }
}

def next[T](b: Box[Iterator[T]])
  (implicit p: CanAccess { type C = b.C
     type State = Available })
  (cont: Spore[(T, Packed[Iterator[T]]), Unit] { type Excluded = CanAccess { type C = b.C }
  }): Nothing
Conclusion

• LaCasa: Object capabilities + path-dependent types + CPS = lightweight affinity (OOPSLA ’16)

• Hypothesis: LaCasa suitable as basis for typestate system
  • Typestates as types
  • CPS unwieldy