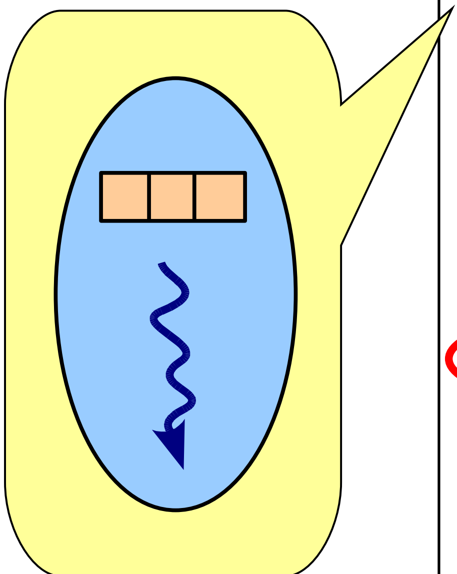
Capabilities for Uniqueness and Borrowing

Philipp Haller and Martin Odersky
EPFL

24th European Conference on Object-Oriented Programming
June 24, 2010
Maribor, Slovenia

Motivating Example



```
actor {
  val buf = new ArrayBuffer[Int]
  buf += 5
  someActor ! buf
  buf.remove(0)
}
val someActor = actor {
  receive {
    case b: ArrayBuffer[Int] =>
      val first = b.remove(0)
      if (first > 4) b += 7
  }
}
```

Potential data race!

Race-Free Imperative Actors

- Wide adoption of actors in industry
- Goal: *Safe and efficient exchange of mutable objects*
  - High performance of sequential code
  - Zero-copy message passing (for data-heavy pipelines, protocol stacks, etc.)
- Challenges:
  - Low conceptual and syntactical overhead
  - Sound foundations
  - Low/no run-time overhead

Isolation through Uniqueness

```
actor {
  val buf: ArrayBuffer[Int] @unique = new ArrayBuffer[Int]
  buf += 5
  someActor ! buf
  buf.remove(0)
}
val someActor = actor {
  react {
    case b: ArrayBuffer[Int] =>
      val first = b.remove(0)
      if (first > 4) b += 7
  }
}
```

buf stores a unique reference

Consumes buf

Compiler output:
test.scala:6: error: buf has been consumed
    buf.remove(0)
    ^

External vs. Separate Uniqueness

External Uniqueness

[*Clarke, Wrigstad 2003; Müller, Rudich 2007; Clarke et al. 2008*]

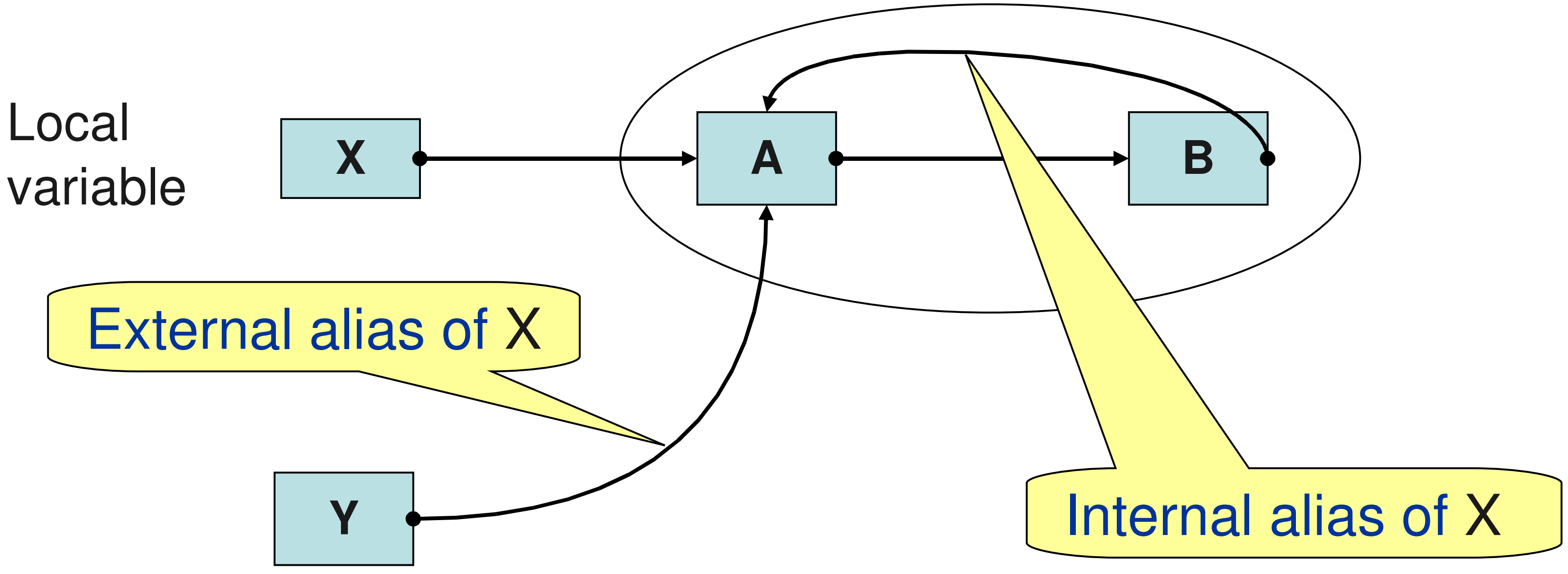
- No external aliases
- No unique method receivers
- Deep/full encapsulation
- Based on ownership types

Separate Uniqueness

(this paper)

- Local external aliases
- Unique method receivers (self transfer)
- Full encapsulation
- No ownership types presumed

Internal vs. External Aliases



Pluggable Type System

• @unique	Unique variable/parameter/result
• @transient	Non-consumable (borrowed) unique parameter
• @peer(x)	Parameter/result in the same region as x
• capture(x, y)	Transfer x into region of y, return alias of x
• swap(x.f, y)	Return unique x.f and replace with unique y

The @unique Annotation

```
val logList: LogList @unique = new LogList
for (test <- tests) {
  val logFile: LogFile @unique = createLogFile(test, kind)
  // run test...
  logList.add(logFile)
}
report(logList)

def report(logList: LogList @unique) {
  master ! new Results(logList)
}
```

Mutating Unique Objects

```
class LogList {
  var elem: LogFile = null
  var next: LogList = this
  @transient def add(file: LogFile @peer(this)) =
    if (isEmpty) { elem = file; next = new LogList }
    else next.add(file)
}
```

- Receiver must remain unique after adding file
- @transient is equivalent to @unique except it does **not consume the receiver**
- file must point into the same region as the receiver, expressed using @peer(**this**)

Transient Classes

- Common pattern:
  - All methods type check with @transient receiver and all parameters @peer(**this**)
  - Method invocations preserve uniqueness
  - Self-contained aggregate objects
  - Class-level @transient annotation

Transferring Unique Objects

- How to transfer a separately-unique object from one region to another?

```
val logList: LogList @unique = new Lo
for (test <- tests) {
  val logFile: LogFile @unique = createLogFile(test)
  // run test...
  logList.add(capture(logFile, logList))
}
```

logFile in disjoint region from region of logList

Returns alias of logFile in region of logList

Consumes logFile

Alias Invariant

- Two local variables x, y are **separate** (in heap H) iff there is no object reachable from both x and y (in H)
- Definition (Separate Uniqueness)

  A variable x is **separately-unique** in heap H iff for all y ≠ x.
  y is live => separate(H, x, y)
- Definition (Alias Invariant)

  Unique parameters are separately-unique

Unique Fields

```
val logList: LogList @unique = swap(this.logFiles, null)
logList.add(capture(logFile, logList))
swap(this.logFiles, logList)
```

Definition (Unique Fields Invariant)

References returned by swap are separately-unique

Pluggable Type System

• @unique	Unique variable/parameter/result
• @transient	Non-consumable (borrowed) unique parameter
• @peer(x)	Parameter/result in the same region as x
• capture(x, y)	Transfer x into region of y, return alias of x
• swap(x.f, y)	Return unique x.f and replace with unique y

Formal Type System

- Class-based object calculus with capabilities and capture/swap
- A unique variable has type $\rho \triangleright C$
- **Capability** ρ = access permission to a **region** in heap
- Definition (Capability Type Invariant)
  - Let $x : \rho \triangleright C$ and $y : \rho' \triangleright C'$ be local variables ($\rho \neq \rho'$)
  - If there is a heap H at program point P such that both x and y are usable at P, then $separate(H, x, y)$

Type Checking

- Typing judgment: $\Gamma \,;\, \Delta \vdash t : T \,;\, \Delta'$
- Type rules consume capability set Δ and produce capability set Δ'
- Capabilities in Δ grant access to variables in t
  - A variable of type $\rho \triangleright C$ can only be accessed if ρ is contained in Δ
- Capabilities in Δ' available after type checking t

Capability Creation/Consumption

Instance creation:

$$\frac{\Gamma\ ;\ \Delta \vdash \overline{y : \rho \triangleright D} \qquad \Delta = \Delta' \oplus \overline{\rho} \qquad fields(C) = \overline{\alpha\ l : D} \qquad \rho' \text{ fresh}}{\Gamma\ ;\ \Delta \vdash \textbf{new}\ C(\overline{y}) : \rho' \triangleright C\ ;\ \Delta' \oplus \rho'}$$

Separation and Internal Aliasing

Field assignment:

$$\frac{\Gamma\ ;\ \Delta\ \vdash\ y : \rho \triangleright C \qquad \Gamma\ ;\ \Delta\ \vdash\ z : \rho \triangleright D_i \qquad fields(C) = \overline{\alpha\ l : D} \qquad \alpha_i \neq \texttt{unique}}{\Gamma\ ;\ \Delta\ \vdash\ y.l_i := z : \rho \triangleright C\ ;\ \Delta}$$

Separate Uniqueness

- Assume x has type $\rho \triangleright C$
- *Capability type invariant:* if there is a heap H where $\neg separate(H, x, y)$, then y has type $\rho \triangleright D$
- Consuming ρ makes all variables of type $\rho \triangleright D$ unusable

→ Consuming ρ makes **all external aliases of x unusable**

Soundness

- Small-step operational semantics
- Soundness established using syntactic Wright-Felleisen technique
- Preservation
  - Static capabilities correspond to dynamic capabilities
  - Reduction preserves separation and uniqueness invariants
- Progress
  - Well-typed programs do not get stuck because of missing capabilities

$$\frac{H, V, R, t_1 \longrightarrow H', V', R', t_1'}{H, V, R, \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 \longrightarrow H', V', R', \mathtt{let}\ x = t_1'\ \mathtt{in}\ t_2} \quad \text{(R-LET)}$$

$$\frac{\begin{array}{c} V(y) = \delta \triangleright r \qquad \delta \in R \\ H(r) = C(\overline{r}) \end{array}}{H, V, R, \mathtt{let}\ x = y.l_i\ \mathtt{in}\ t \longrightarrow H, (V, x \mapsto \delta \triangleright r_i), R, t} \quad \text{(R-SELECT)}$$

$$\frac{\begin{array}{c} V(y) = \delta \triangleright r \qquad V(z) = \delta \triangleright r' \\ H(r) = C(\overline{r}) \qquad \delta \in R \\ H' = H[r \mapsto C([r'/r_i]\overline{r})] \end{array}}{H, V, R, y.l_i := z \longrightarrow H', V, R, y} \quad \text{(R-ASSIGN)}$$

$$\frac{V(\overline{y}) = \overline{\beta \triangleright r} \qquad H' = (H, r \mapsto C(\overline{r})) \qquad r \notin dom(H) \qquad \gamma\ \text{fresh}}{H, V, R \oplus \overline{\beta}, \mathtt{let}\ x = \mathsf{new}\ C(\overline{y})\ \mathtt{in}\ t \longrightarrow H', (V, x \mapsto \gamma \triangleright r), R \oplus \gamma, t} \quad \text{(R-NEW)}$$

$$\frac{\begin{array}{c} V(y) = \beta_1 \triangleright r_1 \qquad H(r_1) = C_1(\_) \\ V(\overline{z}) = \beta_2 \triangleright r_2 \ldots \beta_n \triangleright r_n \qquad \overline{\beta} \subseteq R \\ mbody(m, C_1) = (\overline{x}, e) \end{array}}{H, V, R, \mathtt{let}\ x = y.m(\overline{z})\ \mathtt{in}\ t \longrightarrow H, (V, \overline{x \mapsto \beta \triangleright r}), R, \mathtt{let}\ x = e\ \mathtt{in}\ t} \quad \text{(R-INVOKE)}$$

$$\frac{V(y) = \beta \triangleright r \qquad V(z) = \gamma \triangleright \_}{H, V, R \oplus \beta \oplus \gamma, \mathtt{let}\ x = \mathsf{capture}(y, z)\ \mathtt{in}\ t \longrightarrow H, (V, x \mapsto \gamma \triangleright r), R \oplus \gamma, t} \quad \text{(R-CAPTURE)}$$

$$\frac{\begin{array}{c} V(y) = \beta \triangleright r \qquad H(r) = C(\overline{r}) \qquad \gamma\ \text{fresh} \\ V(z) = \beta' \triangleright r' \qquad H' = H[r \mapsto C([r'/r_i]\overline{r})] \end{array}}{H, V, R \oplus \beta \oplus \beta', \mathtt{let}\ x = \mathsf{swap}(y.l_i, z)\ \mathtt{in}\ t \longrightarrow H', (V, x \mapsto \gamma \triangleright r_i), R \oplus \beta \oplus \gamma, t} \quad \text{(R-SWAP)}$$

$$\frac{\Gamma ; \Delta \vdash e : T' ; \Delta' \qquad T' <: T}{\Gamma ; \Delta \vdash e : T ; \Delta'} \quad \text{(T-SUB)}$$

$$\frac{\Gamma ; \Delta \vdash e : T ; \Delta' \qquad \Gamma, x : T ; \Delta' \vdash t : T' ; \Delta''}{\Gamma ; \Delta \vdash \texttt{let}\ x = e\ \texttt{in}\ t : T' ; \Delta''} \quad \text{(T-LET)}$$

$$\frac{\Gamma(y) = \rho \triangleright C \qquad \rho \in \Delta}{\Gamma ; \Delta \vdash y : \rho \triangleright C ; \Delta} \quad \text{(T-VAR)}$$

$$\frac{\Gamma ; \Delta \vdash y : \rho \triangleright C \qquad fields(C) = \overline{\alpha\ l : D} \qquad \alpha_i \neq \textsf{unique}}{\Gamma ; \Delta \vdash y.l_i : \rho \triangleright D_i ; \Delta} \quad \text{(T-SELECT)}$$

$$\frac{\Gamma ; \Delta \vdash y : \rho \triangleright C \qquad \Gamma ; \Delta \vdash z : \rho \triangleright D_i \qquad fields(C) = \overline{\alpha\ l : D} \qquad \alpha_i \neq \textsf{unique}}{\Gamma ; \Delta \vdash y.l_i := z : \rho \triangleright C ; \Delta} \quad \text{(T-ASSIGN)}$$

$$\frac{\Gamma ; \Delta \vdash \overline{y : \rho \triangleright D} \qquad \Delta = \Delta' \oplus \overline{\rho} \qquad fields(C) = \overline{\alpha\ l : D} \qquad \rho' \text{ fresh}}{\Gamma ; \Delta \vdash \textsf{new}\ C(\overline{y}) : \rho' \triangleright C ; \Delta' \oplus \rho'} \quad \text{(T-NEW)}$$

$$\frac{\begin{array}{c} \Gamma ; \Delta \vdash y : \rho_1 \triangleright D_1 \\ \Gamma ; \Delta \vdash z_{i-1} : \rho_i \triangleright D_i,\ i = 2..n \\ mtype(m, D_1) = \exists \delta.\ \overline{\delta \triangleright D} \rightarrow (R, \Delta_m) \\ \sigma = \overline{\delta \mapsto \rho} \circ \delta \mapsto \rho \text{ injective} \qquad \rho \text{ fresh} \\ \Delta = \Delta' \uplus \{\rho \mid \rho \in \overline{\rho}\} \end{array}}{\Gamma ; \Delta \vdash y.m(\overline{z}) : \sigma R ; \sigma \Delta_m \oplus \Delta'} \quad \text{(T-INVOKE)}$$

$$\frac{\Gamma ; \Delta \vdash y : \rho \triangleright C \qquad \Gamma ; \Delta \vdash z : \rho' \triangleright C' \qquad \Delta = \Delta' \oplus \rho}{\Gamma ; \Delta \vdash \textsf{capture}(y, z) : \rho' \triangleright C ; \Delta'} \quad \text{(T-CAPTURE)}$$

$$\frac{\begin{array}{c} \Gamma ; \Delta \vdash y : \rho \triangleright C \qquad \Gamma ; \Delta \vdash z : \rho' \triangleright D_i \\ fields(C) = \overline{\alpha\ l : D} \qquad \alpha_i = \textsf{unique} \\ \Delta = \Delta' \oplus \rho' \qquad \rho'' \text{ fresh} \end{array}}{\Gamma ; \Delta \vdash \textsf{swap}(y.l_i, z) : \rho'' \triangleright D_i ; \Delta' \oplus \rho''} \quad \text{(T-SWAP)}$$

Implementation and Experience

- Plug-in for Scala 2.8 compiler
  - Erases capabilities, capture, and swap for code generation
- Mutable collection classes
  - DoubleLinkedList, ListBuffer, and HashMap
  - 2046 LOC including all classes/traits that these classes transitively extend
  - Making all classes @transient required changing 60 LOC

Checking Concurrent Programs

- **partest framework, 4182 LOC**
  - Used for check-in and nightly tests of Scala compiler and standard library
  - Uniqueness annotations for isolation checking: 32 LOC changed, 29 LOC added
    - Transient message classes
    - Unique variables/parameters
    - 3 swaps to access a unique field
- **Ray tracer, 414 LOC**
  - Uniqueness annotations for isolation checking: 18 LOC changed or added (3x @transient, 8x @unique)

More Related Work (1)

- Linear Types and Typestate Checking
  - Adoption and focus
    - No internal aliasing [*Fähndrich and DeLine 2002*]
    - Type qualifiers, region declarations, and effect annotations [*Boyland and Retert 2005*]
  - Capabilities [*Walker et al. 2000; Charguéraud and Pottier 2008*]
    - No merging of heterogeneous regions (no capture)
  - Typestate checking
    - Typestates, invariants, and permission packing/unpacking [*Bierhoff and Aldrich 2007*]
    - More refined permissions [*Degen et al. 2007*]

More Related Work (2)

- Process/Thread Isolation
  - PRFJ [*Boyapati et al. 2002*]
    - No internal aliasing, explicit ownership types, effects
  - PacLang [*Ennals et al. 2004*], StreamFlex [*Spring et al. 2007*]
    - Message fields of primitive or primitive array types
  - Sing# [*Fähndrich et al. 2006*]
    - Explicit message exchange heap
  - Kilim [*Srinivasan and Mycroft 2008*]
    - Message shape restricted to trees

Sound?

Conclusion

- *New approach to internally-aliased unique references*
  - Simple capability tokens are enough
  - Uniform treatment of uniqueness and temporary aliasing
- *Lightweight type system*
  - No explicit regions/owners
  - No static alias analysis
  - Syntactic soundness proof
- *Practical annotation system* (lamp.epfl.ch/~phaller/)
  - Applied to real-world, concurrent Scala code
  - Support for closures and nested classes