Environmental Acquisition Revisited

Richard Cobbe and Matthias Felleisen

Northeastern University
What is Acquisition?
Example: Swing Containers

JRootPane located only at top level
JButton must chase pointers to access root pane
Example: Financial Application

Operations on *Fund* s must know tax policy
Example: Financial Application

Operations on $Fund$'s must know tax policy
With acquisition, no longer need to maintain and chase parent refs
Example: IDE Wizard
Example: Wizard with Acquisition
Containment Invariants

Invariants ensured by language support for acquisition:
Containment Invariants

Invariants ensured by language support for acquisition:

- Objects allow access to their containers
Containment Invariants

Invariants ensured by language support for acquisition:

- Objects allow access to their containers
- Two-way links (or their analog) are consistent
Restrictions on Acquisition

- Limit object’s “environment” to its containers
- Only specifically marked fields establish containment relationship
- An object may have at most one container
- Object containment cycles forbidden
Jacques: the Formal Model
Jacques

Based on *ClassicJava*, formal model of Java by Flatt, Krishnamurthi, and Felleisen (1998).

Supported features:

- core OO: classes, inheritance, method dispatch
- field assignment
Based on *ClassicJava*, formal model of Java by Flatt, Krishnamurthi, and Felleisen (1998).

**Supported features:**

- core OO: classes, inheritance, method dispatch
- field assignment
- field and method acquisition
- explicit marks for “containment” fields
- list of possible containers in class definitions
Wizard Example

class UnionInfo extends ClassUnionWizard {
    VariantPanel vPanel;
    ...
}

class VariantPanel extends HorizontalPanel {

    Button editButton;
    void add(...) { ... }
    void produce(...) { ... }
    void errorMessage(String msg) { ... }
}

Environmental Acquisition Revisited — POPL 2005 — p.11/30
class UnionInfo extends ClassUnionWizard {
    contains VariantPanel vPanel;
    ...
}

class VariantPanel extends HorizontalPanel
    contained ClassUnionWizard {
    Button editButton;
    void add(...) { ... }
    void produce(...) { ... }
    acquires void errorMessage(String msg);
}
Static Check I

Environmental Acquisition Revisited — POPL 2005 — p.13/30
\[ D \text{ acquires } f_d \text{ from } B, \text{ and types match.} \]

Program is well-typed.
Static Check II

A
bool fd
contains B b

B : contained A
int fd
contains C c

C : contained B
bool fd
contains D d

D : contained C
acquires int fd
D acquires fd from C, and types are not compatible.
Program is not well-typed.
Design Decisions
Running Example

Ctnr1
- Prop1 fd
  - contains Item it
- Prop1 meth(Property p) { ... }

Item : contained Ctnr1, Ctnr2
- acquires Property fd
- acquires Property meth(Prop2)

Ctnr2
- Prop2 fd
  - contains Item it
- Property meth(Prop2 x) { ... }

Prop1...
- ...

Prop2...
- ...

Property...
  - ...

Environmental Acquisition Revisited — POPL 2005 — p.16/30
Acquisition by Value and by Name

When does \texttt{anItem} acquire \texttt{fd’s value}?
When does \texttt{anItem} acquire \texttt{fd}'s value?

- By value: when \texttt{anItem} is placed into \texttt{aCtnr1}.
Acquisition by Value and by Name

When does \texttt{anItem} acquire \texttt{fd}’s value?

- By value: when \texttt{anItem} is placed into \texttt{aCtnr1}.
- By name: when \texttt{anItem.fd} is referenced.
Acquisition by Value and by Name

When does \texttt{anItem} acquire \texttt{fd}'s value?

- By value: when \texttt{anItem} is placed into \texttt{aCtnr1}.
- By name: when \texttt{anItem.fd} is referenced.

Both are sound; primarily affects visibility of assignments.
Acquisition by Value and by Name

Two questions with acquisition-by-value:
Acquisition by Value and by Name

Two questions with acquisition-by-value:

1. `aCtnr1.it := null;`
   `anItem.fd`: previous value or undefined?
Acquisition by Value and by Name

Two questions with acquisition-by-value:

1. `aCtnr1.it := null;`
   `anItem.fd`: previous value or undefined?

2. `aCtnr2.it := anItem;`
   `anItem.fd`: previous value, or value of `aCtnr2.fd`?
Acquisition by Value and by Name

Two questions with acquisition-by-value:

1. \( \text{aCtnr1.it := null; } \)
   \( \text{anItem.fd: previous value or undefined? } \)

2. \( \text{aCtnr2.it := anItem; } \)
   \( \text{anItem.fd: previous value, or value of aCtnr2.fd? } \)

We implement acquisition-by-name; it avoids both issues.
Gil and Lorenz claim that the above program is type-safe, because of normal method-type co/contravariance.
Gil and Lorenz claim that the above program is type-safe, because of normal method-type co/contravariance.
Gil and Lorenz claim that the above program is type-safe, because of normal method-type co/contravariance.
Gil and Lorenz claim that the above program is type-safe, because of normal method-type co/contravariance.

Unsafe!
Gil and Lorenz claim that the above program is type-safe, because of normal method-type co/contravariance.

Unsafe!

Co/contravariance don’t apply.
Variance is still possible.  
Acquiring class may expect more *general* type.
Assignment to Acquired Fields

aCtnr1 : Ctnr1
contains Item it
Prop1 fd

anItem : Item
acquires Property fd
In a naïve system, `anItem.fd := new Prop2()` type-checks.
In a naïve system, \texttt{anItem.fd := new Prop2()} type-checks. But \texttt{anItem.fd} is an alias to \texttt{aCtnr1.fd}.
In a naïve system, `anItem.fd := new Prop2()` type-checks. But `anItem.fd` is an alias to `aCtnr1.fd`. **Unsafe:** `aCtnr1.fd` is no longer a `Prop1`. 
Assignment to Acquired Fields

Three possible solutions:

1. Forbid subsumption on the right-hand side of assignments to acquired fields.
Assignment to Acquired Fields

Three possible solutions:

1. Forbid subsumption on the right-hand side of assignments to acquired fields.
   Introduces bad asymmetry into language.
Assignment to Acquired Fields

Three possible solutions:

1. Forbid subsumption on the right-hand side of assignments to acquired fields.
   Introduces bad asymmetry into language.

2. Forbid type variance for acquired fields.
Assignment to Acquired Fields

Three possible solutions:

1. Forbid subsumption on the right-hand side of assignments to acquired fields.
   Introduces bad asymmetry into language.

2. Forbid type variance for acquired fields.
   Too inflexible.
Assignment to Acquired Fields

Three possible solutions:

1. Forbid subsumption on the right-hand side of assignments to acquired fields.
   Introduces bad asymmetry into language.

2. Forbid type variance for acquired fields.
   Too inflexible.

3. Forbid assignment to acquired fields.
Assignment to Acquired Fields

Three possible solutions:

1. Forbid subsumption on the right-hand side of assignments to acquired fields.
   Introduces bad asymmetry into language.

2. Forbid type variance for acquired fields.
   Too inflexible.

3. Forbid assignment to acquired fields.
   *Jacques* implements option 3: right balance between flexibility and safety.
Changing Containers

```
<table>
<thead>
<tr>
<th>aCtnr1 : Ctnr1</th>
<th>anItem : Item</th>
<th>aCtnr2 : Ctnr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains Item it</td>
<td>acquires Property fd</td>
<td>contains Item it</td>
</tr>
<tr>
<td>Prop1 fd</td>
<td></td>
<td>Prop2 fd</td>
</tr>
</tbody>
</table>
```
Changing Containers

Assignment $aCtnr1.it := anItem$ automatically updates hidden parent ref.
Changing Containers

Assignment \( a\text{Ctnr1}.it := an\text{Item} \) automatically updates hidden parent ref.

Can change existing containment tree: \( a\text{Ctnr2}.it := an\text{Item} \).

Violates two-way reference invariant.
Changing Containers

Assignment \( \text{aCtnr1}.\text{it} := \text{anItem} \) automatically updates hidden parent ref.

Can change existing containment tree: \( \text{aCtnr2}.\text{it} := \text{anItem} \).

Violates two-way reference invariant.

So we forbid this assignment.
What is **this** when executing acquired method `anItem.meth(...)`?

- **Delegation**: **this** refers to acquiring object (`anItem`)
- **Forwarding**: **this** refers to providing object (`aCtnr1`)
**Forwarding and Delegation**

What is **this** when executing acquired method `anItem.meth(…)`?

- **Delegation:** **this** refers to acquiring object (`anItem`)
- **Forwarding:** **this** refers to providing object (`aCtnr1`)

Delegation unsafe: body of `Ctnr1.meth` type-checked under assumption that **this** : `Ctnr1`.
Type Soundness
Jacques Soundness

If program $P$ has type $t$, then evaluating $P$ has one of the following results:

- The result is an object reference with the right type, or
- The result is `null`, or
- The program diverges, or
- The program halts with an error:
  - dereferenced `null`
  - bad cast
Jacques Soundness

If program $P$ has type $t$, then evaluating $P$ has one of the following results:

- The result is an object reference with the right type, or
- The result is `null`, or
- The program diverges, or
- The program halts with an error:
  - dereferenced `null`
  - bad cast
  - incomplete context
  - object already contained
  - container cycle
Conclusions
Contributions

We have placed demonstrated acquisition’s technical feasibility and placed it on a firm theoretical foundation.

- We developed a formal model for reasoning about acquisition in the context of a Java-like language.
- We used the formal model to re-examine Gil & Lorenz’s conclusions about type safety.
- We explored the interactions between acquisition and assignment.
Future Work

- Wider range of examples of acquisition.
- Practical experience: implement this and use it.
- More advanced type systems:
  - Can we infer list of possible containers for a class?
  - Can a resource-aware type system ensure that the “incomplete context” exception is never generated?
Related Work

Ownership types (Clarke et al):
Related Work

Ownership types (Clarke et al): 

- Also constrain object containment—to limit object aliasing
Related Work

Ownership types (Clarke et al):

- Also constrain object containment—to limit object aliasing
- Could help us ensure no object has multiple containers
Related Work

Ownership types (Clarke et al):

- Also constrain object containment—to limit object aliasing
- Could help us ensure no object has multiple containers
- But resulting constraints on aliasing too restrictive
Related Work

Ownership types (Clarke et al):

- Also constrain object containment—to limit object aliasing
- Could help us ensure no object has multiple containers
- But resulting constraints on aliasing too restrictive
- Cannot statically prevent “incomplete context” exceptions
Thank you.

cobbe@ccs.neu.edu