On Inner Classes

Atsushi Igarashi (Univ. of Tokyo)

joint work with

Benjamin Pierce (Univ. of Penn.)

Inner Classes in Java

- Class declaration in a class
  - as a member just as fields/methods
  - inside a method (a.k.a. local/anonymous classes)
- Direct access to members of its enclosing definitions
  - Programming style similar to nested functions
- Extensively used in event-driven programming (e.g., AWT)

Inner Classes in AWT programming

- Counter and event handler (ActionListener)

```
class Counter {
  int x=0;
  void regButton(Button b) {
    b.addActionListener(new Listener());}
  class Listener implements ActionListener {
    public void actionPerformed(ActionEvent e)
     { x++; }}}
```

  - Method `regButton` associates a listener with a given button
  - Every time the button is pressed, `x` will be incremented through the invocation of `actionPerformed` of the listener object

Complication (1): Scoping Rules

- Directly accessible members are in
  - superclasses
  - enclosing classes
  - even superclasses of enclosing classes

```
class A {
  Object f; …
}
class B {
  Object f;
  class C extends A {
    void m() {
      … f … // which f?
}}}
```

Complication (2): Inheritance

- Almost any form of inheritance is allowed
  - Inner class is inherited outside of the defined scope

```
class A { …
  class B { … }
}
class C extends A.B { … }
```

  - Inner class can extend its enclosing class

```
class A { …
  class B extends A { … }
}
```

Official Specification by JavaSoft

- Explains what inner classes are and how to compile them to Java Virtual Machine Language
- However, it is not quite sufficient
  - indirect
    - the semantics is given only via compilation
  - informal and sometimes ambiguous
    - written in English prose
    - interpreted differently by different versions of JDK

Our Goal

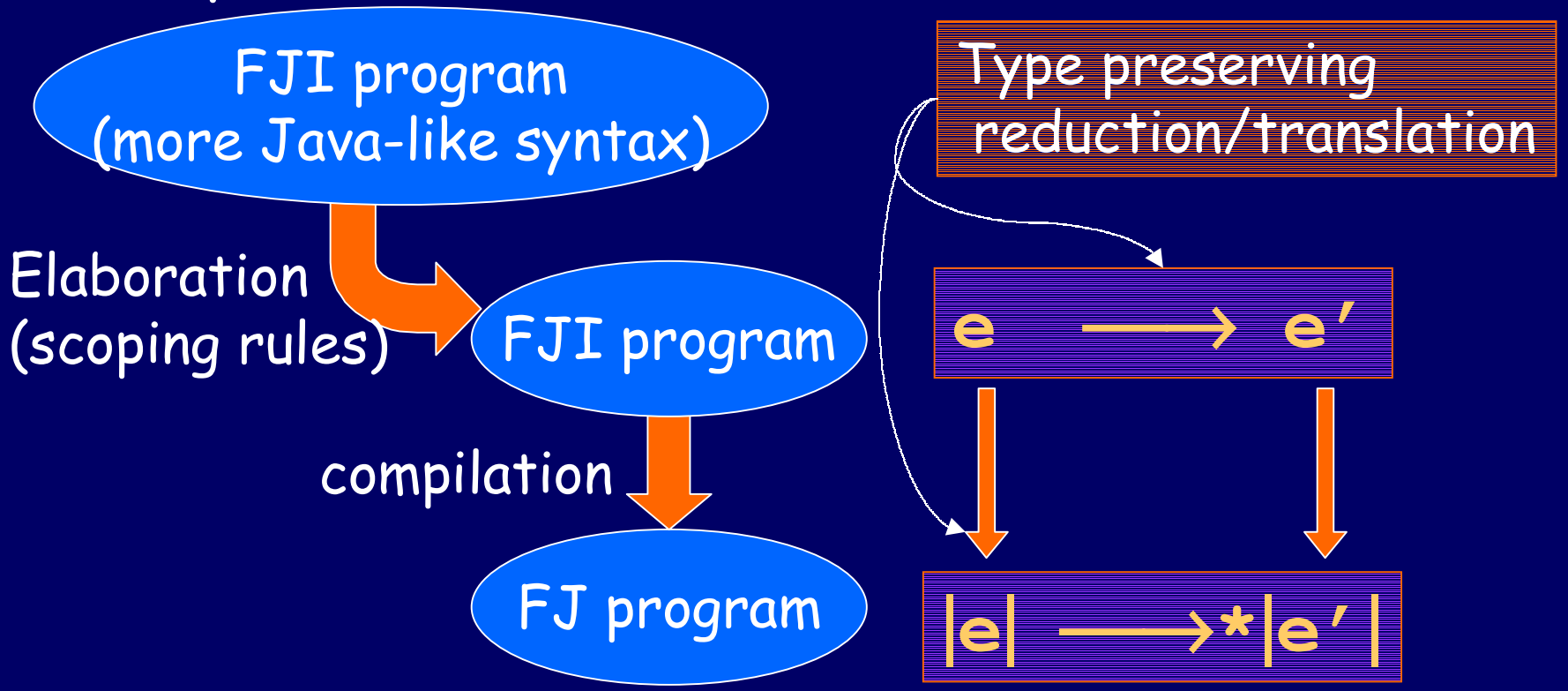
- Clarification of the essence of inner classes by formalization of core features
  - direct semantics
  - compilation and its correctness
  - scoping rules
- Feedback to language design

Contributions

- Design of a toy OO language, FJI, with inner classes
  - extension of Featherweight Java (FJ) [Igarashi&Pierce&Wadler OOPSLA99]
  - w/o local/anonymous classes and access control annotations
- Formalization of the language
  - direct operational semantics and type system
  - scoping rules

Contributions (contd.)

- Formalization of the current compilation scheme
- Proof of type soundness and correctness of compilation



Outline

- Basic Constructs about Inner Classes
- Overview of Formalization
- Interaction with Inheritance
- Concluding Remarks

Enclosing Instances

- Every instance of an inner class is associated with an *enclosing instance* of the enclosing class
  - e.g., `Listener` object has its `Counter` object to increment

  - cf., function closure = code + environment
- `new Listener()` in `Counter` creates a listener whose enclosing instance is `this`

.this Notation

- Enclosing instance is accessible with `C.this`
  - `C` is the enclosing class's name

- Direct access to `x` is just an abbreviation of access to the field `x` of the enclosing instance

```
class Counter {
  int x; …
  class Listener … {
    public void actionPerformed(…){
      Counter.this.x++; } … }}
```

Prefix to `new`

- Enclosing instance of a newly created object can be explicitly specified by prefixing **`new`**
  - **`e.new Listener()`** creates a new listener where **`Counter.this`** refers to **`e`**, which is a **`Counter`** object
  - **`new Listener()`** in the class **`Counter`** is an abbreviation of **`this.new Listener()`**
- You can instantiate a **`Listener`** even outside the class **`Counter`**

```
Counter c = new Counter();
Counter.Listner l = c.new Listener();
```

Formalization: FJ+Inner Classes (FJI)

- FJ [Igarashi&Pierce&Wadler OOPSLA99]
  - Slogan: "λ-calculus" for Java
    - a tiny sublanguage of Java with ...
      - top-level class declarations, inheritance
      - subtyping
      - fields, (side-effect-free) methods
      - method recursion through `this`
    - simple reduction semantics
      - `e` → `e'` (w.r.t. a certain set of class definitions)
  - Testbed for formalizing extensions of Java such as parameterized classes

Formalization: FJ+Inner Classes (FJI)

- FJ [IgarashiPierceWadler99]

  +

- Inner Classes as Members
  - notation `C.this` for enclosing instances
  - prefix to `new`
  - notations to support inheritance (explained later)

Example of FJI Program

External syntax

```
class A {
  class B {
    Object f; Object g;
    B(Object f, Object g) { this.f=f; this.g=g; }
    B set_g(Object x) {
      return new B(f, x); }}}
```

- Method body consisting of a single **return**
  - The method **set_g** returns a new object rather than updating the field **g**

Example of FJI Program

Internal syntax

```
class A {
  class B {
    Object f; Object g;
    B(Object f, Object g) { this.f=f; this.g=g; }
    A.B set_g(Object x) {
      return A.this.new B(this.f, x); }}}
```

- Method body consisting of a single **return**
- Everything is made explicit by elaboration
  - "fully-qualified" type names
  - **this**/**A.this** before field accesses and **new**

Reduction Relation: e → e′

- When a method is invoked, the redex becomes the method body (the expression after **return**), replacing formal parameters, **this** and **A.this**

```
(new A().new B(a,b)).set_g(c)
→ [c/x, new A()/A.this, new A().new B(a,b)/this]
        A.this.new B(this.f, x)
    (= new A().new B(new A().new B(a,b).f, c))
→ new A().new B(a,c)
```

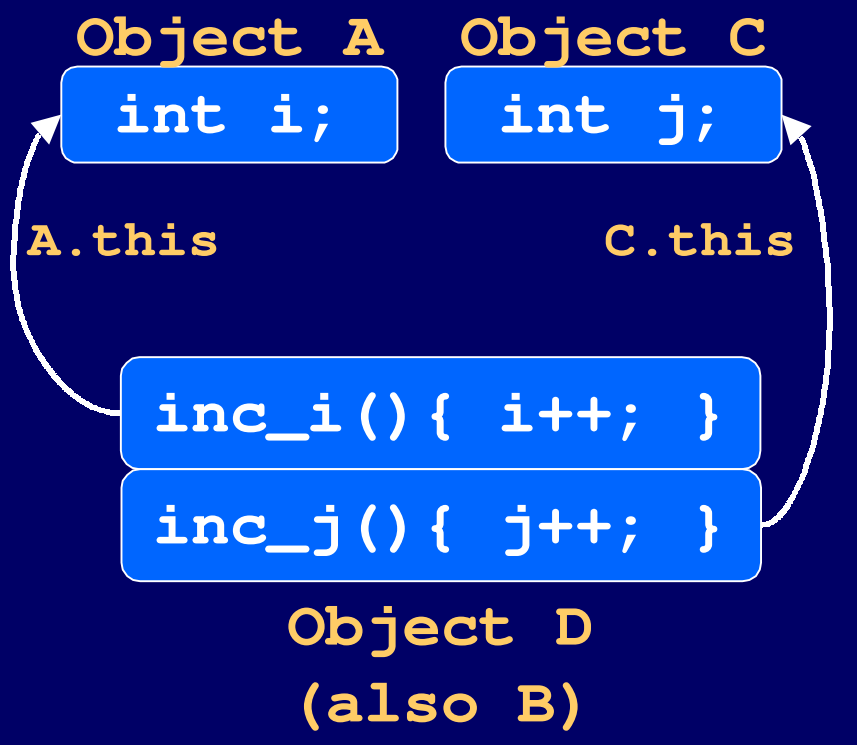
  - The value for **A.this** is immediate from the form of the receiver

Inheritance and Inner Classes

- Inherited methods may require a *different environment* i.e., enclosing instance
  - *How do you create an object of* `C.D`*?*

```
Class A {
  int i;
  class B {
    void inc_i(){
      A.this.i++; }}}

class C {
  int j;
  class D extends A.B {
    void inc_j() {
      C.this.j++; }}}
```



Prefix to `super();`

- Constructor of `D` must specify the value for `A.this` used in `B` by prefixing `super();`

```
class C {
 int j;
 class D extends A.B {
   D(A a){ a.super(); } … }}
```

  - Both `a.new B().inc_i()` and `c.new D(a).inc_i()` increments `i` of `a`
- In FJI, a prefix is restricted to be a constructor argument
  - The prefix can be any expression in Java
  - Even under such a restriction, computing `A.this` gets rather complicated

Technical Results

- Definition of FJI
  - reduction semantics and type system
    - main complexity in the computation of enclosing instances in method invocations
  - Elaboration (scoping rules) in the full version
- Formal rules of compilation to pure FJ
  - Closure conversion of class definitions
- Theorem: type soundness
- Theorem: correctness of compilation

What was "Real" Complication?

- ✔ An instance of an inner class is just like a closure!
  - ✔ Convenient for exporting a small piece of code outside
  - ◆ Environment is represented by an object called enclosing instance
- ✘ Liberal subclassing lets methods from one object refer to distinct environments
  - ◆ By certain restrictions,
    - ◆ we wouldn't need a prefix to `super`
    - ◆ simpler semantics could be given

Subtlety in Scoping Rules

- Some classes/packages can be inaccessible!
  - Necessity of syntax for "top-level"?
    - c.f. "::" in C++

```
class A { … }
class B { …
  class A {
    // you can't use the top-level class A,
    // or package A here!!
  }
}
```

Related Work

- Block expressions in Smalltalk
  - "closure" object
  - no inheritance
- Nested patterns in Beta
  - Is it possible to write such weird programs?
  - Virtual classes
    - `e.new C()` selects the invoked constructor depending on the *run-time* class of `e`

Future Work

- Local/anonymous classes
  - can be free from weird inheritance!!
  - may refer to method parameters as free variables
    - `final` is required for method parameters referred to in such classes
    - interaction with side-effects is expected
- Access control annotations (`public`, `private`,...)
  - Correctness of compilation harder to prove
    - JDK compiler may translate a private member to a package member
      - insecure?
      - Bhowmik and Pugh's implementation scheme [PLDI99 poster session]