Enabling the Automation of Handler Bindings in Event-Driven Programming

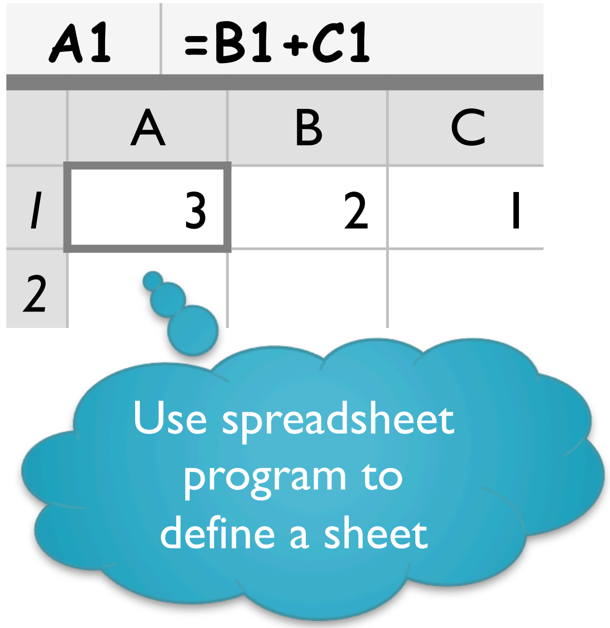
YungYu Zhuang and Shigeru Chiba
The University of Tokyo

FRP and Signals

- FRP (Functional-reactive programming) was developed for reactive programs
  - Signals (behaviors)
    - To describe continuous data streams, e.g. mouse position
  - Event streams
    - A series of events on the timeline, e.g. mouse click
    - Used to get a snapshot of signals at a specific time

A simple example: spreadsheet

- Can be regarded as an interactive programming environment
  - Cells are fields
  - Sheets are some sort of objects



A1 =B1+C1

	A	B	C
1	3	2	1
2			

- A cell can contain a constant value or an expression
  - B1 = 2
  - C1 = 1
  - A1 = B1 + C1 ← updated automatically when the value of B1 or C1 is changed

It's easy to implement the sheet with signals

```
<body onload="loader()">
  <table width=200>
    <tr><th>A1</th>
        <th>B1</th>
        <th>C1</th>
    </tr>
    <tr><th><input id="A1" size=2 value="0" /></th>
        <th><input id="B1" size=2 value="2" /></th>
        <th><input id="C1" size=2 value="1" /></th>
    </tr>
  </table>
</body>

<script type="text/flapjax">
function loader() {
  var b = extractValueB("B1");
  var c = extractValueB("C1");
  var a = b + c;
  insertValueB(a, "A1", "value");
}
</script>
```

Flapjax: JavaScript-based FRP
[L.A. Meyerovich et al, OOPSLA'09]

Extract a signal (behavior) from UI component

Simply describe them as in spreadsheet programs!

Insert a signal (behavior) into UI component

Wait a minute! We already have events

- Event-driven programming has been developed for a long time
  - Widely used in OO libraries

- Can't we use events to implement this sheet example?

  → Yes, of course we can.

We can also implement it with events

```
class Sheet {
  var a: Int = _
  var b: Int = _
  var c: Int = _
  def setB(nb: Int) { b = nb; }
  def setC(nc: Int) { c = nc; }

  evt eb[Unit] = afterExec(setB)
  evt ec[Unit] = afterExec(setC)
  def ha() { a = b + c; }
  eb += ha;
  ec += ha;
}
```

EScala: Scala-based event system
[V. Gasiunas et al, AOSD'11]

An event occurring after the specified method is executed

The relation is described in a handler

Add a handler (right-hand side) to the event (left-hand side)

Both of them state "a = b + c", but…

- In the signal version
  - The assignment looks like an equation
- In the event version
  - Only effective just after it is executed
  - Might not be "true" until it is executed again

signal version:

```
  :
var a = b + c;
  :
```

event version:

```
  :
evt eb[Unit] = afterExec(setB)
evt ec[Unit] = afterExec(setC)
def ha() { a = b + c; }
eb += ha;
ec += ha;
  :
```

Our proposal: expanding event systems

- To provide the automation in handlers
  → allow binding a handler to all events inside itself automatically
  1. Automatic inference
     - Can find out $\mathtt{e}_\mathtt{b}$ and $\mathtt{e}_\mathtt{c}$ according to $\mathtt{h}_\mathtt{a}$
  2. Implicit binding
     - Can bind $\mathtt{h}_\mathtt{a}$ to $\mathtt{e}_\mathtt{b}$ and $\mathtt{e}_\mathtt{c}$
- A prototype implementation: ReactiveDominoJ (RDJ)

RDJ is a Java-based language

- Such a method-like declaration can be both event and handler.

```
public void setX(int nx) {
  this.x = nx;
}
```

  - Add a handler to the event `s.setX`

```
s.setX += o.update;
```

  - Add the handler `s.setX` to an event

```
t.tick += s.setX;
```

Syntax:
⟨event⟩ ⟨assignment operator⟩ ⟨handler⟩;

The braces operator in RDJ

- ⟨event⟩ can be also a handler within {}
  - To select all the involved events inside a handler

- E.g. {h_a} refers to all the involved events in h_a
  - // enables the automation
    `{this.updateA} += this.updateA();`
  - `{_} += this.updateA;`   // "_" refers to the right one

→ RDJ compiler will do the rest
  - Infers the involved events and binds the handler to them

Compare RDJ version with Flapjax-like version

No need to manually enumerate all events

RDJ version with {}

```
public class PlusSheet
 extends Sheet {
  public PlusSheet() {
      :
    {_} += this.updateA();
  }
  public void updateA() {
    a.setValue(b.getValue() + c.getValue());
  }
}
```

RDJ version without {}

```
public class PlusSheet
 extends Sheet {
  public PlusSheet() {
      :
    b.setValue += this.changed;
    c.setValue += this.changed;
    this.changed += this.updateA;
  }
  public void changed(int v);
  public void updateA(int v) {
    a.setValue(b.getValue() + c.getValue());
  }
}
```

Flapjax-like version

```
public class PlusSheet
 extends Sheet {
  public PlusSheet() {
      :
    Behavior b = b1.extractValueB();
    Behavior c = c1.extractValueB();
    Behavior a = b + c;
    a1.insertValueB(a);
  }
}
```

In order to make it easier to compare we assume that there were a Flapjax-like Java-based language

What RDJ compiler does

- A binding with {} is boiled down
  - {_} += this.updateA();

```
→ b.setValue += this.updateA();
  c.setValue += this.updateA();
```

- No need to manually ensure the consistency between bindings and handler body
  - Reduce the risk of bugs
  - Make code shorter

Implicit vs. Explicit!

- That's the difference between FRP and existing event systems

- The braces operator makes it implicit
  - As what signals do

- What is the insufficiency in existing event systems?
  - i.e. how the braces operator does?

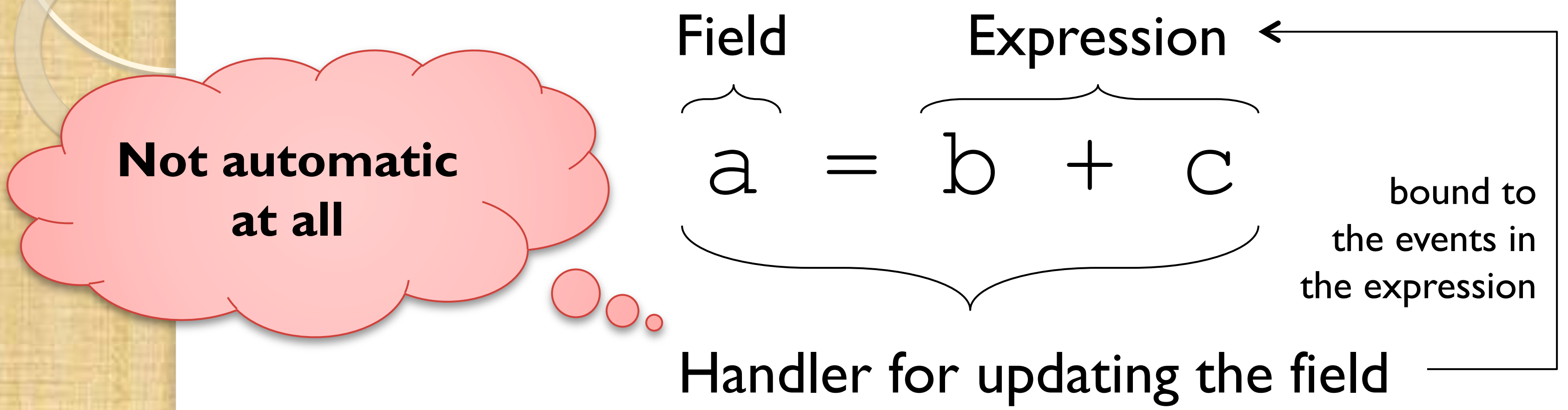
Let’s check the meaning of signals again

$$\underbrace{\overbrace{\texttt{a}}^{\text{Signal}} = \overbrace{\texttt{b} + \texttt{c}}^{\text{Expression}}}_{\text{Signal assignment}}$$

- A signal (a)
  - Is reevaluated when any of the signals (b, c) in its expression is reevaluated
  - Then all signals whose expressions contain this signal (a) will be reevaluated as well

For detailed description of signals and signal assignments, please refer to our paper

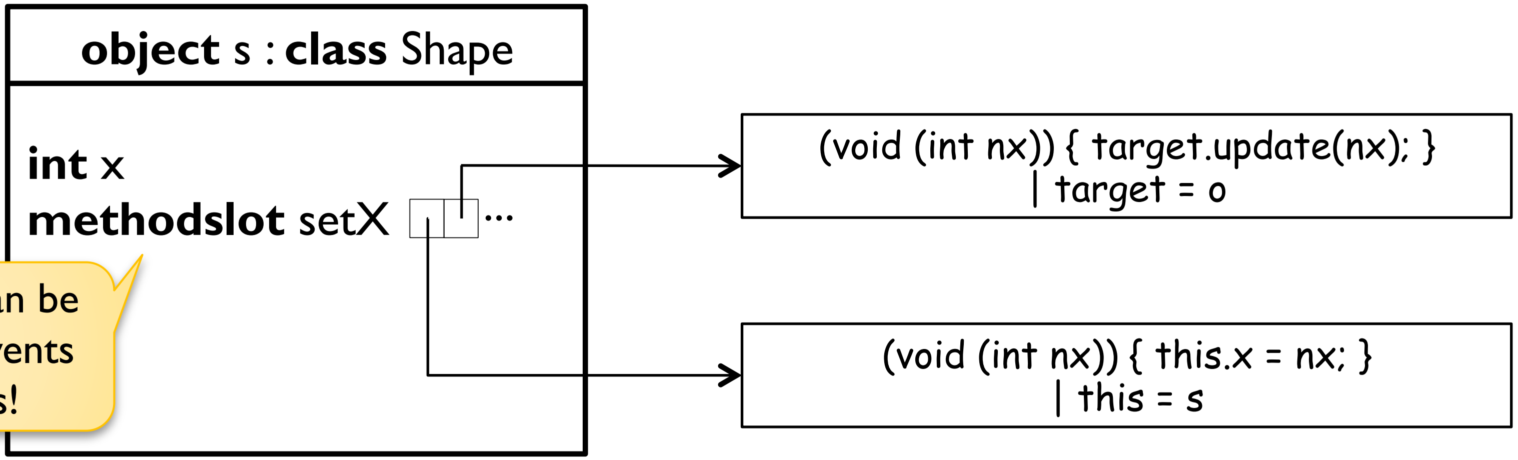
How signals are translated in event systems


Not automatic at all
Field
Expression
a = b + c
bound to the events in the expression
Handler for updating the field

- A signal is a field (a) with an event (e_a)
- Its assignment is a handler (h_a)
  - The handler is executed when any of events (e_b, e_c) in its expression occurs
  - Then its event (e_a) occurs and other fields using the field (a) in their expressions will be set

Actually RDJ is built on top of DJ

- RDJ expands DJ (DominoJ*)
  - By enabling the automation in method slots*
  - But the idea can be applied to any event system
- A method slot is an object's property
  - A "field" holds an array of function closures



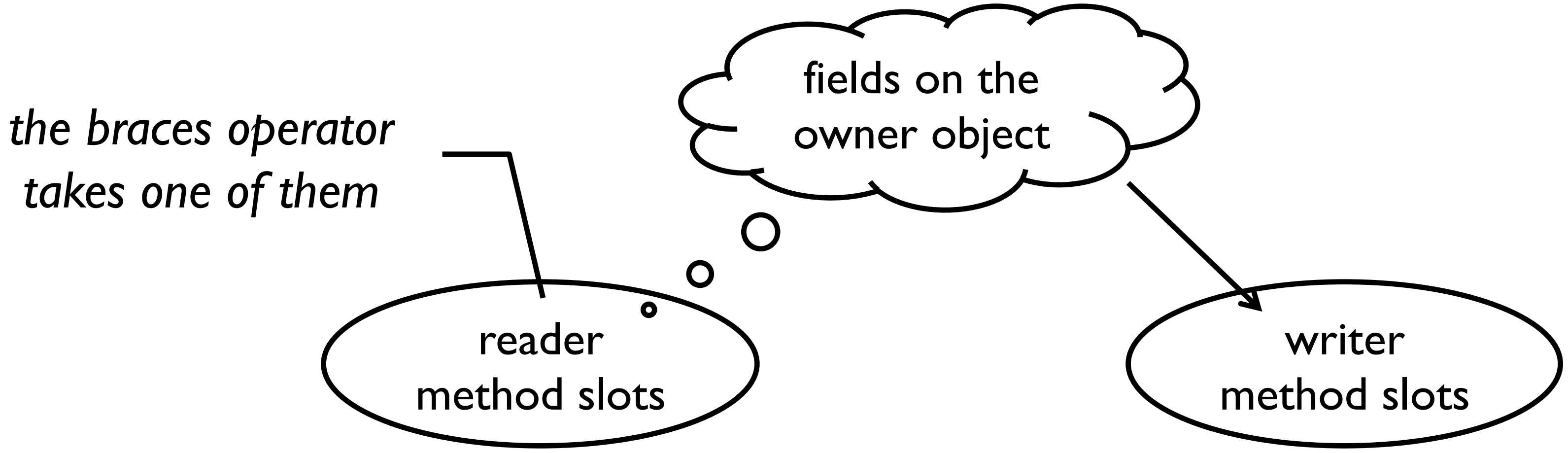
object s : **class** Shape
int x
methodslot setX
…
(void (int nx)) { target.update(nx); } | target = o
(void (int nx)) { this.x = nx; } | this = s

method slots can be used as both events and handlers!

*Our previous work presented at AOSD'13: *Method slots: supporting methods, events, and advices by a single language construct*

How to infer events in OO design

- Finding writers by a given reader through fields
    1. Check all closures in the given method slot
    2. Find out fields read by the closures
    3. Find out the method slots that write these fields
- An extended getter-setter relation
    - Reader-writer

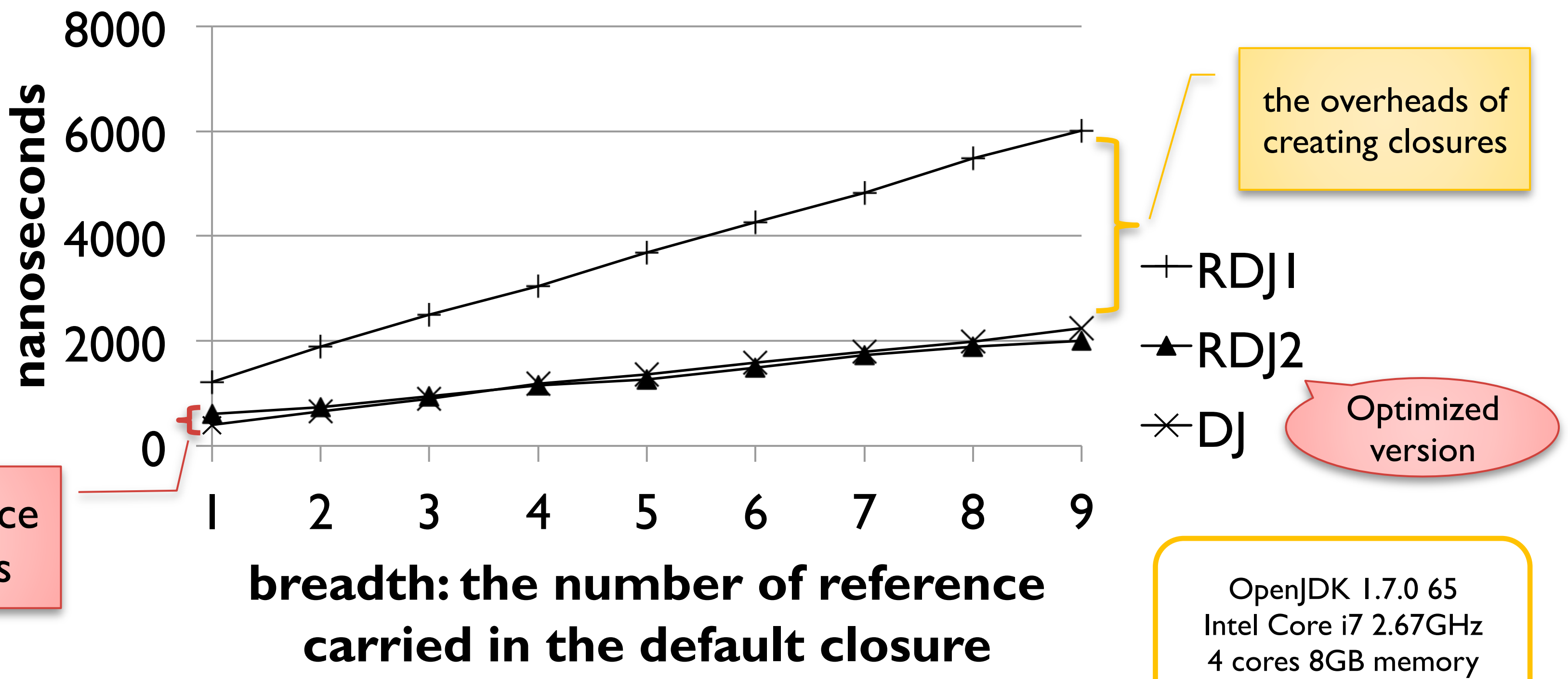

the braces operator takes one of them
fields on the owner object
reader method slots
writer method slots

The inference overheads at runtime are negligible

- Since the automation is boiled down at compile-time
  - But not zero in RDJ since closures can be added at runtime

The performance of DJ, RDJ w/ and w/o optimization

(Execute binding/unbinding with {} one million times to get the average)



the inference overheads

OpenJDK 1.7.0 65
Intel Core i7 2.67GHz
4 cores 8GB memory

The limitations

- Difficult to filter only the events for value change
  - Setting the same value also causes reevaluation
- Propagation loop cannot be totally avoided
  - RDJ only avoids binding a handler to the events in itself

- Other limitations due to DJ
  - Only fields can be translated to signals
  - Only fields are used to infer involved events
  - The usage of the braces operator is not declarative

Related work

- Existing event systems
  - Ptolemy, EventJava, EScala
  - Lack the implicit style
- Using events and signals together
  - Frappé, SuperGlue
  - Only use signals in a limited scope/component
- Library approach
  - Flapjax, Scala.React, Rx.NET
  - Need to manually specify which fields are signals

Conclusions and future directions

- We pointed out the insufficiency in existing event systems
  - By analyzing signals from the viewpoint of events
- Expanding event systems to support the implicit style in reactive programming
  - Show the feasibility by a prototype implementation

- Future directions
  - Make the automation more declarative in RDJ?
  - React at object-level?