Formalizing an Object-Oriented Programming Language with Delimited Control

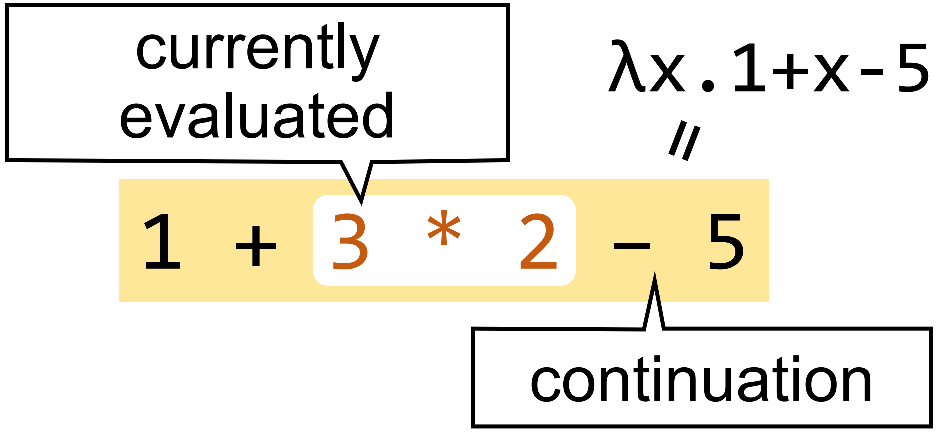
Akane Taniguchi, Youyou Cong, Hidehiko Masuhara

Tokyo Institute of Technology (→ Institute of Science Tokyo)

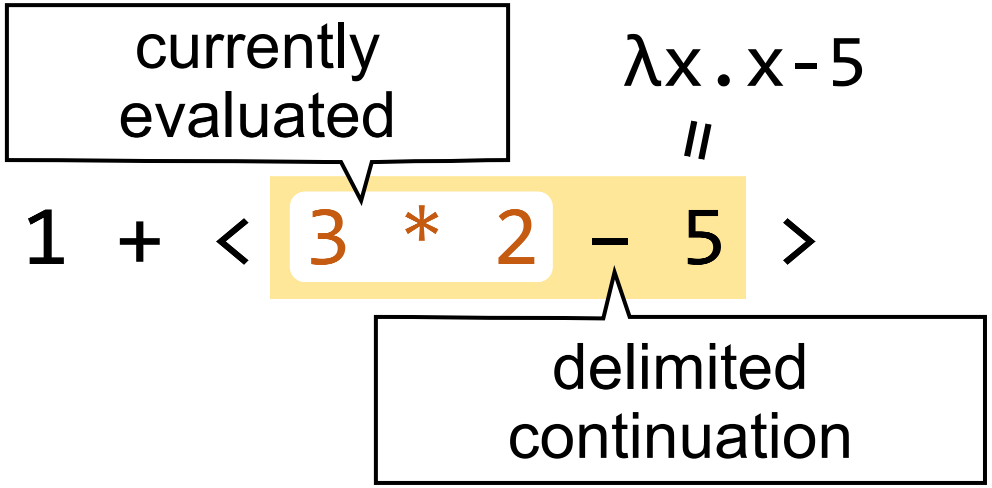
Background

Continuations and Delimited Continuations

- Continuation



- Delimited continuation (DC)



Background

Expressiveness of First-Class DC

- Exception
- Non-determinism
- State
- Generators/iterators
- Futures/promises
- Async/await

Implementation of DC

FP

Racket

OCaml

Haskell

Koka

Links

Effekt

Non-FP

Java: Project Loom

Ruby

Formalization of DC

FP

- Felleisen '88
- Danvy and Filinski '89
- Gunter et. al. '95
- Plotkin & Power '03
- Asai & Kameyama '07
- Materzok & Biernacki '11

⋮

Non-FP

- Inostroza & Storm '18

second-class continuations

Motivation of Formalizing OOPL + DC

- Relation to Java's checked exceptions

```
Int div(Int x) throws Exception {
    // code may cause exception
            ⋮
}
```

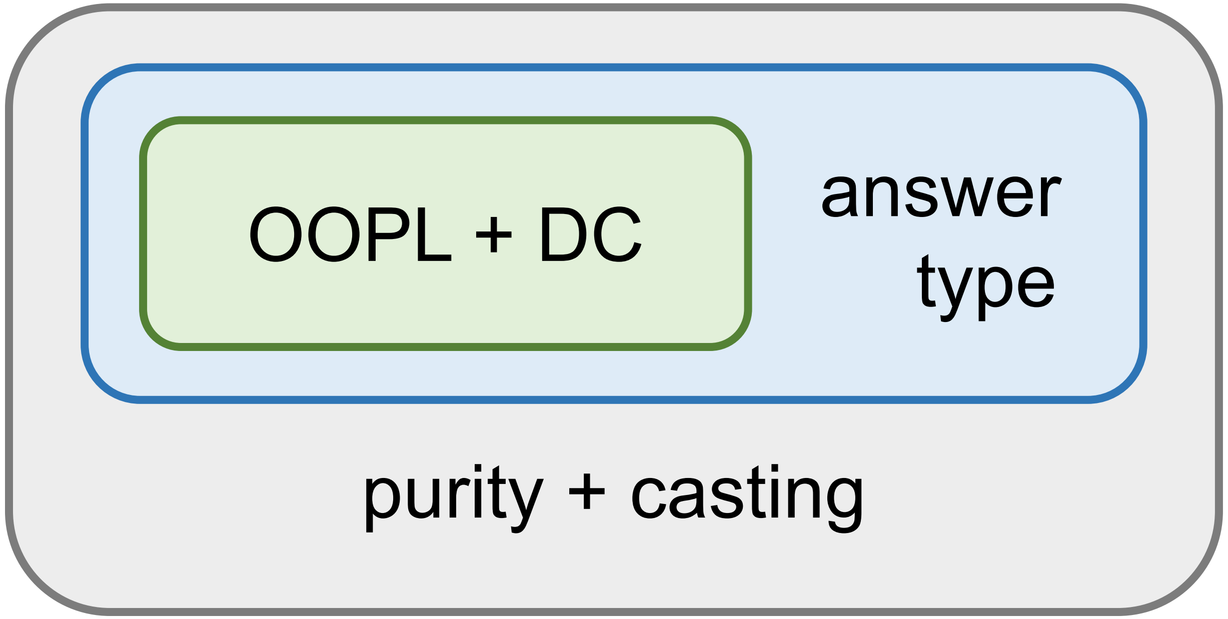
- Combination with bounded polymorphism

This Work: Formalization of OOPL with DC

`shift/reset`[Danvy&Filinski '89]

Approach:

Follow existing formalizations of FPL + DC

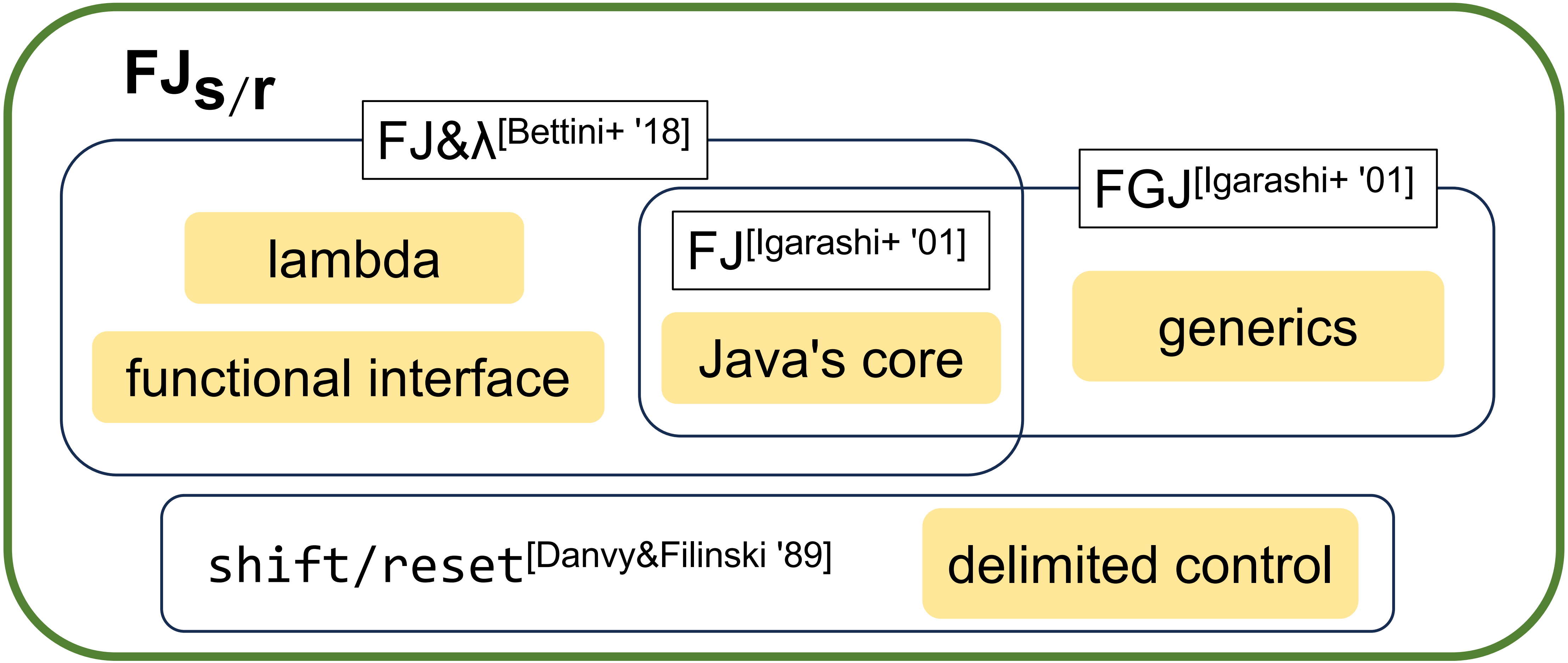


Current Status:

- Formalization done
- Soundness proof done

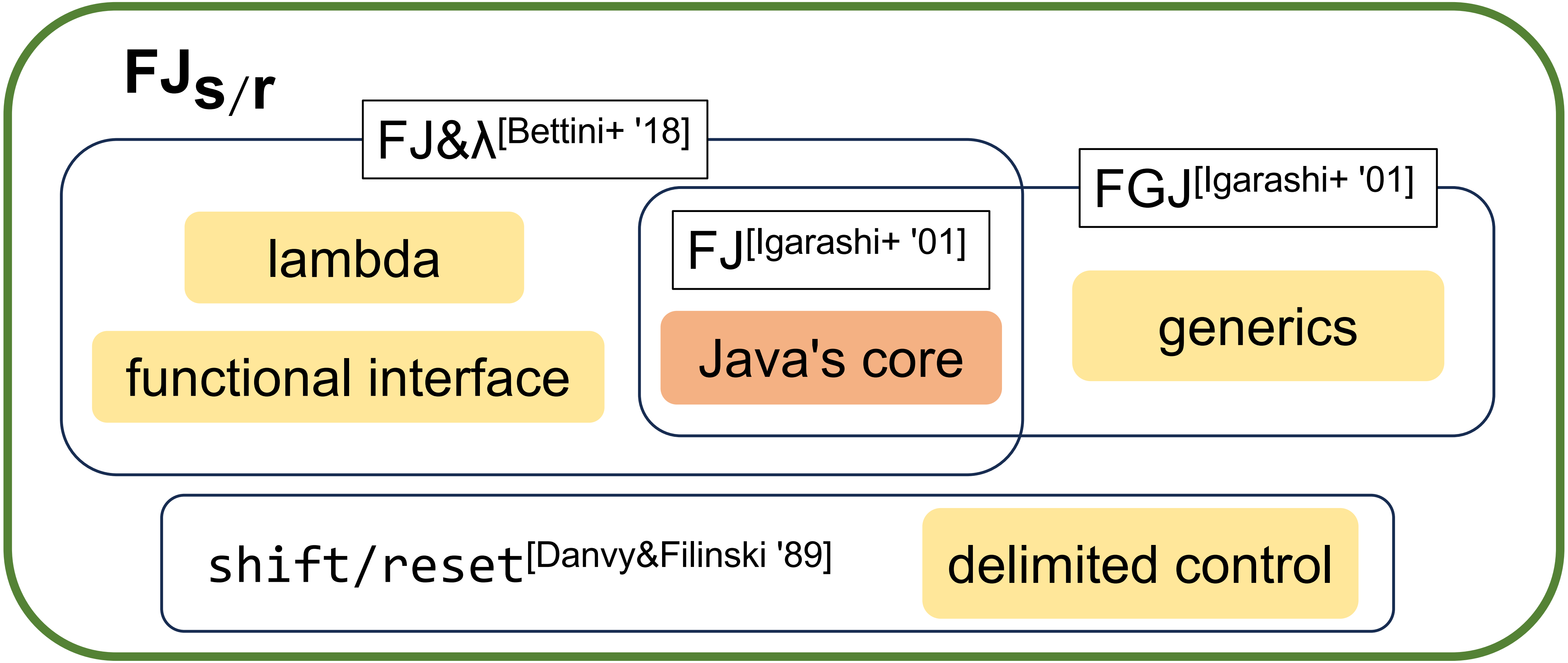
Formalization

$FJ_{s/r}$: an OOP Language with Shift/Reset



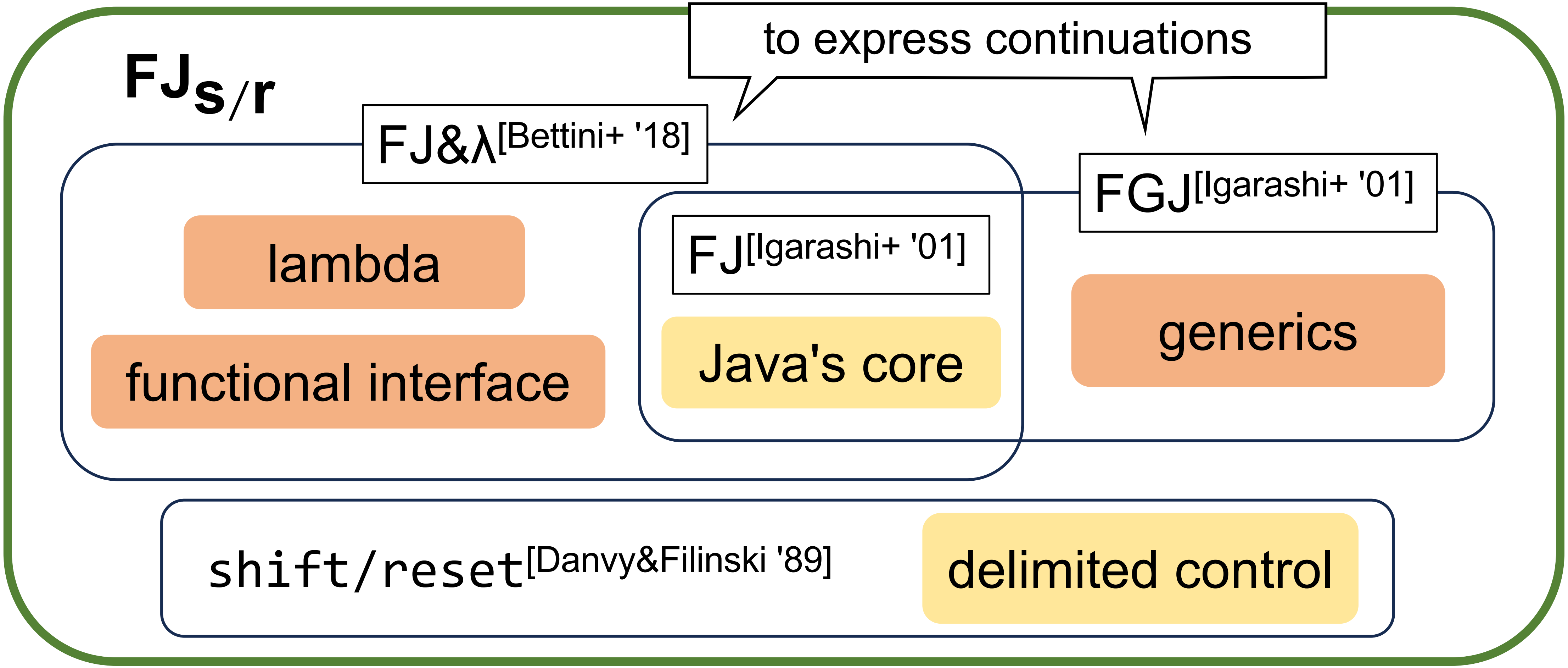
Formalization

$FJ_{s/r}$: an OOP Language with Shift/Reset



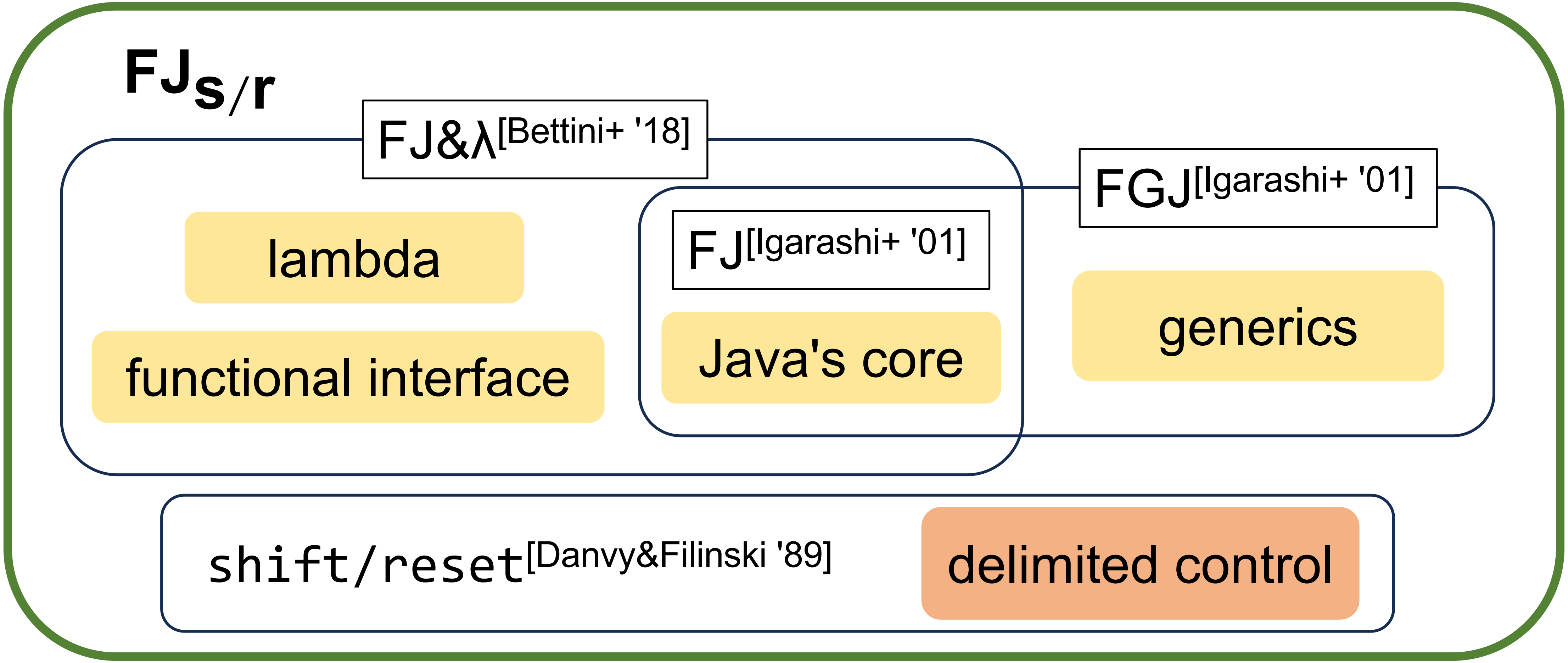
Formalization

$FJ_{s/r}$: an OOP Language with Shift/Reset



Formalization

$FJ_{s/r}$: an OOP Language with Shift/Reset



Formalization

Continuations in FP and OOP

FP

```
⟨ if (Sk. k true + k false)
  then 1
  else -1 ⟩
```

$FJ_{s/r}$ (OOP)

```
⟨ if (Sk<Bool,Int>.
        k.apply(true)
        + k.apply(false))
  then 1
  else -1 ⟩
```

Formalization

Continuations in FP and OOP

FP

```
⟨ if (Sk. k true + k false)
  then 1
  else -1 ⟩
```

$FJ_{s/r}$ (OOP)

```
⟨ if (Sk<Bool,Int>.
       k.apply(true)
       + k.apply(false))
  then 1
  else -1 ⟩
```

reset : delimit continuation

Continuations in FP and OOP

FP

```
⟨ if (Sk. k true + k false)
  then 1
  else -1 ⟩
```

$FJ_{s/r}$ (OOP)

```
⟨ if (Sk<Bool,Int>.
     k.apply(true)
     + k.apply(false))
  then 1
  else -1 ⟩
```

shift : capture continuation

Continuations in FP and OOP

FP

```
⟨ if (Sk. k true + k false)
  then 1
  else -1 ⟩
```

$FJ_{s/r}$ (OOP)

```
⟨ if (Sk<Bool,Int>.
      k.apply(true)
      + k.apply(false))
  then 1
  else -1 ⟩
```

input and output types of continuation

Continuations in FP and OOP

FP

```
⟨ if (Sk. k true + k false)
  then 1
  else -1                      ⟩
                   =
  λx. ⟨ if x then 1 else -1 ⟩
```

lambda expressions needs a type annotation

$FJ_{s/r}$ (OOP)

```
⟨ if (Sk<Bool,Int>.
       k.apply(true)
       + k.apply(false))
  then 1
  else -1                 ⟩
             =
( x → ⟨ if x then 1
        else -1 ⟩ )Fun<Bool,Int,Int>
```

Formalization

Continuations in FP and OOP

FP

```
⟨ if (Sk. k true + k false)
  then 1
  else -1 ⟩
```

continuation call

$FJ_{s/r}$ (OOP)

```
⟨ if (Sk<Bool,Int>.
        k.apply(true)
        + k.apply(false))
  then 1
  else -1 ⟩
```

Continuations in FP and OOP

FP

```
⟨ if (Sk. k true + k false)
  then 1
  else -1 ⟩

⟹ 0
```

$FJ_{s/r}$ (OOP)

```
⟨ if (Sk<Bool,Int>.
        k.apply(true)
        + k.apply(false))
  then 1
  else -1 ⟩

⟹ 0
```

Formalization

Syntax

- Declarations

```
class Pair<X extends Object> {
  X fst; X snd;
  Pair<X>@Object setfst(X newfst){
    return new Pair<X>(newfst,this.snd);
  }
}
interface Fun<X extends Object,
              Y extends Object,
              Z extends Object> {
  Y@Z apply (X x);
}
```

- Expressions

```
e ::= x
      new C<T̄>(ē)
      e.f
      e.m<T̄>(ē)
      (x→e)
      (x→e)^I<T,T,T>
      Sk<T,T>.e
      ⟨e⟩
```

Formalization

Syntax

- Declarations

```
class Pair<X extends Object> {
  X fst; X snd;
  Pair<X>@Object setfst(X newfst){
    return new Pair<X>(newfst,this.snd);
  }
}
interface Fun<X extends Object,
              Y extends Object,
              Z extends Object> {
  Y@Z apply (X x);
}
```

- Expressions

```
e ::= x
      new C<T̄>(ē)
      e.f
      e.m<T̄>(ē)
      (x→e)
      (x→e)^I<T,T,T>
      Sk<T,T>.e
      ⟨e⟩
```

Formalization

Syntax

- Declarations

```
class Pair<X extends Object> {
  X fst; X snd;
  Pair<X>@Object setfst(X newfst){
    return new Pair<X>(newfst,this.snd);
  }
}
interface Fun<X extends Object,
              Y extends Object,
              Z extends Object> {
  Y@Z apply (X x);
}
```

- Expressions

```
e ::= x
      new C<T̄>(ē)
      e.f
      e.m<T̄>(ē)
      (x→e)
      (x→e)^I<T,T,T>
      Sk<T,T>.e
      ⟨e⟩
```

Formalization

Syntax

- Declarations

```
class Pair<X extends Object> {
  X fst; X snd;
  Pair<X>@Object setfst(X newfst){
    return new Pair<X>(newfst,this.snd);
  }
}
interface Fun<X extends Object,
              Y extends Object,
              Z extends Object> {
  Y@Z apply (X x);
}
```

- Expressions

Java's core

```
e ::= x
      new C<T̄>(ē)
      e.f
      e.m<T̄>(ē)
      (x→e)
      (x→e)^I<T,T,T>
      Sk<T,T>.e
      ⟨e⟩
```

Formalization

Syntax

- Declarations

```
class Pair<X extends Object> {
  X fst; X snd;
  Pair<X>@Object setfst(X newfst){
    return new Pair<X>(newfst,this.snd);
  }
}
interface Fun<X extends Object,
              Y extends Object,
              Z extends Object> {
  Y@Z apply (X x);
}
```

- Expressions

```
e ::= x
      new C<T̄>(ē)
      e.f
      e.m<T̄>(ē)
      (x→e)
      (x→e)^I<T,T,T>
      Sk<T,T>.e
      ⟨e⟩
```

lambda

Formalization

Syntax

- Declarations

```
class Pair<X extends Object> {
  X fst; X snd;
  Pair<X>@Object setfst(X newfst){
    return new Pair<X>(newfst,this.snd);
  }
}
interface Fun<X extends Object,
              Y extends Object,
              Z extends Object> {
  Y@Z apply (X x);
}
```

- Expressions

```
e ::= x
      new C<T̄>(ē)
      e.f
      e.m<T̄>(ē)
      (x→e)
      (x→e)^I<T,T,T>
      Sk<T,T>.e
      ⟨e⟩
```

delimited control operators

Formalization

Evaluation

Evaluation of shift expression:

$$\langle \mathtt{F[Sk<S,T>.e]} \rangle \longrightarrow \langle \mathtt{e[k} \mapsto \mathtt{(x} \to \langle \mathtt{F[x]} \rangle \mathtt{)}^{\mathtt{Fun<S,T,U>}} \mathtt{]} \rangle$$

Evaluation

Evaluation of shift expression:

$$\langle F[\texttt{Sk<S,T>.e}] \rangle \longrightarrow \langle \texttt{e}[\texttt{k} \mapsto (\texttt{x} \to \langle F[\texttt{x}] \rangle)^{\texttt{Fun<S,T,U>}}] \rangle$$

e.g., `1 + ⟨ Sk<Int,Int>.2*3 - 5 ⟩`

`F[] = [] - 5`

Evaluation Context:

`F[] ::= [] | F[[].f] | … | F[[].m<T>(e)]`

Formalization

Evaluation

Evaluation of shift expression:

type information

$$\langle \mathtt{F[Sk<S,T>.e]} \rangle \longrightarrow \langle \mathtt{e[k} \mapsto \mathtt{(x} \rightarrow \langle \mathtt{F[x]} \rangle \mathtt{)}^{\mathtt{Fun<S,T,U>}} \mathtt{]} \rangle$$

captured continuation

Formalization

Type Judgement

environment for expression variables

expression type

$$\Gamma ; \Delta \vdash e : T@U$$

answer type

environment for type variables

evaluation of e requires context returning type **U**

Soundness (Ongoing)

Progress

$\emptyset;\emptyset \vdash e : T@U$

$\Rightarrow$ e is a value or $e \rightarrow e'$ or $e = F[Sk.e]$ for some F

stuck : shift has no matching reset ☹

Preservation

$\Gamma;\Delta \vdash e : T@U$ and $e \rightarrow e'$ $\Rightarrow$ $\Gamma;\Delta \vdash e' : S@U$ for some $S <: T$

Future work

- Introduce "purity"
  - Pure = no effect
  - Requires two type judgments

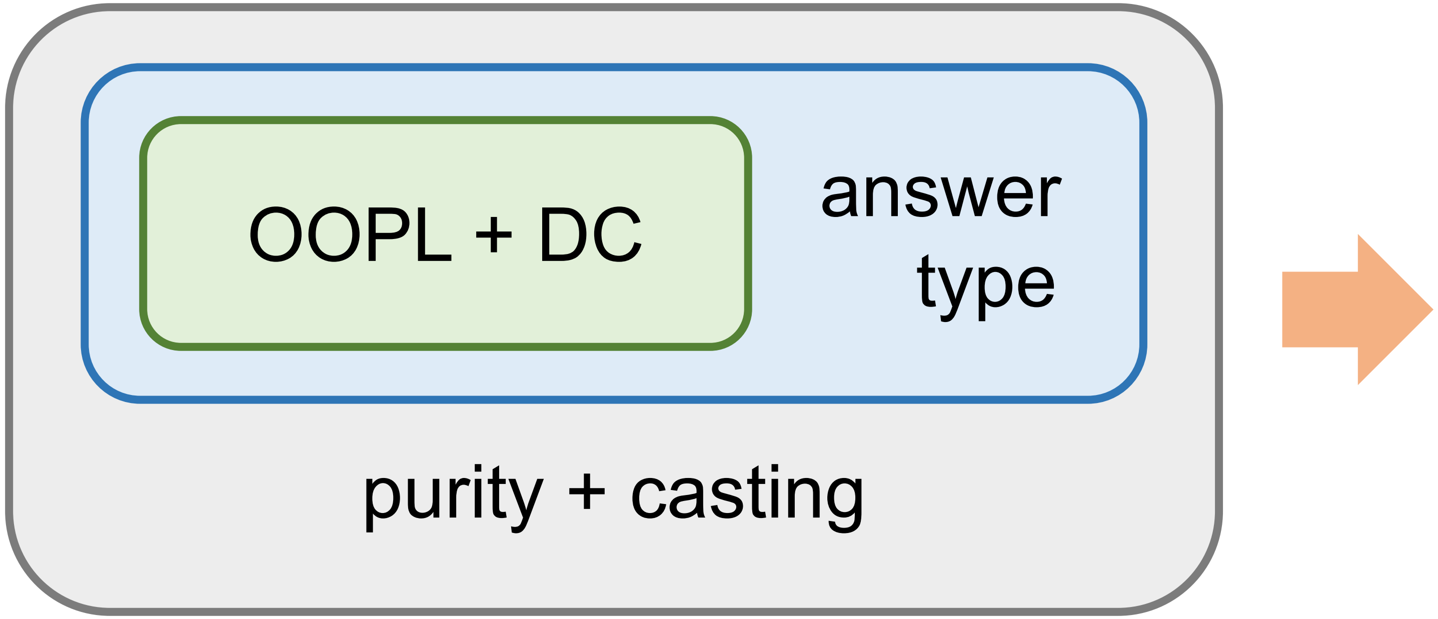
(pure)	$\Gamma;\Delta \vdash_p e:T$
(impure)	$\Gamma;\Delta \vdash e:T@U$

- Add upcasting
  - Convert the type of an expression to its supertype
  - Allow use of pure expressions as impure expressions

    e.g., `if isZero(x) then Sk.0 else 10/x`

Summary

$FJ_{s/r}$: Formalization of an OOPL with shift/reset



Discussion of

- checked exception
- bounded polymorphism

⋮