Effekt

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

MATHEMATISCH-
NATURWISSENSCHAFTLICHE
FAKULTÄT

Effect Handlers for the Masses

Jonathan Immanuel Brachthäuser and **Philipp Schuster** and **Klaus Ostermann**

University of Tübingen, Germany

github.com/b-studios/java-effekt

Overview

1. Introduction & Design Decisions
2. Implementation Details
3. Effect Handlers in Java

Part I

Effect Handlers: Introduction & Library Design

Effect Handlers

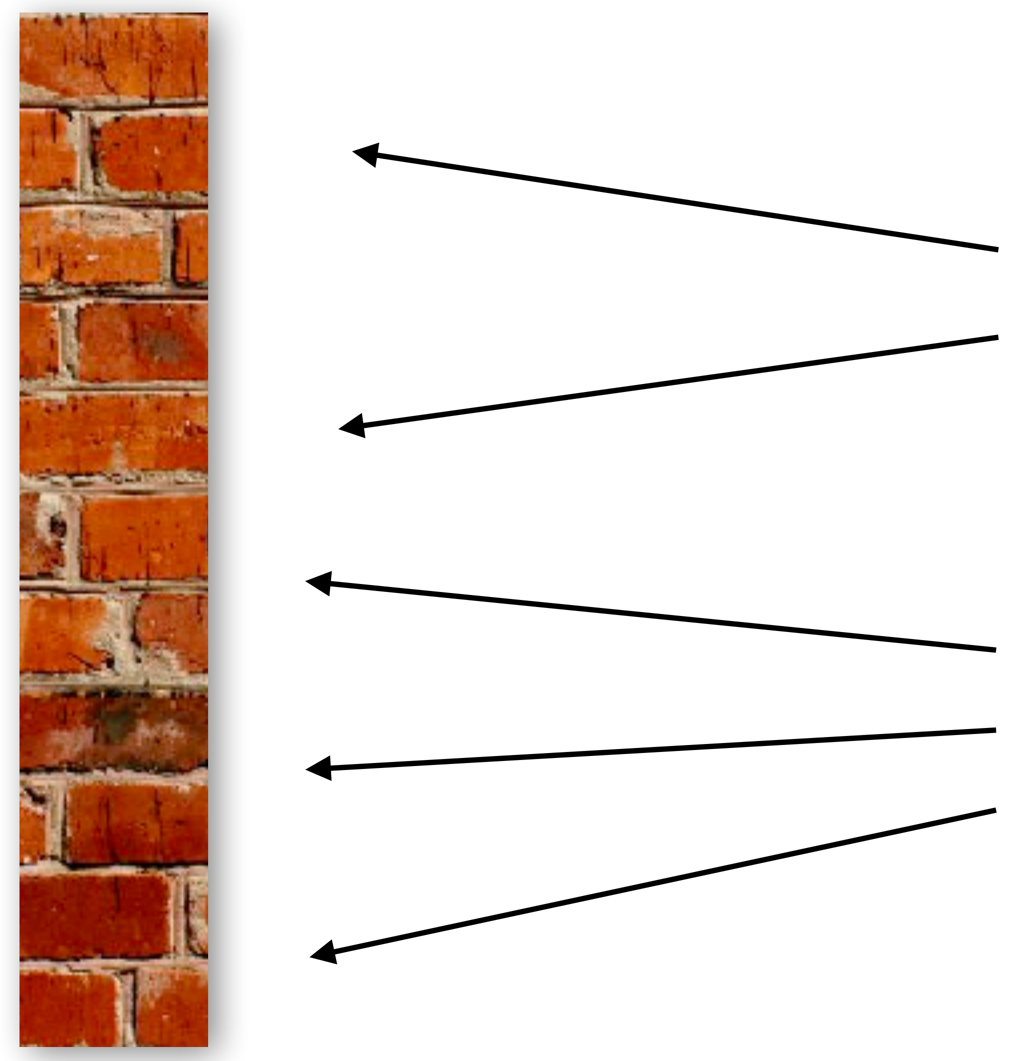
… split effectful programs into three parts / responsibilities:

Effect Signatures

Interfaces, specifying available effect operations

Effect Handlers

… split effectful programs into three parts / responsibilities:



Effectful Program

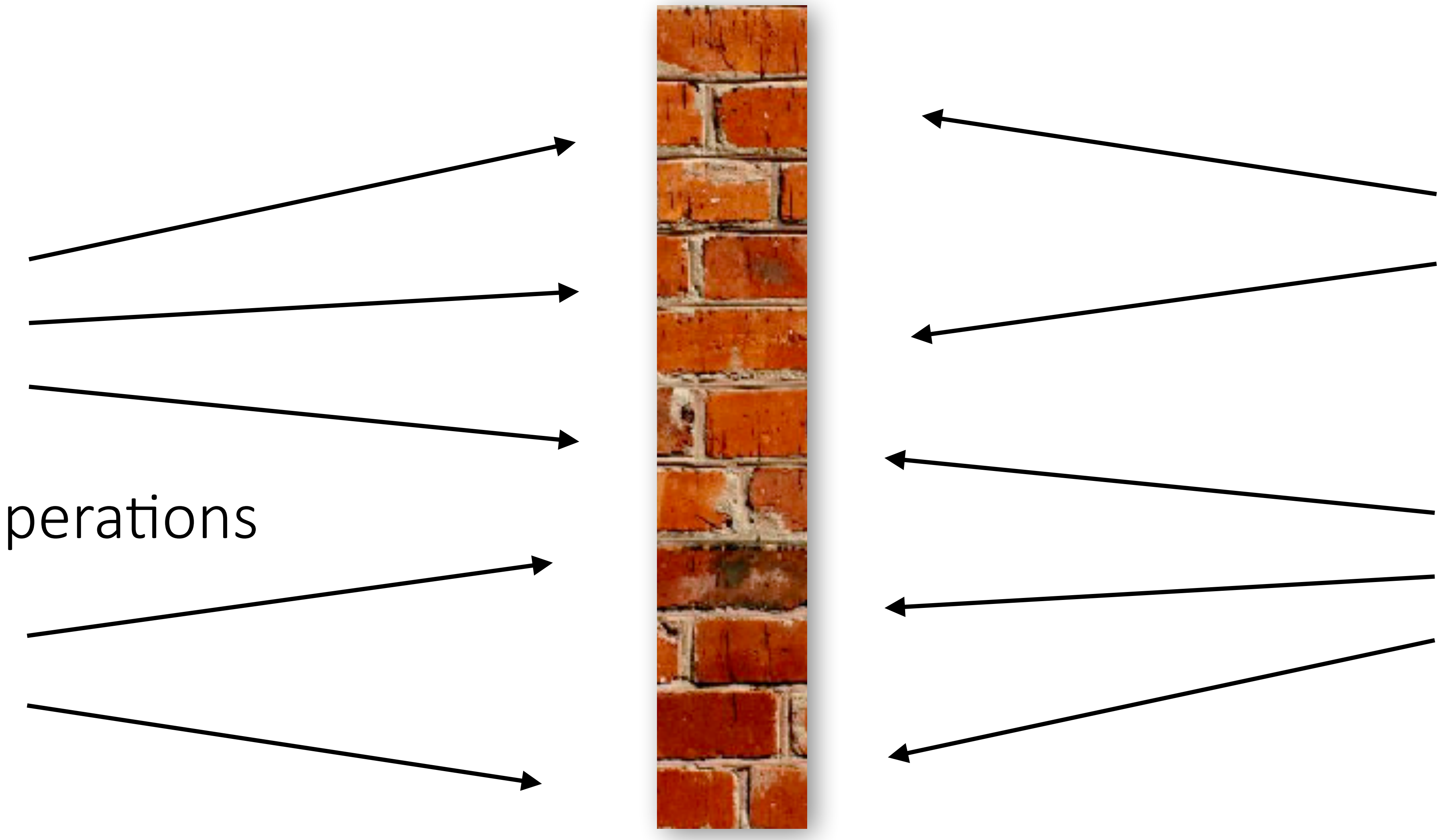
Using effect operations

Effect Signatures

Interfaces, specifying available effect operations

Effect Handlers

… split effectful programs into three parts / responsibilities:



Effect Handlers

Giving semantics to effect operations

Effectful Program

Using effect operations

Effect Signatures

Interfaces, specifying available effect operations

Effect Handlers for Java

- Effect handlers can be seen as **structured programming** with delimited continuations:

- Effect handlers support many use cases of delim. cont. but with **simplified typing**

Effect Handlers for Java

- Effect handlers can be seen as **structured programming** with delimited continuations:

goto	*vs.*	if / for / while
delimited continuations	*vs.*	effect handlers

- Effect handlers support many use cases of delim. cont. but with **simplified typing**

Effect Handlers for Java

- Effect handlers can be seen as **structured programming** with delimited continuations:

goto	*vs.*	if / for / while
delimited continuations	*vs.*	effect handlers

- Effect handlers support many use cases of delim. cont. but with **simplified typing**

Contributions

- The first library design for effect handlers **in Java**
- Our effect handler library only requires simple generics
- An implementation of multi-prompt delim. continuations in Java
- A type-selective bytecode transformation using closures

Example: Drunk Coin Flipping

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  if (amb.flip()) {
    return exc.raise("too drunk");
  } else {
    return amb.flip() ? "heads" : "tails";
  }
}
```

Example: Drunk Coin Flipping

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  if (amb.flip()) {
    return exc.raise("too drunk");
  } else {
    return amb.flip() ? "heads" : "tails";
  }
}
```

Example: Drunk Coin Flipping

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  if (amb.flip()) {
    return exc.raise("too drunk");
  } else {
    return amb.flip() ? "heads" : "tails";
  }
}
```

Effect Operations

Semantics of the operations is left open

Example: Drunk Coin Flipping

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  if (amb.flip()) {
    return exc.raise("too drunk");
  } else {
    return amb.flip() ? "heads" : "tails";
  }
}
```

github.com/b-studios/java-effekt

Example: Drunk Coin Flipping

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  if (amb.flip()) {
    return exc.raise("too drunk");
  } else {
    return amb.flip() ? "heads" : "tails";
  }
}
```

Effect Capabilities

Entitles the function to use these effects

Example: Drunk Coin Flipping

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  if (amb.flip()) {
    return exc.raise("too drunk");
  } else {
    return amb.flip() ? "heads" : "tails";
  }
}
```

Effect Capabilities

Entitles the function to use these effects

Marker Exception

Communicates the usage of effects

Example: Drunk Coin Flipping

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  if (amb.flip()) {
    return exc.raise("too drunk");
  } else {
    return amb.flip() ? "heads" : "tails";
  }
}

interface Exc {
  <A> A raise(String msg) throws Effects;
}
interface Amb {
  boolean flip() throws Effects;
}
```

Effect Signatures

Declare and group effect operations

Example: Drunk Coin Flipping

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  if (amb.flip()) {
    return exc.raise("too drunk");
  } else {
    return amb.flip() ? "heads" : "tails";
  }
}

class Native implements Exc {
  <A> A raise(String msg) throws Effects { throw new NativeExc(msg); }
}
class Random implements Amb {
  boolean flip() throws Effects { return Math.random > 0.5; }
}
```

Example: Drunk Coin Flipping

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  if (amb.flip()) {
    return exc.raise("too drunk");
  } else {
    return amb.flip() ? "heads" : "tails";
  }
}
```

```
drunkFlip(new Native(), new Random())
```

```
class Native implements Exc {
  <A> A raise(String msg) throws Effects { throw new NativeExc(msg); }
}
class Random implements Amb {
  boolean flip() throws Effects { return Math.random > 0.5; }
}
```

Effect Handlers

```
class Maybe<R>   implements Exc, Handler<R, Optional<R>> { ... }
class Collect<R> implements Amb, Handler<R, List<R>>     { ... }
```

Effect Handlers

```
class Maybe<R>   implements Exc, Handler<R, Optional<R>> { ... }
class Collect<R> implements Amb, Handler<R, List<R>>     { ... }
```

Effect Handlers

```
class Maybe<R>   implements Exc, Handler<R, Optional<R>> { ... }
class Collect<R> implements Amb, Handler<R, List<R>>     { ... }
```

Original Result Type

Effect Domain

Effect Handlers

```
class Maybe<R>   implements Exc, Handler<R, Optional<R>> { ... }
class Collect<R> implements Amb, Handler<R, List<R>>     { ... }
```

```
    drunkFlip(???, ???) : String
```

Effect Handlers

```
class Maybe<R>   implements Exc, Handler<R, Optional<R>> { ... }
class Collect<R> implements Amb, Handler<R, List<R>>     { ... }
```

```
handle(new Maybe<String>(), exc ->
  drunkFlip(???, exc) : String
) :  Optional<String>
```

Effect Handlers

```
class Maybe<R>   implements Exc, Handler<R, Optional<R>> { ... }
class Collect<R> implements Amb, Handler<R, List<R>>     { ... }



handle(new Collect<Optional<String>>(), amb ->
  handle(new Maybe<String>(), exc ->
    drunkFlip(amb, exc) : String
  ) :  Optional<String>
) : List<Optional<String>>
```

Effect Handlers

```
class Maybe<R>   implements Exc, Handler<R, Optional<R>> { ... }
class Collect<R> implements Amb, Handler<R, List<R>>     { ... }
```

```
handle(new Collect<Optional<String>>(), amb ->
  handle(new Maybe<String>(), exc ->
    drunkFlip(amb, exc) : String
  ) :  Optional<String>
) : List<Optional<String>>
```

```
res> [Optional["heads"], Optional["tails"], Optional.empty]
```

Effect Handlers

```
class Maybe<R>   implements Exc, Handler<R, Optional<R>> { ... }
class Collect<R> implements Amb, Handler<R, List<R>>     { ... }
```

```
handle(new Maybe<List<String>>(), exc ->
  handle(new Collect<String>(), amb ->
    drunkFlip(amb, exc) : String
  ) :  List<String>
) : Optional<List<String>>
```

Effect Handlers

```
class Maybe<R>   implements Exc, Handler<R, Optional<R>> { ... }
class Collect<R> implements Amb, Handler<R, List<R>>     { ... }
```

```
handle(new Maybe<List<String>>(), exc ->
  handle(new Collect<String>(), amb ->
    drunkFlip(amb, exc) : String
  ) :  List<String>
) : Optional<List<String>>
```

```
res> Optional.empty
```

Effect Handlers

```
class Maybe<R>   implements Exc, Handler<R, Optional<R>> { ... }
class Collect<R> implements Amb, Handler<R, List<R>>     { ... }



handle(new Maybe<List<String>>(), exc ->
  handle(new Collect<String>(), amb ->
    drunkFlip(amb, exc) : String
  ) :  List<String>
) : Optional<List<String>>
```

Handlers provide local capabilities

```
res> Optional.empty
```

Effect Handler Implementations

```
class Collect<R> implements Amb, Handler<R, List<R>> {
  List<R> pure(R r) { return Lists.singleton(r); }
  boolean flip() throws Effects {
    return use(k ->
      Lists.concat(k.resume(true), k.resume(false))
    );
  }
}
```

Effect Handler Implementations

```
class Collect<R> implements Amb, Handler<R, List<R>> {
  List<R> pure(R r) { return Lists.singleton(r); }
  boolean flip() throws Effects {
    return use(k ->
      Lists.concat(k.resume(true), k.resume(false))
    );
  }
}
```

Effect Handler Implementations

```
class Collect<R> implements Amb, Handler<R, List<R>> {
  List<R> pure(R r) { return Lists.singleton(r); }
  boolean flip() throws Effects {
    return use(k ->
      Lists.concat(k.resume(true), k.resume(false))
    );
  }
}
```

Effect Handler Implementations

```
class Collect<R> implements Amb, Handler<R, List<R>> {
  List<R> pure(R r) { return Lists.singleton(r); }
  boolean flip() throws Effects {
    return use(k ->
      Lists.concat(k.resume(true), k.resume(false))
    );
  }
}
```

Effect Handler Implementations

```
class Collect<R> implements Amb, Handler<R, List<R>> {
  List<R> pure(R r) { return Lists.singleton(r); }
  boolean flip() throws Effects {
    return use(k ->
      Lists.concat(k.resume(true), k.resume(false))
    );
  }
}

Optional<List<String>> res =
  handle(new Maybe<>(), exc ->
    handle(new Collect<>(), amb ->
      amb.flip() ? "heads" : "tails"
    )
  );
```

Effect Handler Implementations

```
class Collect<R> implements Amb, Handler<R, List<R>> {
  List<R> pure(R r) { return Lists.singleton(r); }
  boolean flip() throws Effects {
    return use(k ->
      Lists.concat(k.resume(true), k.resume(false))
    );
  }
}

Optional<List<String>> res =
  handle(new Maybe<>(), exc ->
    handle(new Collect<>(), amb ->
      □ ? "heads" : "tails"
    )
  );
```

= k

Effect Handler Implementations

```
class Collect<R> implements Amb, Handler<R, List<R>> {
  List<R> pure(R r) { return Lists.singleton(r); }
  boolean flip() throws Effects {
    return use(k ->
      Lists.concat(k.resume(true), k.resume(false))
    );
  }
}
```

```
    handle(new Collect<>(), amb ->
      □ ? "heads" : "tails"
    )
```

```
= k.resume(true)
```

Effect Handler Implementations

```
class Collect<R> implements Amb, Handler<R, List<R>> {
  List<R> pure(R r) { return Lists.singleton(r); }
  boolean flip() throws Effects {
    return use(k ->
      Lists.concat(k.resume(true), k.resume(false))
    );
  }
}
```

```
    handle(new Collect<>(), amb ->
          true   ? "heads" : "tails"
    )
```

```
= k.resume(true)
```

Effect Handler Implementations

```
class Collect<R> implements Amb, Handler<R, List<R>> {
  List<R> pure(R r) { return Lists.singleton(r); }
  boolean flip() throws Effects {
    return use(k ->
      Lists.concat(k.resume(true), k.resume(false))
    );
  }
}
```

```
    handle(new Collect<>(), amb ->
          "heads"
    )
```

```
= k.resume(true)
```

Effect Handler Implementations

```
class Collect<R> implements Amb, Handler<R, List<R>> {
  List<R> pure(R r) { return Lists.singleton(r); }
  boolean flip() throws Effects {
    return use(k ->
      Lists.concat(k.resume(true), k.resume(false))
    );
  }
}
```

```
handle(new Collect<>(), amb ->
     "heads"
)
```

```
= k.resume(true)
```

Effect Handler Implementations

```
class Collect<R> implements Amb, Handler<R, List<R>> {
  List<R> pure(R r) { return Lists.singleton(r); }
  boolean flip() throws Effects {
    return use(k ->
      Lists.concat(k.resume(true), k.resume(false))
    );
  }
}
```

```
Lists.singleton("heads")      = k.resume(true)
```

The Design of the Effekt Library

Effect Signatures

Interfaces, specifying available effect operations

→

Java Interfaces

Marking effectful methods with a special exception

The Design of the Effekt Library

Effect Signatures → **Java Interfaces**

Interfaces, specifying available effect operations → Marking effectful methods with a special exception

Effectful Program → **Java Method**

Using effect operations → Parametrized over effect handler instances / capabilities

The Design of the Effekt Library

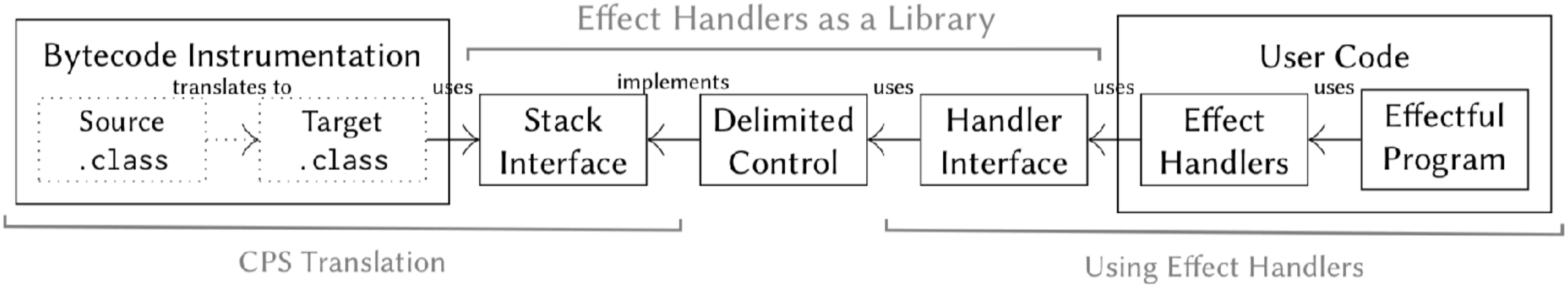
Effect Signatures Interfaces, specifying available effect operations	→	Java Interfaces Marking effectful methods with a special exception
Effectful Program Using effect operations	→	Java Method Parametrized over effect handler instances / capabilities
Effect Handlers Giving semantics to effect operations	→	Java Classes Implementing the effect signatures, potentially using control effects / delimited continuations

github.com/b-studios/java-effekt

Part II

Implementing
Effect Handlers &
Delimited Control

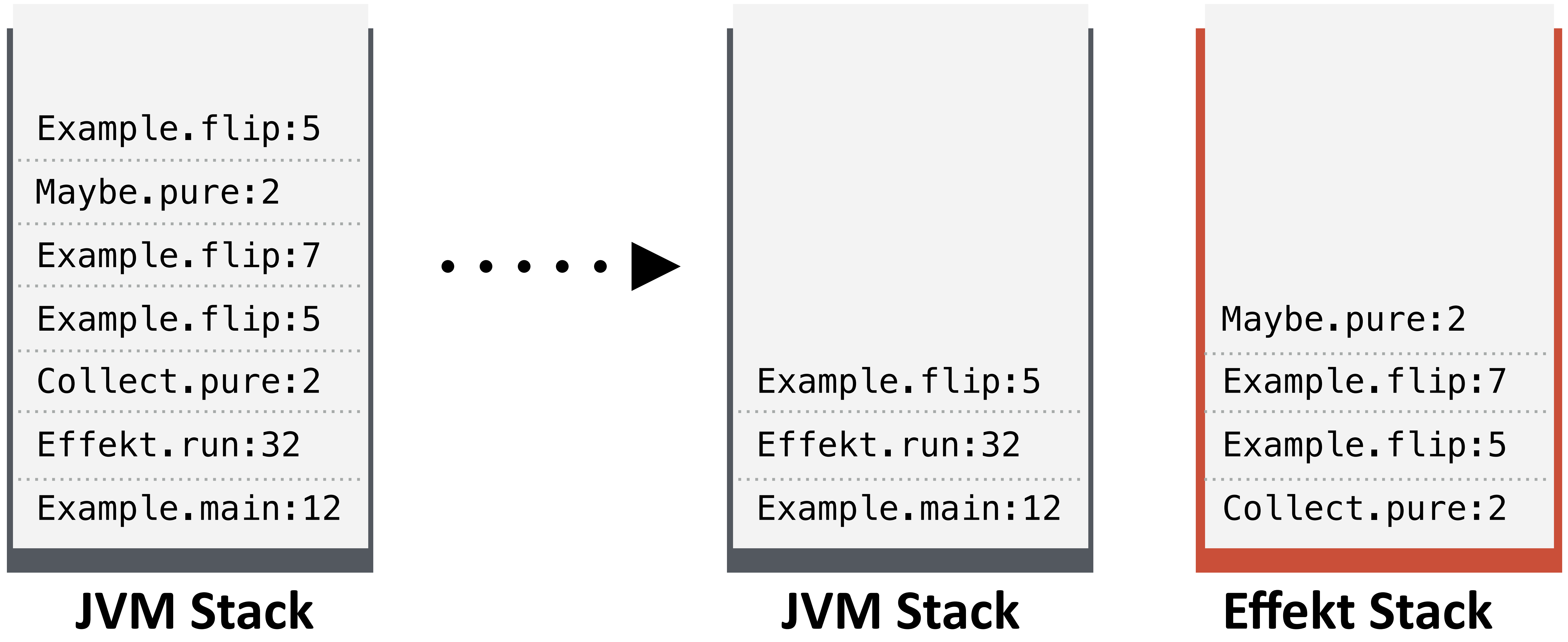
Architectural Overview of Java Effekt



- Programs are written in direct style, but CPS translated via **bytecode transformation**
- Translated programs use a **separate Stack** interface for effectful frames
- **Delimited control** is implemented as a library, implementing the Stack interface
- **Restriction**: Translation preserves signatures, we only transform method bodies

Replacing the JVM Stack

For effectful methods, we maintain our own custom stack, which allows us to manipulate it (searching, slicing, copying).



```
Example.flip:5
Maybe.pure:2
Example.flip:7
Example.flip:5
Collect.pure:2
Effekt.run:32
Example.main:12
```

JVM Stack

```
Example.flip:5
Effekt.run:32
Example.main:12
```

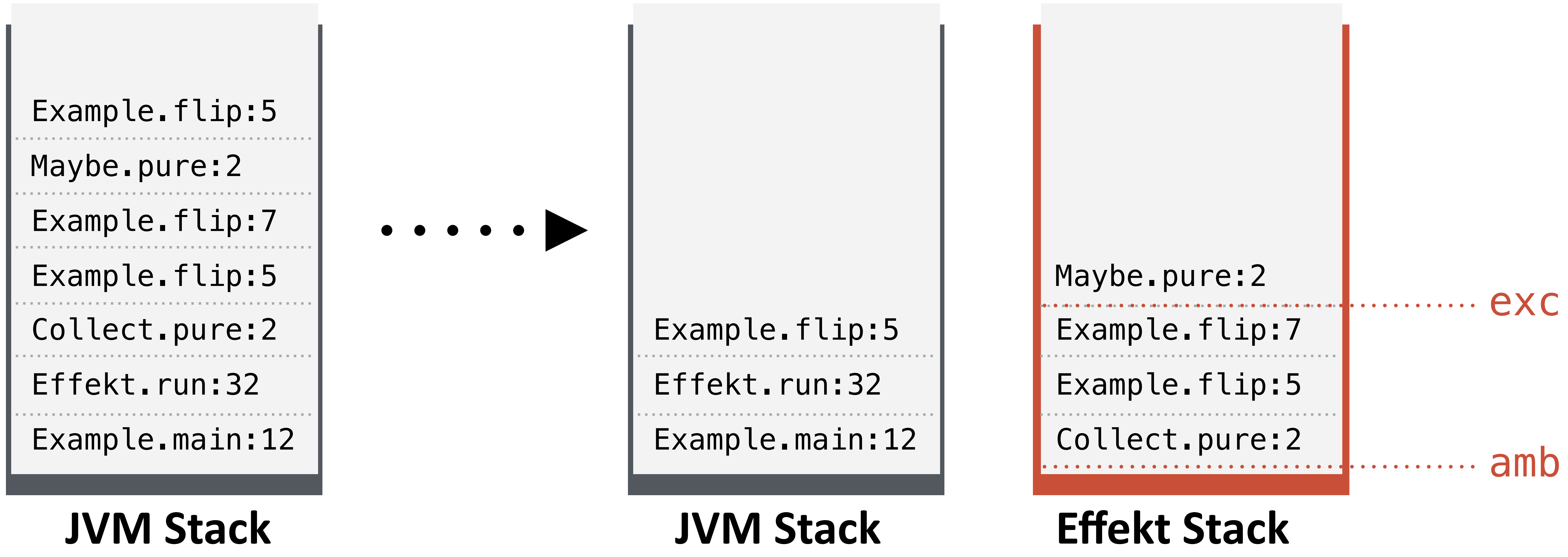
JVM Stack

```
Maybe.pure:2
Example.flip:7
Example.flip:5
Collect.pure:2
```

Effekt Stack

Replacing the JVM Stack

For effectful methods, we maintain our own custom stack, which allows us to manipulate it (searching, slicing, copying).



CPS Translation

- Transformation of bytecode
- Uses a separate stack (`Effekt.push(frame)`)
- Uses its own calling convention (`Effekt.returnWith(result)`, `Effekt.result()`)
- Preserves signatures

CPS Translation

- Transformation of bytecode
- Uses a separate stack (`Effekt.push(frame)`)
- Uses its own calling convention (`Effekt.returnWith(result)`, `Effekt.result()`)
- Preserves signatures

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  boolean dropped = amb.flip();
  if (dropped) {
    return exc.raise("too drunk");
  } else {
    return amb.flip() ? "heads" : "tails";
  }
}
```

CPS Translation

```java
String drunkFlip(Amb amb, Exc exc) throws Effects {
  Effekt.push(() -> drunkFlip₁(amb, exc));
  amb.flip();
  return null;
}
```

```java
String drunkFlip(Amb amb, Exc exc) throws Effects {
  boolean dropped = amb.flip();
  if (dropped) {
    return exc.raise("too drunk");
  } else {
    return amb.flip() ? "heads" : "tails";
  }
}
```

CPS Translation

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  boolean dropped = amb.flip();
  if (dropped) {
    return exc.raise("too drunk");
  } else {
    return amb.flip() ? "heads" : "tails";
  }
}
```

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  Effekt.push(() -> drunkFlip1(amb, exc));
  amb.flip();
  return null;
}

static void drunkFlip1(Amb amb, Exc exc) throws Effects {
  boolean dropped = Effekt.result();
  if (dropped) { exc.raise("too drunk"); }
  else {
    Effekt.push(() -> drunkFlip2(amb, exc, dropped));
    amb.flip();
  }
}
```

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  boolean dropped = amb.flip();
  if (dropped) {
    return exc.raise("too drunk");
  } else {
    return amb.flip() ? "heads" : "tails";
  }
}
```

CPS Translation

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  Effekt.push(() -> drunkFlip₁(amb, exc));
  amb.flip();
  return null;
}

static void drunkFlip₁(Amb amb, Exc exc) throws Effects {
  boolean dropped = Effekt.result();
  if (dropped) { exc.raise("too drunk"); }
  else {
    Effekt.push(() -> drunkFlip₂(amb, exc, dropped));
    amb.flip();
  }
}

static void drunkFlip₂(Amb amb, Exc exc, boolean dropped) throws Effects {
  Effekt.returnWith(Effekt.result() ? "heads" : "tails");
}
```

CPS Translation - Saving Function Local State

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  Effekt.push(() -> drunkFlip1(amb, exc));
  amb.flip();
  return null;
}

static void drunkFlip1(Amb amb, Exc exc) throws Effects {
  boolean dropped = Effekt.result();
  if (dropped) { exc.raise("too drunk"); }
  else {
    Effekt.push(() -> drunkFlip2(amb, exc, dropped));
    amb.flip();
  }
}

static void drunkFlip2(Amb amb, Exc exc, boolean dropped) throws Effects {
  Effekt.returnWith(Effekt.result() ? "heads" : "tails");
}
```

CPS Translation - Saving Function Local State

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  Effekt.push(() -> drunkFlip1(amb, exc));
  amb.flip();
  return null;
}

static void drunkFlip1(Amb amb, Exc exc) throws Effects {
  boolean dropped = Effekt.result();
  if (dropped) { exc.raise("too drunk"); }
  else {
    Effekt.push(() -> drunkFlip2(amb, exc, dropped));
    amb.flip();
  }
}

static void drunkFlip2(Amb amb, Exc exc, boolean dropped) throws Effects {
  Effekt.returnWith(Effekt.result() ? "heads" : "tails");
}
```

CPS Translation - Saving Function Local State

```
String drunkFlip(Amb amb, Exc exc) throws Effects {
  Effekt.push(() -> drunkFlip₁(amb, exc));
  amb.flip();
  return null;
}

static void drunkFlip₁(Amb amb, Exc exc) throws Effects {
  boolean dropped = Effekt.result();
  if (dropped) { exc.raise("too drunk"); }
  else {
    Effekt.push(() -> drunkFlip₂(amb, exc, dropped));
    amb.flip();
  }
}

static void drunkFlip₂(Amb amb, Exc exc, boolean dropped) throws Effects {
  Effekt.returnWith(Effekt.result() ? "heads" : "tails");
}
```

Part III

Modularity through Effect Handlers & Object Orientated Programming

Effect Handlers in Java

Consequences of representing Effect Signatures as Interfaces:

- Signatures can be **mixed** to a desired granularity
- **One handler** can implement **multiple effect signatures** and share the effect domain
- **One effect signature** can be implemented by **multiple handlers** with potentially different effect domains
- Interface subtyping immediately also gives **effect subtyping**

Effect Modularization (Handler Passing)

Using effect signatures

```
interface Exc   { <A> A raise(String msg) throws Effects; }
interface Amb   { boolean flip() throws Effects; }
interface Input { char read() throws Effects; }
```

we can implement parsers

Effect Modularization (Handler Passing)

Using effect signatures

```
interface Exc   { <A> A raise(String msg) throws Effects; }
interface Amb   { boolean flip() throws Effects; }
interface Input { char read() throws Effects; }
```

we can implement parsers

```
void accept(char c, Input in, Exc exc) throws Effects {
  if (in.read() != c) exc.raise("Expected " + c);
}
```

Effect Modularization (Handler Passing)

Using effect signatures

```
interface Exc   { <A> A raise(String msg) throws Effects; }
interface Amb   { boolean flip() throws Effects; }
interface Input { char read() throws Effects; }
```

we can implement parsers

```
void accept(char c, Input in, Exc exc) throws Effects {
  if (in.read() != c) exc.raise("Expected " + c);
}

// <P> ::= 'A' <P> | 'B'
int parse(Input in, Exc exc, Amb amb) throws Effects {
  if (amb.flip()) { accept('A', in, exc); return parse(in, exc, amb) + 1; }
  else            { accept('B', in, exc); return 0; }
}
```

github.com/b-studios/java-effekt

Effect Modularization (Composition)

Using effect signatures

```
interface Exc   { <A> A raise(String msg) throws Effects; }
interface Amb   { boolean flip() throws Effects; }
interface Input { char read() throws Effects; }
interface P extends Exc, Amb, Input {
  default void accept(char c) throws Effects {
    if (this.read() != c) this.raise("Expected " + c);
  }
}
```

Effect Modularization (Composition)

Using effect signatures

```
interface Exc   { <A> A raise(String msg) throws Effects; }
interface Amb   { boolean flip() throws Effects; }
interface Input { char read() throws Effects; }
interface P extends Exc, Amb, Input {
  default void accept(char c) throws Effects {
    if (this.read() != c) this.raise("Expected " + c);
  }
}
```

Now "this" is the capability

Effect Modularization (Composition)

Using effect signatures

```
interface Exc   { <A> A raise(String msg) throws Effects; }
interface Amb   { boolean flip() throws Effects; }
interface Input { char read() throws Effects; }
interface P extends Exc, Amb, Input {
  default void accept(char c) throws Effects {
    if (this.read() != c) this.raise("Expected " + c);
  }
}
```

Now "this" is the capability

```
// <P> ::= 'A' <P> | 'B'
int parse(P p) throws Effects {
  if (p.flip()) { p.accept('A'); return parse(p) + 1; }
  else          { p.accept('B'); return 0; }
}
```

Modularity Benefits

- Effect Signatures just describe the parser **interface**
- Handlers can implement **different parsing strategies**:
  - backtracking vs. enumerating all parse results
  - depth first vs. breadth first
  - pull vs. push
- **Compose with other effects** like ANF transformation / Let-insertion

Conclusions

- "handler- / capability passing style"
- user defined effects ✔
- dynamic effect instances ✔
- modular and extensible effect signatures and handlers ✔
- user programs are written in direct style ✔
- performance: competitive with JVM continuation libraries ✔
- safety (capabilities can leak) ✘

Conclusions

- "handler- / capability passing style"
- user defined effects ✔
- dynamic effect instances ✔
- modular and extensible effect signatures and handlers ✔
- user programs are written in direct style ✔
- performance: competitive with JVM continuation libraries ✔
- safety (capabilities can leak) ✘

Conclusions

- "handler- / capability passing style"
- user defined effects ✔
- dynamic effect instances ✔
- modular and extensible effect signatures and handlers ✔
- user programs are written in direct style ✔
- performance: competitive with JVM continuation libraries ✔
- safety (capabilities can leak) ✘

Thank you!