Algebraic Effects on the JVM

Jonathan Immanuel Brachthäuser
University of Tübingen, Germany

github.com/b-studios/scala-effekt
We developed algebraic effect libraries for

-scala-effekt
-JVM-effekt

Scala Effekt

JVM Effekt
Part I
Effect Handlers as a Library for Scala
Key Specs: Scala Effekt

- shallow embedding vs. deep embedding of handlers
- "capability passing style"
- shallow handlers vs. deep handlers
- user defined effects ✔
- dynamic effect instances ✔
- modular and extensible effect signatures and handlers ✔
- safety (capabilities can leak) ✗
- user programs are written in direct style ✗
- performance: still (orders of magnitude) slower than primitive effects ✗
How to USE Scala Effekt
Example: Drunk Coin Flipping

```scala
val drunkFlip: String using Amb and Exc = for {
  caught ← flip()
  heads ← if (caught) flip() else raise("too drunk")
} yield if (heads) "heads" else "tails"
```
val drunkFlip: String using Amb and Exc = for {
  caught ← flip()
  heads ← if (caught) flip() else raise("too drunk")
} yield if (heads) "heads" else "tails"

Effect Operations
Semantics of the operations is left open
val drunkFlip: String using Amb and Exc = for {
  caught ← flip()
  heads ← if (caught) flip() else raise("too drunk")
} yield if (heads) "heads" else "tails"
val drunkFlip: String using Amb and Exc = for {
  caught ← flip()
  heads ← if (caught) flip() else raise("too drunk")
} yield if (heads) "heads" else "tails"

AmbList { ExcOption { drunkFlip } }
val drunkFlip: String using Amb and Exc = for {
  caught ← flip()
  heads ← if (caught) flip() else raise("too drunk")
} yield if (heads) "heads" else "tails"

AmbList { ExcOption { drunkFlip } }

Effect Handlers
Provide semantics to effect operations
val drunkFlip: String using Amb and Exc = for {
  caught ← flip()
  heads ← if (caught) flip() else raise("too drunk")
} yield if (heads) "heads" else "tails"

AmbList { ExcOption { drunkFlip } }
val drunkFlip: String using Amb and Exc = for {
  caught ← flip()
  heads ← if (caught) flip() else raise("too drunk")
} yield if (heads) "heads" else "tails"

AmbList { ExcOption { drunkFlip } }
  > List(Some("heads"), Some("tails"), None)
val drunkFlip: String using Amb and Exc = for {
  caught ← flip()
  heads ← if (caught) flip() else raise("too drunk")
} yield if (heads) "heads" else "tails"

AmbList { ExcOption { drunkFlip } }
  > List(Some("heads"), Some("tails"), None)

ExcOption { AmbList { drunkFlip } }
val drunkFlip: String using Amb and Exc = for {
  caught ← flip()
  heads ← if (caught) flip() else raise("too drunk")
} yield if (heads) "heads" else "tails"

AmbList { ExcOption { drunkFlip } }
  > List(Some("heads"), Some("tails"), None)

ExcOption { AmbList { drunkFlip } }
  > None
The role of implicit
Design Decisions - Scala Effekt

We were faced with the following three design questions:
Design Decisions - Scala Effekt

We were faced with the following three design questions:

1. how to **capture the context** of effect operations?
   ⇒ monadic implementation for multi-prompt delimited continuations
We were faced with the following three design questions:

1. how to capture the context of effect operations

⇒ monadic implementation for delimited continuations
Design Decisions - Scala Effekt

We were faced with the following three design questions:

1. how to capture the context of effect operations?
   ⟹ monadic implementation for multi-prompt delimited continuations

2. how should effect handlers provide semantics for effect operations?
   ⟹ shallow embedding of effect handlers
Design Decisions - Scala Effekt

We were faced with the following three design questions:

1. how to **capture the context** of effect operations?
   \[\Rightarrow\text{monadic implementation for multi-prompt delimited continuations}\]

2. how should **effect handlers provide semantics** for effect operations?
   \[\Rightarrow\text{shallow embedding of effect handlers}\]

3. how to establish an **effect typing discipline**?
   \[\Rightarrow\text{capability passing style}\]
Design Decisions - Scala Effekt

We were faced with the following three design questions:

1. how to **capture the context** of effect operations?
   ⟹ monadic implementation for multi-prompt delimited continuations

2. how should **effect handlers provide semantics** for effect operations?
   ⟹ shallow embedding of effect handlers

3. how to establish an **effect typing discipline**?
   ⟹ capability passing style

For all three answers **implicit function types** turned out to be a perfect fit!
Implicit Function Types
Implicit Function Types

```scala
def drunkFlip: String using Amb and Exc = for {
  caught ← flip()
  heads ← if (caught) flip() else raise("too drunk")
} yield if (heads) "heads" else "tails"
```
Implicit Function Types

```scala
def drunkFlip: String using Amb and Exc = for {
  caught ← flip()
  heads ← if (caught) flip() else raise("too drunk")
} yield if (heads) "heads" else "tails"
```

Library defined type aliases

```scala
```
Effekt

Implicit Function Types

```
def drunkFlip: String using Amb and Exc = for {
  caught ← flip()
  heads ← if (caught) flip() else raise("too drunk")
} yield if (heads) "heads" else "tails"
```

Library defined type aliases

```

type and[A, E] = implicit Cap[E] ⇒ A
```
Making Capability Passing Explicit

Explicitly desugaring implicit function types gives:

``` scala
def drunkFlip(amb: Cap[Amb], exc: Cap[Exc]): Control[String] = for {
  caught ← amb.handler.flip()
  heads ← if (caught) amb.handler.flip()
    else exc.handler.raise("too drunk")
} yield if (heads) "heads" else "tails"
```
Making Capability Passing Explicit

Explicitly desugaring implicit function types gives:

```scala
def drunkFlip(amb: Cap[Amb], exc: Cap[Exc]): Control[String] = for {
  caught ← amb.handler.flip()
  heads ← if (caught) amb.handler.flip()
    else exc.handler.raise("too drunk")
} yield if (heads) "heads" else "tails"
```
Making Capability Passing Explicit

Explicitly desugaring implicit function types gives:

```scala
def drunkFlip(amb: Cap[Amb], exc: Cap[Exc]): Control[String] = for {
  caught ← amb.handler.flip()
  heads ← if (caught) amb.handler.flip()
  else exc.handler.raise("too drunk")
} yield if (heads) "heads" else "tails"
```

Handler create capabilities:

```
AmbList { amb ⇒ ExcOption { exc ⇒ drunkFlip(amb, exc) } }
```
Capabilities in Effekt

Capabilities \( \text{Cap} [E] \) encapsulate three different things:
Capabilities in Effekt

Capabilities $\text{Cap}[E]$ encapsulate three different things:

1. they contain a unique prompt marker that delimits the scope of the continuation to be captured.
Capabilities in Effekt

Capabilities $\text{Cap}[E]$ encapsulate three different things:

1. they contain a unique prompt marker that *delimits the scope* of the continuation to be captured.
2. they contain the *effect handler implementation* to be passed down (shallow embedding of handlers).
Capabilities in Effekt

Capabilities $\text{Cap}[E]$ encapsulate three different things:

1. they contain a unique prompt marker that delimits the scope of the continuation to be captured.
2. they contain the effect handler implementation to be passed down (shallow embedding of handlers).
3. they entitle the holder of the capability to invoke effectful operations specified in effect signature $E$ (effect typing discipline).
shallow embedding of effect handlers
Calling an Effect Operation
Calling an Effect Operation

We can think of effect operations as uninterpreted constructors of an effect-language. An effectful program then could be represented as a tree of operations:

\[
\begin{align*}
0p_1(\text{args}..., \text{res}_1 & \Rightarrow \\
0p_2(\text{args}..., \text{res}_2 & \Rightarrow \\
\cdots & \\
\text{Pure(value)}))
\end{align*}
\]
Effekt

Calling an Effect Operation

We can think of effect operations as uninterpreted constructors of an effect-language. An effectful program then could be represented as a tree of operations:

\[
\text{Op}_1(\text{args...}, \text{res}_1 \Rightarrow \\
\text{Op}_2(\text{args...}, \text{res}_2 \Rightarrow \\
\ldots \\
\text{Pure(value)}))
\]

we can write a recursive, pattern matching recursive interpreter to provide semantics to effectful operations.

In PL terms: a \textit{deep embedding} of effect operations.
Shallow Embedding of Effect Handlers

In Scala Effekt, effect operations are immediately called on effect handlers.

Schematically:

```scala
handler.op1(args..., res1 =>
    handler.op2(args..., res2 =>
        ...
    )
)```

 Effekt
Shallow Embedding of Effect Handlers
In Scala Effekt, effect operations are immediately called on effect handlers.

Schematically:
```
handler.op1(args..., res1 ⇒
  handler.op2(args..., res2 ⇒
    ...))
```

Technical Insights
Shallow Embedding of Effect Handlers
In Scala Effekt, effect operations are immediately called on effect handlers.

Schematically:

\[
\text{handler}.\text{op}_1(\text{args}..., \text{res}_1 \Rightarrow \\
\text{handler}.\text{op}_2(\text{args}..., \text{res}_2 \Rightarrow \\
...))
\]

Technical Insights
(a) Shallow embedding of effect handlers simplifies typing – no GADTs are necessary!
Effekt

Shallow Embedding of Effect Handlers

In Scala Effekt, effect operations are immediately called on effect handlers.

Schematically:

```
handler.op1(args..., res1 ⇒
    handler.op2(args..., res2 ⇒
        ...))
```

Technical Insights

(a) Shallow embedding of effect handlers simplifies typing – no GADTs are necessary!

(b) Pattern matching is replaced by dynamic dispatch – benefits performance on the JVM.
Shallow Embedding of Effect Handlers
In Scala Effekt, effect operations are immediately called on effect handlers.

Schematically:
```
handler.op\_1(args\..., res\_1 \\Rightarrow
    handler.op\_2(args\..., res\_2 \\Rightarrow
        ...))
```

Technical Insights
(a) Shallow embedding of effect handlers simplifies typing – no GADTs are necessary!
(b) Pattern matching is replaced by dynamic dispatch – benefits performance on the JVM.
(c) Direct call to corresponding handler – no need to lookup handler.
Part II
Algebraic Effects as Libraries for Java / JVM
Key Specs: JVM / Java Effekt

- shallow embedding vs. deep embedding of handlers
- "handler passing style"
- shallow handlers vs. deep handlers
- user defined effects ✓
- dynamic effect instances ✓
- modular and extensible effect signatures and handlers (✓)
- safety (capabilities can leak) ✗
- user programs are written in direct style ✓
- performance: competitive with JVM continuation libraries ✓
Key Specs: JVM / Java Effekt

- shallow embedding vs. deep embedding of handlers
- "handler passing style"
- shallow handlers vs. deep handlers
- user defined effects ✔
- dynamic effect instances ✔
- modular and extensible effect signatures and handlers ✔
- safety (capabilities can leak) ❌
- user programs are written in direct style ✔
- performance: competitive with JVM continuation libraries ✔
Overview of JVM 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
- We redesigned the algebraic effects library to **only require simple generics**
- **Restriction**: We only transform the terms, not types / signatures
For effectful methods, we maintain our own custom stack, which allows us to manipulate it (searching, slicing, copying).
Example: Drunk Coin Flipping

```scala
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

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

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

class AmbList<R> extends Handler<R, List<R>> implements Amb {
  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 AmbList<Optional<String>>()(), amb ->
  handle(new Maybe<String>()(), exc -> drunkFlip(amb, exc))
>
[Optional["heads"], Optional["tails"], Optional.empty]
Stateful / Parametrized Handlers

```java
interface Reader<In> { In read() throws Effects; }

class StringReader<R> extends Handler<R, R> implements Reader<Char> {
  final String input;
  int pos = 0;

  Char read() throws Effects { return input.charAt(pos++); }
}
```
Stateful / Parametrized Handlers

```java
interface Reader<In> { In read() throws Effects; }

class StringReader<R> extends Handler<R, R> implements Reader<Char>,
    Stateful<Integer> {
    final String input;
    int pos = 0;

    Char read() throws Effects { return input.charAt(pos++); }

    Integer exportState() { return pos; }
    void importState(Integer n) { pos = n; }
}
```
Design Decisions

- **Effectful methods** are marked with a special, checked exception Effects
- **Effect signatures** are interfaces that contain effectful methods
- **Effect handlers** are implementations of those interfaces.
- Users need to manually follow the capability passing style.
- Effect handlers can extend the library class Handler to capture the continuation (but don't need to).
- We use the handler instances as prompt markers.
Bytecode Transformation Example (CPS)

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

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

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

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

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

CPS

Effekt.push(() -> drunkFlip1(amb, exc));
amb.flip();
return DUMMY;
Alternative Transformations

**CPS**

Effekt.push(() -> drunkFlip1(amb, exc));
amb.flip();
return DUMMY;

**Gen. Stack Inspection / Bubble Sem.**

Effekt.beforeCall();
amb.flip();
if (Effekt.isImpure()) {

    Effekt.push(() -> drunkFlip1(amb, exc));

    return DUMMY;
}

Effekt.push(() -> drunkFlip1(amb, exc));
amb.flip();
if (Effekt.isImpure()) {

    Effekt.push(() -> drunkFlip1(amb, exc));

    return DUMMY;
}
Alternative Transformations

CPS

Effekt.push(() -> drunkFlip1(amb, exc));
amb.flip();
return DUMMY;

- all effect calls are tail calls
- cont. is constructed eagerly and immediately available
- unnecessary push/pop/enter cycles
- full reification of the stack

Gen. Stack Inspection / Bubble Sem.

Effekt.beforeCall();
amb.flip();
if (Effekt.isImpure()) {
    Effekt.push(() -> drunkFlip1(amb, exc));
    return DUMMY;
}
Effekt

Alternative Transformations

CPS

Effekt.push(() -> drunkFlip1(amb, exc));
amb.flip();
return DUMMY;

- all effect calls are tail calls
- cont. is constructed eagerly and immediately available
- unnecessary push/pop/enter cycles
- full reification of the stack

Gen. Stack Inspection / Bubble Sem.

Effekt.beforeCall();
amb.flip();
if (Effekt.isImpure()) {
    Effekt.push(() -> drunkFlip1(amb, exc));
    return DUMMY;
}

- two ways to leave a method, distinguished by a flag
- cont. is constructed on demand
- reduced overhead for pure code
- prompt markers are trampolines
## Alternative Transformations (Performance)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline</th>
<th>EFFEKT</th>
<th>EFFEKT\textsubscript{opt}</th>
<th>Coroutines</th>
<th>Quasar</th>
<th>JavaFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stateloop 1M</td>
<td>1.61 ± 0.09</td>
<td>29.76 ± 2.57</td>
<td>1.91 ± 0.04</td>
<td>5.52 ± 0.35</td>
<td>69.02 ± 2.59</td>
<td>14.82 ± 0.48</td>
</tr>
<tr>
<td>RecursiveOnce 1K</td>
<td>0.01 ± 0.0</td>
<td>0.69 ± 0.22</td>
<td>0.34 ± 0.01</td>
<td>0.07 ± 0.0</td>
<td>0.23 ± 0.03</td>
<td>8.18 ± 0.19</td>
</tr>
<tr>
<td>RecursiveMany 1K</td>
<td>0.01 ± 0.0</td>
<td>1.05 ± 0.38</td>
<td>0.4 ± 0.07</td>
<td>10.29 ± 1.41</td>
<td>68.07 ± 2.07</td>
<td>3363.74 ± 23.46</td>
</tr>
<tr>
<td>Skynet 1M</td>
<td>2.74 ± 0.03</td>
<td>171.34 ± 5.55</td>
<td>62.13 ± 3.87</td>
<td>35.19 ± 2.51</td>
<td>762.1 ± 155.95</td>
<td>1277.51 ± 54.18</td>
</tr>
<tr>
<td>SkynetSuspend 1M</td>
<td>2.74 ± 0.03</td>
<td>414.56 ± 9.2</td>
<td>147.4 ± 5.44</td>
<td>50.46 ± 2.95</td>
<td>1113.15 ± 112.78</td>
<td>7198.72 ± 122.56</td>
</tr>
</tbody>
</table>
# Alternative Transformations (Performance)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline</th>
<th>EFFEKT</th>
<th>EFFEKT_{opt}</th>
<th>Coroutines</th>
<th>Quasar</th>
<th>JavaFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stateloop 1M</td>
<td>1.61 ± 0.09</td>
<td>29.76 ± 2.57</td>
<td>1.91 ± 0.04</td>
<td>5.52 ± 0.35</td>
<td>69.02 ± 2.59</td>
<td>14.82 ± 0.48</td>
</tr>
<tr>
<td>RecursiveOnce 1K</td>
<td>0.01 ± 0.0</td>
<td>0.69 ± 0.22</td>
<td>0.34 ± 0.01</td>
<td>0.07 ± 0.0</td>
<td>0.23 ± 0.03</td>
<td>8.18 ± 0.19</td>
</tr>
<tr>
<td>RecursiveMany 1K</td>
<td>0.01 ± 0.0</td>
<td>1.05 ± 0.38</td>
<td>0.4 ± 0.07</td>
<td>10.29 ± 1.41</td>
<td>68.07 ± 2.07</td>
<td>3363.74 ± 23.46</td>
</tr>
<tr>
<td>Skynet 1M</td>
<td>2.74 ± 0.03</td>
<td>171.34 ± 5.55</td>
<td>62.13 ± 3.87</td>
<td>35.19 ± 2.51</td>
<td>762.1 ± 155.95</td>
<td>1277.51 ± 54.18</td>
</tr>
<tr>
<td>SkynetSuspend 1M</td>
<td>2.74 ± 0.03</td>
<td>414.56 ± 9.2</td>
<td>147.4 ± 5.44</td>
<td>50.46 ± 2.95</td>
<td>1113.15 ± 112.78</td>
<td>7198.72 ± 122.56</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>EFFEKT</th>
<th>EFFEKT_{opt}</th>
<th>Scala Effekt</th>
<th>Scala Eff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Countdown 10K</td>
<td>3.35 ± 0.07</td>
<td>2.47 ± 0.12</td>
<td>6.07 ± 0.32</td>
<td>34.39 ± 2.59</td>
</tr>
<tr>
<td>Countdown8 1K</td>
<td>1.31 ± 0.39</td>
<td>1.77 ± 0.1</td>
<td>2.31 ± 0.12</td>
<td>36.92 ± 3.0</td>
</tr>
<tr>
<td>NQueens (10)</td>
<td>19.5 ± 0.38</td>
<td>16.09 ± 0.19</td>
<td>40.95 ± 0.54</td>
<td>49.89 ± 2.17</td>
</tr>
</tbody>
</table>
# Alternative Transformations (Performance)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline</th>
<th>EFFEKT</th>
<th>EFFEKT&lt;sub&gt;opt&lt;/sub&gt;</th>
<th>Coroutines</th>
<th>Quasar</th>
<th>JavaFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stateloop 1M</td>
<td>1.61 ± 0.09</td>
<td>29.76 ± 2.57</td>
<td>1.91 ± 0.04</td>
<td>5.52 ± 0.35</td>
<td>69.02 ± 2.59</td>
<td>14.82 ± 0.48</td>
</tr>
<tr>
<td>RecursiveOnce 1K</td>
<td>0.01 ± 0.0</td>
<td>0.69 ± 0.22</td>
<td>0.34 ± 0.01</td>
<td>0.07 ± 0.0</td>
<td>0.23 ± 0.03</td>
<td>8.18 ± 0.19</td>
</tr>
<tr>
<td>RecursiveMany 1K</td>
<td>0.01 ± 0.0</td>
<td>1.05 ± 0.38</td>
<td>0.4 ± 0.07</td>
<td>10.29 ± 1.41</td>
<td>68.07 ± 2.07</td>
<td>3363.74 ± 23.46</td>
</tr>
<tr>
<td>Skynet 1M</td>
<td>2.74 ± 0.03</td>
<td>171.34 ± 5.55</td>
<td>62.13 ± 3.87</td>
<td>35.19 ± 2.51</td>
<td>762.1 ± 155.95</td>
<td>1277.51 ± 54.18</td>
</tr>
<tr>
<td>SkynetSuspend 1M</td>
<td>2.74 ± 0.03</td>
<td>414.56 ± 9.2</td>
<td>147.4 ± 5.44</td>
<td>50.46 ± 2.95</td>
<td>1113.15 ± 112.78</td>
<td>7198.72 ± 122.56</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>EFFEKT</th>
<th>EFFEKT&lt;sub&gt;opt&lt;/sub&gt;</th>
<th>Scala Eff</th>
<th>Scala Eff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Countdown 10K</td>
<td>3.35 ± 0.07</td>
<td>2.47 ± 0.12</td>
<td>34.39 ± 2.59</td>
<td></td>
</tr>
<tr>
<td>Countdown8 1K</td>
<td>1.31 ± 0.39</td>
<td>1.77 ± 0.1</td>
<td>36.92 ± 3.0</td>
<td></td>
</tr>
<tr>
<td>NQueens (10)</td>
<td>19.5 ± 0.38</td>
<td>16.09 ± 0.19</td>
<td>49.89 ± 2.17</td>
<td></td>
</tr>
</tbody>
</table>

- Coroutines (https://github.com/offbynull/coroutines)
- Quasar (http://docs.paralleluniverse.co/quasar)
- Javaflow (https://github.com/vsilaev/tascalate-javaflow)
- Eff (https://github.com/athanos-org/eff)
Part III
Even More Extensible Effects
The (Effect) Expression Problem

Original Expression Problem
Variant of a Datatype vs. Effect Operation
(Recursive) Operation vs. Handler Implementation
The (Effect) Expression Problem

We rephrase the expression problem in context of algebraic effects as:

Modularly being able to
a) implement new handlers for an effect signature.

b) add new effect operations to an existing effect signature
Extensibility supported by Effekt
Extensibility supported by Effekt

a) *implement new handlers for an effect signature.*

```scala
trait ExcOption[R] extends Exc with Handler[R, Option[R]] { ... }
trait ExcEither[R] extends Exc with Handler[R, Either[String, R]] { ... }
```
Extensibility supported by Effekt

a) *implement new handlers for an effect signature.*

```
trait ExcOption[R] extends Exc with Handler[R, Option[R]] { … }
trait ExcEither[R] extends Exc with Handler[R, Either[String, R]] { … }
```

b) *add new effect operations …*

… by adding a new signature (like Amb and Exc)

… by adding operations to a signature
Extensibility supported by Effekt

a) *implement new handlers* for an effect signature.

```scala
trait ExcOption[R] extends Exc with Handler[R, Option[R]] { ... }
trait ExcEither[R] extends Exc with Handler[R, Either[String, R]] { ... }
```

b) *add new effect operations* ...

... by adding a new signature (like Amb and Exc)
... by adding operations to a signature

```scala
trait AmbChoose extends Amb { def choose[A](choices: List[A]): Op[A] }
trait AmbChooseList[R] extends AmbChoose with AmbList[R] {
  def choose[A](choices: List[A]): Op[A] = ...
}
```
Effekt

Extensibility supported by Effekt (2)

Handling two effects with one handler:

```scala
trait ExcList[R] extends Exc with Handler[R, List[R]] {
  def raise[A](msg: String): Op[A] = resume => pure(List.empty)
}

trait ExcAmbList[R] extends ExcList[R] with AmbList[R] {}

ExcAmbList { drunkFlip }
> List("heads", "tails")
```
Extensibility supported by Effekt (2)

Handling two effects with one handler:

```scala
trait ExcList[R] extends Exc with Handler[R, List[R]] {
  def raise[A](msg: String): Op[A] = resume ⇒ pure(List.empty)
}

trait ExcAmbList[R] extends ExcList[R] with AmbList[R] {}

ExcAmbList { drunkFlip }
  > List("heads", "tails")

Desugares to:

ExcAmbList { both ⇒ drunkFlip(both, both) }
```
Part IV
Effect Typing
and OO:
A Problem
Statement
Object Oriented Programming

The mantra of OOP:

- **subtyping** and Liskov's substitution principle
- hiding **implementation details** behind interfaces
- implementation is **existentially** hidden
- Information hiding happens on the granularity of **single objects**
Subtyping & Information Hiding

```scala
trait Person {
    def greet(): Unit
}

trait IOPerson extends Person {
    def greet(): Unit using Console
}

trait AlertPerson extends Person {
    def greet(): Unit using GUI
}
```
Subtyping & Information Hiding

```scala
trait Person {
    def greet(): Unit
}

trait IOPerson extends Person {
    def greet(): Unit using Console
}

trait AlertPerson extends Person {
    def greet(): Unit using GUI
}
```

Not a subtype!
Users of Person now also need to be effect polymorphic!

```scala
trait Person[E] {
    def greet(): Unit using E
}

trait IOPerson extends Person[Console]

trait AlertPerson extends Person[GUI]
```
Solution Attempt 1

```scala
trait Person {
  type E
  def greet(): Unit using E
}

trait IOPerson extends Person {
  type E = Console
}

trait AlertPerson extends Person {
  type E = GUI
}
```

Effect types are now path dependent:

```scala
def user(p: Person): Int using p.E
```

Only works for stable values!
Solution Attempt 2

```scala
// Person
trait Person {
  def greet(): Control[Unit]
}

// IOPerson extends Person
trait IOPerson extends Person {
  implicit val console: Cap[Console]
}

// AlertPerson extends Person
trait AlertPerson extends Person {
  implicit val gui: Cap[GUI]
}

// Define user function
def user(p: Person): Control[Int]
```

The effect now is truly hidden

```scala
def user(p: Person): Control[Int]
```
Capability Safety

```scala
def leaking(implicit amb: Cap[Amb]): Control[String] = {
  pure("hello world")
}
```
null

def leaking(implicit amb: Cap[Amb]): Control[String] = {
  c = amb;
  pure("hello world")
}
 Capability Safety

```scala
val c: Cap[Amb] = null
def leaking(implicit amb: Cap[Amb]): Control[String] = {
  c = amb;
  pure("hello world")
}

AmbList { leaking }.run()
```
% Effekt

**Capability Safety**

```scala
var c: Cap[Amb] = null
def leaking(implicit amb: Cap[Amb]): Control[String] = {
    c = amb;
    pure("hello world")
}

AmbList { leaking }.run()
{ flip()(c) }.run()
```
```scala
var c: Cap[Amb] = null
def leaking(implicit amb: Cap[Amb]): Control[String] = {
  c = amb;
pure("hello world")
}{ flip()(c) }.run()
AmbList { leaking }.run()
```
Possible Solution

Make (capability) objects second class again
Second Class Values in Scala Escape

```scala
var c: Cap[Amb] = null
def leaking(implicit @local amb: Cap[Amb]): Control[String] = {
    c = amb;
    pure("hello world")
}
```
Second Class Values in Scala Escape

```scala
var c: Cap[Amb] = null
def leaking(implicit @local amb: Cap[Amb]): Control[String] = {
  c = amb;
  pure("hello world")
}
```

**Error:** local value `amb` cannot be assigned to variable `c` since it would leave the scope of function `leaking`.
Second Class Values in Scala Escape

```scala
var c: Cap[Amb] = null
def leaking(implicit @local amb: Cap[Amb]): Control[String] = {
  c = amb;
  pure("hello world")
}
```

**Error:** local value `amb` cannot be assigned to variable `c` since it would leave the scope of function `leaking`.

Restricts scope of capabilities so that they can be stack allocated.
Second Class Values in Scala Escape

```scala
var c: Cap[Amb] = null
def leaking(implicit @local amb: Cap[Amb]): Control[String] = {
  c = amb;
  pure("hello world")
}
```

**Error:** local value `amb` cannot be assigned to variable `c` since it would leave the scope of function `leaking`.

Restricts scope of capabilities so that they can be stack allocated. This perfectly fits algebraic effects.
Solution Attempt 2

```
trait Person {
  def greet(): Control[Unit]
}

trait IOPerson extends Person {
  implicit val console: Cap[Console]
}

trait AlertPerson extends Person {
  implicit val gui: Cap[GUI]
}
```
Solution Attempt 2

```scala
trait Person {
    def greet(): Control[Unit]
}

trait IOPerson extends Person {
    implicit val console: Cap[Console]
}

trait AlertPerson extends Person {
    implicit val gui: Cap[GUI]
}
```

Object lifetime < Capability lifetime
The Root of Evil

There is a simple connection:
- Algebraic effects and delimited continuations are all about the stack
- Object oriented programming is all about heap allocated objects

Conflicting requirements:
- capabilities should be stack allocated, objects don't
- but object lifetime should not be coupled to capability lifetime
- in particular, objects should be able to escape the handler scope losing the capabilities
Part V
Effectful
Syntax
Towards Naturalistic EDSLs using Algebraic Effects

Effectful Syntax

Linguistic phenomena like anaphora, scoping, quantification, implicature, focus and more can be modeled uniformly using algebraic effects.
Towards Naturalistic EDSLs using Algebraic Effects

**Effectful Syntax**

Linguistic phenomena like anaphora, scoping, quantification, implicature, focus and more can be modeled uniformly using algebraic effects.

- Support linguistic phenomena in EDSLs using algebraic effects
- Use (algebraic) effects for AST construction
Example 1: The Speaker Effect

```scala
val s1: Sentence using Speaker = john said { mary loves me }
```
Example 1: The Speaker Effect

Effect Signature
Groups effect operations in a type

```scala
val s1: Sentence using Speaker = john said { mary loves me }
```

Effect Operations
Semantics of the operations is left open
Example 1: The Speaker Effect

Effect Signature
Groups effect operations in a type

```
val s1: Sentence using Speaker = john said { mary loves me }
```

Effect Operations
Semantics of the operations is left open

```
pete saidQuote { s1 }
```
Example 1: The Speaker Effect

Effect Signature

Groups effect operations in a type

| val s₁: Sentence using Speaker = john said { mary loves me } |

Effect Handlers

Provide semantics to effect operations

| pete saidQuote { s₁ } |

Effect Operations

Semantics of the operations is left open
Example 1: The Speaker Effect

Effect Signature
Groups effect operations in a type

```
val s1: Sentence using Speaker = john said { mary loves me }
```

Effect Handlers
Provide semantics to effect operations
```
pete saidQuote { s1 }
```

Effect Operations
Semantics of the operations is left open
```
> Said(Pete, Said(John, Loves(Mary, Pete)))
```
Towards Naturalistic EDSLs using Algebraic Effects

Example 2: The Scope Effect

```scala
val s2: Sentence using Scope = john saidQuote { every(woman) loves me }
```
Example 2: The Scope Effect

```scala
val s2: Sentence using Scope = john saidQuote { every(woman) loves me }
```

Effect Operations
Semantics of the operations is left open
Example 2: The Scope Effect

Effect Signature

Groups effect operations in a type

\[
\text{val } s_2 : \text{Sentence using Scope } = \text{john saidQuote } \{ \text{every(woman) loves me} \}
\]

Effect Operations

Semantics of the operations is left open
Example 2: The Scope Effect

Effect Signature

Groups effect operations in a type

\[
\text{val } s_2: \text{Sentence using Scope = john saidQuote } \{ \text{every(woman) loves me} \}
\]

Effect Operations

Semantics of the operations is left open

\[
\text{scoped } \{ \ s_2 \ }
\]
Example 2: The Scope Effect

Effect Signature
Groups effect operations in a type

```scala
val s2: Sentence using Scope = john saidQuote { every(woman) loves me }
```

Effect Handlers
Provide semantics to effect operations

```scala
scoped { s2 }
```

Effect Operations
Semantics of the operations is left open
Example 2: The Scope Effect

Effect Signature
Groups effect operations in a type

```scala
val s2: Sentence using Scope = john saidQuote { every(woman) loves me }
```

Effect Operations
Semantics of the operations is left open

```scala
scoped { s2 }
> Forall(x => Implies(Woman(x), Said(John, Loves(x, John))))
```
Example 3: The Implicature Effect

```scala
val s3: Sentence using Speaker and Implicature =
mary loves { john whoIs { _ bestFriendOf me } }
```
Example 3: The Implicature Effect

```scala
val s3: Sentence using Speaker and Implicature =
mary loves { john whoIs { _ bestFriendOf me } }
```

Effect Operations

Semantics of the operations is left open
Example 3: The Implicature Effect

```scala
val s3: Sentence using Speaker and Implicature =
  mary loves { john whoIs { _ bestFriendOf me } }

pete saidQuote { accommodate { s3 } }
```
Towards Naturalistic EDSLs using Algebraic Effects

Example 3: The Implicature Effect

```scala
val s3: Sentence using Speaker and Implicature =
  mary loves { john whoIs { _ bestFriendOf me } }

pete saidQuote { accommodate { s3 } }
> Said(Pete,
  And(BestFriendOf(John, Pete),
    Loves(Mary, John)))
```
Example 3: The Implicature Effect

val s₃: Sentence using Speaker and Implicature =
  mary loves { john whoIs { _ bestFriendOf me } }}

pete saidQuote { accommodate { s₃ } }
> Said(Pete,
  And(BestFriendOf(John, Pete),
       Loves(Mary, John)))

accommodate { pete saidQuote { s₃ } }
Example 3: The Implicature Effect

```scala
val s3: Sentence using Speaker and Implicature =
  mary loves { john whoIs { _ bestFriendOf me } }

pete saidQuote { accommodate { s3 } }
> Said(Pete,
    And(BestFriendOf(John, Pete),
        Loves(Mary, John)))

accommodate { pete saidQuote { s3 } }
> And(Said(Pete, BestFriendOf(John, Pete)),
    Said(Pete, Loves(Mary, John)))
```
Towards Naturalistic EDSLs using Algebraic Effects

**Effectful Syntax**

Use (algebraic) effects for AST construction.

More precisely, we propose to group syntax elements of DSLs into:
Towards Naturalistic EDSLs using Algebraic Effects

**Effectful Syntax**

Use (algebraic) effects for AST construction.

More precisely, we propose to group syntax elements of DSLs into:

- **Pure Syntax**: Plain constructors.
Towards Naturalistic EDSLs using Algebraic Effects

**Effectful Syntax**

Use (algebraic) effects for AST construction.

More precisely, we propose to group syntax elements of DSLs into:

- **Pure Syntax**: Plain constructors.

  - loves
  - woman
  - john
  - mary

- **Effectful Syntax**: Non local syntax that requires contextual handling.

  - me
  - whoIs
  - every
Towards Naturalistic EDSLs using Algebraic Effects

**Effectful Syntax**

Use (algebraic) effects for AST construction.

More precisely, we propose to group syntax elements of DSLs into:

**Effectful Syntax**
- Non local syntax that requires contextual handling.

**Pure Syntax**
- Plain constructors.

**Handling Syntax**
- Handles effects by restructuring the AST, potentially again using effects.
Towards Naturalistic EDSLs using Algebraic Effects

Properties

Effectful syntax based on algebraic effects is...
Towards Naturalistic EDSLs using Algebraic Effects

Properties

Effectful syntax based on algebraic effects is...

Modular

Linguistic phenomena can be encapsulated into reusable modules.
Towards Naturalistic EDSLs using Algebraic Effects

Properties

Effectful syntax based on algebraic effects is...

Modular
Linguistic phenomena can be encapsulated into reusable modules.

Effectful Syntax
Handling Syntax

Pure Syntax
Towards Naturalistic EDSLs using Algebraic Effects

Properties

Effectful syntax based on algebraic effects is...

**Modular**
Linguistic phenomena can be encapsulated into reusable modules.

**Learnable**
Separating linguistic phenomena from other domain concepts allows separate understanding

**Effectful Syntax Handling Syntax**

**Pure Syntax**
Towards Naturalistic EDSLs using Algebraic Effects

Properties

Effectful syntax based on algebraic effects is...

**Modular**
Linguistic phenomena can be encapsulated into reusable modules.

**Learnable**
Separating linguistic phenomena from other domain concepts allows separate understanding

**Maintainable**
Types precisely communicate usage of effectful syntax.