Effekt

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT

Algebraic Effects on the JVM

Jonathan Immanuel Brachthäuser
University of Tübingen, Germany

Dagstuhl Seminar 18172
Algebraic Effect Handlers go Mainstream

github.com/b-studios/scala-effekt

We developed algebraic effect libraries for

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 magnitute) slower than primitive effects ✘

Effekt

b-studios.de/scala-effekt

005

How to USE Scala Effekt

Example: Drunk Coin Flipping

```
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

Effect Signatures

Group effect operations in one type

```
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"


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
```

Effekt

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

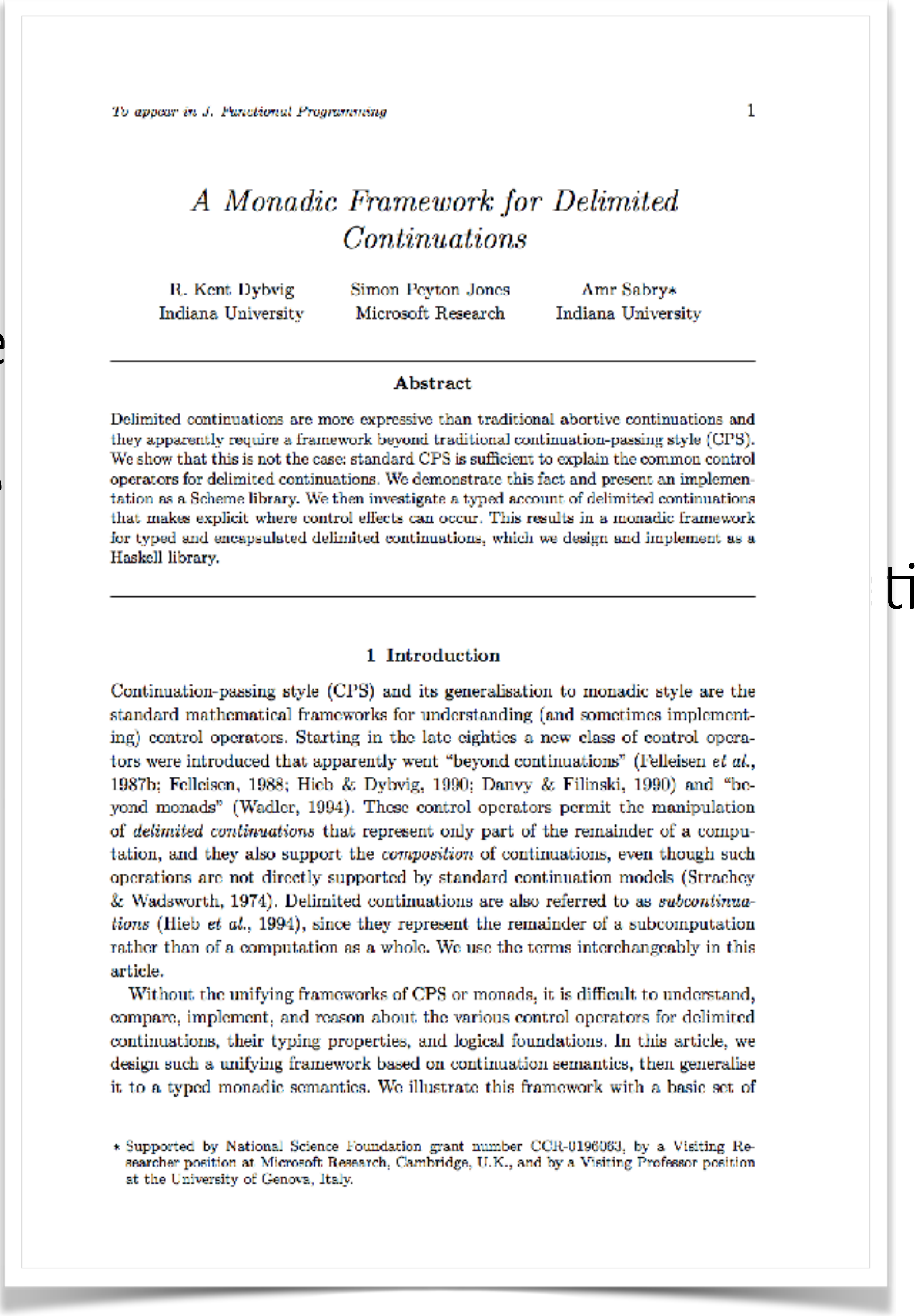
Effekt

Design Decisions - Sca

We were faced with the following thre

1. how to **capture the context** of effe

   ⟹ monadic implementation for m tions



A Monadic Framework for Delimited Continuations

R. Kent Dybvig, Indiana University
Simon Peyton Jones, Microsoft Research
Amr Sabry⋆, Indiana University

Abstract

Delimited continuations are more expressive than traditional abortive continuations and they apparently require a framework beyond traditional continuation-passing style (CPS). We show that this is not the case: standard CPS is sufficient to explain the common control operators for delimited continuations. We demonstrate this fact and present an implementation as a Scheme library. We then investigate a typed account of delimited continuations that makes explicit where control effects can occur. This results in a monadic framework for typed and encapsulated delimited continuations, which we design and implement as a Haskell library.

1 Introduction

Continuation-passing style (CPS) and its generalisation to monadic style are the standard mathematical frameworks for understanding (and sometimes implementing) control operators. Starting in the late eighties a new class of control operators were introduced that apparently went "beyond continuations" (Felleisen *et al.*, 1987b; Felleisen, 1988; Hieb & Dybvig, 1990; Danvy & Filinski, 1990) and "beyond monads" (Wadler, 1994). These control operators permit the manipulation of *delimited continuations* that represent only part of the remainder of a computation, and they also support the *composition* of continuations, even though such operations are not directly supported by standard continuation models (Strachey & Wadsworth, 1974). Delimited continuations are also referred to as *subcontinuations* (Hieb *et al.*, 1994), since they represent the remainder of a subcomputation rather than of a computation as a whole. We use the terms interchangeably in this article.

Without the unifying frameworks of CPS or monads, it is difficult to understand, compare, implement, and reason about the various control operators for delimited continuations, their typing properties, and logical foundations. In this article, we design such a unifying framework based on continuation semantics, then generalise it to a typed monadic semantics. We illustrate this framework with a basic set of

⋆ Supported by National Science Foundation grant number CCR-0196063, by a Visiting Researcher position at Microsoft Research, Cambridge, U.K., and by a Visiting Professor position at the University of Genova, Italy.

Design Decisions - Scala Effekt

We were faced with the following three design questions:

1. how to **capture the context** of effect operations?

   $\Longrightarrow$ monadic implementation for multi-prompt delimited continuations

2. how should **effect handlers provide semantics** for effect operations?

   $\Longrightarrow$ 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?
   $\Longrightarrow$ monadic implementation for multi-prompt delimited continuations
2. how should **effect handlers provide semantics** for effect operations?
   $\Longrightarrow$ shallow embedding of effect handlers
3. how to establish an **effect typing discipline**?
   $\Longrightarrow$ 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?

   $\Longrightarrow$ monadic implementation for multi-prompt delimited continuations

2. how should **effect handlers provide semantics** for effect operations?

   $\Longrightarrow$ shallow embedding of effect handlers

3. how to establish an **effect typing discipline**?

   $\Longrightarrow$ capability passing style

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

Implicit Function Types

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"
```

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 using[A, E] = implicit Cap[E] ⇒ Control[A]
```

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 using[A, E] = implicit Cap[E] ⇒ Control[A]
```

```
type and[A, E]   = implicit Cap[E] ⇒ A
```

Making Capability Passing Explicit

Explicitly desugaring implicit function types gives:

```
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:

```
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:

```
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 `Cap[E]` encapsulate three different things:

Capabilities in Effekt

Capabilities `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 $\mathsf{Cap}[E]$ 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 `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).

Effekt

shallow embedding

of effect handlers

b-studios.de/scala-effekt

0015

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:

```
Op₁(args…, res₁ ⇒
  Op₂(args…, res₂ ⇒
    …
      Pure(value)))
```

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:

```
Op₁(args…, res₁ ⇒
  Op₂(args…, res₂ ⇒
    …
      Pure(value)))
```

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

In PL terms: a ***deep embedding*** of effect operations.

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 ⇒
    …))
```

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:

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

Technical Insights

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

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.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.

(c) Direct call to corresponding handler – no need to lookup handler.

Part II

Algebraic Effects as Libraries for Java / JVM

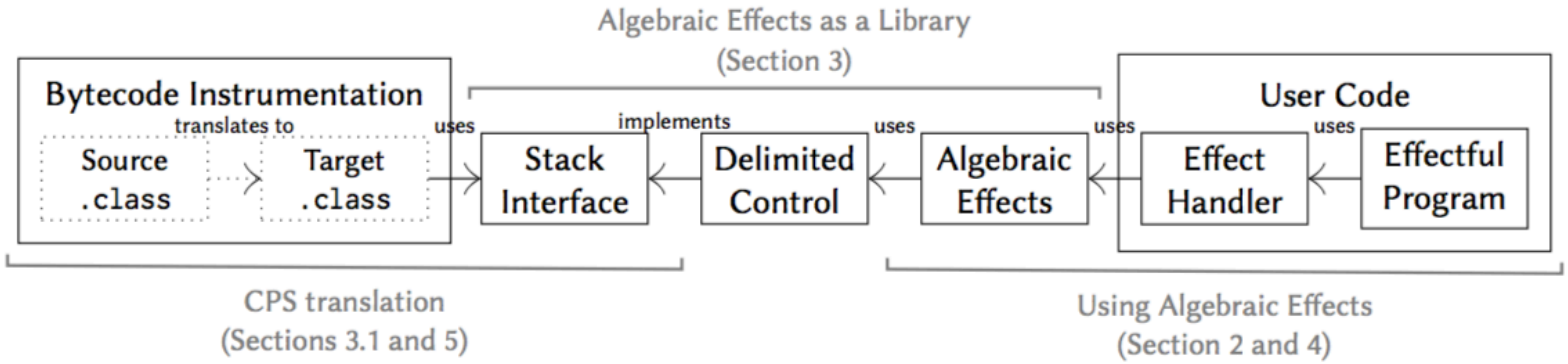
Key Specs: JVM / Java Effekt

- 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 ✔

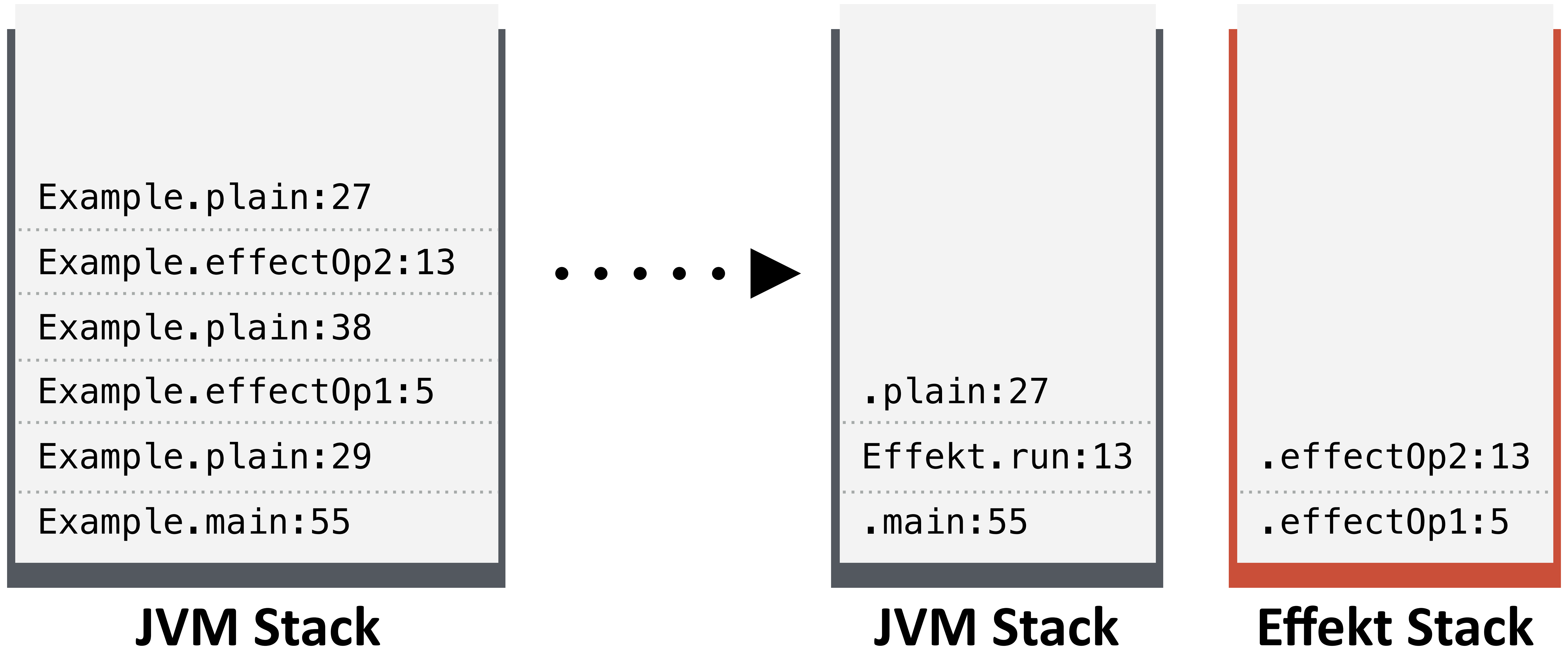
Overview of JVM Effekt



- 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

Replacing the JVM Stack

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



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

```
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

```
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

```
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();
  }
}
```

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();
  }
}

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;
}
```

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;
}
```

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)

	Time in ms (Confidence Interval)					
Benchmark	**Baseline**	**Effekt**	**Effekt$_{opt}$**	**Coroutines**	**Quasar**	**JavaFlow**
Stateloop 1M	1.61 ±0.09	29.76 ±2.57	1.91 ±0.04	5.52 ±0.35	69.02 ±2.59	14.82 ±0.48
RecursiveOnce 1K	0.01 ±0.0	0.69 ±0.22	0.34 ±0.01	0.07 ±0.0	0.23 ±0.03	8.18 ±0.19
RecursiveMany 1K	0.01 ±0.0	1.05 ±0.38	0.4 ±0.07	10.29 ±1.41	68.07 ±2.07	3363.74 ±23.46
Skynet 1M	2.74 ±0.03	171.34 ±5.55	62.13 ±3.87	35.19 ±2.51	762.1 ±155.95	1277.51 ±54.18
SkynetSuspend 1M	2.74 ±0.03	414.56 ±9.2	147.4 ±5.44	50.46 ±2.95	1113.15 ±112.78	7198.72 ±122.56

Alternative Transformations (Performance)

	Time in ms (Confidence Interval)					
Benchmark	**Baseline**	**Effekt**	**Effekt$_{opt}$**	**Coroutines**	**Quasar**	**JavaFlow**
Stateloop 1M	1.61 ±0.09	29.76 ±2.57	1.91 ±0.04	5.52 ±0.35	69.02 ±2.59	14.82 ±0.48
RecursiveOnce 1K	0.01 ±0.0	0.69 ±0.22	0.34 ±0.01	0.07 ±0.0	0.23 ±0.03	8.18 ±0.19
RecursiveMany 1K	0.01 ±0.0	1.05 ±0.38	0.4 ±0.07	10.29 ±1.41	68.07 ±2.07	3363.74 ±23.46
Skynet 1M	2.74 ±0.03	171.34 ±5.55	62.13 ±3.87	35.19 ±2.51	762.1 ±155.95	1277.51 ±54.18
SkynetSuspend 1M	2.74 ±0.03	414.56 ±9.2	147.4 ±5.44	50.46 ±2.95	1113.15 ±112.78	7198.72 ±122.56

Benchmark	**Effekt**	**Effekt$_{opt}$**	**Scala Effekt**	**Scala Eff**
Countdown 10K	3.35 ±0.07	2.47 ±0.12	6.07 ±0.32	34.39 ±2.59
Countdown8 1K	1.31 ±0.39	1.77 ±0.1	2.31 ±0.12	36.92 ±3.0
NQueens (10)	19.5 ±0.38	16.09 ±0.19	40.95 ±0.54	49.89 ±2.17

Alternative Transformations (Performance)

	Time in ms (Confidence Interval)					
Benchmark	**Baseline**	**Effekt**	**Effekt_{opt}**	**Coroutines**	**Quasar**	**JavaFlow**
Stateloop 1M	1.61 ±0.09	29.76 ±2.57	1.91 ±0.04	5.52 ±0.35	69.02 ±2.59	14.82 ±0.48
RecursiveOnce 1K	0.01 ±0.0	0.69 ±0.22	0.34 ±0.01	0.07 ±0.0	0.23 ±0.03	8.18 ±0.19
RecursiveMany 1K	0.01 ±0.0	1.05 ±0.38	0.4 ±0.07	10.29 ±1.41	68.07 ±2.07	3363.74 ±23.46
Skynet 1M	2.74 ±0.03	171.34 ±5.55	62.13 ±3.87	35.19 ±2.51	762.1 ±155.95	1277.51 ±54.18
SkynetSuspend 1M	2.74 ±0.03	414.56 ±9.2	147.4 ±5.44	50.46 ±2.95	1113.15 ±112.78	7198.72 ±122.56

Benchmark	**Effekt**	**Effekt_{opt}**	**Scala Effekt**	**Scala Eff**
Countdown 10K	3.35 ±0.07	2.47 ±0.12	6.07 ±0.32	34.39 ±2.59
Countdown8 1K	1.31 ±0.39	1.77 ±0.1	2.31 ±0.12	36.92 ±3.0
NQueens (10)	19.5 ±0.38	16.09 ±0.19	40.95 ±0.54	49.89 ±2.17

- Coroutines (https://github.com/offbynull/coroutines)
- Quasar (http://docs.paralleluniverse.co/quasar)
- Javaflow (https://github.com/vsilaev/tascalate-javaflow)
- Eff (https://github.com/atnos-org/eff)

Part III
Even More Extensible Effects

The (Effect) Expression Problem

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

The (Effect) Expression Problem

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

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.*

```
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.*

```
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

```
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] = …
}
```

Extensibility supported by Effekt (2)

Handling two effects with one handler:

```
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:

```
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) }
```

Effekt

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

```
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

```
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!

Solution Attempt 1

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

Users of Person now also need to be effect polymorphic!

```
def user[E](p: Person[E]): Int using E
```

```
trait IOPerson extends Person[Console]
```

```
trait AlertPerson extends Person[GUI]
```

Solution Attempt 1

```
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:

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

Only works for stable values!

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]
}
```

The effect now is truly hidden

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

Capability Safety

```
def leaking(implicit amb: Cap[Amb]): Control[String] = {

  pure("hello world")
}
```

Capability Safety

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

Capability Safety

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

AmbList { leaking }.run()
```

Capability Safety

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

AmbList { leaking }.run()

{ flip()(c) }.run()
```

1. fish /Users/jonathan/AeroFS/work/github/algebraic-effects (fish)

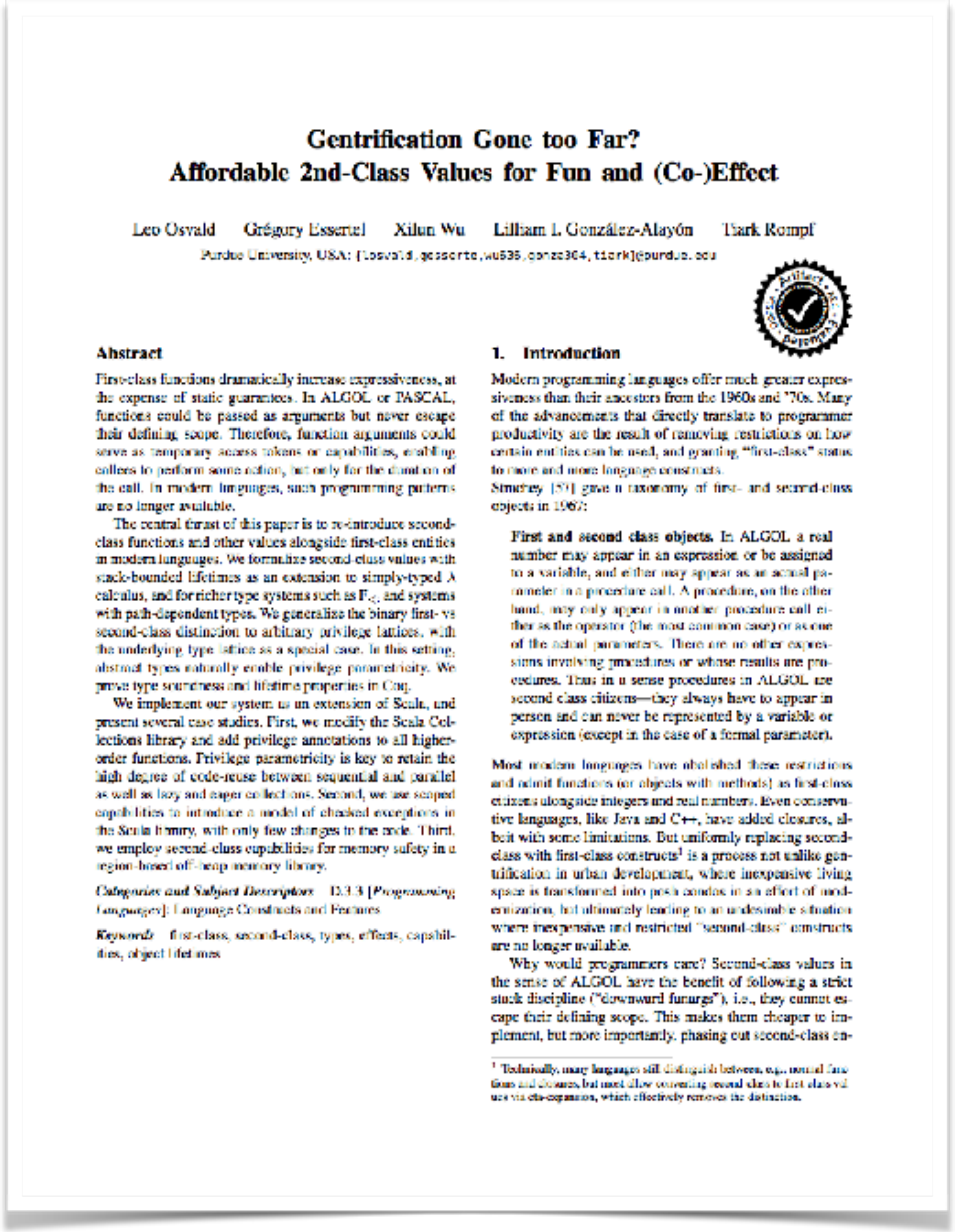
```
hello 4
java.lang.RuntimeException: Prompt effekt.Capability$$anon$1@741e0037 not found on the stack.
  at scala.sys.package$.error(package.scala:27)
  at effekt.ReturnCont.splitAt(MetaCont.scala:24)
  at effekt.ReturnCont.splitAt(MetaCont.scala:19)
  at effekt.FramesCont.splitAt(MetaCont.scala:64)
  at effekt.Control$$anonfun$use$1.apply(Control.scala:121)
  at effekt.Control$$anonfun$use$1.apply(Control.scala:118)
  at effekt.Computation.apply(Control.scala:80)
  at effekt.Result$.trampoline(Result.scala:26)
  at effekt.Control$class.run(Control.scala:48)
  at effekt.Computation.run(Control.scala:78)
  at events.asyncPiping$suspended$$anonfun$interleave$1.apply(asyncPiping.scala:112)
  at events.asyncPiping$suspended$$anonfun$interleave$1.apply(asyncPiping.scala:108)
  at events.asyncPiping$Default$$anonfun$await$1$$anonfun$apply$4.apply(asyncPiping.scala:51)
  at events.asyncPiping$Default$$anonfun$await$1$$anonfun$apply$4.apply(asyncPiping.scala:51)
  at effekt.Control$$anonfun$use$1.apply(Control.scala:142)
  at effekt.Control$$anonfun$use$1.apply(Control.scala:118)
  at effekt.Computation.apply(Control.scala:80)
  at effekt.Result$.trampoline(Result.scala:26)
  at effekt.Control$class.run(Control.scala:48)
  at effekt.Computation.run(Control.scala:78)
  at events.asyncPiping$.runInterleaveUser(asyncPiping.scala:187)
  ... 43 elided

scala>
```

Effekt

Possible Solution

Make (capability) objects *second class* again



Gentrification Gone too Far? Affordable 2nd-Class Values for Fun and (Co-)Effect

Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I. González-Alayón, Tiark Rompf
Purdue University, USA

Abstract

1. Introduction

First and second class objects.

A Study of Capability-Based Effect Systems

Fengyun Liu

A thesis submitted for the degree of Master of Computer Science at École Polytechnique Fédérale de Lausanne

Supervisors

Martin Odersky, Professor
Nada Amin, PhD Student
Sandro Stucki, PhD Student

School of Computer and Communication Sciences

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Lausanne, January 2016

Second Class Values in Scala Escape

```
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


```
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


```
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


```
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

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

Object lifetime < Capability lifetime

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

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

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:

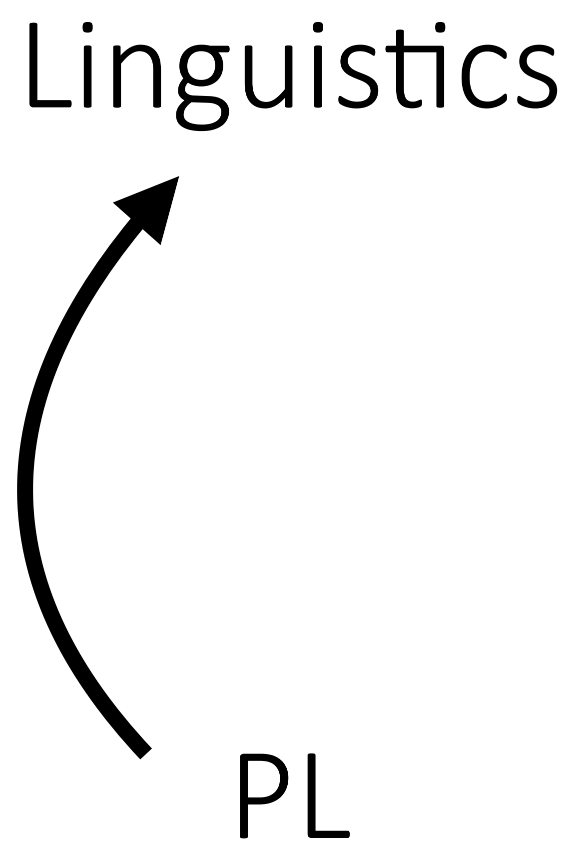
- capabilities should be stack allocated, objects don't
- but object lifetime should not be coupled to capability lifetime
- in particular, objects should be able to escape the handler scope losing the capabilities

Part V
Effectful

Syntax

Effectful Syntax

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

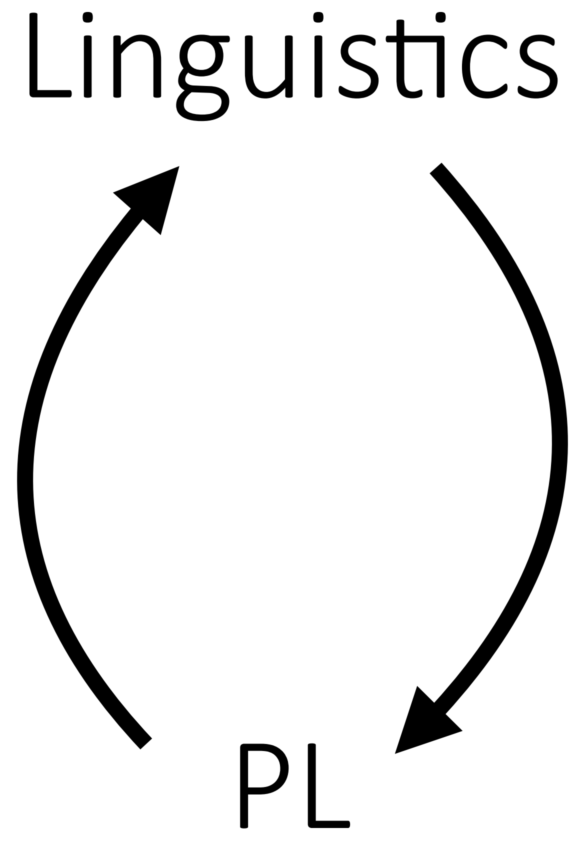


Algebraic Effects
Jiří Maršík and Maxime Amblard, 2016

Effectful Syntax

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

Algebraic Effects
Jiří Maršík and Maxime Amblard, 2016



Effectful Syntax
This Talk

- **Support linguistic phenomena in EDSLs using algebraic effects**
- **Use (algebraic) effects for AST construction**

Example 1: The Speaker Effect

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

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

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

Effect Operations

Semantics of the operations is left open

```
pete saidQuote { s₁ }
> Said(Pete, Said(John, Loves(Mary, Pete)))
```

Example 2: The Scope Effect

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

Example 2: The Scope Effect


```
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

```
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

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

Effect Operations

Semantics of the operations is left open

```
scoped { s₂ }
```

Example 2: The Scope Effect

Effect Signature

Groups effect operations in a type

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

Effect Operations

Semantics of the operations is left open

Effect Handlers

Provide semantics to effect operations

```
scoped { s2 }
```

Example 2: The Scope Effect

Effect Signature

Groups effect operations in a type

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

Effect Operations

Semantics of the operations is left open

Effect Handlers

Provide semantics to effect operations

```
scoped { s₂ }

> Forall(x => Implies(Woman(x), Said(John, Loves(x, John))))
```

Example 3: The Implicature Effect

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

Example 3: The Implicature Effect

```
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

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

pete saidQuote { accommodate { s₃ } }
```

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)))
```

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

```
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)))
```

Effectful Syntax

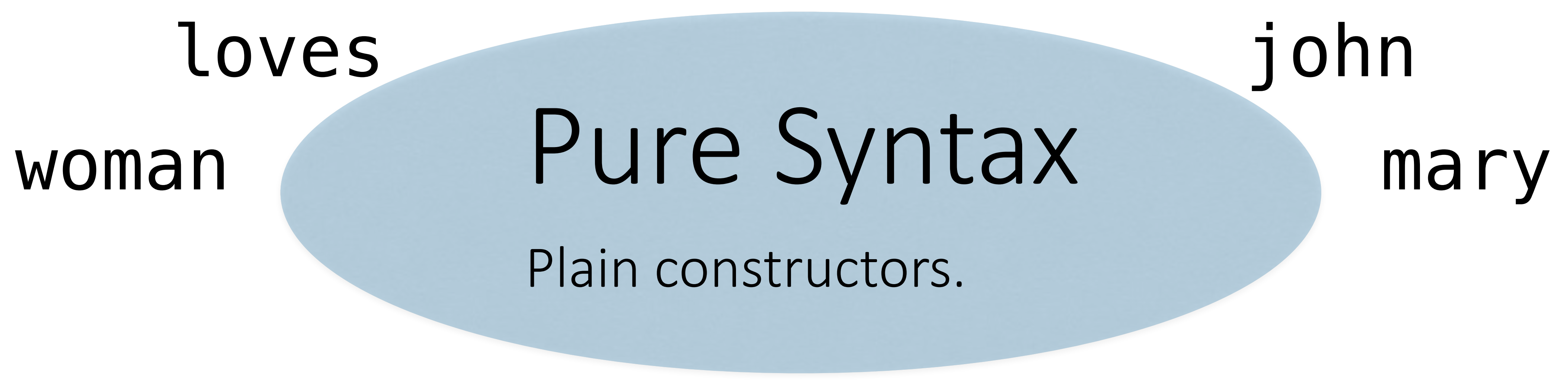
Use (algebraic) effects for AST construction.

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

Effectful Syntax

Use (algebraic) effects for AST construction.

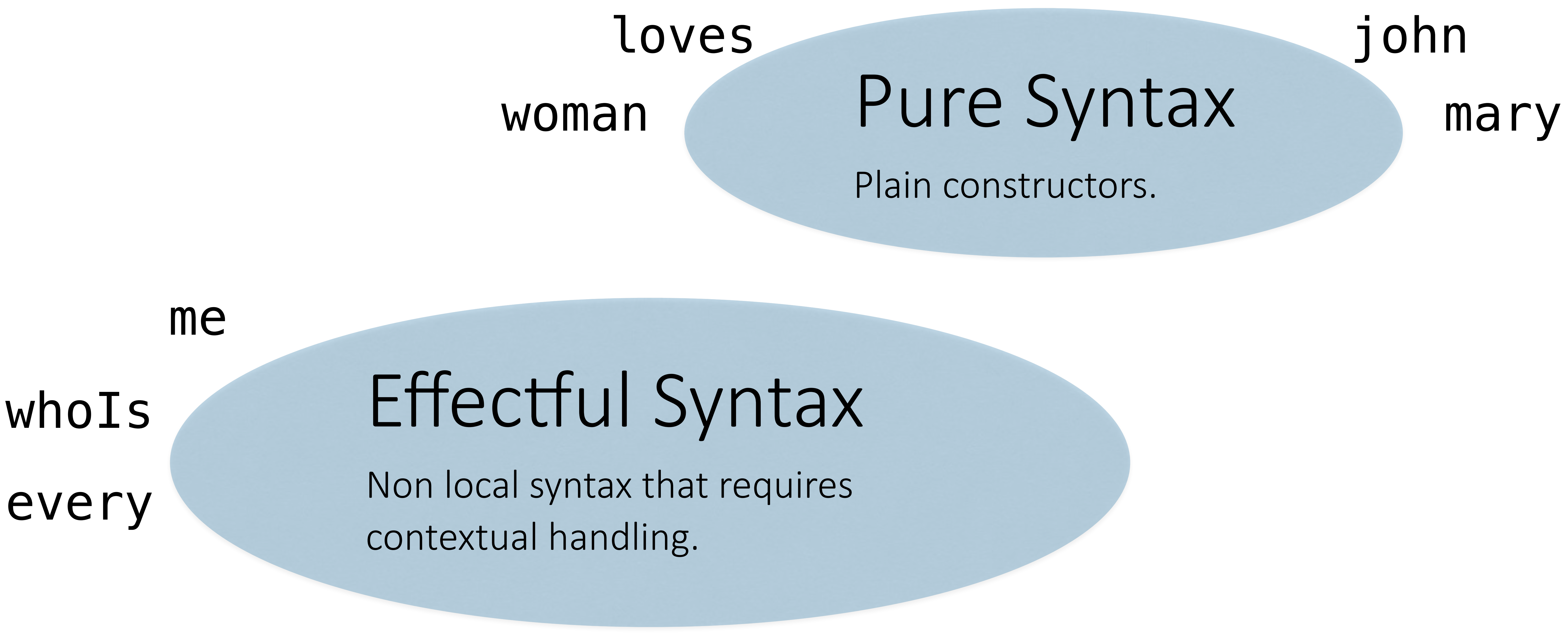
More precisely, we propose to group syntax elements of DSLs into:



Effectful Syntax

Use (algebraic) effects for AST construction.

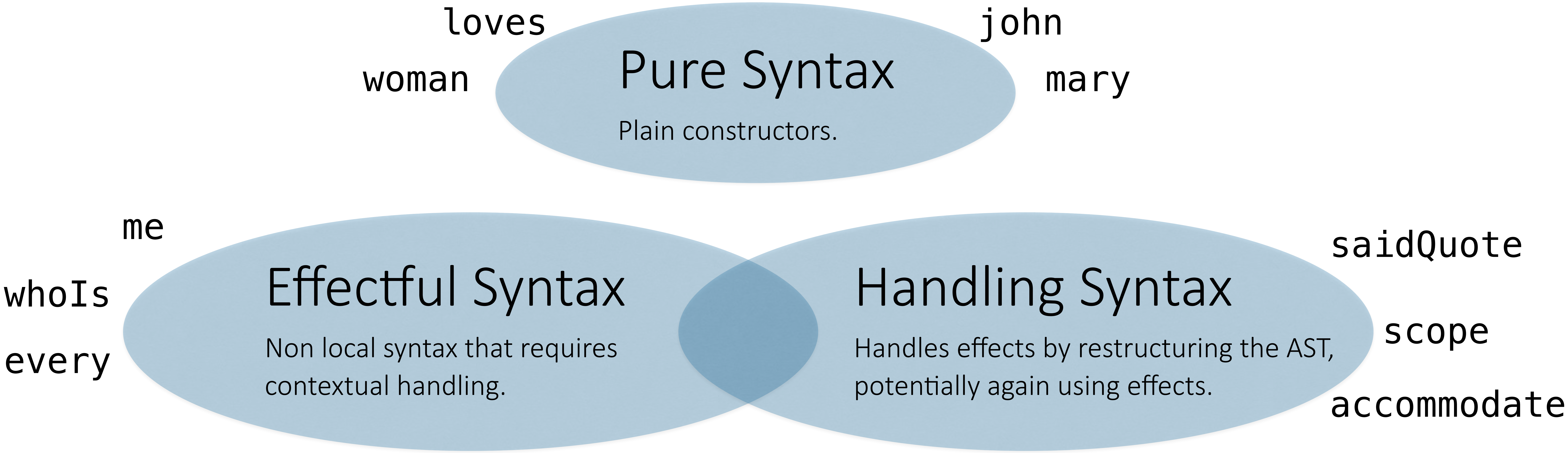
More precisely, we propose to group syntax elements of DSLs into:



Effectful Syntax

Use (algebraic) effects for AST construction.

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



Properties

Effectful syntax based on algebraic effects is...

Properties

Effectful syntax based on algebraic effects is...

Modular
Linguistic phenomena
can be encapsulated
into reusable modules.

Properties

Effectful syntax based on algebraic effects is...

Modular
Linguistic phenomena can be encapsulated into reusable modules.

Effectful Syntax
Handling Syntax

Pure Syntax

Properties

Effectful syntax based on algebraic effects is...

Modular
Linguistic phenomena can be encapsulated into reusable modules.

Effectful Syntax
Handling Syntax

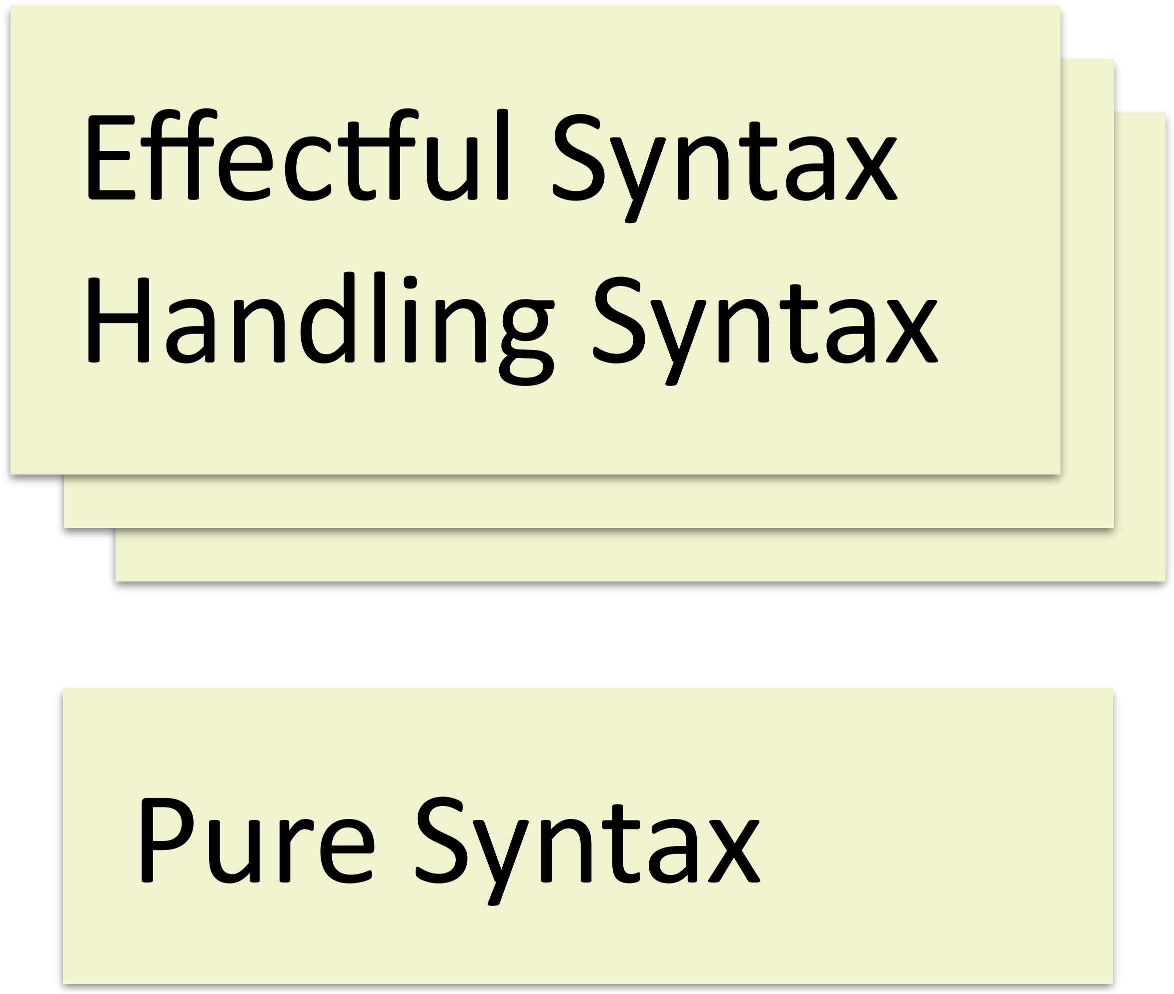
Pure Syntax

Learnable
Separating linguistic phenomena from other domain concepts allows separate understanding

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.