Staged Abstract Interpreters

Guannan Wei, Yuxuan Chen, and Tiark Rompf
Purdue University

OOPSLA 2019
Athens, Greece

From Semantics to *Correct* Tools

concrete
semantics

From Semantics to *Correct* Tools

abstract
semantics

concrete
semantics

- *Abstract interpretation* [POPL '77, '79]

  A theory of approximation – design sound static analysis from concrete semantics

From Semantics to *Correct* Tools

abstract
interpreters

concrete
interpreters

- *Abstract interpretation* [POPL '77, '79]

  A theory of approximation – design sound static analysis from concrete semantics

- *Abstracting Abstract Machines* [ICFP '10]
  Abstracting Definitional Interpreters [ICFP '17]

  Deriving semantic artifacts for abstract interpretation from concrete semantic artifacts

From Semantics to *Correct* Tools

abstract
interpreters

concrete
interpreters

- *Abstract interpretation* [POPL ‘77, ‘79]

  A theory of approximation – design sound static analysis from concrete semantics

- *Abstracting Abstract Machines* [ICFP ‘10]
  Abstracting Definitional Interpreters [ICFP ‘17]

  Deriving semantic artifacts for abstract interpretation from concrete semantic artifacts

- *Easy to construct, but not efficient*

From Semantics to *Efficient* Tools

From Semantics to *Efficient* Tools

- *Futamura Projections* [1971]

  Deriving compilers from interpreters, via a hierarchy of specializations

From Semantics to *Efficient* Tools

- *Futamura Projections* [1971]

  Deriving compilers from interpreters, via a hierarchy of specializations

- *Multi-stage Programming*

  Allows programmers to control which part of the program will be specialized

From Semantics to *Efficient* Tools

- *Futamura Projections* [1971]

  Deriving compilers from interpreters, via a hierarchy of specializations

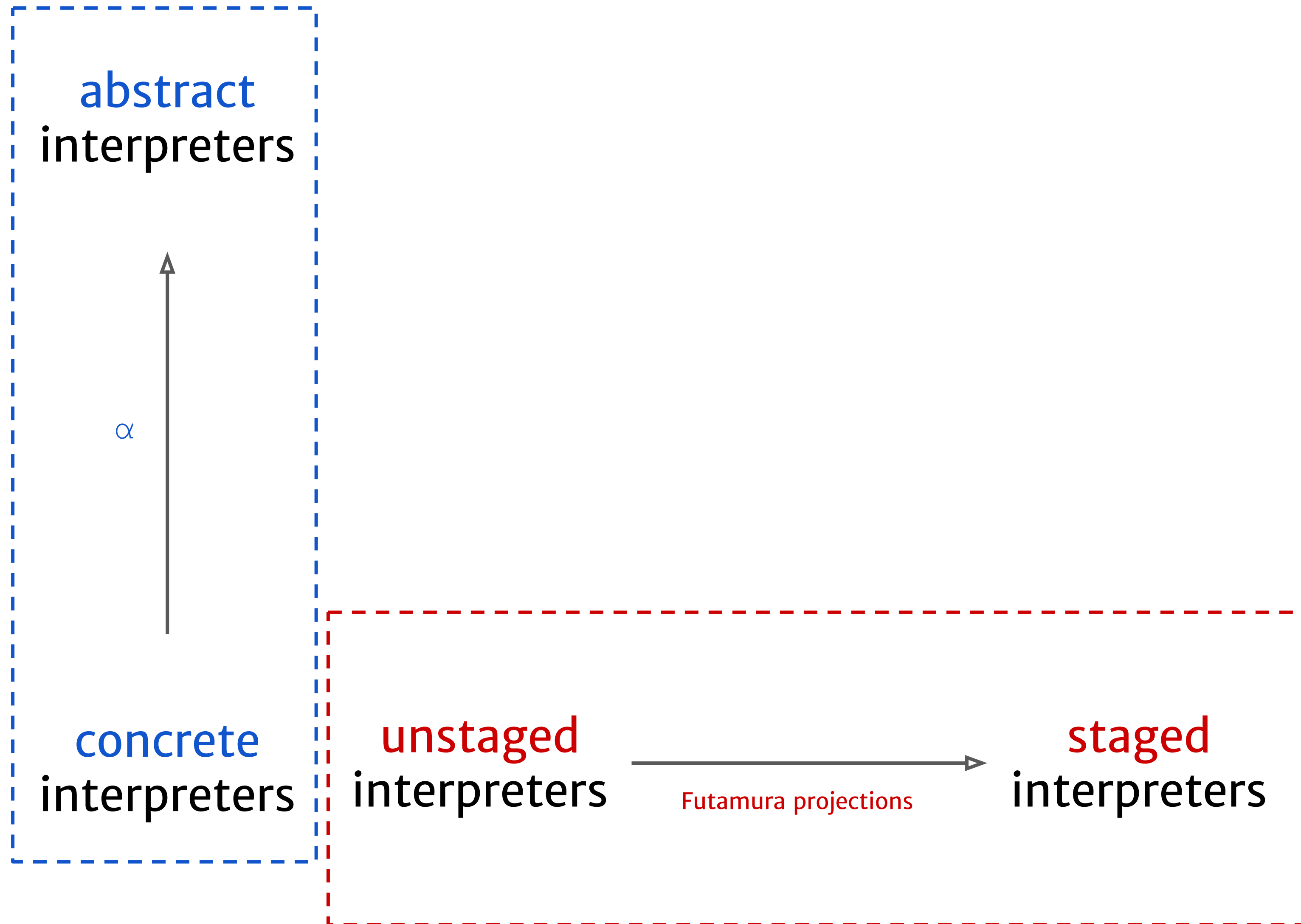
- *Multi-stage Programming*

  Allows programmers to control which part of the program will be specialized

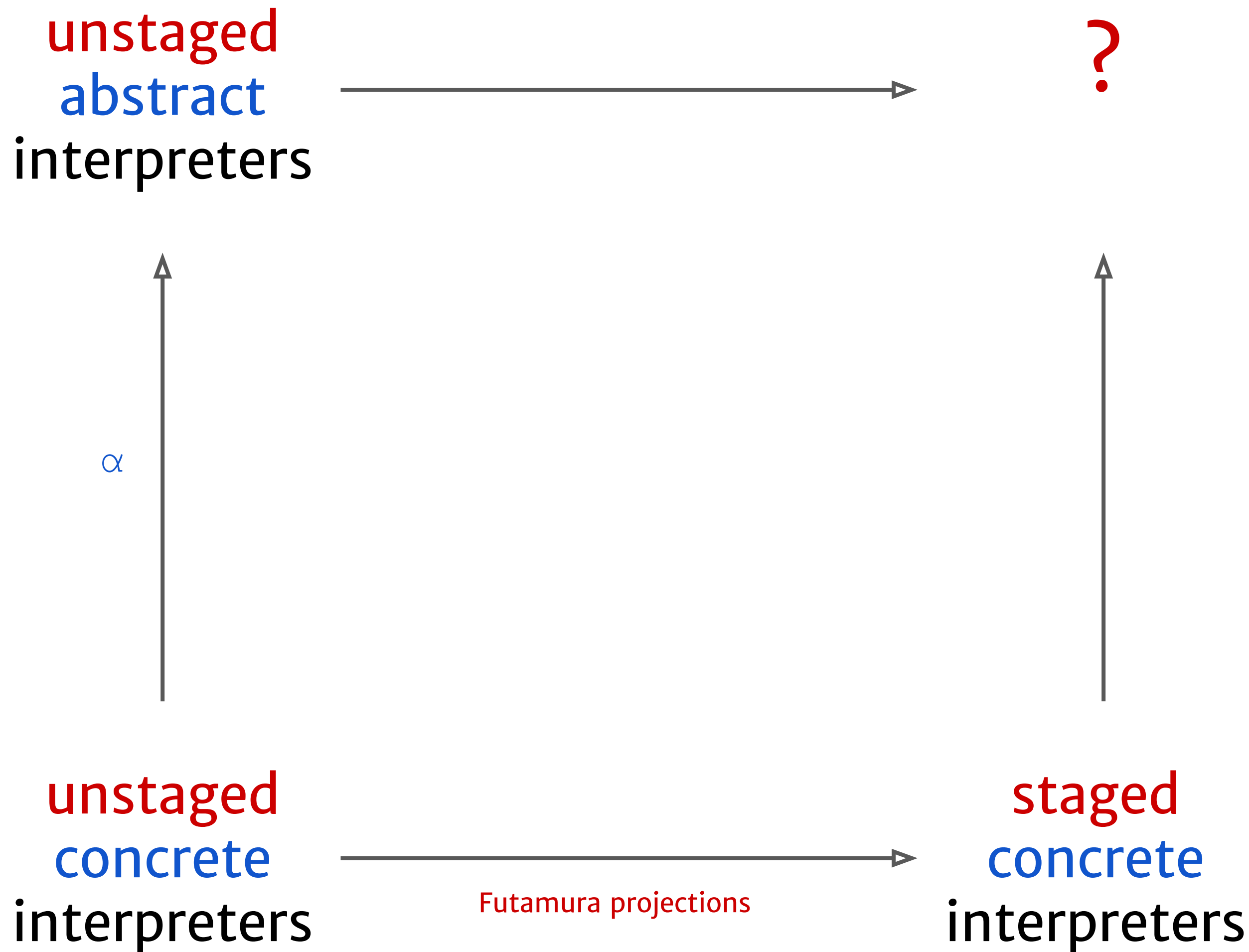
- *Examples*

  SQL query compilers [SIGMOD ‘18], etc.

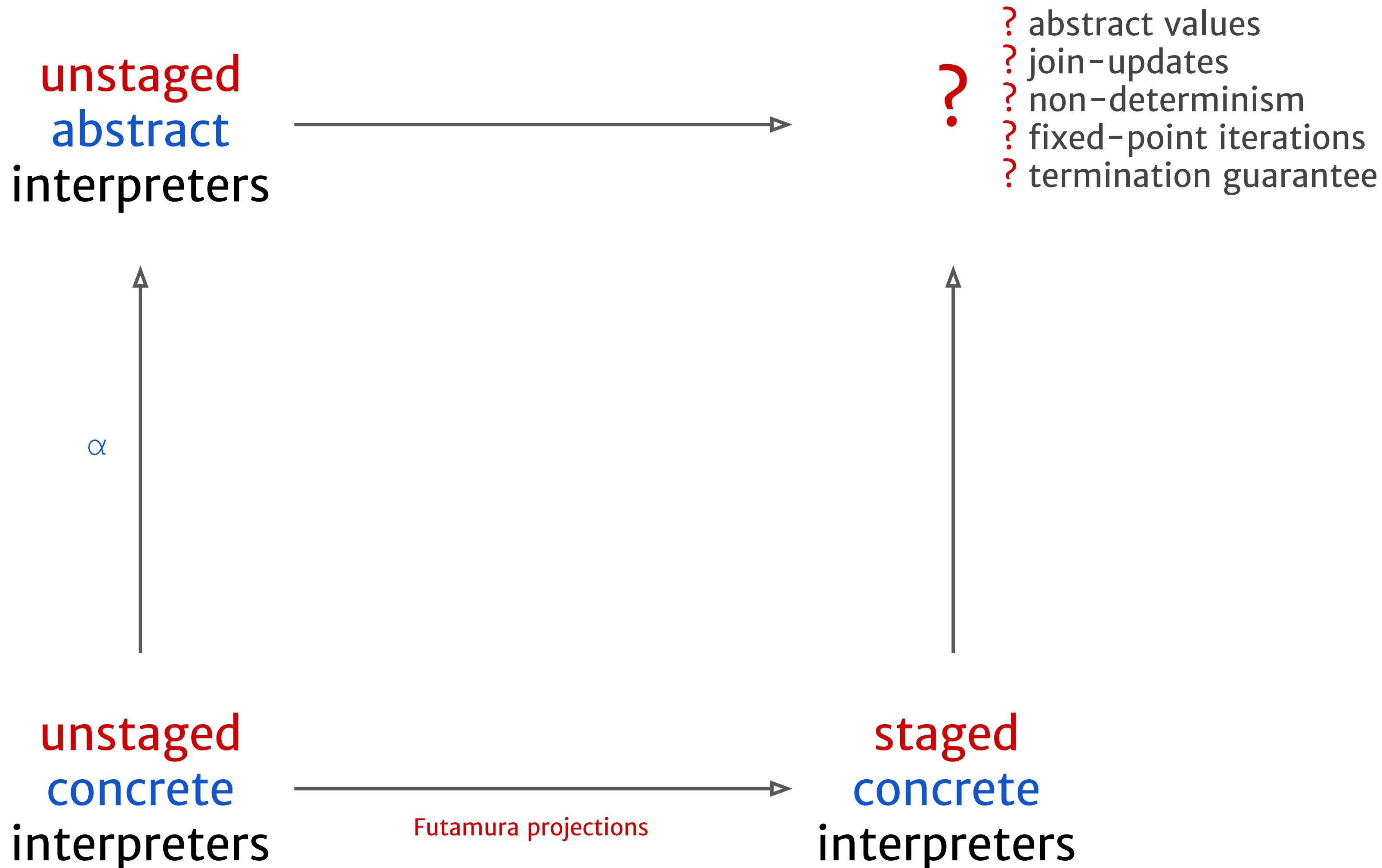
From Semantics to *Efficient* and *Correct* Tools



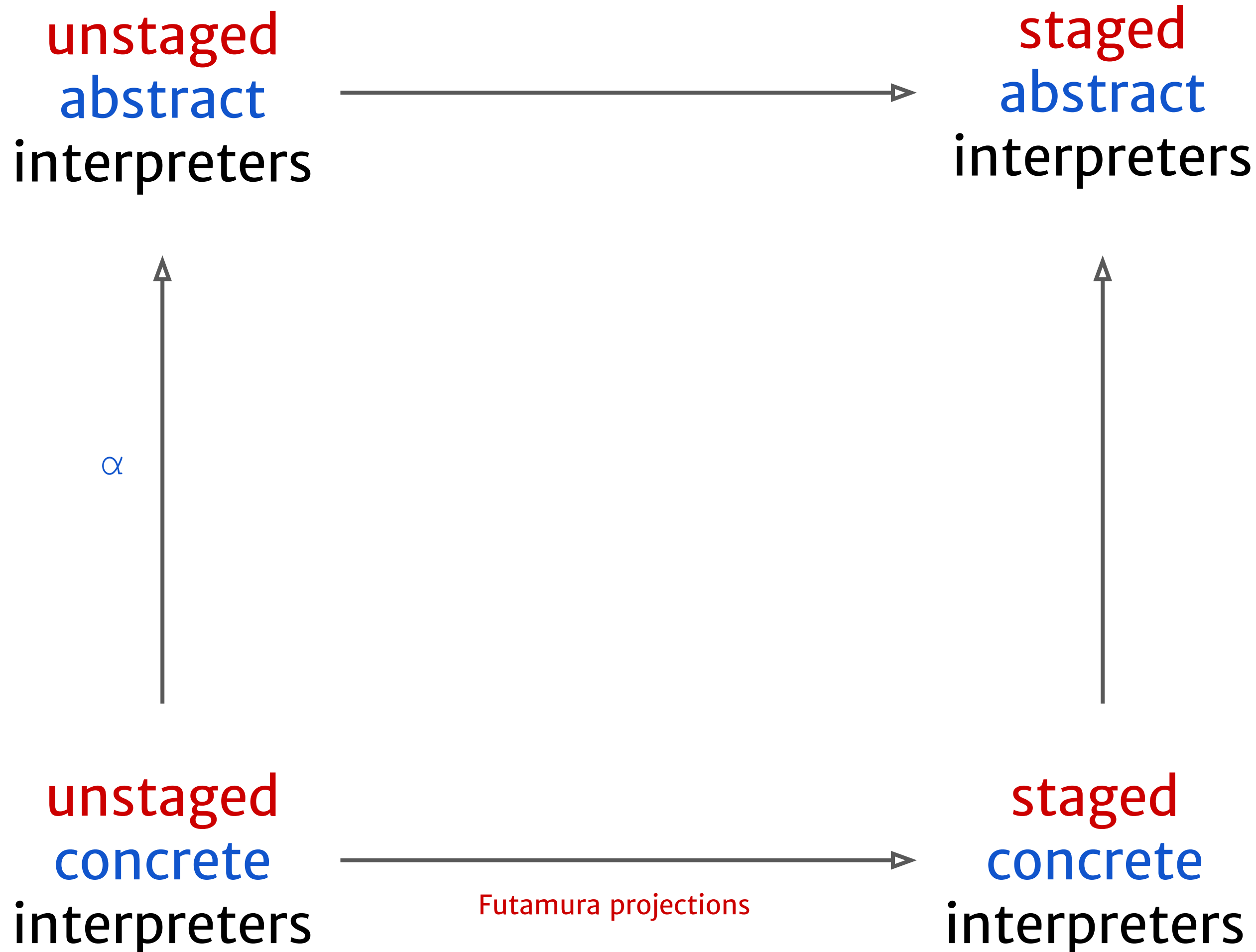
From Semantics to *Efficient* and *Correct* Tools



From Semantics to *Efficient* and *Correct* Tools



From Semantics to *Efficient* and *Correct* Tools



Overview: Staged Abstract Interpreters

This paper: 1st Futamura Projection of abstract interpreters

Specializing an abstract interpreter to an input program within a multi-stage programming framework.

Overview: Staged Abstract Interpreters

This paper: 1st Futamura Projection of abstract interpreters

Specializing an abstract interpreter to an input program within a multi-stage programming framework.

Target language: a small higher-order functional language

Baseline: a monadic abstract definitional interpreter [ICFP '17]

Host language: Scala + Lightweight Modular Staging (LMS) framework [GPCE '10]

Result: generating low-level code with no monads and no interpretation overhead

Overview: Staged Abstract Interpreters

This paper: 1st Futamura Projection of abstract interpreters

Specializing an abstract interpreter to an input program within a multi-stage programming framework.

How?

Overview: Staged Abstract Interpreters

This paper: 1st Futamura Projection of abstract interpreters

Specializing an abstract interpreter to an input program within a multi-stage programming framework.

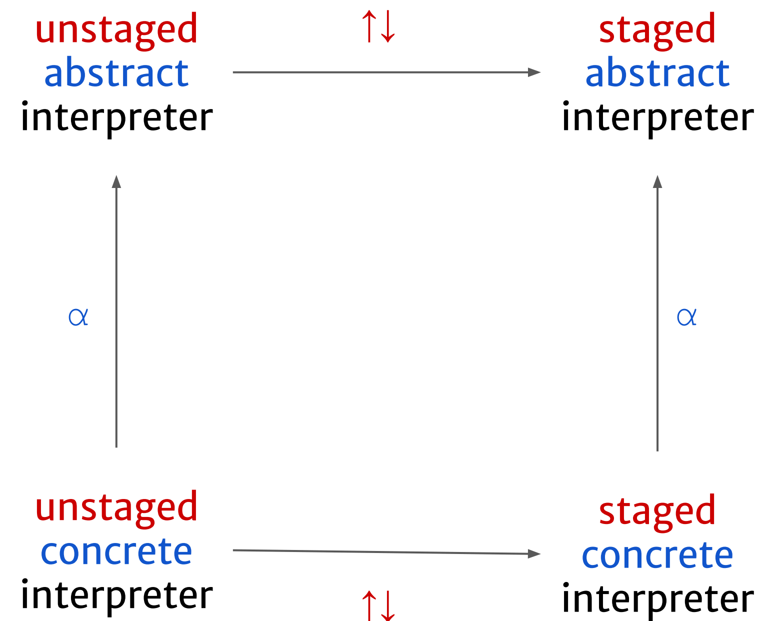
How?
Building a generic definitional interpreter that *abstracts over both value domains and binding times using monads*

Overview: Staged Abstract Interpreters

This paper: 1st Futamura Projection of abstract interpreters

Specializing an abstract interpreter to an input program within a multi-stage programming framework.

How?
Building a generic definitional interpreter that *abstracts over both value domains and binding times using monads*, deriving the four semantic artifacts modularly.



Multi-stage Programming

- *Meta-programming*: write programs that manipulate or generate other programs
- *Multi-stage programming*: write programs with stage annotations, which allows program specialization + type-safe runtime code generation

Multi-stage Programming

- *Meta-programming*: write programs that manipulate or generate other programs
- *Multi-stage programming*: write programs with stage annotations, which allows program specialization + type-safe runtime code generation
- *An example*: power(b, x) ≡ b^x

Multi-stage Programming

- *Meta-programming*: write programs that manipulate or generate other programs
- *Multi-stage programming*: write programs with stage annotations, which allows program specialization + type-safe runtime code generation
- *An example*: power(b, x) ≡ b^x + Lightweight Modular Staging

Multi-stage Programming

- *Meta-programming*: write programs that manipulate or generate other programs
- *Multi-stage programming*: write programs with stage annotations, which allows program specialization + type-safe runtime code generation
- *An example*: power(b, x) ≡ b^x + Lightweight Modular Staging

```
def power(b: ℕ, x: ℕ): ℕ =
if (x == 0) 1 else b * power(b, x - 1)
```

Multi-stage Programming

- *Meta-programming*: write programs that manipulate or generate other programs
- *Multi-stage programming*: write programs with stage annotations, which allows program specialization + type-safe runtime code generation
- *An example*: power(b, x) ≡ b^x + Lightweight Modular Staging

```
def power(b: ℕ, x: ℕ): ℕ =
if (x == 0) 1 else b * power(b, x - 1)

def power5(b: ℕ) = power(b, 5)
```

Multi-stage Programming

- *Meta-programming*: write programs that manipulate or generate other programs
- *Multi-stage programming*: write programs with stage annotations, which allows program specialization + type-safe runtime code generation
- *An example*: power(b, x) ≡ b^x + Lightweight Modular Staging

```
def power(b: Rep[ℕ], x: ℕ): Rep[ℕ] =
if (x == 0) 1 else b * power(b, x - 1)
```

Multi-stage Programming

- *Meta-programming*: write programs that manipulate or generate other programs
- *Multi-stage programming*: write programs with stage annotations, which allows program specialization + type-safe runtime code generation
- *An example*: power(b, x) ≡ b^x + Lightweight Modular Staging

```
def power(b: Rep[ℕ], x: ℕ): Rep[ℕ] =
if (x == 0) 1 else b * power(b, x - 1)

def power5(b: Rep[ℕ]) = power(b, 5)
```

Multi-stage Programming

- *Meta-programming*: write programs that manipulate or generate other programs
- *Multi-stage programming*: write programs with stage annotations, which allows program specialization + type-safe runtime code generation
- *An example*: power(b, x) ≡ b^x + Lightweight Modular Staging

```
def power(b: Rep[ℕ], x: ℕ): Rep[ℕ] =
if (x == 0) 1 else b * power(b, x - 1)


def power5(b: Rep[ℕ]) = power(b, 5)
```

specialization/code generation

```
def power5(b: ℕ) = b * b * b * b * b
```

Multi-stage Programming

- *Meta-programming*: write programs that manipulate or generate other programs
- *Multi-stage programming*: write programs with stage annotations, which allows program specialization + type-safe runtime code generation
- How does it relate to interpreter specialization?

Multi-stage Programming

- *Meta-programming*: write programs that manipulate or generate other programs
- *Multi-stage programming*: write programs with stage annotations, which allows program specialization + type-safe runtime code generation
- How does it relate to interpreter specialization?

```
def eval(e: Exp, ρ: Env, σ: Store): (Value, Store)
```

Multi-stage Programming

- *Meta-programming*: write programs that manipulate or generate other programs
- *Multi-stage programming*: write programs with stage annotations, which allows program specialization + type-safe runtime code generation
- How does it relate to interpreter specialization?

```
def eval(e: Exp, ρ: Env, σ: Store): (Value, Store)
```

```
def eval(e: Exp, ρ: Rep[Env], σ: Rep[Store]):
    Rep[(Value, Store)]
```

Multi-stage Programming

- *Meta-programming*: write programs that manipulate or generate other programs
- *Multi-stage programming*: write programs with stage annotations, which allows program specialization + type-safe runtime code generation
- How does it relate to interpreter specialization?

```
def eval(e: Exp, ρ: Env, σ: Store): (Value, Store)
```

```
def eval(e: Exp, ρ: Rep[Env], σ: Rep[Store]):
        Rep[(Value, Store)]
```

Given a program e ,

$\texttt{eval}_e(\rho, \sigma)$

Multi-stage Programming

- *Meta-programming*: write programs that manipulate or generate other programs
- *Multi-stage programming*: write programs with stage annotations, which allows program specialization + type-safe runtime code generation
- How does it relate to interpreter specialization?

```
def eval(e: Exp, ρ: Env, σ: Store): (Value, Store)
```

```
def eval(e: Exp, ρ: Rep[Env], σ: Rep[Store]):
    Rep[(Value, Store)]
```

Given a program `e`, such that ∀ `ρ σ`,

$$\mathtt{eval}_e(\rho, \sigma) \equiv \mathtt{eval}(e, \rho, \sigma)$$

Generic Definitional Interpreter

- Abstract over value domain types
- Abstract over binding-time types [GPCE '17]

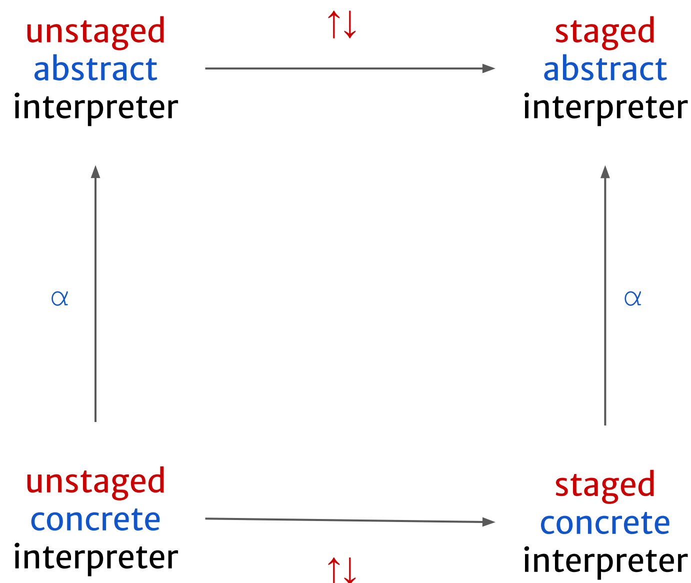
```
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)

    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)

    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```

Generic Definitional Interpreter

- Abstract over value domain types
- Abstract over binding-time types [GPCE '17]
- Derive 4 different interpreters without changing the generic definitional interpreter




```
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)

    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)

    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```

Generic Definitional Interpreter

- Abstract over value domain types
- Abstract over binding-time types [GPCE '17]

```
type Value
```

```
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)

    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)

    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```

Generic Definitional Interpreter

- Abstract over value domain types
- Abstract over binding-time types [GPCE '17]

```
type Value
```

- Type Ans is an abstract monad type:

```
type Ans = AnsM[Value]
```

```
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)

    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)

    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```

Generic Definitional Interpreter

- Abstract over value domain types
- Abstract over binding-time types [GPCE '17]

```
type Value
```

- Type Ans is an abstract monad type:

```
type Ans = AnsM[Value]
type AnsM[T] <: MonadOps[AnsM, T]
type MonadOps[M[_], A] = {
  def flatMap[B](f: A ⇒ M[B]): M[B]
  def map[B](f: A ⇒ B): M[B]
}
```

```
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)

    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)

    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```

Generic Definitional Interpreter

- Abstract over value domain types
- Abstract over binding-time types [GPCE '17]

```
type Value
```

- Type Ans is an abstract monad type:

```
type Ans = AnsM[Value]
type AnsM[T] <: MonadOps[AnsM, T]
type MonadOps[M[_], A] = {
  def flatMap[B](f: A ⇒ M[B]): M[B]
  def map[B](f: A ⇒ B): M[B]
}
```

- Primitive operations (abstract methods):

```
def close(ev: Expr ⇒ Ans)(λ: Lam, ρ: Env): Value
def ap_clo(ev: Expr ⇒ Ans)(fun: Value, arg: Value): Ans
...
```

```
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)

    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)

    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```

Generic Definitional Interpreter

- Abstract over value domain types
- Abstract over binding-time types [GPCE '17]

```
type Value
type R[_] // * -> *
```

```
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)

    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)

    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```

Generic Definitional Interpreter

- Abstract over value domain types
- Abstract over binding-time types [GPCE '17]

```
type Value
type R[_] // * -> *

// interpreter - flat binding time:
type R[T] = T
// compiler - next-stage annotation
type R[T] = Rep[T]
```

```
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)

    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)

    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```

Generic Definitional Interpreter

- Abstract over value domain types
- Abstract over binding-time types [GPCE ‘17]

```
type Value
type R[_] // * -> *
```

- Type Ans is an abstract *stage-polymorphic* monad type:

```
type Ans = AnsM[Value]
type AnsM[T] <: MonadOps[R, AnsM, T]
type MonadOps[R[_], M[_], A] = {
  def flatMap[B](f: R[A] ⇒ M[B]): M[B]
  def map[B](f: R[A] ⇒ R[B]): M[B]
}
```

```
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)

    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)

    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```

Generic Definitional Interpreter

- Abstract over value domain types
- Abstract over binding-time types [GPCE '17]

```
type Value
type R[_] // * -> *
```

- Type Ans is an abstract *stage-polymorphic* monad type:

```
type Ans = AnsM[Value]
type AnsM[T] <: MonadOps[R, AnsM, T]
type MonadOps[R[_], M[_], A] = {
  def flatMap[B](f: R[A] ⇒ M[B]): M[B]
  def map[B](f: R[A] ⇒ R[B]): M[B]
}
```

- *Stage-polymorphic* primitive operations:

```
def close(ev: Expr ⇒ Ans)(λ: Lam, ρ: R[Env]): R[Value]
...
```

```
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)

    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)

    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```

Generic Definitional Interpreter

- Abstract over value domain types
- Abstract over binding-time types [GPCE '17]

```
type Ans = AnsM[Value]
type AnsM[T] <: MonadOps[R, AnsM, T]
type MonadOps[R[_], M[_], A] = {
  def flatMap[B](f: R[A] ⇒ M[B]): M[B]
  def map[B](f: R[A] ⇒ R[B]): M[B]
}
```

Key idea: *we stage the data that manipulated by the monads; we do not stage the monad objects.*

```
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)

    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)

    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```

Unstaged Concrete Interpreter

```
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)

    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)

    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```

- Values

```
trait Value
case class CloV(λ: Lam, ρ: Env)
  extends Value
```

- Flat binding-time type

```
type R[T] = T
```

- Monads / Monad transformers [POPL '95]

```
type AnsM[T] = ReaderT[StateT[IdM, Store, ?], Env, T]
```

Staged Concrete Interpreter

```
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)

    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)

    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```

- Next-stage values

  Staged closures are compiled next-stage functions
  `Rep[((Value, Store)) ⇒ (Value, Store)]`

- Two-level binding-times

  `type R[T] = Rep[T]`

- Monads *manipulating staged data*

  `type AnsM[T] = RepReaderT[RepStateT[RepIdM, Store, ?], Env, T]`

Unstaged *Abstract* Interpreter

```
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)

    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)

    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```

- Abstract values

```
trait RawValue
case class CloV(λ: Lam, ρ: Env)
  extends RawValue
type Value = Set[RawValue]
```

- Flat binding-time type

```
type R[T] = T
```

- Monads for big-step abstract semantics [ICFP '17]

```
type AnsM[T] =
  ReaderT[StateT[NondetT[CacheM, ?], Store, ?], Env, T]
```

Staged *Abstract* Interpreter

```scala
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)

    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)

    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```

- Next-stage abstract values

```scala
type Value = Set[CompiledClo]
```

- Two-level binding-times

```scala
type R[T] = Rep[T]
```

- Monads *manipulating staged data* for big-step abstract semantics

```scala
type AnsM[T] =
  RepReaderT[RepStateT[RepNondetT[RepCacheM, ?], Store, ?], Env, T]
```

What does the generated code look like?

What does the generated code look like?

```
(define (map xs f) (if (null? xs) '() (cons (f (car xs)) (map (cdr xs) f))))
```

What does the generated code look like?

```scheme
(define (map xs f) (if (null? xs) '() (cons (f (car xs)) (map (cdr xs) f))))
```

```scala
val x39 = {(x40:Set[AbsValue]],x41:Map[Addr,Set[AbsValue]],x42:Map[Config,Set[ValStore]],x43:Map[Config,Set[ValStore]]) =>
val x47 = x43
val x45 = x41
val x44 = x40
val x51 = x50.zip(x44)
val x61 = x51.foldLeft (x45) { case (x52, x53) =>
val x54 = x53._1
val x55 = x53._2
val x58 = x52.getOrElse(x54, x57)
val x59 = x58.union(x55)
val x60 = x52 + (x54 -> x59)
x60
}
val x76 = new Tuple3[Int,Env,Map[Addr,Set[AbsValue]]](1359862071,x49,x61)
val x82 = x47.contains(x76)
def x1105_then() = {
val x83 = x47(x76)
val x84 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x83,x47)
x84
}
def x1105_else() = {
val x46 = x42
val x85 = x46.getOrElse(x76, x6)
val x86 = x47 + (x76 -> x85)
val x98 = new Tuple3[Int,Env,Map[Addr,Set[AbsValue]]](1508898119,x49,x61)
val x104 = x86.contains(x98)
def x144_then() = {
val x105 = x86(x98)
val x106 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x105,x86)
x106
}
def x144_else() = {
val x133 = new Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]](x125,x61)
val x134 = Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]](x133)
val x107 = x46.getOrElse(x98, x6)
val x108 = x86 + (x98 -> x107)
val x137 = x108.getOrElse(x98, x6)
val x138 = x137.union(x134)
val x139 = x108 + (x98 -> x138)
val x143 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x134,x139)
x143
}
val x144 = if (x104) x144_then() else x144_else()
val x151 = x144._1
val x152 = x144._2
val x157 = new Tuple2[Set[Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x156,x152)
val x165 = x151.foldLeft (x157) { case ((x158, x159), x160) =>
val x161 = new Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]](x160,x61)
val x162 = Set[Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]]](x161)
val x163 = x158 ++ x162
val x164 = new Tuple2[Set[Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x163,x159)
x164
}
val x166 = x165._1
val x167 = x165._2
val x168 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x6,x167)
val x184 = x166.foldLeft (x168) { case ((x169, x170), x171) =>
val x172 = x171._1
val x174 = x172._1
val x175 = x172._2
val x179 = new Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]](x174,x175)
val x180 = Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]](x179)
val x182 = x169 ++ x180
val x183 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x182,x170)Map[Addr,Set[AbsValue]]
x183
}
val x185 = x184._1
val x186 = x184._2
val x188 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x6,x186)
val x1074 = x185.foldLeft (x188) { case ((x189, x190), x191) =>
val x193 = x191._2
val x278 = new Tuple3[Int,Env,Map[Addr,Set[AbsValue]]](756624586,x49,x193)
val x283 = x190.contains(x278)
def x1009_then() = {
val x284 = x190(x278)
val x285 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x284,x190)
x285
}
def x1009_else() = {
val x286 = x46.getOrElse(x278, x6)
```

```scala
val x287 = x190 + (x278 -> x286)
val x297 = new Tuple3[Int,Env,Map[Addr,Set[AbsValue]]](964775054,x49,x193)
val x303 = x287.contains(x297)
def x554_then() = {
val x304 = x287(x297)
val x305 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x304,x287)
x305
}
def x554_else() = {
val x306 = x46.getOrElse(x297, x6)
val x307 = x287 + (x297 -> x306)
val x321 = new Tuple3[Int,Env,Map[Addr,Set[AbsValue]]](1901772330,x49,x193)
val x327 = x307.contains(x321)
def x355_then() = {
val x328 = x307(x321)
val x329 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x328,x307)
x329
}
def x355_else() = {
val x344 = new Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]](x130,x193)
val x345 = Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]](x344)
val x330 = x46.getOrElse(x321, x6)
val x331 = x307 + (x321 -> x330)
val x348 = x331.getOrElse(x321, x6)
val x349 = x348.union(x345)
val x350 = x331 + (x321 -> x349)
val x354 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x345,x350)
x354
}
val x355 = if (x327) x355_then() else x355_else()
val x361 = x355._1
val x362 = x355._2
val x366 = new Tuple2[Set[Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x156,x362)
val x374 = x361.foldLeft (x366) { case ((x367, x368), x369) =>
val x370 = new Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]](x369,x193)
val x371 = Set[Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]]](x370)
val x372 = x367 ++ x371
val x373 = new Tuple2[Set[Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x372,x368)
x373
}
val x375 = x374._1
val x376 = x374._2
val x377 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x6,x376)
val x393 = x375.foldLeft (x377) { case ((x378, x379), x380) =>
val x381 = x380._1
val x383 = x381._1
val x384 = x381._2
val x388 = new Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]](x383,x384)
val x389 = Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]](x388)
val x391 = x378 ++ x389
val x392 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x391,x379)
x392
}
val x394 = x393._1
val x395 = x393._2
val x397 = new Tuple2[Set[Tuple2[Set[AbsValue]],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x116,x395)
val x412 = x394.foldLeft (x397) { case ((x398, x399), x400) =>
val x402 = x400._2
val x401 = x400._1
val x406 = List[Set[AbsValue]](x401)
val x407 = new Tuple2[Set[AbsValue]],Map[Addr,Set[AbsValue]]](x406,x402)
val x408 = Set[Tuple2[Set[AbsValue]],Map[Addr,Set[AbsValue]]]](x407)
val x410 = x398 ++ x408
val x411 = new Tuple2[Set[Tuple2[Set[AbsValue]],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x410,x399)
x411
}
val x413 = x412._1
val x414 = x412._2
val x415 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x6,x414)
val x311 = x193(ZCFAAddr("var3"))
val x522 = x413.foldLeft (x415) { case ((x416, x417), x418) =>
val x427 = x311.filter { x424 =>
val x425 = x424.isInstanceOf[CompiledClo]
x425
}
val x429 = new Tuple2[Set[Tuple2[AbsValue,Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x428,x417)
val x420 = x418._2
val x437 = x427.foldLeft (x429) { case ((x430, x431), x432) =>
val x433 = new Tuple2[AbsValue,Map[Addr,Set[AbsValue]]](x432,x420)
val x434 = Set[Tuple2[AbsValue,Map[Addr,Set[AbsValue]]]](x433)
val x435 = x430 ++ x434
...
```

What does the generated code look like?

```
(define (map xs f) (if (null? xs) '() (cons (f (car xs)) (map (cdr xs) f))))
```

```
val x39 = {(x40:Set[AbsValue]],x41:Map[Addr,Set[AbsValue]],x42:Map[Config,Set[ValStore]],x43:Map[Config,Set[ValStore]]) =>
val x47 = x43
val x45 = x41
val x44 = x40
val x51 = x50.zip(x44)
val x61 = x51.foldLeft (x45) { case (x52, x53) =>
val x54 = x53._1
val x55 = x53._2
val x58 = x52.getOrElse(x54, x57)
val x59 = x58.union(x55)
val x60 = x52 + (x54 -> x59)
x60
}
val x76 = new Tuple3[Int,Env,Map[Addr,Set[AbsValue]]](1359862071,x49,x61)
val x82 = x47.contains(x76)
def x1105_then() = {
val x83 = x47(x76)
val x84 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x83,x47)
}
def x1105_else() = {
val x46 = x42
val x85 = x46.getOrElse(x76, x6)
val x86 = x47 + (x76 -> x85)
val x98 = new Tuple3[Int,Env,Map[Addr,Set[AbsValue]]](1508898119,x49,x61)
val x104 = x86.contains(x98)
def x144_then() = {
val x105 = x86(x98)
val x106 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x105,x86)
x106
}
def x144_else() = {
val x133 = new Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]](x125,x61)
val x134 = Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]](x133)
val x107 = x46.getOrElse(x98, x6)
val x108 = x86 + (x98 -> x107)
val x137 = x108.getOrElse(x98, x6)
val x138 = x137.union(x134)
val x139 = x108 + (x98 -> x138)
val x143 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x134,x139)
x143
}
val x144 = if (x104) x144_then() else x144_else()
val x151 = x144._1
val x152 = x144._2
val x157 = new Tuple2[Set[Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x156,x152)
val x165 = x151.foldLeft (x157) { case ((x158, x159), x160) =>
val x161 = new Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]](x160,x61)
val x162 = Set[Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]]](x161)
val x163 = x158 ++ x162
val x164 = new Tuple2[Set[Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x163,x159)
x164
}
val x166 = x165._1
val x167 = x165._2
val x168 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x6,x167)
val x184 = x166.foldLeft (x168) { case ((x169, x170), x171) =>
val x172 = x171._1
val x174 = x172._1
val x175 = x172._2
val x179 = new Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]](x174,x175)
val x180 = Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]](x179)
val x182 = x169 ++ x180
val x183 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x182,x170)Map[Addr,Set[AbsValue]]
x183
}
val x185 = x184._1
val x186 = x184._2
val x188 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x6,x186)
val x1074 = x185.foldLeft (x188) { case ((x189, x190), x191) =>
val x193 = x191._2
val x278 = new Tuple3[Int,Env,Map[Addr,Set[AbsValue]]](756624586,x49,x193)
val x283 = x190.contains(x278)
def x1009_then() = {
val x284 = x190(x278)
val x285 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x284,x190)
x285
}
def x1009_else() = {
val x286 = x46.getOrElse(x278, x6)
val x287 = x190 + (x278 -> x286)
val x297 = new Tuple3[Int,Env,Map[Addr,Set[AbsValue]]](964775054,x49,x193)
val x303 = x287.contains(x297)
def x554_then() = {
val x304 = x287(x297)
val x305 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x304,x287)
x305
}
def x554_else() = {
val x306 = x46.getOrElse(x297, x6)
val x307 = x287 + (x297 -> x306)
val x321 = new Tuple3[Int,Env,Map[Addr,Set[AbsValue]]](1901772330,x49,x193)
val x327 = x307.contains(x321)
def x355_then() = {
val x328 = x307(x321)
val x329 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x328,x307)
x329
}
def x355_else() = {
val x344 = new Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]](x130,x193)
val x345 = Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]](x344)
val x330 = x46.getOrElse(x321, x6)
val x331 = x307 + (x321 -> x330)
val x348 = x331.getOrElse(x321, x6)
val x349 = x348.union(x345)
val x350 = x331 + (x321 -> x349)
val x354 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x345,x350)
x354
}
val x355 = if (x327) x355_then() else x355_else()
val x361 = x355._1
val x362 = x355._2
val x366 = new Tuple2[Set[Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x156,x362)
val x374 = x361.foldLeft (x366) { case ((x367, x368), x369) =>
val x370 = new Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]](x369,x193)
val x371 = Set[Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]]](x370)
val x372 = x367 ++ x371
val x373 = new Tuple2[Set[Tuple2[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x372,x368)
x373
}
val x375 = x374._1
val x376 = x374._2
val x377 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x6,x376)
val x393 = x375.foldLeft (x377) { case ((x378, x379), x380) =>
val x381 = x380._1
val x383 = x381._1
val x384 = x381._2
val x388 = new Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]](x383,x384)
val x389 = Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]](x388)
val x391 = x378 ++ x389
val x392 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x391,x379)
x392
}
val x394 = x393._1
val x395 = x393._2
val x397 = new Tuple2[Set[Tuple2[Set[AbsValue]],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x116,x395)
val x412 = x394.foldLeft (x397) { case ((x398, x399), x400) =>
val x402 = x400._2
val x401 = x400._1
val x406 = List[Set[AbsValue]](x401)
val x407 = new Tuple2[Set[AbsValue]],Map[Addr,Set[AbsValue]]](x406,x402)
val x408 = Set[Tuple2[Set[AbsValue]],Map[Addr,Set[AbsValue]]]](x407)
val x410 = x398 ++ x408
val x411 = new Tuple2[Set[Tuple2[Set[AbsValue]],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x410,x399)
x411
}
val x413 = x412._1
val x414 = x412._2
val x415 = new Tuple2[Set[Tuple2[Set[AbsValue],Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x6,x414)
val x311 = x193(ZCFAAddr("var3"))
val x522 = x413.foldLeft (x415) { case ((x416, x417), x418) =>
val x427 = x311.filter { x424 =>
val x425 = x424.isInstanceOf[CompiledClo]
x425
}
val x429 = new Tuple2[Set[Tuple2[AbsValue,Map[Addr,Set[AbsValue]]]],Map[Config,Set[ValStore]]](x428,x417)
val x420 = x418._2
val x437 = x427.foldLeft (x429) { case ((x430, x431), x432) =>
val x433 = new Tuple2[AbsValue,Map[Addr,Set[AbsValue]]](x432,x420)
val x434 = Set[Tuple2[AbsValue,Map[Addr,Set[AbsValue]]]](x433)
val x435 = x430 ++ x434
...
```

What does the generated code look like?

```
(define (map xs f) (if (null? xs) '() (cons (f (car xs)) (map (cdr xs) f))))
```

```
val x39 = {(x40: List[Set[AbsValue]],          // list of argument values
            x41: Map[Addr, Set[AbsValue]],     // abstract store
            x42: Map[Config, Set[ValSt]],      // in cache
            x43: Map[Config, Set[ValSt]]) ⇒   // out cache
  // addresses of arguments
  val x50 = List[Addr](ZCFAAddr("xs"), ZCFAAddr("f"))
  val x51 = x50.zip(x40)
  val x61 = x51.foldLeft (x41) { case (x52, x53) ⇒  // join into the store
    val x54 = x53._1; val x55 = x53._2
    val x58 = x52.getOrElse(x54, x57)
    val x59 = x58.union(x55);
    x52 + (x54 → x59)
  }
  ...
}
```

What does the generated code look like?

```
(define (map xs f) (if (null? xs) '() (cons (f (car xs)) (map (cdr xs) f))))
```

```
val x39 = {(x40: List[Set[AbsValue]],          // list of argument values
            x41: Map[Addr, Set[AbsValue]],     // abstract store
            x42: Map[Config, Set[ValSt]],      // in cache
            x43: Map[Config, Set[ValSt]]) ⇒   // out cache
  // addresses of arguments
  val x50 = List[Addr](ZCFAAddr("xs"), ZCFAAddr("f"))
  val x51 = x50.zip(x40)
  val x61 = x51.foldLeft (x41) { case (x52, x53) ⇒  // join into the store
    val x54 = x53._1; val x55 = x53._2
    val x58 = x52.getOrElse(x54, x57)
    val x59 = x58.union(x55);
    x52 + (x54 → x59)
  }
  ...
}
```

- There is no monadic layers in the generated code
- The compiled analysis is modular and reusable

More in the Paper

- Optimizations
- Comparison with Abstract Compilation [CC ‘96]
  - Both apply the idea of partial evaluation
  - Advantage: only adding type-level annotations and rewriting a few primitive operations, no need to refactor the whole analyzer to closure generation form
- Integration with other control-flow analysis techniques
  - Staging is orthogonal to those analyses
- Performance evaluation

Performance Evaluation

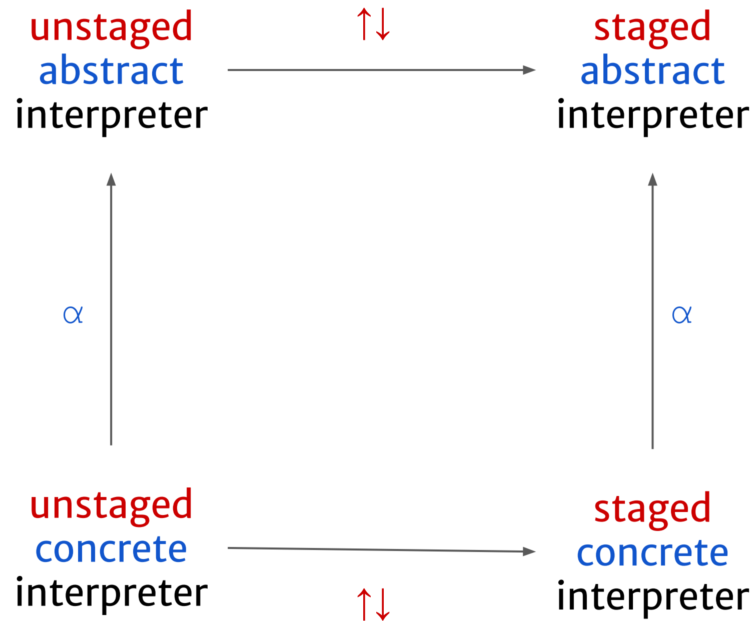
- Analysis: context-insensitive control-flow analysis for a subset of Scheme
  - 0CFA and 0CFA with store-widening
- Baseline: unstaged monadic abstract interpreter (Scala)
  - Uses monad transformers (e.g., ReaderT, StateT and SetT)
- Our prototype: staged monadic abstract interpreter (generates Scala)

Performance Evaluation

program	#AST	unstaged	staged	unstaged/staged	unstaged	staged	unstaged/staged
		w/o store-widening			w/ store-widening		
`fib`	32	3.288 ms	0.154 ms	21.33x	1.434 ms	0.098 ms	14.62x
`rsa`	451	238.171 ms	23.333 ms	10.20x	11.977 ms	1.197 ms	10.00x
`church`	120	61.001 s	4.277 s	14.26x	2.338 ms	0.534 ms	4.37x
`fermat`	310	23.540 ms	2.885 ms	8.05x	7.146 ms	0.915 ms	7.81x
`mbrotZ`	331	665.456 ms	66.070 ms	10.07x	11.008 ms	1.476 ms	7.45x
`lattice`	609	29.230 s	2.627 s	11.12x	16.432 ms	2.427 ms	6.76x
`kcfa-worst-16`	182	44.431 ms	3.211 ms	13.83x	4.425 ms	0.850 ms	5.20x
`kcfa-worst-32`	358	284.268 ms	9.065 ms	31.35x	10.109 ms	1.661 ms	6.08x
`kcfa-worst-64`	710	2.165 s	0.0425 s	50.85x	23.269 ms	3.312 ms	7.02x
`solovay-strassen`	523	5.078 s	0.766 s	6.62x	18.757 ms	3.142 ms	5.96x
`regex`	550	-	-	-	6.803 ms	1.088 ms	6.24x
`matrix`	1732	-	-	-	85.611 ms	9.297 ms	9.20x

An order of magnitude faster than the unstaged version.

Conclusion



- Constructing 1st Futamura Projection of abstract interpreters
- An abstraction-without-regret approach to compile static analysis via staging/specialization
- Semantic-based + clarity + efficiency

Thanks!

Question?