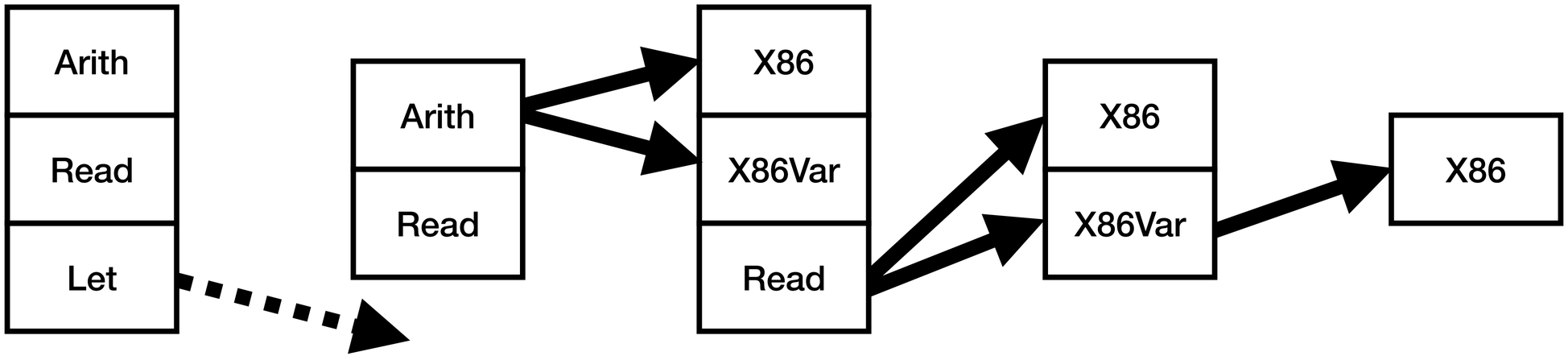
Towards Modular Compilation

Using Higher-order Effects

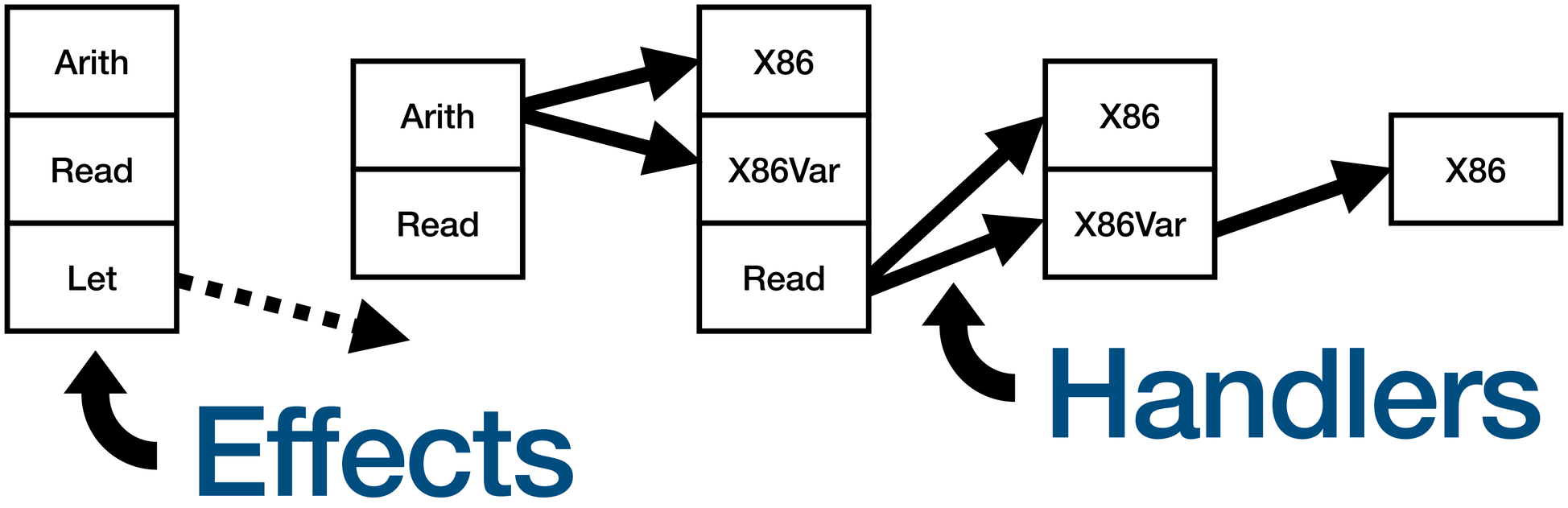
Jaro Reinders

TU Delft

A Modular Compiler Architecture



A Modular Compiler Architecture



A Simple Language

```
effect Arith v where
  int : Integer → m v
  add : v × v   → m v
```

A Simple Language

```
effect Arith v where
  int : Integer → m v
  add : v × v   → m v

effect Read v where
  read : m v
```

A Simple Language

```
effect Arith v where
  int : Integer → m v
  add : v × v   → m v

effect Read v where
  read : m v

effect Let v where
  let  : m v × (v → m v)
       → m v
```

A Simple Language

```
effect Arith v where
  int : Integer → m v
  add : v × v   → m v

effect Read v where
  read : m v

effect Let v where
  let  : m v × (v → m v)
       → m v
```

A Simple Language

```
effect Arith v where
  int : Integer → m v
  add : v × v   → m v

effect Read v where
  read : m v

effect Let v where
  let  : m v × (v → m v)
       → m v
```

$$
\begin{aligned}
[\![n]\!] &= \mathbf{int}\ n \\
[\![e_1 + e_2]\!] &= \mathbf{do}\ x \leftarrow [\![e_1]\!] \\
&\qquad\quad\ \ y \leftarrow [\![e_2]\!] \\
&\qquad\quad\ \ \mathbf{add}\ x\ y
\end{aligned}
$$

A Simple Language

```
effect Arith v where
  int : Integer → m v
  add : v × v   → m v

effect Read v where
  read : m v

effect Let v where
  let  : m v × (v → m v)
       → m v
```

```
⟦n⟧         = int n
⟦e₁ + e₂⟧ = do x ← ⟦e₁⟧
               y ← ⟦e₂⟧
               add x y

⟦read⟧     = read
```

A Simple Language

```
effect Arith v where
  int : Integer → m v
  add : v × v   → m v

effect Read v where
  read : m v

effect Let v where
  let  : m v × (v → m v)
       → m v
```

```
⟦n⟧       = int n
⟦e₁ + e₂⟧ = do x ← ⟦e₁⟧
               y ← ⟦e₂⟧
               add x y

⟦read⟧    = read
```

```
⟦let x = e₁ in e₂⟧ = let ⟦e₁⟧ \x → ⟦e₂ x⟧
```

X86

```
effect X86 where
  imm   : Integer  → m v
  reg   : Register → m v
  deref : Register × Integer → m v
  movq  : v × v    → m ()
  addq  : v × v    → m ()
  callq : Label    → m ()
```

X86

```
effect X86 where
  imm   : Integer  → m v
  reg   : Register → m v
  deref : Register × Integer → m v
  movq  : v × v    → m ()
  addq  : v × v    → m ()
  callq : Label    → m ()
```

```
effect X86Var where
  x86var : m v
```

Compiling Let, Arith, Read

```
handle
  (let e f) k → do
    x ← e
    z ← f x
    k z
```

Compiling Let, Arith, Read

```
handle
  (let e f) k → do
    x ← e
    z ← f x
    k z
```

```
handle
  (int n) k → do
    x ← imm n
    z ← x64var
    movq x z
    k z
  (add x y) k → do
    z ← x64var
    movq y z
    addq x z
    k z
```

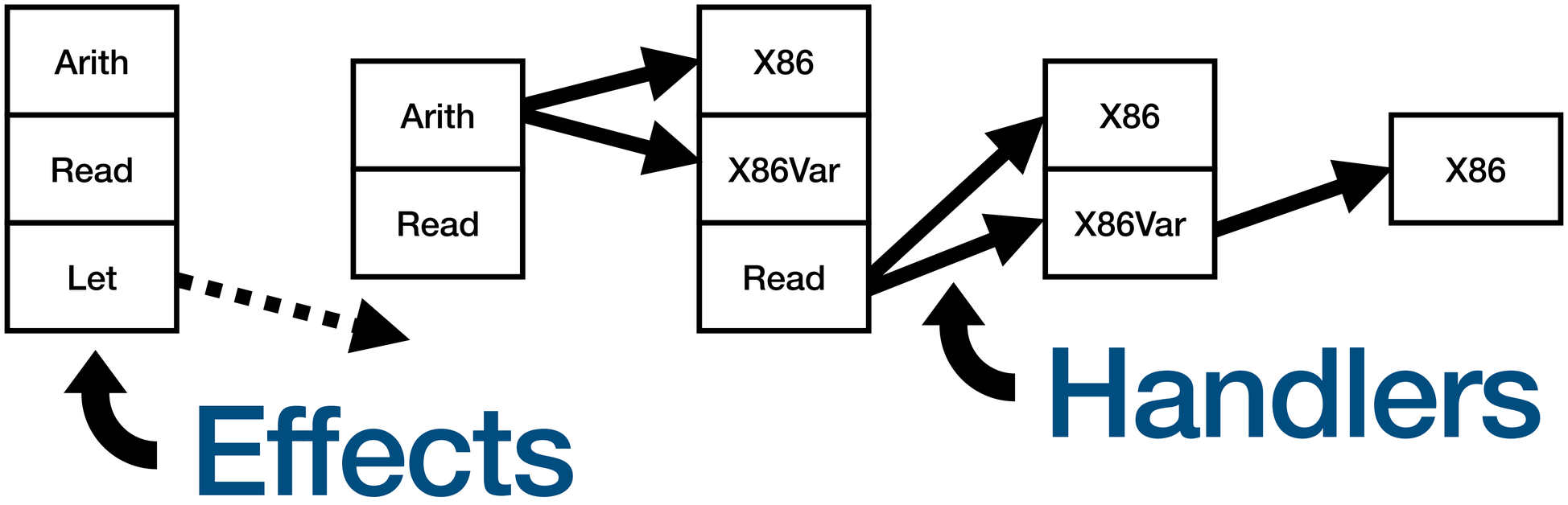
Compiling Let, Arith, Read

```
handle
  (let e f) k → do
    x ← e
    z ← f x
    k z
```

```
handle
  (int n) k → do
    x ← imm n
    z ← x64var
    movq x z
    k z
  (add x y) k → do
    z ← x64var
    movq y z
    addq x z
    k z
```

```
handle
  read k → do
    callq _read_int
    x ← reg %rax
    z ← x64var
    movq x z
    k z
```

Thank you!


Arith
Read
Let
Arith
Read
X86
X86Var
Read
X86
X86Var
X86
Effects
Handlers