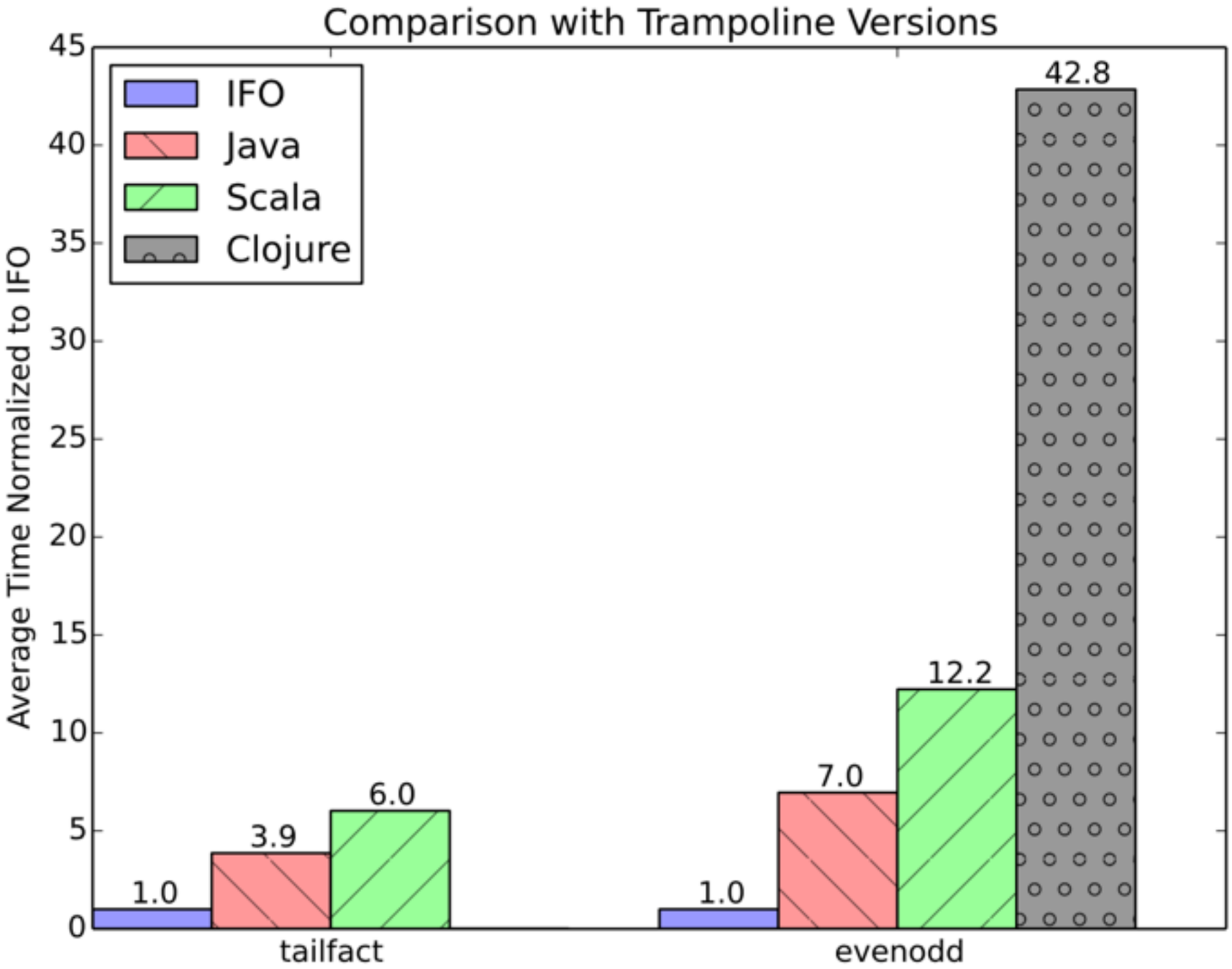
What Makes Our Compiler Fast

Jeremy



A picture is worth a thousand words!

“Avoid method calls at any cost!”

– Anonymous

(CJ-App)

$$\frac{\Gamma \vdash E_1 : \forall(x : T_2)\Delta.T_1 \rightsquigarrow J_1 \textbf{ in } S_1 \qquad \Gamma \vdash E_2 : T_2 \rightsquigarrow J_2 \textbf{ in } S_2 \qquad \Delta; T_1 \Downarrow T_3 \qquad f,\ x_f\ fresh}{\Gamma \vdash E_1\ E_2 : T_3 \rightsquigarrow x_f \textbf{ in } S_1 \uplus S_2 \uplus S_3}$$

S_3 := {
Function f = J_1;
f.arg = J_2;
f.apply();
$\langle T_3 \rangle$ x_f = ($\langle T_3 \rangle$) f.res;}

In base, one method call for one argument

Recap

```
public abstract class Closure
{
    public Object arg;
    public Object res;
    public abstract void apply ();
}
```

```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 4
```

let add2 (a:Int) (b:Int): Int = a + b
in add2 3 4

base

```
class Fun1 extends f2j.Closure
{...}
final f2j.Closure x0 = new Fun1();
f2j.Closure x8 = x0;
x8.arg = 3;
x8.apply();
final f2j.Closure x9 = (f2j.Closure) x8.res;
f2j.Closure x10 = x9;
x10.arg = 4;
x10.apply();
final java.lang.Integer x11 = (java.lang.Integer) x10.res;
```

let add2 (a:Int) (b:Int): Int = a + b
in add2 3 4

base

```
class Fun1 extends f2j.Closure
{...}
final f2j.Closure x0 = new Fun1();
f2j.Closure x8 = x0;
x8.arg = 3;
x8.apply();
final f2j.Closure x9 = (f2j.Closure) x8.res;
f2j.Closure x10 = x9;
x10.arg = 4;
x10.apply();
final java.lang.Integer x11 = (java.lang.Integer) x10.res;
```

let add2 (a:Int) (b:Int): Int = a + b
in add2 3 4

base

```
class Fun1 extends f2j.Closure
{...}
final f2j.Closure x0 = new Fun1();
f2j.Closure x8 = x0;
x8.arg = 3;
x8.apply();
final f2j.Closure x9 = (f2j.Closure) x8.res;
f2j.Closure x10 = x9;
x10.arg = 4;
x10.apply();
final java.lang.Integer x11 = (java.lang.Integer) x10.res;
```

apply-opt

```
class Fun1 extends f2j.Closure
{...}
f2j.Closure x1 = new Fun1();
final f2j.Closure x0 = new Fun1();
f2j.Closure x8 = x0;
x8.arg = 3;
final f2j.Closure x9 = (f2j.Closure) x8.res;
f2j.Closure x10 = x9;
x10.arg = 4;
x10.apply();
final java.lang.Integer x11 = (java.lang.Integer) x10.res;
```

let add2 (a:Int) (b:Int): Int = a + b
in add2 3 4

base

apply-opt

```
class Fun1 extends f2j.Closure
{...}
final f2j.Closure x0 = new Fun1();
f2j.Closure x8 = x0;
x8.arg = 3;
x8.apply();
final f2j.Closure x9 = (f2j.Closure) x8.res;
f2j.Closure x10 = x9;
x10.arg = 4;
x10.apply();
final java.lang.Integer x11 = (java.lang.Integer) x10.res;
```

```
class Fun1 extends f2j.Closure
{...}
f2j.Closure x1 = new Fun1();
final f2j.Closure x0 = new Fun1();
f2j.Closure x8 = x0;
x8.arg = 3;
final f2j.Closure x9 = (f2j.Closure) x8.res;
f2j.Closure x10 = x9;
x10.arg = 4;
x10.apply();
final java.lang.Integer x11 = (java.lang.Integer) x10.res;
```

let add2 (a:Int) (b:Int): Int = a + b
in add2 3 4

base

apply-opt

```
class Fun1 extends f2j.Closure
{...}
final f2j.Closure x0 = new Fun1();
f2j.Closure x8 = x0;
x8.arg = 3;
x8.apply();
final f2j.Closure x9 = (f2j.Closure) x8.res;
f2j.Closure x10 = x9;
x10.arg = 4;
x10.apply();
final java.lang.Integer x11 = (java.lang.Integer) x10.res;
```

Why?

```
class Fun1 extends f2j.Closure
{...}
f2j.Closure x1 = new Fun1();
final f2j.Closure x0 = new Fun1();
f2j.Closure x8 = x0;
x8.arg = 3;
final f2j.Closure x9 = (f2j.Closure) x8.res;
f2j.Closure x10 = x9;
x10.arg = 4;
x10.apply();
final java.lang.Integer x11 = (java.lang.Integer) x10.res;
```

```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 4
```

let add2 (a:Int) (b:Int): Int = a + b
in add2 3 4

```
class Fun1 extends f2j.Closure
{
    f2j.Closure x2 = this;
    public void apply ()
    {
        final java.lang.Integer x3 = (java.lang.Integer) x2.arg;
        class Fun4 extends f2j.Closure
        {...}
        res = new Fun4();
    }
}
```

base

let add2 (a:Int) (b:Int): Int = a + b
in add2 3 4

base

```
class Fun1 extends f2j.Closure
{
    f2j.Closure x2 = this;
    public void apply ()
    {
        final java.lang.Integer x3 = (java.lang.Integer) x2.arg;
        class Fun4 extends f2j.Closure
        {...}
        res = new Fun4();
    }
}
```

apply-opt

```
class Fun1 extends f2j.Closure
{
    f2j.Closure x2 = this;
    {
        class Fun4 extends f2j.Closure
        {...}
        res = new Fun4();
    }
    public void apply ()
    {
    }
    public f2j.Closure clone ()
    {...}
}
```

What about unknown function?

\(f : Int -> Int -> Int) (a : Int) (b : Int). f a b

What about unknown function?

\(f : Int -> Int -> Int) (a : Int) (b : Int). f a b

```
f.arg = a;
f.apply();
final f2j.Closure x10 = (f2j.Closure) f.res;
f2j.Closure x11 = x10;
x11.arg = b;
x11.apply();
final java.lang.Integer x12 = (java.lang.Integer) x11.res;
```

What about unknown function?

\(f : Int -> Int -> Int) (a : Int) (b : Int). f a b

```
f.arg = a;
f.apply();
final f2j.Closure x10 = (f2j.Closure) f.res;
f2j.Closure x11 = x10;
x11.arg = b;
x11.apply();
final java.lang.Integer x12 = (java.lang.Integer) x11.res;
```

Or

What about unknown function?

\(f : Int -> Int -> Int) (a : Int) (b : Int). f a b

```
f.arg = a;
f.apply();
final f2j.Closure x10 = (f2j.Closure) f.res;
f2j.Closure x11 = x10;
x11.arg = b;
x11.apply();
final java.lang.Integer x12 = (java.lang.Integer) x11.res;
```

Or

```
f.arg = a;
final f2j.Closure x10 = (f2j.Closure) f.res;
f2j.Closure x11 = x10;
x11.arg = b;
x11.apply();
final java.lang.Integer x12 = (java.lang.Integer) x11.res;
```

We don’t know!

The statically unknown function *f* can be like *add2*, which takes two arguments, and returns the result.

Or it might be some function that takes one argument, compute for a while before returning a function that consumes the next argument, like the following one

\(a : Int). if a == 0 then \(b : Int). b + 1 else \(b : Int). b + 2

We don’t know!

The statically unknown function *f* can be like *add2*, which takes two arguments, and returns the result.

Or it might be some function that takes one argument, compute for a while before returning a function that consumes the next argument, like the following one

\(a : Int). if a == 0 then \(b : Int). b + 1 else \(b : Int). b + 2

In either case, they all have the same type: **Int -> Int -> Int**

Not long before, we just blithely eliminated all apply method calls for good, which of course causes …

Not long before, we just blithely eliminated all apply method calls for good, which of course causes …

```
Exception in thread "main" java.lang.NullPointerException
        at Test$1Fun1$1Fun4$1Fun7.apply(Test.java:27)
        at Test.apply(Test.java:138)
        at Test.main(Test.java:145)
```

Insight

The idea is simple. When a function of statically-unknown arity is applied, two pieces of information come together:

1. The arity of the function
2. The number of arguments in the call

Let the function itself tells us how many arguments it needs.

Insight

The idea is simple. When a function of statically-unknown arity is applied, two pieces of information come together:

1. The arity of the function
2. The number of arguments in the call

Let the function itself tells us how many arguments it needs.

Boolean to Rescue!

Run-time check

When translating a function, we set a *boolean flag* to indicate whether it needs to call apply method or not.

This is feasible because the function itself statically knows its arity.

\(a:Int) (b:Int): Int = a + b

\(a : Int). if a == 0 then \(b : Int). b + 1 else \(b : Int). b + 2

Run-time check

When translating a function, we set a *boolean flag* to indicate whether it needs to call apply method or not.

This is feasible because the function itself statically knows its arity.

\(a:Int) (b:Int): Int = a + b

No need for apply call

\(a : Int). if a == 0 then \(b : Int). b + 1 else \(b : Int). b + 2

Run-time check

When translating a function, we set a *boolean flag* to indicate whether it needs to call apply method or not.

This is feasible because the function itself statically knows its arity.

\(a:Int) (b:Int): Int = a + b

No need for apply call

\(a : Int). if a == 0 then \(b : Int). b + 1 else \(b : Int). b + 2

need apply call

\(a : Int). if a == 0 then \(b : Int). b + 1 else \(b : Int). b + 2

```
class Fun0 extends f2j.Closure
{
    f2j.Closure x1 = this;
    public void apply ()
    {
        final java.lang.Integer x2 = (java.lang.Integer) x1.arg;
        final java.lang.Boolean x4 = x2 == 0;
        f2j.Closure ifres3;
        if (x4)
        {
            class Fun5 extends f2j.Closure
            {...}
            ifres3 = new Fun5();
        }
        else
        {
            class Fun9 extends f2j.Closure
            {...}
            ifres3 = new Fun9();
        }
        res = ifres3;
    }
    public f2j.Closure clone ()
    {...}
}
```

```
public abstract class Closure
{
    public Object arg;
    public Object res;
    public boolean hasApply = true;
    public abstract void apply ();
}
```

\(a:Int) (b:Int): Int = a + b

```
class Fun0 extends f2j.Closure
{
    f2j.Closure x1 = this;
    {
        x1.hasApply = false;
        class Fun3 extends f2j.Closure
        {...}
        res = new Fun3();
    }
    public void apply ()
    {
    }
    public f2j.Closure clone ()
    {...}
}
```

\(a : Int). if a == 0 then \(b : Int). b + 1 else \(b : Int). b + 2

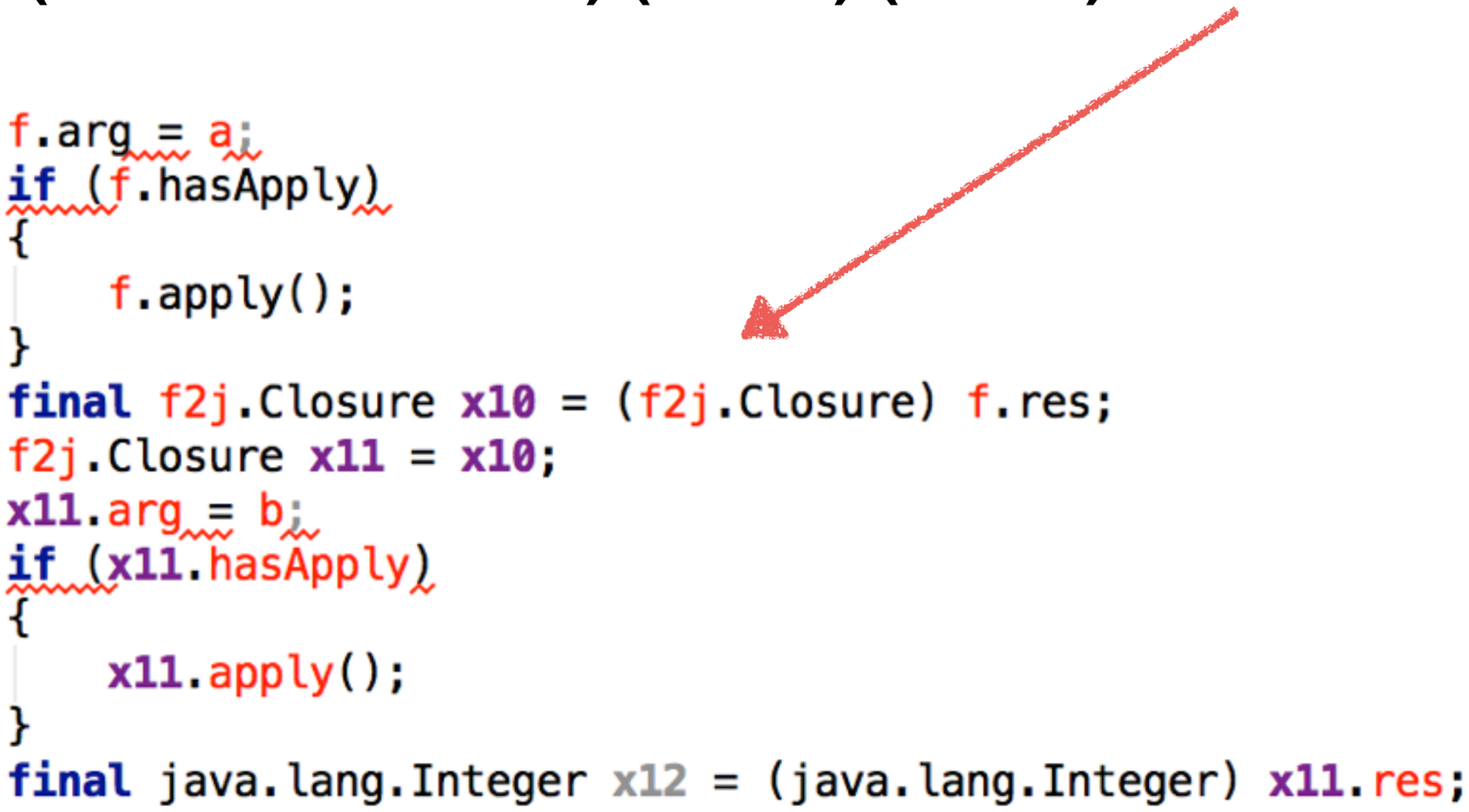
```
class Fun0 extends f2j.Closure
{
    f2j.Closure x1 = this;
    public void apply ()
    {
        final java.lang.Integer x2 = (java.lang.Integer) x1.arg;
        final java.lang.Boolean x4 = x2 == 0;
        f2j.Closure ifres3;
        if (x4)
        {
            class Fun5 extends f2j.Closure
            {...}
            ifres3 = new Fun5();
        }
        else
        {
            class Fun9 extends f2j.Closure
            {...}
            ifres3 = new Fun9();
        }
        res = ifres3;
    }
    public f2j.Closure clone ()
    {...}
}
```

```
public abstract class Closure
{
    public Object arg;
    public Object res;
    public boolean hasApply = true;
    public abstract void apply ();
}
```

\(a:Int) (b:Int): Int = a + b

```
class Fun0 extends f2j.Closure
{
    f2j.Closure x1 = this;
    {
        x1.hasApply = false;
        class Fun3 extends f2j.Closure
        {...}
        res = new Fun3();
    }
    public void apply ()
    {
    }
    public f2j.Closure clone ()
    {...}
}
```

\(f : Int -> Int -> Int) (a : Int) (b : Int). f a b



```
f.arg = a;
if (f.hasApply)
{
    f.apply();
}
final f2j.Closure x10 = (f2j.Closure) f.res;
f2j.Closure x11 = x10;
x11.arg = b;
if (x11.hasApply)
{
    x11.apply();
}
final java.lang.Integer x12 = (java.lang.Integer) x11.res;
```

Now we can rest assured that the function f takes care of whether it needs to call apply method or not.

Life is all too wonderful if we don’t …

```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```

Which one produces the correct result?

Life is all too wonderful if we don’t …

```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```

Which one produces the correct result?


```
⇒  f2j test.sf -m naive -r
test using [Naive]
  Compiling to Java source code ( ./Test.java )
12
```

Life is all too wonderful if we don't …

```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```

Which one produces the correct result?

base

```
⇒  f2j test.sf -m naive -r
test using [Naive]
  Compiling to Java source code ( ./Test.java )
12
```

apply-opt

```
⇒  f2j test.sf -m apply -r
test using [Apply,Naive]
  Compiling to Java source code ( ./Test.java )
13
```

Apparently something terrible happens …

Apparently something terrible happens …

$$\text{(CJD-Bind1)}\qquad \frac{\Gamma\,(y : T_1 \mapsto x_2); \Delta \vdash E : T \rightsquigarrow J \textbf{ in } S \qquad FC,\ x_1,\ x_2,\ f\ fresh}{\Gamma; (y : T_1)\ \Delta \vdash E : T \rightsquigarrow f \textbf{ in } S'}$$

S' := {
```
  class FC extends Function {
    Function x1 = this;
    void apply() {
      ⟨T1⟩ x2 = (⟨T1⟩) x1.arg;
      S;
      res = J;
    }
  };
  Function f = new FC();}
```

Apparently something terrible happens …

$$\text{(CJD-Bind1)}\quad \frac{\Gamma\,(y : T_1 \mapsto x_2); \Delta \vdash E : T \rightsquigarrow J \textbf{ in } S \qquad FC,\ x_1,\ x_2,\ f\ fresh}{\Gamma; (y : T_1)\ \Delta \vdash E : T \rightsquigarrow f \textbf{ in } S'}$$

S' := {
```
  class FC extends Function {
    Function x1 = this;
    void apply() {
      ⟨T1⟩ x2 = (⟨T1⟩) x1.arg;
      S;
      res = J;
    }
  };
Function f = new FC();}
```

Apparently something terrible happens …

$$\text{(CJD-Bind1)}\quad \frac{\Gamma\,(y : T_1 \mapsto x_2); \Delta \vdash E : T \rightsquigarrow J \textbf{ in } S \qquad FC,\ x_1,\ x_2,\ f\ fresh}{\Gamma; (y : T_1)\ \Delta \vdash E : T \rightsquigarrow f \textbf{ in } S'}$$

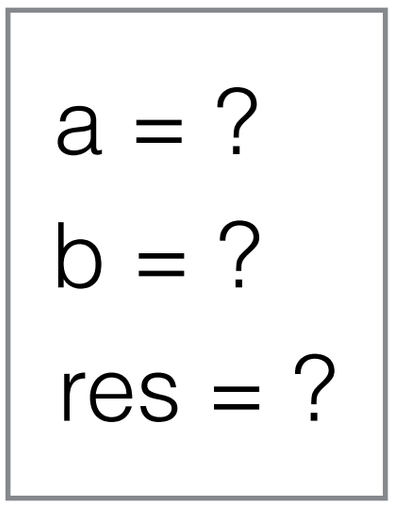
S' := {
```
  class FC extends Function {
    Function x1 = this;
    void apply() {
      ⟨T1⟩ x2 = (⟨T1⟩) x1.arg;
      S;
      res = J;
    }
  };
Function f = new FC();}
```

Wherever there is a function definition, we immediately allocate memory for that.

This is no problem for base, because each apply call will create fresh copies of all the variables.

But for apply-opt, this is not the case …

let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)

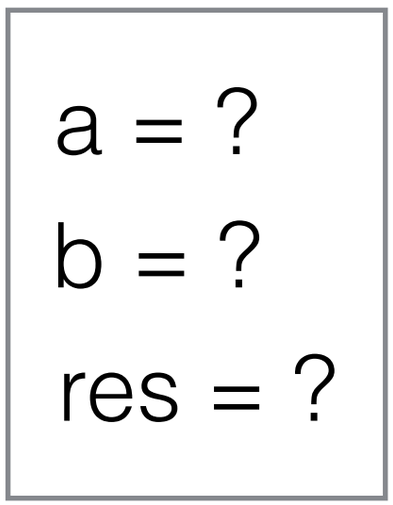


add2

This is no problem for base, because each apply call will create fresh copies of all the variables.

But for apply-opt, this is not the case …

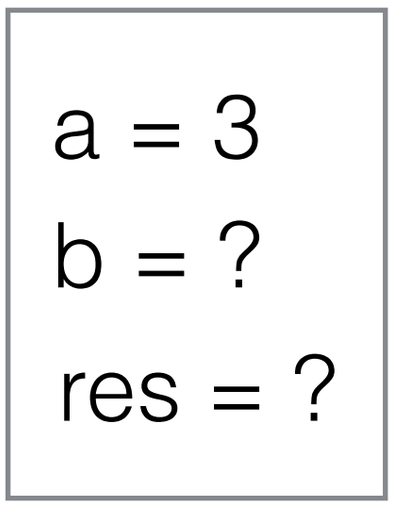
```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```



This is no problem for base, because each apply call will create fresh copies of all the variables.

But for apply-opt, this is not the case …

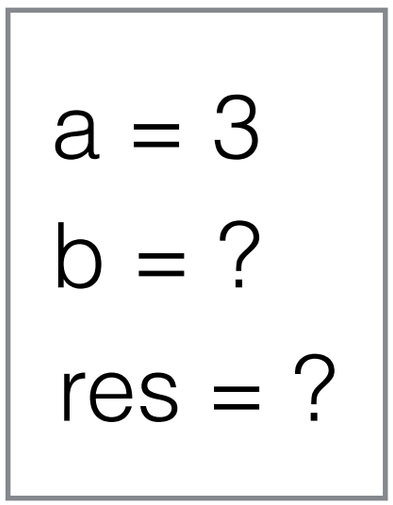
```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```



This is no problem for base, because each apply call will create fresh copies of all the variables.

But for apply-opt, this is not the case …

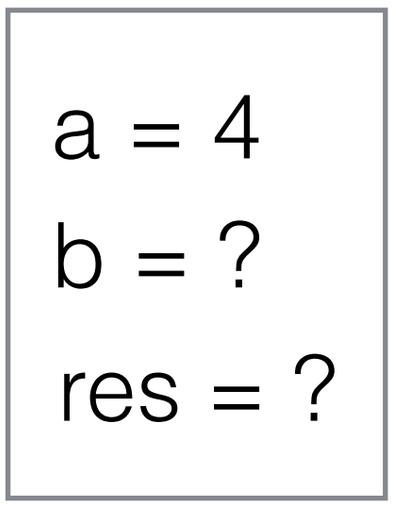
```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```



This is no problem for base, because each apply call will create fresh copies of all the variables.

But for apply-opt, this is not the case …

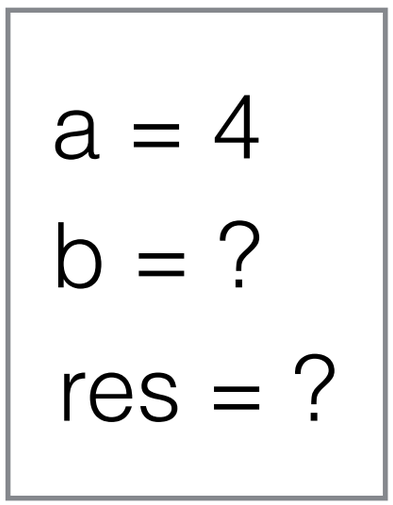
```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```



This is no problem for base, because each apply call will create fresh copies of all the variables.

But for apply-opt, this is not the case …

```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```

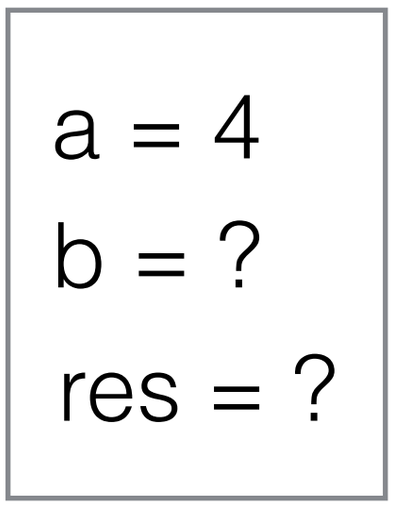


Overriding happens!

This is no problem for base, because each apply call will create fresh copies of all the variables.

But for apply-opt, this is not the case …

```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```



This is no problem for base, because each apply call will create fresh copies of all the variables.

But for apply-opt, this is not the case …

let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)

a = 4

b = 5

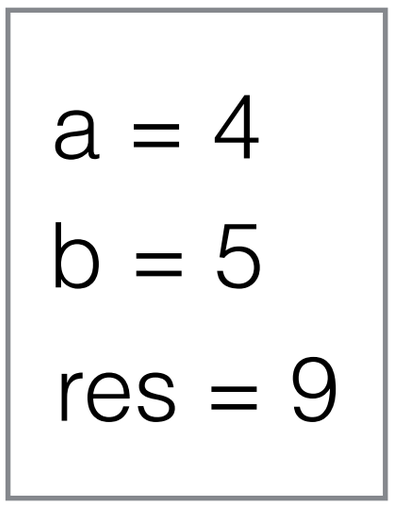
res = ?

add2

This is no problem for base, because each apply call will create fresh copies of all the variables.

But for apply-opt, this is not the case …

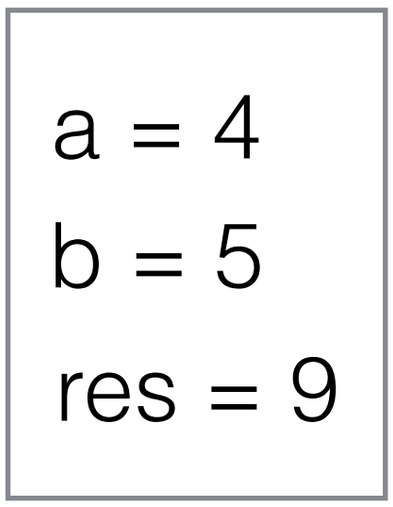
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)



This is no problem for base, because each apply call will create fresh copies of all the variables.

But for apply-opt, this is not the case …

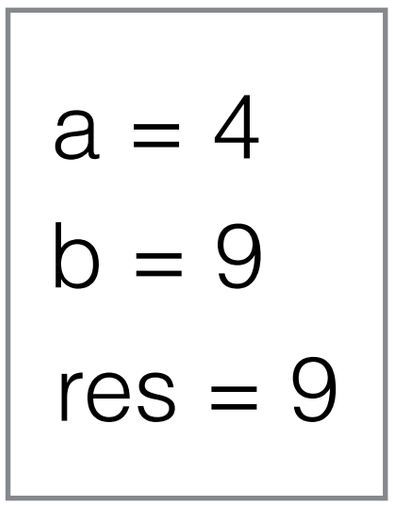
```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```



This is no problem for base, because each apply call will create fresh copies of all the variables.

But for apply-opt, this is not the case …

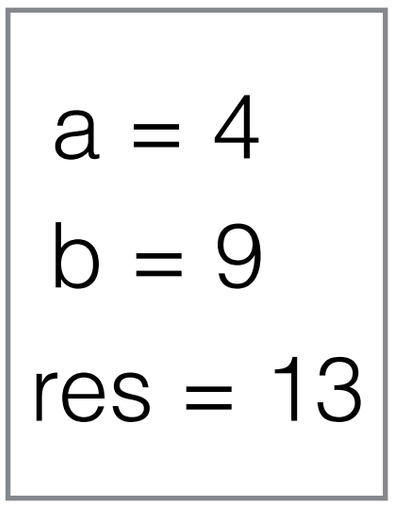
```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```



This is no problem for base, because each apply call will create fresh copies of all the variables.

But for apply-opt, this is not the case …

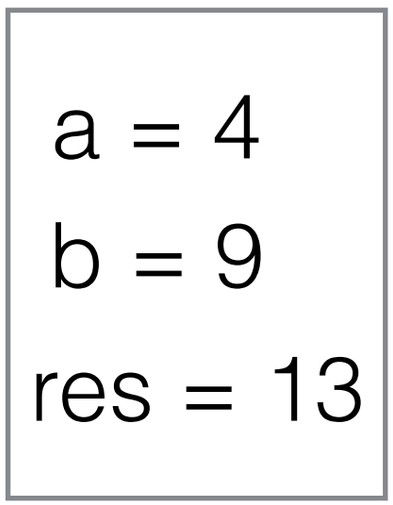
```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```



This is no problem for base, because each apply call will create fresh copies of all the variables.

But for apply-opt, this is not the case …

```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```



Observation is that, those two *add2* applications should use two different instances in memory instead of sharing one.

Non-Solution

Use a set to record all function names which have been used before.

```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```

set

Non-Solution

Use a set to record all function names which have been used before.

```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```

set

Non-Solution

Use a set to record all function names which have been used before.

```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```

set

Non-Solution

Use a set to record all function names which have been used before.

```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```

set

Non-Solution

Use a set to record all function names which have been used before.

```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```

set

add2 is in the set!
Cloning is needed…

let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)

```
class Fun1 extends f2j.Closure
{
    f2j.Closure x2 = this;
    {
        x2.hasApply = false;
        class Fun4 extends f2j.Closure
        {...}
        res = new Fun4();
    }
    public void apply ()
    {...}
    public f2j.Closure clone ()
    {
        f2j.Closure c = new Fun1();
        return (f2j.Closure) c;
    }
}
```

```
let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)
```

```
class Fun1 extends f2j.Closure
{
    f2j.Closure x2 = this;
    {
        x2.hasApply = false;
        class Fun4 extends f2j.Closure
        {...}
        res = new Fun4();
    }
    public void apply ()
    {...}
    public f2j.Closure clone ()
    {
        f2j.Closure c = new Fun1();
        return (f2j.Closure) c;
    }
}
```

let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)

add2.clone()

```
class Fun1 extends f2j.Closure
{
    f2j.Closure x2 = this;
    {
        x2.hasApply = false;
        class Fun4 extends f2j.Closure
        {...}
        res = new Fun4();
    }
    public void apply ()
    {...}
    public f2j.Closure clone ()
    {
        f2j.Closure c = new Fun1();
        return (f2j.Closure) c;
    }
}
```

let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)

add2.clone()

```
class Fun1 extends f2j.Closure
{
    f2j.Closure x2 = this;
    {
        x2.hasApply = false;
        class Fun4 extends f2j.Closure
        {...}
        res = new Fun4();
    }
    public void apply ()
    {...}
    public f2j.Closure clone ()
    {
        f2j.Closure c = new Fun1();
        return (f2j.Closure) c;
    }
}
```

We get a fresh copy of *add2* :)

Problem solved?

Problem solved?

```
let add2 (a : Int) (b : Int): Int = a + b
in let g (a : Int) (b : Int) : Int = add2 a b
    in let f = add2
        in f 3 (g 4 5)
```

Problem solved?

```
let add2 (a : Int) (b : Int): Int = a + b
in let g (a : Int) (b : Int) : Int = add2 a b
    in let f = add2
        in f 3 (g 4 5)
```

set

Problem solved?

```
let add2 (a : Int) (b : Int): Int = a + b
in let g (a : Int) (b : Int) : Int = add2 a b
    in let f = add2
        in f 3 (g 4 5)
```

set

Problem solved?

```
let add2 (a : Int) (b : Int): Int = a + b
in let g (a : Int) (b : Int) : Int = add2 a b
    in let f = add2
        in f 3 (g 4 5)
```

set

Problem solved?

```
let add2 (a : Int) (b : Int): Int = a + b
in let g (a : Int) (b : Int) : Int = add2 a b
    in let f = add2
        in f 3 (g 4 5)
```

set

Problem solved?

```
let add2 (a : Int) (b : Int): Int = a + b
in let g (a : Int) (b : Int) : Int = add2 a b
    in let f = add2
        in f 3 (g 4 5)
```

set

Problem solved?

```
let add2 (a : Int) (b : Int): Int = a + b
in let g (a : Int) (b : Int) : Int = add2 a b
    in let f = add2
        in f 3 (g 4 5)
```

set

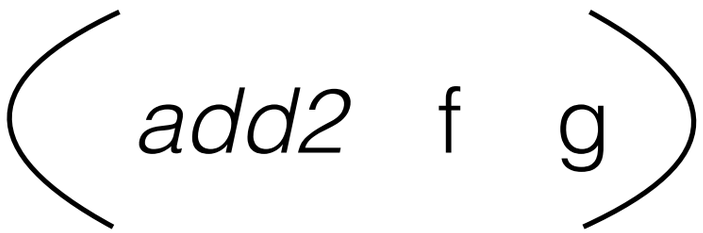
Problem solved?

```
let add2 (a : Int) (b : Int): Int = a + b
in let g (a : Int) (b : Int) : Int = add2 a b
    in let f = add2
        in f 3 (g 4 5)
```

set

Problem solved?

```
let add2 (a : Int) (b : Int): Int = a + b
in let g (a : Int) (b : Int) : Int = add2 a b
    in let f = add2
        in f 3 (g 4 5)
```



set

Overriding happens!

Even worse …

Even worse …

\(f : Int -> Int -> Int) (g -> Int -> Int -> Int). f 3 (g 4 5)

We have no clue at all whether *g* will have a call to *f* or not!

Even worse …

\(f : Int -> Int -> Int) (g -> Int -> Int -> Int). f 3 (g 4 5)

We have no clue at all whether *g* will have a call to *f* or not!

Cloning everywhere is possible, but slows down performance.

Potential Solution

The problem boils down to allocating memory immediately after function definition.

Potential Solution

The problem boils down to allocating memory immediately after function definition.

$$\text{(CJD-Bind1)}\quad \frac{\Gamma\ (y : T_1 \mapsto x_2); \Delta \vdash E : T \rightsquigarrow J\ \textbf{in}\ S \qquad FC,\ x_1,\ x_2,\ f\ fresh}{\Gamma; (y : T_1)\ \Delta \vdash E : T \rightsquigarrow f\ \textbf{in}\ S'}$$

S' := {
  class FC **extends** Function {
    Function x_1 = **this**;
    void apply() {
      $\langle T_1 \rangle$ x_2 = ($\langle T_1 \rangle$) x1.arg;
      S;
      res = J;
    }
  };
  Function f = **new** FC();}

Potential Solution

The problem boils down to allocating memory immediately after function definition.

$$\text{(CJD-Bind1)}\quad \frac{\Gamma\ (y : T_1 \mapsto x_2); \Delta \vdash E : T \rightsquigarrow J\ \textbf{in}\ S \qquad FC,\ x_1,\ x_2,\ f\ fresh}{\Gamma; (y : T_1)\ \Delta \vdash E : T \rightsquigarrow f\ \textbf{in}\ S'}$$

S' := {
```
  class FC extends Function {
    Function x1 = this;
    void apply() {
      ⟨T1⟩ x2 = (⟨T1⟩) x1.arg;
      S;
      res = J;
    }
  };
  Function f = new FC();}
```

Potential Solution

The problem boils down to allocating memory immediately after function definition.

(CJD-Bind1)
$$\frac{\Gamma\ (y : T_1 \mapsto x_2); \Delta \vdash E : T \rightsquigarrow J\ \textbf{in}\ S \qquad FC,\ x_1,\ x_2,\ f\ fresh}{\Gamma; (y : T_1)\ \Delta \vdash E : T \rightsquigarrow f\ \textbf{in}\ S'}$$

```
S' := {
  class FC extends Function {
    Function x1 = this;
    void apply() {
      ⟨T1⟩ x2 = (⟨T1⟩) x1.arg;
      S;
      res = J;
    }
  };
  Function f = new FC();}
```

Potential Solution

The problem boils down to allocating memory immediately after function definition.

(CJD-Bind1)

$$\frac{\Gamma\ (y : T_1 \mapsto x_2); \Delta \vdash E : T \rightsquigarrow J\ \textbf{in}\ S \qquad FC,\ x_1,\ x_2,\ f\ fresh}{\Gamma; (y : T_1)\ \Delta \vdash E : T \rightsquigarrow f\ \textbf{in}\ S'}$$

let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)

S' := {
  class FC **extends** Function {
    Function x_1 = **this**;
    void apply() {
      $\langle T_1 \rangle$ x_2 = ($\langle T_1 \rangle$) x1.arg;
      S;
      res = J;
    }
  };
  ~~Function f = **new** FC();~~}

Potential Solution

The problem boils down to allocating memory immediately after function definition.

(CJD-Bind1)

$$\frac{\Gamma\ (y : T_1 \mapsto x_2); \Delta \vdash E : T \rightsquigarrow J\ \textbf{in}\ S \qquad FC,\ x_1,\ x_2,\ f\ fresh}{\Gamma; (y : T_1)\ \Delta \vdash E : T \rightsquigarrow f\ \textbf{in}\ S'}$$

S' := {
  class FC **extends** Function {
    Function x_1 = **this**;
    void apply() {
      $\langle T_1 \rangle$ x_2 = ($\langle T_1 \rangle$) x1.arg;
      S;
      res = J;
    }
  };
~~Function f = **new** FC();~~}

let add2 (a:Int) (b:Int): Int = a + b
in add2 3 (add2 4 5)

new FC() new FC()

Still …

let f = ...
in f 2 (f 3 4)

```
f = new FC();
f.arg = 2;
...
f.arg = 3;
...
```

Aliasing still causes the same problem …

Contributions

- Propose a run-time check solution to solve the problem in apply-opt.
- Propose a potential solution which could possibly solve field-overriding problem.
- Build a working prototype compiler that uses apply-opt, which produces reasonably fast code.

Related work

- Marlow, Simon, and Simon Peyton Jones. "Making a fast curry: push/enter vs. eval/apply for higher-order languages." *ACM SIGPLAN Notices*. Vol. 39. No. 9. ACM, 2004.
- Shao, Zhong, and Andrew W. Appel. *Space-efficient closure representations*. Vol. 7. No. 3. ACM, 1994.

Future work

- Continue improving apply-opt to fully solve the field-overriding problem
- Investigate *tail recursive optimisation*

To be continued …

Paper Reading

About Author

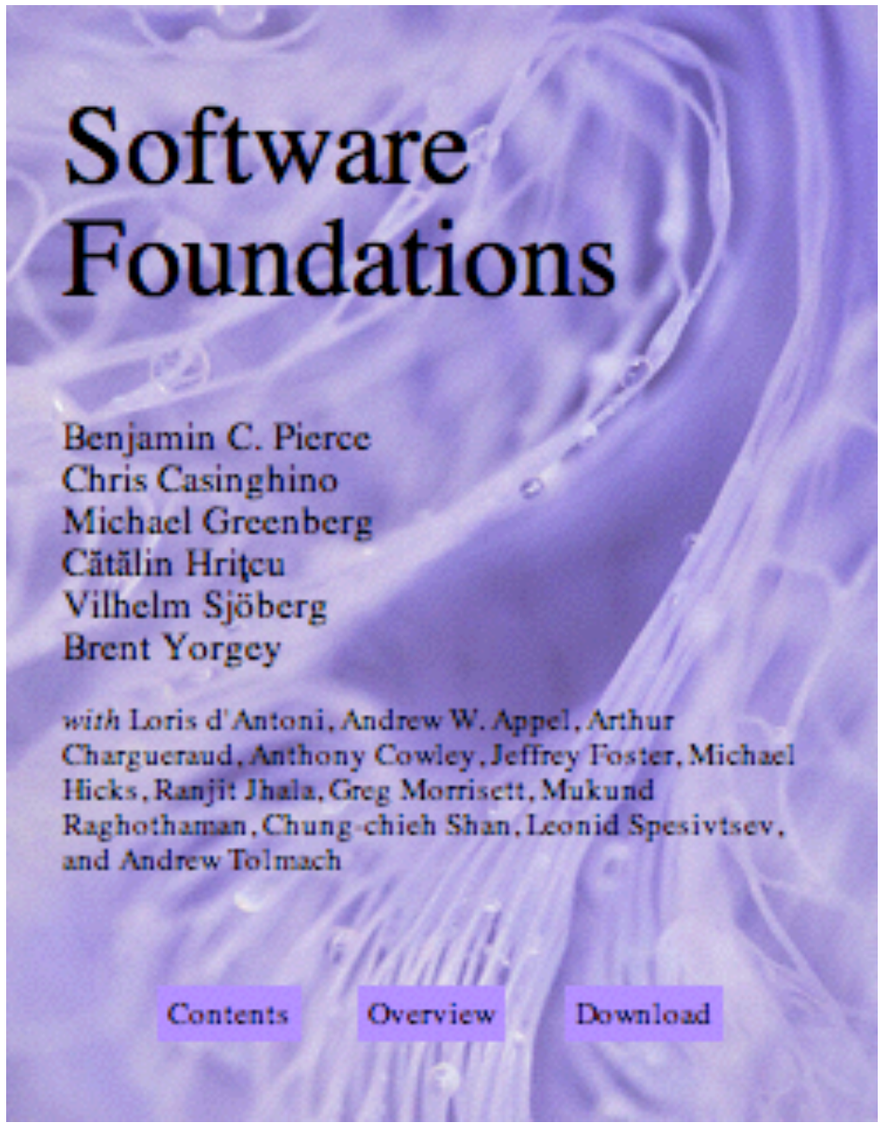
Brent A. Yorgey

University of Pennsylvania

About Author

Brent A. Yorgey

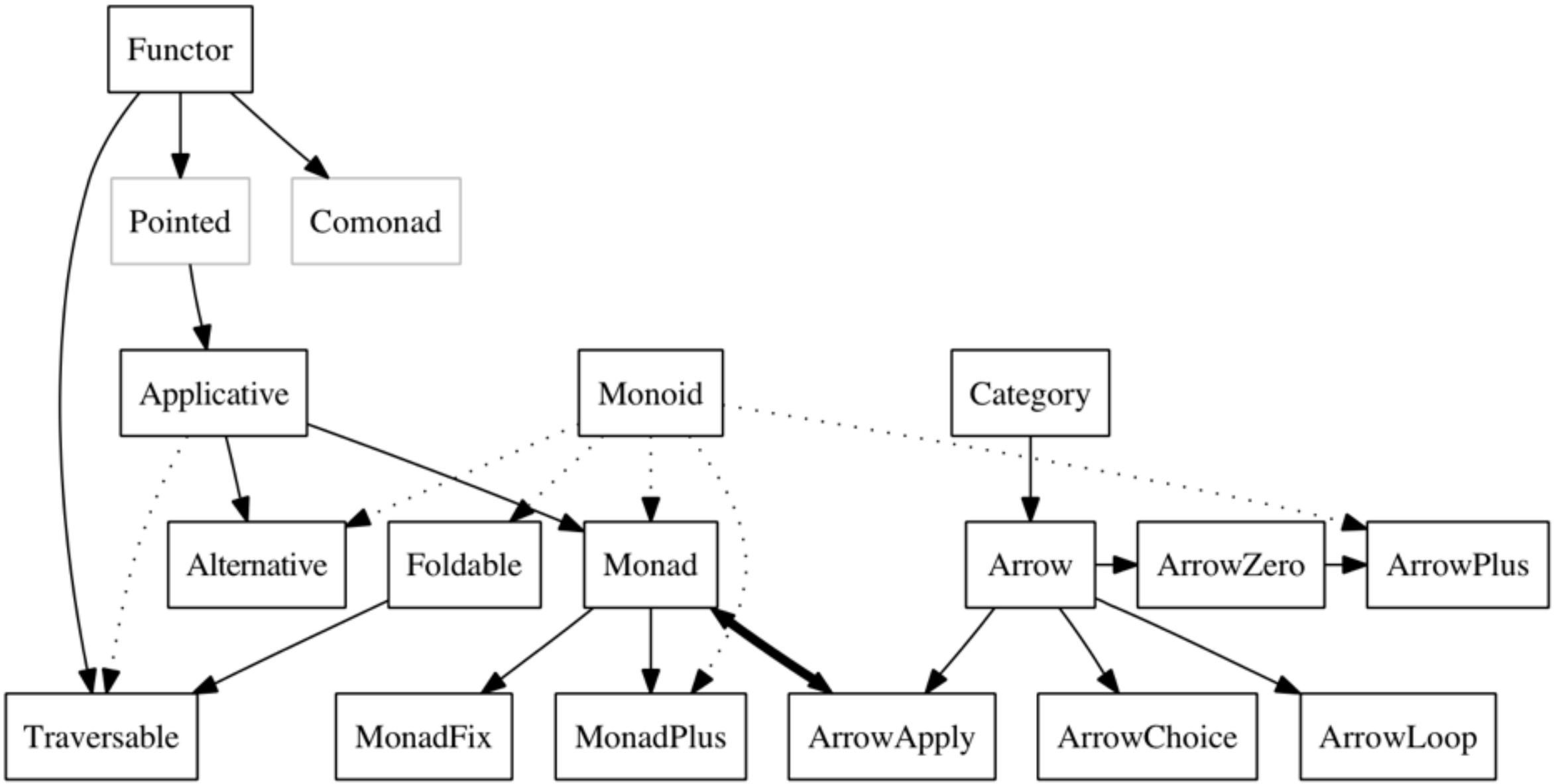
University of Pennsylvania

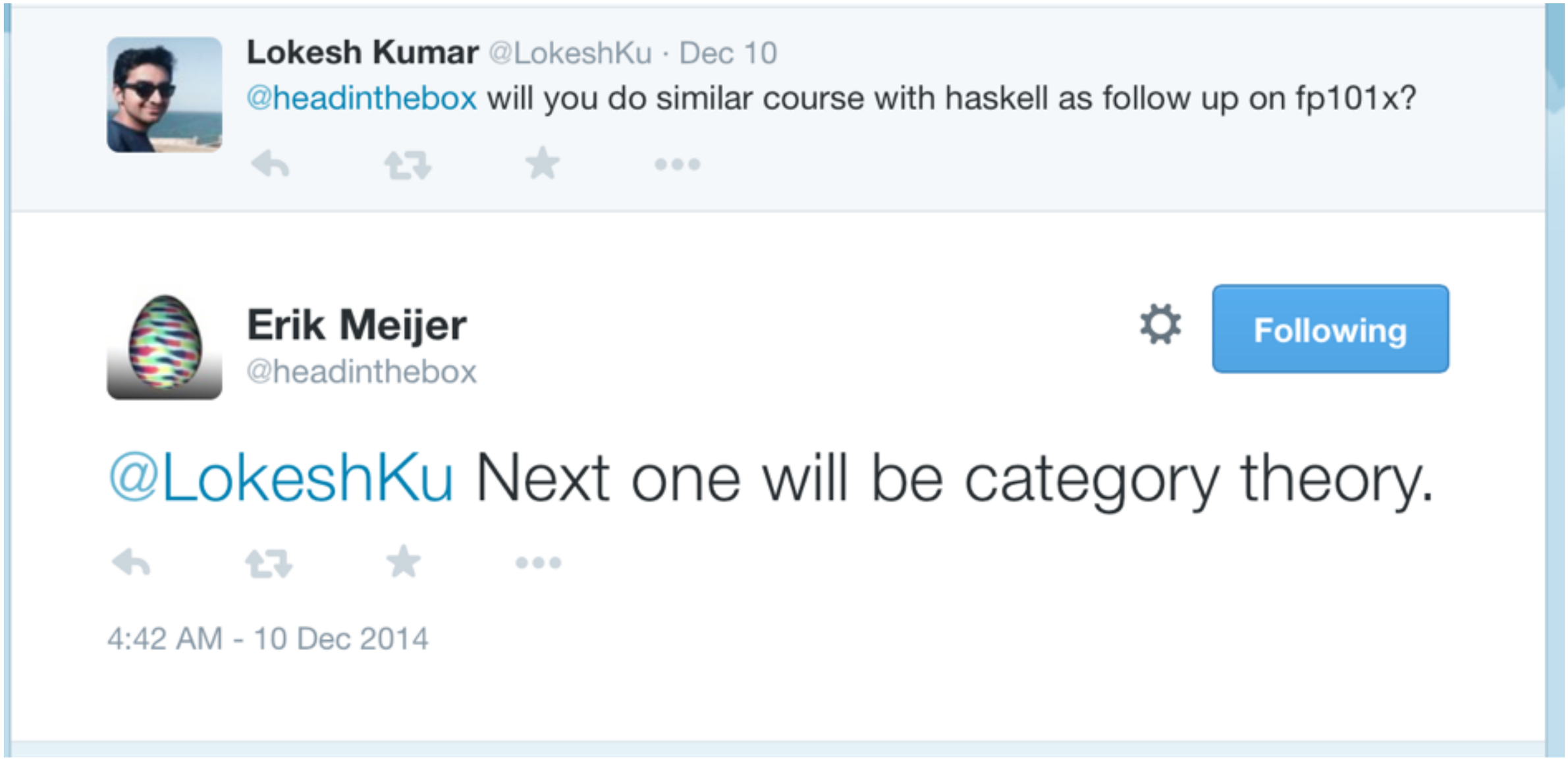


About Author

Note to headhunters: I am not interested in positions in industry, especially in the financial sector. Please don't contact me about job opportunities.

All about type class




Lokesh Kumar @LokeshKu · Dec 10
@headinthebox will you do similar course with haskell as follow up on fp101x?
Erik Meijer
@headinthebox
Following
@LokeshKu Next one will be category theory.
4:42 AM - 10 Dec 2014