How to Make FCore Thread-safe

Jeremy
2015-3-12

Outline

- Thread-safety in FCore
  - Motivation: Why the original is thread-unsafe
  - First try
  - Thunk
- Working with PHOAS
  - Motivation: Make simplifier work correctly
  - Three rewrite rules
  - Implementation
- Future and related work

Motivation

Motivation

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

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

Motivation

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

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

Motivation

(CJD-Bind1)

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

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

Each FC class is followed by a Function instance

Motivation, cont.

```
sum (x : Int) : Int =
  if x == 0
  then 0
  else x + sum (x - 1)
```

Motivation, cont.

```
sum (x : Int) : Int =
  if x == 0
  then 0
  else x + sum (x - 1)
```

```
class Sum extends Function
{
    Function x1 = this;
    public void apply ()
    {
        final Integer x3 = (Integer) x1.arg;
        if (x3 == 0)
        {
            res = 0;
        }
        else
        {
            Function x7 = x1;
            x7.arg = x3 - 1;
            x7.apply();
            final Integer x8 = (Integer) x7.res;
            res = x3 + x8;
        }
    }
}
Function sum = new Sum();
```

Motivation, cont.

```
sum (x : Int) : Int =
  if x == 0
  then 0
  else x + sum (x - 1)
```

```
class Sum extends Function
{
    Function x1 = this;
    public void apply ()
    {
        final Integer x3 = (Integer) x1.arg;
        if (x3 == 0)
        {
            res = 0;
        }
        else
        {
            Function x7 = x1;
            x7.arg = x3 - 1;
            x7.apply();
            final Integer x8 = (Integer) x7.res;
            res = x3 + x8;
        }
    }
}
Function sum = new Sum();
```

What if shared by multiple threads?

Motivation, cont.

```
sum 600
sum 500
```

Motivation, cont.

Sequential code

```
sum 600
sum 500
```

```
sum.arg = 600L;
sum.apply();
System.out.println("sum 6000 = " + sum.res);
sum.arg = 500L;
sum.apply();
System.out.println("sum 5000 = " + sum.res);
```

Motivation, cont.

Sequential code

```
sum 600
sum 500
```

```
sum.arg = 600L;
sum.apply();
System.out.println("sum 6000 = " + sum.res);
sum.arg = 500L;
sum.apply();
System.out.println("sum 5000 = " + sum.res);
```

```
SEQUENTIAL CODE (CORRECT RESULT)
sum 600 = 180300
sum 500 = 125250
```

Motivation, cont.

```
sum 600
sum 500
```

Motivation, cont.

```
sum 600
sum 500
```

Thread-unsafe code

```
Thread thread1 = new Thread() {
    public void run() {
        sum.arg = 600L;
        sum.apply();
        res1.res = (Long) sum.res;
    }
};

Thread thread2 = new Thread() {
    public void run() {
        sum.arg = 500L;
        sum.apply();
        res2.res = (Long) sum.res;
    }
};
```

Motivation, cont.

```
sum 600
sum 500
```

Thread-unsafe code

```
Thread thread1 = new Thread() {
    public void run() {
        sum.arg = 600L;
        sum.apply();
        res1.res = (Long) sum.res;
    }
};

Thread thread2 = new Thread() {
    public void run() {
        sum.arg = 500L;
        sum.apply();
        res2.res = (Long) sum.res;
    }
};
```

```
THREAD-UNSAFE CODE (SOMETIMES GIVES INCORRECT RESULT)
sum 600 = 186945
sum 500 = 34848
```

The Problem …

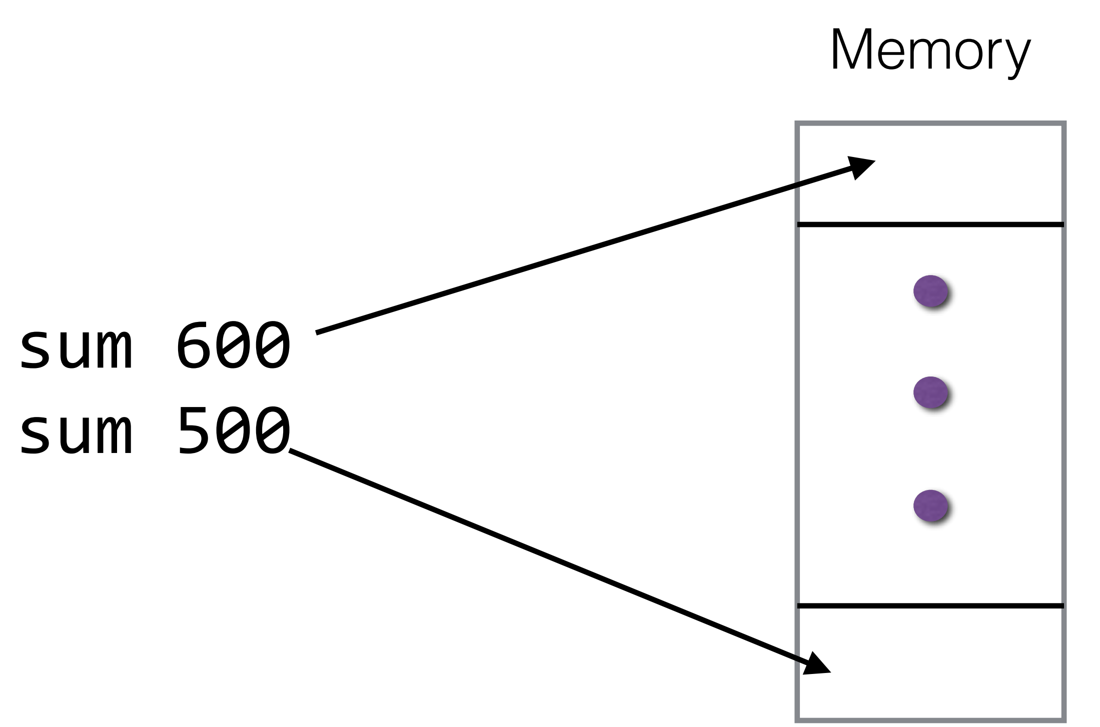
- The *Function* instance representing *sum* is shared by multiple threads
- There will be a race condition and results of function invocations may be incorrect
- Depending on the interleaving of two threads, one might overwrite some fields of the other
- JVM is inherently a current platform

First try

- The idea is to enforce a separate instance for each thread by allocating the object at its **call site** rather than at its definition site

First try

- The idea is to enforce a separate instance for each thread by allocating the object at its **call site** rather than at its definition site



First try, cont.

Thread-unsafe code

```
Thread thread1 = new Thread() {
    public void run() {
        sum.arg = 600L;
        sum.apply();
        res1.res = (Long) sum.res;
    }
};

Thread thread2 = new Thread() {
    public void run() {
        sum.arg = 500L;
        sum.apply();
        res2.res = (Long) sum.res;
    }
};
```

First try, cont.

Thread-unsafe code

```
Thread thread1 = new Thread() {
    public void run() {
        sum.arg = 600L;
        sum.apply();
        res1.res = (Long) sum.res;
    }
};

Thread thread2 = new Thread() {
    public void run() {
        sum.arg = 500L;
        sum.apply();
        res2.res = (Long) sum.res;
    }
};
```

Thread-safe code

```
thread1 = new Thread() {
    public void run() {
        Function sum1 = new Sum();
        sum1.arg = 6000L;
        sum1.apply();
        res1.res = (Long) sum1.res;
    }
};

thread2 = new Thread() {
    public void run() {
        Function sum2 = new Sum();
        sum2.arg = 5000L;
        sum2.apply();
        res2.res = (Long) sum2.res;
    }
};
```

First try, cont.

Thread-unsafe code

```
Thread thread1 = new Thread() {
    public void run() {
        sum.arg = 600L;
        sum.apply();
        res1.res = (Long) sum.res;
    }
};

Thread thread2 = new Thread() {
    public void run() {
        sum.arg = 500L;
        sum.apply();
        res2.res = (Long) sum.res;
    }
};
```

Thread-safe code

```
thread1 = new Thread() {
    public void run() {
        Function sum1 = new Sum();
        sum1.arg = 6000L;
        sum1.apply();
        res1.res = (Long) sum1.res;
    }
};

thread2 = new Thread() {
    public void run() {
        Function sum2 = new Sum();
        sum2.arg = 5000L;
        sum2.apply();
        res2.res = (Long) sum2.res;
    }
};
```

First try, cont.

Thread-unsafe code

```
Thread thread1 = new Thread() {
    public void run() {
        sum.arg = 600L;
        sum.apply();
        res1.res = (Long) sum.res;
    }
};

Thread thread2 = new Thread() {
    public void run() {
        sum.arg = 500L;
        sum.apply();
        res2.res = (Long) sum.res;
    }
};
```

Thread-safe code

```
thread1 = new Thread() {
    public void run() {
        Function sum1 = new Sum();
        sum1.arg = 6000L;
        sum1.apply();
        res1.res = (Long) sum1.res;
    }
};

thread2 = new Thread() {
    public void run() {
        Function sum2 = new Sum();
        sum2.arg = 5000L;
        sum2.apply();
        res2.res = (Long) sum2.res;
    }
};
```

```
THREAD-SAFE CODE (ALWAYS GIVES CORRECT RESULT)
sum 600 = 180300
sum 500 = 125250
```

Unfortunately …

- The previous example only involves first order functions.
- For higher-order functions, we are out of luck.

Unfortunately …

- The previous example only involves first order functions.
- For higher-order functions, we are out of luck.

```
apply (f : Int -> Int) (x : Int) : Int = f x
```

Unfortunately …

- The previous example only involves first order functions.
- For higher-order functions, we are out of luck.

```
apply (f : Int -> Int) (x : Int) : Int = f x
```

corresponding Java code
a programmer may use

```
int apply (Function f, int x) { … }
```

Unfortunately …

- The previous example only involves first order functions.
- For higher-order functions, we are out of luck.

```
apply (f : Int -> Int) (x : Int) : Int = f x
```

corresponding Java code
a programmer may use

```
int apply (Function f, int x) { … }
```

same instance in
each thread

Contribution

- We present a new approach to make FCore thread-safe
- We implement a Thunk interface to delay memory allocation

Caveat

- Thread safety is a complicated issue in read world applications
- Writing thread-safe code is a “black art”, since you can’t reproduce all possible interactions between threads
- There are two approaches: to avoid shared state or use some sort of synchronization

First, let's clarify …

- In our case, what are the exact threading scenarios we are concerned about?

- How a programmer can use the generated Java code in a multithreading setting?

User case

- For the moment, our language doesn’t support some sort of parallel operator yet
- A programmer will import some code generated by the compiler
- What kind of code (Class? Interface? Object?) we provide to programmers is the key to the thread-safety problem in our domain

Avoid shared state

- The idea is the same with our first try. We enforce a separate instance for each thread by allocating the object at its **call site** rather than at its definition site

Avoid shared state

- The idea is the same with our first try. We enforce a separate instance for each thread by allocating the object at its **call site** rather than at its definition site

```
inc (x : Int) = x + 1
```

Avoid shared state

- The idea is the same with our first try. We enforce a separate instance for each thread by allocating the object at its **call site** rather than at its definition site

```
inc (x : Int) = x + 1
```

```
class Inc extends Function
{...}

Function inc = new Inc();
```

Avoid shared state

- The idea is the same with our first try. We enforce a separate instance for each thread by allocating the object at its **call site** rather than at its definition site

```
inc (x : Int) = x + 1
```

```
class Inc extends Function
{...}

Function inc = new Inc();
```

Avoid shared state

- The idea is the same with our first try. We enforce a separate instance for each thread by allocating the object at its **call site** rather than at its definition site

```
inc (x : Int) = x + 1
```

Avoid shared state

- The idea is the same with our first try. We enforce a separate instance for each thread by allocating the object at its **call site** rather than at its definition site

```
inc (x : Int) = x + 1
```

```
class Inc extends Function
{...}

Thunk inc = () -> new Inc();
```

Thunk interface

- Inspired by the concept of “thunk” in programming languages: a zero-argument function that delays the computation

Thunk interface

- Inspired by the concept of “thunk” in programming languages: a zero-argument function that delays the computation

```
public interface Thunk {
    public Function compute();
}
```

How it works

- With the *Thunk* interface, we can wrap the code of object allocation inside the *compute* method in order to delay memory allocation, e.g.

```
Thunk inc = () -> new Inc();
```

- In a multi-threading scenario, even if a *Thunk* instance is shared by multiple threads, each invocation of *compute* will result in a separate object instance

```
public interface Thunk {
    public Function compute();
}
```

How it works, cont.

- What a programmer gets now is a *Thunk* instance. Now he/she can make a thread-safe call.

```
inc (inc 0)
```

How it works, cont.

- What a programmer gets now is a *Thunk* instance. Now he/she can make a thread-safe call.

Thread safe

```
inc (inc 0)
```

→

```
Function inc1 = inc.compute();
inc1.arg = 0;
inc1.apply();
Function inc2 = inc.compute();
inc2.arg = inc1.res;
inc2.apply();
```

How it works, cont.

- What a programmer gets now is a *Thunk* instance. Now he/she can make a thread-safe call.

Thread safe

```
inc (inc 0)
```

```
Function inc1 = inc.compute();
inc1.arg = 0;
inc1.apply();
Function inc2 = inc.compute();
inc2.arg = inc1.res;
inc2.apply();
```

How it works, cont.

- The Thunk interface also works with higher order functions

How it works, cont.

- The Thunk interface also works with higher order functions

```
twice ≡ (λf:Int→Int).(λx:Int).f (f x)
```

How it works, cont.

- The Thunk interface also works with higher order functions

```
twice ≡ (λf:Int→Int).(λx:Int).f (f x)
```

```
interface Twice {
  default int twice (Thunk f, int x) {
    Function fun1 = f.compute();
    fun1.arg = (Integer) x;
    fun1.apply();
    Integer res1 = (Integer) fun1.res;
    Function fun2 = f.compute();
    fun2.arg = (Integer) res1;
    fun2.apply();
    Integer res2 = (Integer) fun2.res;
    return res2;
  }
}
```

How it works, cont.

- The Thunk interface also works with higher order functions

```
twice ≡ (λf:Int→Int).(λx:Int).f (f x)
```

```
interface Twice {
  default int twice (Thunk f, int x) {
    Function fun1 = f.compute();
    fun1.arg = (Integer) x;
    fun1.apply();
    Integer res1 = (Integer) fun1.res;
    Function fun2 = f.compute();
    fun2.arg = (Integer) res1;
    fun2.apply();
    Integer res2 = (Integer) fun2.res;
    return res2;
  }
}
```

How it works, cont.

- The Thunk interface also works with higher order functions

```
twice ≡ (λf:Int→Int).(λx:Int).f (f x)
```

```java
interface Twice {
  default int twice (Thunk f, int x) {
    Function fun1 = f.compute();
    fun1.arg = (Integer) x;
    fun1.apply();
    Integer res1 = (Integer) fun1.res;
    Function fun2 = f.compute();
    fun2.arg = (Integer) res1;
    fun2.apply();
    Integer res2 = (Integer) fun2.res;
    return res2;
  }
}
```

force the programmer to pass Thunk instance

Conclusion & future work

- The Thunk interface works well with System F binders and let expressions.
- At the moment, we are implementing a modified version to work with TCE
- We leave a full implementation for future work.

Second Part (QA)

Problem

- Not long before, simplifier will produce code with superfluous identity functions and lambda wrappings.
- Semantically, the generated code is correct. Practically, it is slow due to the generation of extra closures.

Problem, cont.

```
λ(f : Int -> Int). λ(x : Int). f x
```

Problem, cont.

```
λ(f : Int -> Int). λ(x : Int). f x
```

```
λ(f : Int -> Int). λ(x : Int). f ((λ(y : Int). y) x)
```

Problem, cont.

```
λ(f : Int -> Int). λ(x : Int). f x
```

```
λ(f : Int -> Int). λ(x : Int). f ((λ(y : Int). y) x)
```

Problem, cont.

```
λ(f : Int -> Int). λ(x : Int). f x
```

$\Downarrow$

```
λ(f : Int -> Int). λ(x : Int). f ((λ(y : Int). y) x)
```

```
λ(f : (Int -> Int) -> Int). λ(g : Int -> Int). f g
```

Problem, cont.

```
λ(f : Int -> Int). λ(x : Int). f x
```

```
λ(f : Int -> Int). λ(x : Int). f ((λ(y : Int). y) x)
```

```
λ(f : (Int -> Int) -> Int). λ(g : Int -> Int). f g
```

```
λ(f : (Int -> Int) -> Int). λ(g : Int -> Int).
     f ((λ(x : Int -> Int). λ(y : Int).
                             (λ(z : Int). z)
                               (x ((λ(d : Int). d) y))) g)
```

Problem, cont.

```
λ(f : Int -> Int). λ(x : Int). f x
```

```
λ(f : Int -> Int). λ(x : Int). f ((λ(y : Int). y) x)
```

```
λ(f : (Int -> Int) -> Int). λ(g : Int -> Int). f g
```

```
λ(f : (Int -> Int) -> Int). λ(g : Int -> Int).
     f ((λ(x : Int -> Int). λ(y : Int).
                                (λ(z : Int). z)
                                  (x ((λ(d : Int). d) y))) g)
```

Problem, cont.

```
λ(f : Int -> Int). λ(x : Int). f x
```

```
λ(f : Int -> Int). λ(x : Int). f ((λ(y : Int). y) x)
```

```
λ(f : (Int -> Int) -> Int). λ(g : Int -> Int). f g
```

```
λ(f : (Int -> Int) -> Int). λ(g : Int -> Int).
     f ((λ(x : Int -> Int). λ(y : Int).
                                (λ(z : Int). z)
                                  (x ((λ(d : Int). d) y))) g)
```

Let's do simplification by hand!

```
λ(f : (Int -> Int) -> Int). λ(g : Int -> Int).
     f ((λ(x : Int -> Int). λ(y : Int).
                             (λ(z : Int). z)
                               (x ((λ(d : Int). d) y))) g)
```

Problem, cont.

```
λ(f : (Int -> Int) -> Int). λ(g : Int -> Int).
     f ((λ(x : Int -> Int). λ(y : Int).
                                  (λ(z : Int). z)
                                    (x ((λ(d : Int). d) y))) g)
```

Problem, cont.

```
λ(f : (Int -> Int) -> Int). λ(g : Int -> Int).
     f ((λ(x : Int -> Int). λ(y : Int).
                                 (λ(z : Int). z)
                                   (x ((λ(d : Int). d) y))) g)
```

peval

```
\ (f : (Int -> Int) -> Int).
  \ (g : Int -> Int).
    f
      let x = g
      in
      \ (y : Int).
        let z = x (let d = y in d)
        in z
```

Problem, cont.

```
λ(f : (Int -> Int) -> Int). λ(g : Int -> Int).
     f ((λ(x : Int -> Int). λ(y : Int).
                             (λ(z : Int). z)
                               (x ((λ(d : Int). d) y))) g)
```

peval

```
\ (f : (Int -> Int) -> Int).
  \ (g : Int -> Int).
    f
      let x = g
      in
      \ (y : Int).
        let z = x (let d = y in d)
        in z
```

?

```
λ(f : (Int -> Int) -> Int).
  λ(g : Int -> Int).
    f g
```

Contribution

- We propose three rewrite rules to make simplifier work correctly
- We presents a PHOAS-based implementation of the rewriting process

Rewrite rules

- We need to perform rewriting on the core AST
- There are 3 basic patterns

Rule 1

let x = y in e ~> [x -> y] e

(where y is a variable)

Rule 2

let x = e in x ~> e

Rule 3

\x. y x   ~>   y

(where y is a variable)

Rule 3, cont.

\x. y x   ~>   y

(where y is a variable)

- The condition (y is a variable) is important!
- In general, this is not a valid transformation in call-by-value

Rule 3, cont.

\x. y x ~> y

(where y is a variable)

- The condition (y is a variable) is important!
- In general, this is not a valid transformation in call-by-value

```
(\f . 1)  (\x . infinite x)    ==>  1

(\f . 1)  infinite             ==>   does not terminate
```

Let’s do rewriting by hand!

```
\ (f : (Int -> Int) -> Int).
  \ (g : Int -> Int).
    f
      let x = g
      in
      \ (y : Int).
        let z = x (let d = y in d)
        in z
```

1. let x = y in e ~> [x -> y] e (where y is a variable)
2. let x = e in x ~> e
3. \x. y x ~> y (where y is a variable)

PHOAS

- In the back-end, we represent ASTs with Parametric Higher Order Abstract Syntax (PHOAS)
- The function space of the meta language is used to encode the binders of the object language
- It has benefits of avoiding common issues of alpha-equivalence and name-capturing

PHOAS, cont.

```haskell
data Expr t e
  = Var Src.ReaderId e
  | Lit Src.Lit

  -- Binders we have: λ, fix, letrec, and Λ
  | Lam Src.ReaderId (Type t) (e -> Expr t e)
  | Fix Src.ReaderId Src.ReaderId
        (e -> e -> Expr t e)
        (Type t)  -- t1
        (Type t)  -- t
  | Let Src.ReaderId (Expr t e) (e -> Expr t e)
```

Implementation

- It turns out, implementing Rule 1 is fairly easy, while for Rule 2 and 3, it's a bit involved

- The difficulty arises because of the abstract type arguments

```
data Expr t e
```

- They should not be be instantiated to concrete types when constructing ASTs

Implementation, cont.

```
-- Rule 1: let x = y in e                  => [x -> y] e   (where y is a variable)
rewrite1 :: Expr t e -> Expr t e
rewrite1 (Let _ (Var _ n) f) = rewrite1 (f n)
rewrite1 e = mapExpr rewrite1 e
```

Implementation, cont.

Rule 2: let x = e in x ~> e

- The job is to recognise this pattern. We can't just pattern match on it directly
- We need to distinguish different variables
- One way is to instantiate the type argument *e* to Int
- The requirement is, after transformation, the type argument should still be abstract

Implementation, cont.

`Expr t Int` $\xrightarrow{?}$ `Expr t e`

Implementation, cont.

`Expr t Int` $\xrightarrow{?}$ `Expr t e`

- The idea is simple. Use a map to record mappings of Int and e

`Map Int e`

Implementation, cont.

```
Expr t Int        ?        Expr t e
```

- The idea is simple. Use a map to record mappings of Int and e

```
Map Int e
```

```
rewrite2 :: Map Int e -> Expr t Int -> Int -> Expr t e
```

Implementation, cont.

$\texttt{Expr t Int} \xrightarrow{?} \texttt{Expr t e}$

- The idea is simple. Use a map to record mappings of Int and e

```
Map Int e
```

```
rewrite2 :: Map Int e -> Expr t Int -> Int -> Expr t e
```

different variables

Implementation, cont.

```
Expr t Int        ?        Expr t e
```

- The idea is simple. Use a map to record mappings of Int and e

```
Map Int e
```

```
rewrite2 :: Map Int e -> Expr t Int -> Int -> Expr t e
```

different variables

a counter

Implementation, cont.

`Expr t Int` $\xrightarrow{?}$ `Expr t e`

```haskell
rewrite2 :: Map.Map Int e -> Expr t Int -> Int -> Expr t e
rewrite2 env expr num =
  case expr of
    Var n v ->
      Var n
          (fromJust $
           Map.lookup v env)
    Lit n -> Lit n
    Lam n t f ->
      Lam n
          t
          (\e ->
             rewrite2 (Map.insert num e env)
                      (f num)
                      (num + 1))
```

Implementation, cont.

`Expr t Int` $\xrightarrow{?}$ `Expr t e`

Implementation, cont.

`Expr t Int` ——?——> `Expr t e`

```
case expr of
  Let n e f ->
    case f num of
      Var _ num' ->
        if num == num'
           then rewrite2 env e num
           else Let n
                    (rewrite2 env e num)
                    (\b ->
                       rewrite2 (Map.insert num b env)
                                (f num)
                                (num + 1))
      _ ->
        Let n
            (rewrite2 env e num)
            (\b ->
               rewrite2 (Map.insert num b env)
                        (f num)
                        (num + 1))
```

Future & related work

Future & related work

- Related work
  - Heavily inspired by the paper *Functional Programming with Structured Graphs*
  - Using PHOAS to define cyclic structures (e.g. graphs), easy to observe and manipulate sharing and cycles

Future & related work

Future & related work

- Future work
  - The approach needs several iterations of rewritings
  - We need structured equality on ASTs (too time consuming)

Thanks you!