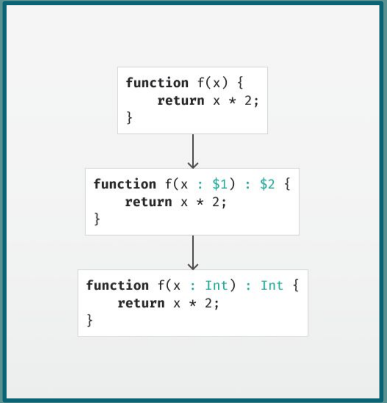
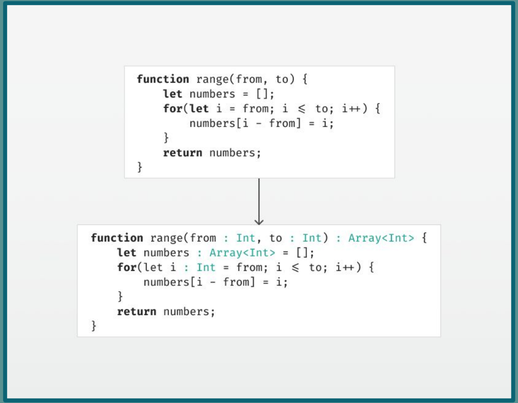
Type Inference by Example



```
function f(x) {
    return x * 2;
}
```

```
function f(x : $1) : $2 {
    return x * 2;
}
```

```
function f(x : Int) : Int {
    return x * 2;
}
```



```
function range(from, to) {
    let numbers = [];
    for(let i = from; i ⩽ to; i++) {
        numbers[i - from] = i;
    }
    return numbers;
}
```

```
function range(from : Int, to : Int) : Array<Int> {
    let numbers : Array<Int> = [];
    for(let i : Int = from; i ⩽ to; i++) {
        numbers[i - from] = i;
    }
    return numbers;
}
```

Joakim Ahnfelt-Rønne
github.com/ahnfelt

Mødegruppe for Funktionelle Københavnere
April 28, 2020, online

```
function f(x) {
   return x * 2;
}
```

Environment:

x : $1

Type variables

```
function f(x : $1) : $2 {
   return x * 2;
}
```

Environment:

x : $1

```
function f(x : $1) : $2 {
   return x * 2;
}
```

* : (Int, Int) => Int

```
function f(x : $1) : $2 {
   return x * 2;
}
```

* : (Int, Int) => Int

Environment:

x : $1

Type constraints:

Int == $1

```
function f(x : $1) : $2 {
    return x * 2;
}
```

* : (Int, Int) => Int

Environment:

x : $1

Type constraints:

Int == $1
$2 == Int

```
function f(x : $1) : $2 {
   return x * 2;
}
```

Environment:

x : $1

Type constraints:

Int == $1
$2 == Int

```
function f(x : $1) : $2 {
   return x * 2;
}
```

Environment:

x : $1

Type constraints:

Int == $1
$2 == Int

↓

Substitution:

$1 := Int
$2 := Int

```
function f(x : Int) : Int {
   return x * 2;
}
```

Environment:

x : $1

Type constraints:

Int == $1
$2 == Int

Substitution:

$1 := Int
$2 := Int

Longer example

```
function range(from, to) {
   let numbers = [];
   for(let i = from; i ⩽ to; i++) {
       numbers[i - from] = i;
   }
   return numbers;
}
```

Environment:

Type constraints:

Substitution:

```
function range(from : $1, to : $2) : $3 {
   let numbers : $4 = [];
   for(let i : $5 = from; i <= to; i++) {
       numbers[i - from] = i;
   }
   return numbers;
}
```

Environment:

from : $1
to : $2

numbers : $4
i : $5

Type constraints:

Substitution:

```
function range(from : $1, to : $2) : $3 {
    let numbers : $4 = [];
    for(let i : $5 = from; i ⩽ to; i++) {
        numbers[i - from] = i;
    }
    return numbers;
}
```

Environment:

from : $1	numbers : $4
to : $2	i : $5

Type constraints:

$4 == Array<$6>

Substitution:

```
function range(from : $1, to : $2) : $3 {
   let numbers : $4 = [];
   for(let i : $5 = from; i ⩽ to; i++) {
      numbers[i - from] = i;
   }
   return numbers;
}
```

Environment:

from : $1
to : $2

numbers : $4
i : $5

Type constraints:

$4 == Array<$6>

$5 == $1
$5 == $2
$5 == Int
$2 == Int

Substitution:

```
function range(from : $1, to : $2) : $3 {
   let numbers : $4 = [];
   for(let i : $5 = from; i ⩽ to; i++) {
       numbers[i - from] = i;
   }
   return numbers;
}
```

Environment:

from : $1
to : $2

numbers : $4
i : $5

Type constraints:

$4 == Array<$6>
$4 == Array<$5>

$5 == $1
$5 == $2
$5 == Int
$2 == Int

Substitution:

```
function range(from : $1, to : $2) : $3 {
    let numbers : $4 = [];
    for(let i : $5 = from; i ⩽ to; i++) {
        numbers[i - from] = i;
    }
    return numbers;
}
```

Environment:

from : $1	numbers : $4
to : $2	i : $5

Type constraints:

$4 == Array<$6>	$5 == $1
$4 == Array<$5>	$5 == $2
...	$5 == Int
$3 == $4	$2 == Int

Substitution:

```
function range(from : $1, to : $2) : $3 {
   let numbers : $4 = [];
   for(let i : $5 = from; i ⩽ to; i++) {
       numbers[i - from] = i;
   }
   return numbers;
}
```

Environment:

from : $1	numbers : $4
to : $2	i : $5

Type constraints:

$4 == Array<$6>	$5 == $1
$4 == Array<$5>	$5 == $2
...	$5 == Int
$3 == $4	$2 == Int

Substitution:

$4 := Array<$6>

```
function range(from : $1, to : $2) : $3 {
   let numbers : $4 = [];
   for(let i : $5 = from; i ⩽ to; i++) {
       numbers[i - from] = i;
   }
   return numbers;
}
```

Environment:

from : $1	numbers : $4
to : $2	i : $5

Type constraints:

$4 == Array<$6>	$5 == $1
Array<$6> == Array<$5>	$5 == $2
...	$5 == Int
$3 == $4	$2 == Int

Substitution:

$4 := Array<$6>

```
function range(from : $1, to : $2) : $3 {
   let numbers : $4 = [];
   for(let i : $5 = from; i ⩽ to; i++) {
       numbers[i - from] = i;
   }
   return numbers;
}
```

Environment:

from : $1
to : $2
numbers : $4
i : $5

Type constraints:

$4 == Array<$6>
$6 == $5
...
$3 == $4
$5 == $1
$5 == $2
$5 == Int
$2 == Int

Substitution:

$4 := Array<$6>

```
function range(from : $1, to : $2) : $3 {
   let numbers : $4 = [];
   for(let i : $5 = from; i ⩽ to; i++) {
       numbers[i - from] = i;
   }
   return numbers;
}
```

Environment:

from : $1
to : $2

numbers : $4
i : $5

Type constraints:

$4 == Array<$6>
$6 == $5
...
$3 == $4

$5 == $1
$5 == $2
$5 == Int
$2 == Int

Substitution:

$1 := Int
$2 := Int
$3 := Array<Int>
$4 := Array<Int>
$5 := Int
$6 := Int

```
function range(from : Int, to : Int) : Array<Int> {
   let numbers : Array<Int> = [];
   for(let i : Int = from; i ⩽ to; i++) {
       numbers[i - from] = i;
   }
   return numbers;
}
```

Environment:

from : $1
to : $2

numbers : $4
i : $5

Type constraints:

$4 == Array<$6>
$6 == $5
...
$3 == $4

$5 == $1
$5 == $2
$5 == Int
$2 == Int

Substitution:

$1 := Int
$2 := Int
$3 := Array<Int>
$4 := Array<Int>
$5 := Int
$6 := Int

Calling generic functions

```
function map<A, B>(array : Array<A>, body : A ⇒ B) : Array<B>
```

```
function square(items) {
   return map(
      items,
      x ⇒ x * x
   );
}
```

Environment:

map : ...

Type constraints:

Substitution:

```
function map(array : Array<$3>, body : $3 ⇒ $4) : Array<$4>
```

```
function square(items) {
   return map(
      items,
      x ⇒ x * x
   );
}
```

Environment:

map : ...

Type constraints:

Substitution:

```
function map(array : Array<$3>, body : $3 ⇒ $4) : Array<$4>
```

```
function square(items : $1) : $2 {
   return map<$3, $4>(
      items,
      (x : $5) ⇒ x * x
   );
}
```

Environment:

items : $1 **map : ...**

x : $5 (only within the lambda)

Type constraints:

Substitution:

```
function map(array : Array<$3>, body : $3 ⇒ $4) : Array<$4>
```

```
function square(items : $1) : $2 {
   return map<$3, $4>(
      items,
      (x : $5) ⇒ x * x
   );
}
```

Environment:

items : $1　　　**map : ...**

x : $5 (only within the lambda)

Type constraints:

Substitution:

```
function map(array : Array<$3>, body : $3 ⇒ $4) : Array<$4>
```

```
function square(items : $1) : $2 {
   return map<$3, $4>(
      items,
      (x : $5) ⇒ x * x
   );
}
```

Environment:

items : $1 **map : ...**

x : $5 (only within the lambda)

Type constraints:

$1 == Array<$3>

Substitution:

```
function map(array : Array<$3>, body : $3 ⇒ $4) : Array<$4>
```

```
function square(items : $1) : $2 {
   return map<$3, $4>(
      items,
      (x : $5) ⇒ x * x
   );
}
```

Environment:

items : $1 **map : ...**

x : $5 (only within the lambda)

Type constraints:

$1 == Array<$3>
($3 => $4) == ($5 => $6)

Substitution:

```
function map(array : Array<$3>, body : $3 ⇒ $4) : Array<$4>
```

```
function square(items : $1) : $2 {
   return map<$3, $4>(
      items,
      (x : $5) ⇒ x * x
   );
}
```

Environment:

items : $1 **map : ...**

x : $5 (only within the lambda)

Type constraints:

$1 == Array<$3>
($3 => $4) == ($5 => $6)
$5 == Int
$6 == Int

Substitution:

```
function map(array : Array<$3>, body : $3 ⇒ $4) : Array<$4>
```

```
function square(items : $1) : $2 {
   return map<$3, $4>(
      items,
      (x : $5) ⇒ x * x
   );
}
```

Environment:

items : $1 **map : ...**

x : $5 (only within the lambda)

Type constraints:

$1 == Array<$3>
($3 => $4) == ($5 => $6)
$5 == Int
$6 == Int $2 == Array<$4>

Substitution:

```
function map(array : Array<$3>, body : $3 ⇒ $4) : Array<$4>
```

```
function square(items : $1) : $2 {
    return map<$3, $4>(
        items,
        (x : $5) ⇒ x * x
    );
}
```

Environment:

items : $1 **map : ...**

x : $5 (only within the lambda)

Type constraints:

$1 == Array<$3>
$3 == $5 $4 == $6
$5 == Int
$6 == Int $2 == Array<$4>

Substitution:

```
function map(array : Array<$3>, body : $3 ⇒ $4) : Array<$4>
```

```
function square(items : $1) : $2 {
   return map<$3, $4>(
      items,
      (x : $5) ⇒ x * x
   );
}
```

Environment:

items : $1 **map : ...**

x : $5 (only within the lambda)

Type constraints:

$1 == Array<$3>	
$3 == $5	$4 == $6
$5 == Int	
$6 == Int	$2 == Array<$4>

Substitution:

$1 := Array<Int>
$2 := Array<Int>
$3 := Int
$4 := Int
$5 := Int
$6 := Int

```
function map(array : Array<$3>, body : $3 ⇒ $4) : Array<$4>
```

```
function square(items : Array<Int>) : Array<Int> {
   return map<Int, Int>(
      items,
      (x : Int) ⇒ x * x
   );
}
```

Environment:

items : $1 **map : ...**

x : $5 (only within the lambda)

Type constraints:

$1 == Array<$3>
$3 == $5 $4 == $6
$5 == Int
$6 == Int $2 == Array<$4>

Substitution:

$1 := Array<Int>
$2 := Array<Int>
$3 := Int
$4 := Int
$5 := Int
$6 := Int

Unification

Or: How to solve equality constraints

$$\text{unify} : \overbrace{\text{Type} \times \text{Type}}^{\text{Equality constraint}} \rightarrow \text{Substitution}$$

"Given two types,
find the substitution (if any),
that makes them equal."

unify : Type × Type ⟶ Substitution

Types:

`Int` — a plain type

`Array<Int>` — a generic type

`(Int, Int) ⇒ Int` — a function type

`$1` — a type variable

unify : Type × Type → Substitution

Types:

$$\left.\begin{array}{l}\texttt{Int} \text{ — a plain type} \\ \texttt{Array<Int>} \text{ — a generic type} \\ \texttt{(Int, Int)} \Rightarrow \texttt{Int} \text{ — a function type}\end{array}\right\} \text{Type constructors, fully applied.}$$

$$\left.\texttt{\$1} \text{ — a type variable}\right\} \text{Type variable.}$$

unify : Type × Type → Substitution

```
sealed abstract class Type

case class TConstructor(
    name : String,
    generics : List[Type] = List()
) extends Type

case class TVariable(
    index : Int
) extends Type
```

```
Int — TConstructor("Int")

Array<Int> — TConstructor("Array", List(TConstructor("Int")))

(Int, Int) ⇒ Int — TConstructor("Function2", List(TConstructor("Int"), TConstructor("Int"), TConstructor("Int")))

$1 — TVariable(1)
```

unify : Type × Type → Substitution

```
$1 := Int
$2 := Array<$1>
$3 := $1 ⇒ $2
…
```

```
Map[Int, Type]
```

unify : Type × Type → Substitution

```
freshTypeVariable() : Type

get(index : Int) : Type

substitute(t : Type) : Type

substituteExpression(e : Expression) : Expression
```

unify : Type × Type → Substitution

```
class Substitution() {

  private var typeVariables : Map[Int, Type] = Map()

  def freshTypeVariable() : TVariable = {
    val result = TVariable(typeVariables.size)
    typeVariables += (typeVariables.size → result)
    result
  }

  // ... as well as the following methods ...

}
```

$0 := $0
$1 := $1
$2 := $2
...

unify : Type × Type ⟶ Substitution

```
def get(index : Int) : Type =
  typeVariables(index) match {
    case TVariable(i) if i ≠ index ⇒ get(i)
    case t ⇒ t
  }
```

$1 := $2
$2 := $3
$3 := $4
$4 := Int

unify : Type × Type ⟶ Substitution

```
def get(index : Int) : Type =
  typeVariables(index) match {
    case TVariable(i) if i ≠ index ⇒
      val t = get(i)
      typeVariables(index) = t
      t
    case t ⇒ t
  }
```

Path compression

\$1 := \$2	\$1 := Int
\$2 := \$3	\$2 := Int
\$3 := \$4	\$3 := Int
\$4 := Int	\$4 := Int

unify : Type × Type → Substitution

```
def substitute(t : Type) : Type =
  t match {
    case TVariable(i) ⇒
      if(has(i)) substitute(get(i)) else t
    case TConstructor(name, generics) ⇒
      TConstructor(name, generics.map(t ⇒ substitute(t)))
  }
```

```
def has(index : Int) : Boolean =
  typeVariables(index) match {
    case TVariable(i) ⇒ i ≠ index
    case _ ⇒ true
  }
```

unify : Type × Type → Substitution

```
def substituteExpression(e : Expression) : Expression =
  e match {

    case ELet(name, typeAnnotation, value, body) ⇒
      val newTypeAnnotation = typeAnnotation.map(substitute)
      val newValue = substituteExpression(value)
      val newBody = substituteExpression(body)
      ELet(name, newTypeAnnotation, newValue, newBody)

    // ...

  }
```

unify : Type × Type ⟶ Substitution

```
def unify(t1 : Type, t2 : Type) : Unit = (t1, t2) match {
  case (TVariable(i1), TVariable(i2)) if i1 == i2 =>
  case (TVariable(i), _) if has(i) => unify(get(i), t2)
  case (_, TVariable(i)) if has(i) => unify(t1, get(i))
  case (TVariable(i), _) =>
    typeVariables(i) = t2
  case (_, TVariable(i)) =>
    typeVariables(i) = t1
  case (TConstructor(name1, generics1), TConstructor(name2, generics2)) =>
    assert(name1 == name2 && generics1.size == generics2.size)
    for((t1, t2) <- generics1.zip(generics2)) unify(t1, t2)
}
```

unify : Type × Type ⟶ Substitution

```
def unify(t1 : Type, t2 : Type) : Unit = (t1, t2) match {
  case (TVariable(i1), TVariable(i2)) if i1 == i2 ⇒
  case (TVariable(i), _) if has(i) ⇒ unify(get(i), t2)
  case (_, TVariable(i)) if has(i) ⇒ unify(t1, get(i))
  case (TVariable(i), _) ⇒
    assert(!occursIn(i, t2))
    typeVariables(i) = t2
  case (_, TVariable(i)) ⇒
    assert(!occursIn(i, t1))
    typeVariables(i) = t1
  case (TConstructor(name1, generics1), TConstructor(name2, generics2)) ⇒
    assert(name1 == name2 && generics1.size == generics2.size)
    for((t1, t2) ← generics1.zip(generics2)) unify(t1, t2)
}
```

Infinite type: $1 = Array<$1>

```
def occursIn(index : Int, t : Type) : Boolean = t match {
   case TVariable(i) if has(i) ⇒ occursIn(index, get(i))
   case TVariable(i) ⇒ i == index
   case TConstructor(_, generics) ⇒ generics.exists(t ⇒ occursIn(index, t))
}
```

Type inference

infer : Expression $\times$ Environment $\longrightarrow$ Type

"Given an expression
in an environment,
compute its type (if any)."

$$\text{infer} : \text{Expression} \times \underbrace{\text{Environment}}_{\texttt{Map[String, Type]}} \rightarrow \text{Type}$$

The variables that are in scope.

infer : Expression × Environment ⟶ Type

```scala
sealed abstract class Expression
case class ELambda(x : String, e : Expression) extends Expression
case class EApply(e1 : Expression, e2 : Expression) extends Expression
case class EVariable(x : String) extends Expression
```

infer : Expression × Environment ⟶ Type

```
case ELambda(x, e) ⇒
  val t1 = freshTypeVariable()
  val t2 = infer(e, environment.updated(x, t1))
  TConstructor("Function", List(t1, t2))

case EApply(e1, e2) ⇒
  val t1 = infer(e1, environment)
  val t2 = infer(e2, environment)
  val t3 = freshTypeVariable()
  unify(t1, TConstructor("Function", List(t2, t3)))
  t3

case EVariable(x) ⇒
  environment.getOrElse(x,
    throw TypeError("Variable not in scope: " + x)
  )
```

infer : Expression × Environment ⟶ Type

```
val e = ELambda("x", EApply(EVariable("-"), EVariable("x")))

            x ⇒ -x
```

infer : Expression × Environment ⟶ Type

```
val e = ELambda("x", EApply(EVariable("-"), EVariable("x")))

        x ⇒ -x

val t1 = infer(e, Map("-", Int ⇒ Int))

t1 == TConstructor("Function", List(TVariable(0), TVariable(1)))

      $0 ⇒ $1
```

infer : Expression × Environment ⟶ Type

```
val e = ELambda("x", EApply(EVariable("-"), EVariable("x")))

        x ⇒ -x

val t1 = infer(e, Map("-", Int ⇒ Int))

t1 == TConstructor("Function", List(TVariable(0), TVariable(1)))

      $0 ⇒ $1


val t2 = substitute(t1)

t2 == TConstructor("Function",
          List(TConstructor("Int"), TConstructor("Int")))

      Int ⇒ Int
```

infer : Expression × Environment ⟶ Type

```
val e = ELambda("x", EVariable("x"))

        x ⇒ x

val t1 = infer(e, Map())

t1 == TConstructor("Function", List(TVariable(0), TVariable(0)))

      $0 ⇒ $0
```

infer : Expression × Environment ⟶ Type

done?

```
x ⇒ -x  :  Int ⇒ Int
```

```
x ⇒ x  :  $0 ⇒ $0
```

infer : Expression × Environment → Type

```
x ⇒ -x  :  Int ⇒ Int
```

What is the type of x? (x : Int) => -x

```
x ⇒ x  :  $0 ⇒ $0
```

infer : Expression × Environment ⟶ Type

```
x ⇒ -x  :  Int ⇒ Int
```

What is the type of x? (x : Int) => -x

```
x ⇒ x  :  $0 ⇒ $0
```

What about generic types? function identity<A>(x : A) : A

infer : Expression × Environment ⟶ Type

```
x ⇒ -x  :  Int ⇒ Int
```

What is the type of x? (x : Int) => -x

```
x ⇒ x  :  $0 ⇒ $0
```

What about generic types? function identity<A>(x : A) : A

What about recursive functions?

Type reconstruction

… and a richer language!

infer : Environment $\times$ Type $\times$ Expression $\rightarrow$ Expression

"Given an environment
and an expected type,
fill in the missing types
in the expression."

infer : Environment × Type × Expression → Expression

```
(x : Int, y : String) : Bool ⇒ f(x, y)
```

```
case class ELambda(
  parameters : List[Parameter],
  returnType : Option[Type],
  body : Expression
) extends Expression

case class EApply(
  function : Expression,
  arguments : List[Expression]
) extends Expression
```

```
case class Parameter(
  name : String,
  typeAnnotation : Option[Type]
)
```

infer : Environment × Type × Expression → Expression

```
let x : Int = 42;
x + x
```

```
case class ELet(
  name : String,
  typeAnnotation : Option[Type],
  value : Expression,
  body : Expression
) extends Expression
```

infer : Environment × Type × Expression → Expression

```
[1, 2, 3];
"Hello, World"
```

```
case class EInt(
  value : Int
) extends Expression

case class EString(
  value : String
) extends Expression
```

```
case class EArray(
  itemType : Option[Type],
  items : List[Expression],
) extends Expression

case class ESemicolon(
  before : Expression,
  after : Expression
) extends Expression
```

$$\text{infer} : \text{Environment} \times \text{Type} \times \text{Expression} \rightarrow \text{Expression}$$

infer : Environment × Type × Expression ⟶ Expression

```
case ELet(name, typeAnnotation, value, body) ⇒
  val newTypeAnnotation = typeAnnotation.getOrElse(substitution.freshTypeVariable())
  val newValue = infer(environment, newTypeAnnotation, value)
  val newEnvironment = environment.updated(name, newTypeAnnotation)
  val newBody = infer(newEnvironment, expectedType, body)
  ELet(name, Some(newTypeAnnotation), newValue, newBody)
```

infer : Environment × Type × Expression → Expression

```
case ELambda(parameters, returnType, body) ⇒
  val newReturnType = returnType.getOrElse(substitution.freshTypeVariable())
  val newParameterTypes = parameters.map { p ⇒
    p.typeAnnotation.getOrElse(substitution.freshTypeVariable())
  }
  val newParameters = parameters.zip(newParameterTypes).map { case (p, t) ⇒
    p.copy(typeAnnotation = Some(t))
  }
  val newEnvironment = environment ++ newParameters.map { p ⇒
    p.name → p.typeAnnotation.get
  }
  val newBody = infer(newEnvironment, newReturnType, body)
  substitution.unify(expectedType,
    TConstructor(s"Function${parameters.size}", newParameterTypes ++ List(newReturnType))
  )
  ELambda(newParameters, Some(newReturnType), newBody)
```

Generic types

& mutual recursion

infer : Environment × Type × Expression → Expression

```
case class GenericType(
   generics : List[String],
   uninstantiatedType : Type
)
```

```
function identity<A>(x : A) : A

identity : forall A. A ⇒ A

GenericType(List("A"),
    TConstructor("Function1", List(
        TConstructor("A"),
        TConstructor("A"),
    ))
)
```

infer : Environment × Type × Expression → Expression

Mutually recursive, generic functions.

```scala
case class EFunctions(
   functions : List[GenericFunction],
   body : Expression
) extends Expression
```

```scala
case class GenericFunction(
   name : String,
   typeAnnotation : Option[GenericType],
   lambda : Expression
)
```

```javascript
function even(x) { return x == 0 || odd(x - 1); }
function odd(x) { return x != 0 && even(x - 1); }
```

infer : Environment × Type × Expression → Expression

```
case EFunctions(functions, body) ⇒

  val recursiveEnvironment = environment ++ functions.map { function ⇒
    function.name → function.typeAnnotation.getOrElse(
      GenericType(List(), substitution.freshTypeVariable())
    )
  }.toMap

  val ungeneralizedFunctions = functions.map { function ⇒
    val uninstantiatedType = recursiveEnvironment(function.name).uninstantiatedType
    function.copy(lambda =
      infer(recursiveEnvironment, uninstantiatedType, function.lambda)
    )
  }

  // ...
```

```
function even(x) { return x == 0 || odd(x - 1); }
function odd(x) { return x ≠ 0 && even(x - 1); }
```

```
case EFunctions(functions, body) ⇒
  // ...

  val newFunctions = ungeneralizedFunctions.map { function ⇒
    if(function.typeAnnotation.nonEmpty) function else {
      val functionType = recursiveEnvironment(function.name).uninstantiatedType
      val (newTypeAnnotation, newLambda) =
        generalize(environment, functionType, function.lambda)
      function.copy(typeAnnotation = Some(newTypeAnnotation), lambda = newLambda)
    }
  }

  val newEnvironment = environment ++ newFunctions.map { function ⇒
    function.name → function.typeAnnotation.get
  }.toMap

  val newBody = infer(newEnvironment, expectedType, body)
  EFunctions(newFunctions, newBody)
```

```
function compose(f, g) { return x ⇒ f(g(x)) }
function compose(f : $3 ⇒ $1, g : $2 ⇒ $3) : $2 ⇒ $1
function compose<A, B, C>(f : C ⇒ A, g : B ⇒ C) : B ⇒ A
```

infer : Environment × Type × Expression → Expression

```
case EVariable(name, generics) ⇒

  val genericType = environment.get(name)
  val newGenerics = genericType.generics.map(_ ⇒ substitution.freshTypeVariable())
  val instantiation = genericType.generics.zip(newGenerics).toMap
  val variableType = instantiate(instantiation, genericType.uninstantiatedType)

  if(generics.nonEmpty) {
    assert(generics.size == genericType.generics.size)
    for((t, v) ← generics.zip(newGenerics)) substitution.unify(t, v)
  }

  substitution.unify(expectedType, variableType)
  EVariable(name, newGenerics)
```

```
map<Int, String>(values, x ⇒ format(x))
```

Generalization & instantiation

```javascript
function f(x) {
  function g(y) {
    return (x, y);
  }
  return x + 7;
}
```

```scala
def generalize(
  environment : Map[String, GenericType],
  t : Type,
  expression : Expression
) : (GenericType, Expression) = {

  val genericTypeVariables =
    substitution.freeInType(t) -- substitution.freeInEnvironment(environment)

  val genericNames =
    genericTypeVariables.map(_ → genericParameterNames.next()).toList

  val local = substitution.copy()

  for((i, name) ← genericNames) local.unify(local.get(i), TConstructor(name))

  val newExpression = local.substituteExpression(expression)

  val newType = local.substitute(t)

  GenericType(genericNames.map { case (_, name) ⇒ name }, newType) → newExpression

}
```

```scala
def freeInType(t : Type) : SortedSet[Int] = t match {
  case TVariable(i) if has(i) ⇒ freeInType(get(i))
  case TVariable(i) ⇒ SortedSet(i)
  case TConstructor(_, generics) ⇒
    generics.map(freeInType).fold(SortedSet[Int]()) { _ ++ _ }
}

def freeInGenericType(t : GenericType) : SortedSet[Int] = {
  freeInType(t.uninstantiatedType)
}

def freeInEnvironment(environment : Map[String, GenericType]) : SortedSet[Int] = {
  environment.values.map(freeInGenericType).fold(SortedSet[Int]()) { _ ++ _ }
}
```

```
def instantiate(instantiation : Map[String, Type], t : Type) : Type = t match {

  case TConstructor(name, generics) ⇒
    instantiation.get(name).map { instantiationType ⇒
      assert(generics.isEmpty)
      instantiationType
    }.getOrElse {
      TConstructor(name, generics.map(t ⇒ instantiate(instantiation, t)))
    }

  case TVariable(i) if substitution.has(i) ⇒
    instantiate(instantiation, substitution.get(i))

  case TVariable(i) ⇒
    t

}
```

```
function map<A, B>(array : Array<A>, f : A ⇒ B) : Array<B>

(Array<$1>, $1 ⇒ $2) ⇒ Array<$2>
```

Substitution & Inference

~100 lines each

```
function f(x, y) { return x < x + y; }
function f(x, y) { return x < x + y; }
```

Should we give it this type?

```
function f(x : Int, y : Int) : Bool
```

Or this type?

```
function f(x : Float, y : Float) : Bool
function f(x, y) { return x < x + y; }
```

Type classes

```
function f<T>(x : T, y : T) : Bool where Order<T>, Number<T>
function fullName(person) { return r.firstName + " " + r.lastName; }
function fullName(person) { return r.firstName + " " + r.lastName; }
```

Field constraints

```
function fullName<T>(person : T) : String where
   T.firstName : String, T.lastName : String
```

More?

Typing Haskell in Haskell

Mark P Jones

https://web.cecs.pdx.edu/~mpj/thih/

```
// A, B, C,  ...  A1, B1, C1,  ... A2, B2, C2,  ..

val genericParameterNames = Iterator.iterate(('A', 0)) {
  case ('Z', i) ⇒ ('A', i + 1)
  case (x, i) ⇒ ((x + 1).toChar, i)
}.map { case (x, i) ⇒
  if(i == 0) x.toString else x.toString + i.toString
}
```