Do fish have legs?

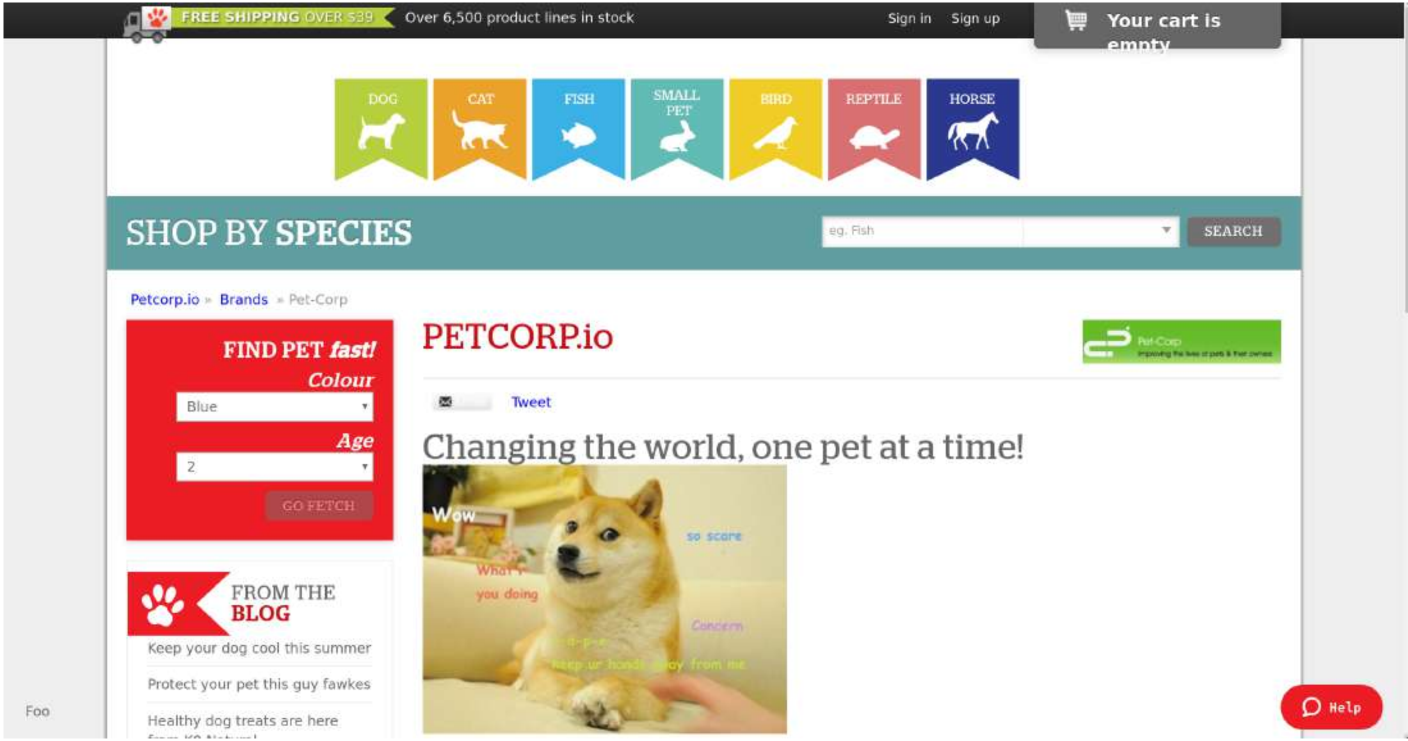
Yilin Wei
@Yilin47276121
https://gitlab.com/yilinwei/domain-for-all

CodeMesh

Bodil Stokke, Esq.
@bodil

On the bright side, we had a last minute speaker cancellation and SPJ immediately offered to do an extra talk out of a selection of four he has prepared, and I'm now secretly hoping three more speakers will cancel at the last minute.

12:20 PM · Nov 6, 2019 · Twitter for Android


FREE SHIPPING OVER $39
Over 6,500 product lines in stock
Sign in
Sign up
Your cart is empty
DOG
CAT
FISH
SMALL PET
BIRD
REPTILE
HORSE
SHOP BY SPECIES
eg. Fish
SEARCH
Petcorp.io » Brands » Pet-Corp
FIND PET fast!
Colour
Blue
Age
2
GO FETCH
PETCORP.io
Tweet
Changing the world, one pet at a time!
Wow
so scare
What r you doing
Concern
FROM THE BLOG
Keep your dog cool this summer
Protect your pet this guy fawkes
Healthy dog treats are here
Foo
Help

Problem

```
POST /pets/search?name=fido HTTP/1.1
```

Problem

```
POST /pets/search?name=fido HTTP/1.1
```

```
POST /pets/search?name=fido_AND_age=3 HTTP/1.1
```

Problem

```
POST /pets/search?name=fido HTTP/1.1
```

```
POST /pets/search?name=fido_AND_age=3 HTTP/1.1
```

```
POST /pets/search?name=fido_AND_age<3_AND_age>5 HTTP/1.1
```

Problem

```
POST /pets/search?name=fido HTTP/1.1
```

```
POST /pets/search?name=fido_AND_age=3 HTTP/1.1
```

```
POST /pets/search?name=fido_AND_age<3_AND_age>5 HTTP/1.1
```

```
POST /pets/search?name=fido_AND_age<3_AND_age>5_OR_name=whiskers HTTP/1.1
```

Problem

```
POST /pets/search?name=fido HTTP/1.1
```

```
POST /pets/search?name=fido_AND_age=3 HTTP/1.1
```

```
POST /pets/search?name=fido_AND_age<3_AND_age>5 HTTP/1.1
```

```
POST /pets/search?name=fido_AND_age<3_AND_age>5_OR_name=whiskers HTTP/1.1
```

And so on...

All roads lead to ~~Rome~~ Lisp

> Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.

10th rule, Philip Greenspun

All roads lead to ~~Rome~~ Lisp

> Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.
>
> ***10th rule, Philip Greenspun***

> Any sufficiently complicated ~~C or Fortran~~ **$COMMERCIAL_LANG** program contains an ~~ad-hoc~~, ~~informally-specified~~ **formally specified**, ~~bug-ridden~~ **bug-free**, ~~slow~~ implementation of half of Common Lisp.
>
> ***10th rule (revisited)***

Scala

```scala
sealed trait ADT

object ADT {
  case class Foo(bar: Int) extends ADT
  case class Baz(qux: Int) extends ADT
}
```

Products

```scala
foo match {
  case Foo(bar) => ???
  case Baz(_) => ???
}
```

Pattern matching

Lisp

```
(println "This is the total: %i" (+ a b))
```

```
(and true (or false true))
```

```
(f x y)
```

```
(eq (name pet) "fido")
```

Simple syntax

AST

```
sealed trait Expr

object Expr {
    sealed trait Lit extends Expr
    object Lit {
      //Literals
      case class Num(value: Double) extends Lit
      case class Str(value: String) extends Lit
    }
    //Symbol
    case class Sym(value: String) extends Expr
    //Function application
    case class Apply(sym: Sym, args: List[Expr]) extends Expr
}
```

```
  Apply(
    Sym("eq"),
    List(
      Apply(
        Sym("name"),
        List(Sym("pet"))
      ),
      Lit.Str("fido")
      )
  )
```

Evaluation

```scala
object Eval {
    def apply[A](expr: Expr): A = {
      expr match {
        case Lit.Num(value) => value.asInstanceOf[A]
        case Lit.Str(value) => value.asInstanceOf[A]
        case Sym(_) => ???
        case Apply(_) => ???
      }
    }
}
```

Context

```scala
final class Context(value: Map[Expr.Sym, Any]) {

  def lookup[A](sym: Expr.Sym): Option[A] =
    value.get(sym).map(_.asInstanceOf[A])

}
```

```scala
new Context(Map(
  Expr.plus -> ((x: Double, y: Double) => x + y)
 )
```

Context

```
final class Context(value: Map[Expr.Sym, Any]) {

  def lookup[A](sym: Expr.Sym): Option[A] =
    value.get(sym).map(_.asInstanceOf[A])

}
```

```
new Context(Map(
  Expr.plus -> ((x: Double, y: Double) => x + y)
 )
```

Symbol case

```
  case sym @ Expr.Sym(_) => context
      .lookup[A](sym)
      .map(Right(_))
      .getOrElse(Left(sym))
```

Apply case

```
case Apply(sym, args) => {
  val len = args.length
  len match {
    case 1 =>
      for {
        f <- apply[Any => Any](sym, context)
        a <- apply[Any](args(0), context)
      } yield f(a).asInstanceOf[A]
    case 2 =>
      for {
        f <- apply[(Any, Any) => Any](sym, context)
        a <- apply[Any](args(0), context)
        b <- apply[Any](args(1), context)
      } yield f(a, b).asInstanceOf[A]
    }
}
```

Demo

Type systems

- Set of rules which allow correct programs
- As expressive as needed

Type systems

- Set of rules which allow correct programs
- As expressive as needed

Let's do it through trial and error

```
object Typer {

  type Result = Either[(String, Expr), Type]

  def apply(expr: Expr): Result = ???
}
```

Do fish have legs?

```
(lt 3 (legs pet))
```

```
Apply(
  Sym("lt"),
  List(
    Lit.Num(3),
    Apply(
      Sym("legs"),
      List(Expr.Sym("pet"))
    )
  )
)
```

Types

```
object Type {
  case object Num extends Type
  case object Str extends Type
  case object Bool extends Type
  case class Func(args: List[Type], ret: Type) extends Type
}
```

Simplest type system

```
def apply(expr: Expr, context: Context): Result = {
    expr match {
      case Lit.Num(_) => Result.success(Type.Num)
      case Lit.Str(_) => Result.success(Type.Str)
      case sym @ Expr.Sym(symbol) => ???
      case Apply(_) => ???
    }
}
```

Simplest type system

```
def apply(expr: Expr, context: Context): Result = {
    expr match {
      case Lit.Num(_) => Result.success(Type.Num)
      case Lit.Str(_) => Result.success(Type.Str)
      case sym @ Expr.Sym(symbol) => ???
      case Apply(_) => ???
    }
}
```

This looks familiar

Symbol case

```
case sym @ Expr.Sym(symbol) =>
  context
    .lookup(sym)
    .map(Result.success _)
    .getOrElse(Result.fail(s"could not find $symbol in context", expr))
```

Type system

Rule 1

A function needs to be applied to arguments of the correct type and length

```
(f x y)
```

```
Apply(
  Sym("f"),
  List(
    x,
    y
  )
)
```

- `f: (A, B) => C`
- `x: A`
- `y: B`

Restrictive type systems

- What type is `pet`?

Restrictive type systems

- What type is `pet`?
- We want to restrict bad programs
- But our type system doesn't allow correct programs

Restrictive type systems

- What type is `pet`?
- We want to restrict bad programs
- But our type system doesn't allow correct programs

Solution

- Make a more expressive type system

```
case class Coprod(types: Set[Type]) extends Type
case class Record(name: String) extends Type
```

- `pet: Dog | Cat | Fish`

Coproduct rules

A type A is considered a **subtype** of another type B, **A < B**

Case 1

If $A = B$

Case 2

- If $B = B_1 \mid B_2 \mid \ldots$
- such that $A = B_n$

Case 3

- If $A = A_1 \mid A_2 \mid \ldots$
- and $B = B_1 \mid B_2 \mid \ldots$
- such that all $A_i = B_j$

Rule 1 (revised)

A function needs to be applied to arguments of the correct type, or a **subtype** of the type

```
(f x y)
```

```
Apply(
  Sym("f"),
  List(
    x,
    y
  )
)
```

- `f: (A, B) => C`
- `x: A` or `x: D` where `D < A`
- `y: B` or `y: E` where `E < B`

More complications

What should the following program return?

```
Apply(
  Sym("if"),
  x,
  List(
    Lit.Str("moo"),
    Lit.Num(3)
  )
)
```

Unions

Case 1

- If A and B are not coproducts
- the union of A, B is $A \mid B$ if A != B
- otherwise, A if $A = B$

Case 2

- If A and B are coproducts
- the union of A, B is $A_1 \mid \ldots \mid B_1 \mid \ldots$

Case 3

- If one of them is a coproduct, C, and the other is not, D
- then the union of A, B is $C \mid D_1 \mid \ldots$

Rule 2

In an if statement, the return of the if statement is the union of the two types.

```
Apply(
  Sym("if"),
  x,
  List(
    Lit.Str("moo"),
    Lit.Num(3)
  )
)
```

- `(if cond a b)`
- `a: A`
- `b: B`
- `(if cond a b): A | B`

Further complications

```
Apply(
  Sym("lt"),
  List(
    Lit.Num(3),
    Apply(
      Sym("legs"),
      List(Expr.Sym("pet"))
    )
  )
)
```

- `legs: (Dog | Cat | Fish) => Int` or `legs: (Dog | Cat) => Int`
- `pet: Dog | Cat | Fish`

Further complications

```
Apply(
  Sym("lt"),
  List(
    Lit.Num(3),
    Apply(
      Sym("legs"),
      List(Expr.Sym("pet"))
    )
  )
)
```

- `legs: (Dog | Cat | Fish) => Int` or `legs: (Dog | Cat) => Int`
- `pet: Dog | Cat | Fish`

Clearly not going to work

Further complications

```
Apply(
  Sym("lt"),
  List(
    Lit.Num(3),
    Apply(
      Sym("legs"),
      List(Expr.Sym("pet"))
    )
  )
)
```

- `legs: (Dog | Cat | Fish) => Int` or `legs: (Dog | Cat) => Int`
- `pet: Dog | Cat | Fish`

Clearly not going to work

Solution

- Make a more powerful expressive system

Is statement

```
Apply(
  Sym("is"),
  List(
    Sym("pet"),
    Type("Fish")
  )
)
```

Is statement

```
Apply(
  Sym("is"),
  List(
    Sym("pet"),
    Type("Fish")
  )
)
```

- `(is pet 'Fish): Bool`
- If the *runtime* value = `true`, `pet: Fish`
- If the *runtime* value = `false`, `pet: !Fish`

What about compile time?

Consider if statements

```
Apply(
  Sym("if"),
  List(
    Apply(
      Sym("is"),
      List(
        Sym("pet"),
        Type("Fish")
      )
    ),
    Lit.Str("pet should be fish"),
    Lit.Str("pet should be anything else")
  )
)
```

Consider if statements

```
Apply(
  Sym("if"),
  List(
    Apply(
      Sym("is"),
      List(
        Sym("pet"),
        Type("Fish")
      )
    ),
    Lit.Str("pet should be fish"),
    Lit.Str("pet should be anything else")
  )
)
```

- What about `not`, `and`, `or`?

Bounds

- What should go into `a`?

```
Apply(
  Sym("is"),
    List(
      a,
      b
    )
)
```

B type

- What about `b`?

Bounds

- What should go into `a`?

```
Apply(
  Sym("is"),
    List(
      a,
      b
    )
)
```

B type

- What about `b`?

Further constraints

- Clearly `b: Type(B)`, where `B` is a valid type
- One further constraint; `a: A`, then `A < B`

Bounds (continued)

- We then need to change the inferred type.
- Context now takes in an `Expr`

```
final class Context(value: Map[Expr, Type]) {
  val lookup = value.get _
}
```

Bounds (continued)

- We then need to change the inferred type.
- Context now takes in an `Expr`

```
final class Context(value: Map[Expr, Type]) {
  val lookup = value.get _
}
```

```
final class Bound(val values: Map[Expr, Type]) {
  def complement(typer: Expr => Result): Bound = ???
}
```

```
type Result = Either[(String, Expr), (Type, Bound)]
```

Adding to the context

```
context + bound
```

- for all `expr` in the `context` replace

Adding to the context

```
context + bound
```

- for all `expr` in the `context` replace

Complement?

- `x : Foo | Bar | Baz`
- `(is x Foo) = Bound(x: Foo)`
- `(not (is x Foo)) = ???`

Adding to the context

```
context + bound
```

- for all `expr` in the `context` replace

Complement?

- `x : Foo | Bar | Baz`
- `(is x Foo) = Bound(x: Foo)`
- `(not (is x Foo)) = ???`

Elaborated

- `x : Foo | Bar | Baz`
- `(not (Bound(x: Foo)) = Bound(x: Bar | Baz)`
- In the case that `!` uninhabited; program incorrect

Operations (revisited)

If statements

- `(if a b c)`
- `a: Boolean`, `B` where `B` is a bound
- `b`, apply `B`
- `c`, apply `!B`

Operations (revisited)

If statements

- `(if a b c)`
- `a: Boolean, B` where `B` is a bound
- `b`, apply `B`
- `c`, apply `!B`

Bound changes for all boolean operations

Finally we can typecheck

```
(if (is pet Fish) 1.0 (legs pet))
```

```
Apply(
  Sym("if"),
  List(
    Apply(
      Sym("is"),
      List(
        Sym("pet"),
        Type("Fish")
      )
    ),
    Lit.Num(1.0),
    Apply(
      Sym("legs"),
      List(
        Sym("pet")
      )
    )
  )
)
```

Demo

Recap

- Created an AST
- Created an evaluator which could run a program
- Created a simple typer
- Made it more expressive

Go forth and scheme!

Evaluation

- Really easy to do simple evaluators
- You can play around with evaluation/compilation strategies

Type systems

- Are as simple as you want
- Are as expressive as you want

Q & A