Odd

Alex Kalicki, Alexandra Medway, Daniel Echikson, Lilly Wang
The Team

Alexandra - The Manager
Alex - The System Architect
Danny - The Language Guru
Lilly - The Tester
The Process

Branch, add a feature
Make a pull request
Travis
Merge with master
Repeat
The Process

Source Code → Scanner → Parser → AST → Analyzer

Generator → PAST → Pythonizer → SAST

Target Code
Language Description

“A simple mathematical distribution language.”

Odds is a functional programming language that centers around mathematical distributions, and expresses operations on them in a direct and uncomplicated way.
Challenges

Functional → Imperative

Static Scoping

Immutable data

Anonymous Functions
Challenges

Type Inference \rightarrow Type Ignorant

Python is type ignorant - catches errors at runtime

Odds catches type errors at compile time

Hindley Milner type inference, no runtime errors in generated python

Distribution type

When to make conversion to python function calls?
**Static Scoping**

Python doesn’t have static scoping, it has dynamic scoping

We mimic “static scoping” with a table →

  each id corresponds to a “statically scoped” id

Keep integer value at top, every time we create a new id, replace name in python with id_integer

This also makes variables immutable, any assignment leads to a new statically scoped id
**Static Scoping**

```plaintext
do a = 5

do adda = (x) -> return x + a

do a = 10

do print(adda(0))

do adda = (x) -> return x + a

do print(adda(0))
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a_0</td>
</tr>
<tr>
<td>adda</td>
<td>adda_1</td>
</tr>
<tr>
<td>x</td>
<td>x_2</td>
</tr>
</tbody>
</table>
Static Scoping

do a = 5

do adda = (x) -> return x + a

do a = 10

<table>
<thead>
<tr>
<th>a</th>
<th>a_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>adda</td>
<td>adda_1</td>
</tr>
<tr>
<td>x</td>
<td>x_2</td>
</tr>
</tbody>
</table>

do print(adda(0))

do adda = (x) -> return x + a

do print(adda(0))
Static Scoping

do a = 5

do adda = (x) -> return x + a

do a = 10

do print(adda(0))

do adda = (x) -> return x + a

do print(adda(0))
Static Scoping

do a = 5

do adda = (x) -> return x + a

\textbf{do a = 10}

\textbf{do print(adda(0))}

\textbf{do adda = (x) -> return x + a}

\textbf{do print(adda(0))}

<table>
<thead>
<tr>
<th>a</th>
<th>a_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>adda</td>
<td>adda_4</td>
</tr>
<tr>
<td>x</td>
<td>x_5</td>
</tr>
</tbody>
</table>
Static Scoping

do a = 5

do adda = (x) -> return x + a

do a = 10

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a_3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>adda</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>adda_4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x_5</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a_3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>adda</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>adda_4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x_5</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a_3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>adda</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>adda_4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x_5</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Static Scoping

do a = 5
do adda = (x) → return x + a
do a = 10
do print(adda(0))
do adda = (x) → return x + a
do print(adda(0))

\[
a_0 = 5
a_0
\]
\[
def adda_1(x_2):
    return (x_2 + a_0)
adda_1
a_3 = 10
a_3
print(adda_1(0))
def adda_4(x_5):
    return (x_5 + a_3)
adda_4
print(adda_4(0))\]
Anonymous Functions

do call = (f, x) -> return f(x)

do y = call((x) -> return x + 2, 4)

do print(y)

... how does this work?
Anonymous Functions

Python doesn’t have anonymous functions

As we move from sast $\rightarrow$ past

- pull up anonymous functions one statement
- replace occurrence of anonymous function with function name
Anonymous Functions

Odds

do call = (f, x) -> return f(x)
do y = call((x) -> return x + 2, 4)
do print(y)

Python

def call_0(f_1, x_2):
    return f_1(x_2)
call_0
def _anon_3(x_4):
    return (x_4 + 2)
y_5 = call_0(_anon_3, 4)
y_5
print(y_5)
Anonymous Functions

Odds

do call = (f, x) -> return f(x)
do y = call((x) -> return x + 2, 4)
do print(y)

Prints “6”

Python

def call_0(f_1, x_2):
    return f_1(x_2)
call_0
def _anon_3(x_4):
    return (x_4 + 2)
y_5 = call_0(_anon_3, 4)
y_5
print(y_5)
Anonymous Functions

What else can we do?

“caking” → calling the function immediately after it is declared

Odds

```haskell
do magic = ((()) -> return 42)()
do magic = ((()) -> return 40)() + 2
```

Python

```python
def _anon_0():
    return 42
magic_1 = _anon_0()
magic_1

def _anon_2():
    return 40
magic_3 = (_anon_2() + 2)
magic_3
```
Everything Is An Expression

In python, most things are statements.

Not in Odds, because we are a functional language!
Everything Is An Expression

So, we needed to replace all instances of “python non-expressions” in odds with their expression value (an id)

Similar to anonymous functions...

Whenever we have an expression in odds which is not an expression in python (assignment, conditionals)

Assign expression value to temporary id, replace expression instance with id
Everything Is An Expression

Conditionals need to be encapsulated in a “conditional” function which returns the value of the conditional evaluation.
Everything Is An Expression

**Odds**

```python
do x = y = z = if true then 4 else 2
```

**Python**

```python
def _cond_0():
    if True:
        return 4
    return 2
z_1 = _cond_0()
y_2 = z_1
x_3 = y_2
x_3
```
**Type Inference**

Python is not type checked; it is ‘type ignorant’. Odds is type checked.

Odds has no type annotations. Problem: how to get type information with which to check?

Solution: Hindley-Milner style type inference →

- variables start out unconstrained
- constrain where and when possible to a type
- If the variable has been constrained and there is a type mismatch, throw a compile-time error at that user!
Type Inference

Simple Case

‘n’ must be a Num

‘n + 2’ is OK because ‘n’ is a number. ‘success’ must also be a number.

Program passes Semantic Checking!

Odds

```plaintext
do n = 2
do success = n + 2
```

Sast Printer Output

```plaintext
do Num n_0 = 2
do Num success_1 = Num n_0 + 2
```
Type Inference

Slightly harder case...

‘x’ and ‘y’ are unconstrained because they are parameters

Is ‘x && y’ valid? We don’t know what types ‘x’ and ‘y’ are...

What do we do?
Type Inference

Solution!

‘x’ and y are unconstrained, so on ‘x && y’ make ‘x’ a Bool and make ‘y’ a Bool. ‘result’ must also be a Bool.

```
do and = (x, y) ->
do result = x && y
   do success = x || y
return result
```
Type Inference

Solution!

‘x’ and y are unconstrained, so on ‘x && y’ make ‘x’ a Bool and make ‘y’ a Bool. ‘result’ must also be a Bool.

‘x || y’ is OK because ‘x’ and ‘y’ are Bools
Type Inference

Solution!

‘x’ and y are unconstrained, so on ‘x && y’ make ‘x’ a Bool and make ‘y’ a Bool. ‘result’ must also be a Bool.

‘x || y’ is OK because ‘x’ and ‘y’ are Bools

‘and’ must be a function that takes 2 Bools and returns a Bool.
Type Inference

Now all we have to do is generalize the process we just outlined:

1. If assigning a literal to a var - do $x = 2$ - give the var the type of the literal.
Type Inference

Now all we have to do is generalize the process we just outlined:

1. If assigning a literal to a var - do x = 2 - give the var the type of the literal.
2. If a var is included in some sort of operation - x && y - ensure that the var is the appropriate type, in this case Bool. If a var is not the appropriate type - If x or y is not a Bool - spit out an error.
Type Inference

Now all we have to do is generalize the process we just outlined:

1. If assigning a literal to a var - do x = 2 - give the var the type of the literal.
2. If a var is included in some sort of operation - x && y - ensure that the var is the appropriate type, in this case Bool. If a var is not the appropriate type - If x or y is not a Bool - spit out an error.
3. If the type of a var is not known - i.e. because the var is a parameter - place constraints on its type where possible. For example:
   
   /* var x has unknown type. The function add_two adds 2 to the argument it is fed and returns */
   do a _num = add_two(x)
   /* We know x must now be a Num */
Type Inference

Generalization was a challenge; there are many corner cases…
What about constraining recursive functions?

`inf_recursion` expected to return `Num`

`inf_recursion` returns a `Bool`

```plaintext
do inf_recursion = () ->
do inf_recursion() + 2
return true
```

Semantic error:
Invalid return type in function 'inf_recursion':
_type 'Num' expected to be returned, but type 'Bool' returned instead.
Distributions

“A distribution is a measurable set of data to which a function of a discrete variable is applied. This function will map the set of data to a new set of outcomes.”
Distributions

Two Type: Continuous and Discrete

Continuous:

Declare minimum, maximum, and the weight to apply to the range of values.

```do dist = <0, 10> | (x) -> return 1 | /* Uniform Distribution */
```

Discrete:

Have two lists, variables and the respective weights of the variables.

```do vals = [1, 2, 3]
do weights = [0.1, 0.5, 0.4]
do discrete = [<vals, weights>]
```
Distributions

Operations:

Addition, multiplication, exponentiation between distributions -- use cross product

Operations with constants -- apply value and operation to each element of distribution

\[\times, +, \times\times\]
Distributions

The probability density function of $X$, the lifetime of a certain type of electronic device (measured in hours), is given by

$$f(x) = \begin{cases} 
\frac{10}{x^2} & x > 10 \\
0 & x \leq 10
\end{cases}$$

Find $P\{X > 20\}$. 
do d = <0, 400> | return if x <= 10 then 0 else 10 / x ** 2 |
do print(1 - P(20, d))
do print(d)
Set min and max of distribution, mimic infinity with large number
The probability density function

```
do d = <0, 400> | return if x <= 10 then 0 else 10 / x ** 2 |
do print(1 - P(20, d))
do print(d)
```

P(20, d) will calculate the probability that X (in d) < 20, works the same way as normal distribution table

Subtract from 1 to get P(X>20)
do print(d)
Lottery Question

You can buy one ticket to one of four different lotteries:

Lottery One:
90% chance of winning $2, 8% → $50, 2% → $5,000, 1% → $10,000

Lottery Two:
Distributed with 1/x along 5->100

Lottery Three:
Distributed with 1/x*x along 10->400

Lottery Four:
99.9% chance of winning $1, 0.1% → $1,000,000
Lottery Question

Which ticket should you buy?

... examine expected value

... sort dists by expected value

merge sort!

buy ticket to lottery with the highest expected value for their distribution
Lottery Question

You have 10 dollars. You can buy a ticket to four different lotteries:

Lottery One: 90% chance of winning $2, 8% of $50, 2% of $5,000, 1% of $10,000

Lottery Two: Distributed with $1/x$ along [5, 100]

Lottery Three: Distributed with $1/x^2$ along [10, 400]

Lottery Four: 99.9% chance of winning $1, 0.1% of winning $1,000,000
Lottery Question

do lot_winnings_one = [2, 50, 5000, 100000]  
do lot_probs_one = [.90, .08, .02, .01]     
do lot_one = |<lot_winnings_one, lot_probs_one>|  


do lot_two = <5, 100> | (x) -> return 1 / x |  


do lot_three = <10, 400> | (x) -> return 1 / x * x |  


do lot_winnings_four = [1, 10000000]  
do lot_probs_four = [.999, .001]   
do lot_four = |<lot_winnings_four, lot_probs_four>|
do merge_sort = (l) ->
  do merge = (lists) ->
    /* get two lists */
    do l1 = head(lists)
    do l2 = head(tail(lists))
    /* if either list is empty, return the list */
    return if len(l1) == 0 then l2 else if len(l2) == 0 then l1
    /* otherwise merge */
    else () ->
      do h1 = head(l1) do t1 = tail(l1)
      do h2 = head(l2) do t2 = tail(l2)
      return if E(h1) <= E(h2) then h1 :: merge([t1, h2 :: t2])
        else h2 :: (merge([h1 :: t1, t2]))
    ()
  ()
  do halve = (l) -> return
    if len(l) <= 1 then [l, []]
    else () ->
      do h = head(l) do t = tail(l)
      do halves = halve(t)
      do t1 = head(halves) do t2 = head(tail(halves))
      return [h :: t2, t1]
  ()
  return
    if len(l) <= 1 then l
    else () ->
      do halves = halve(l)
      do l1 = head(halves) do l2 = head(tail(halves))
      return merge([merge_sort(l1), merge_sort(l2)])
  ()
**Lottery Question**

So which one should you buy? Let’s run the program!