Coral

Jacob Austin
Matthew Bowers
Rebecca Cawkwell
Sanford Miller

* please note that this presentation theme is also called Coral
The Coral Team*

Jacob Austin
Semant Architect
Snakes are nice

Matthew Bowers
Codegen Architect
I lik snek

Rebecca Cawkwell
Manager & Tester
Passionately hates snakes

Sanford Miller
Language Guru
Loves Coral Snakes

*with guidance by Lauren Arnett
Our Inspiration

- Coral to Python as TypeScript to Javascript
- **Type Safety**: optional static typing enforced at compile and runtime.
- **Optimization**: use type-inference to generate code as fast as C.

Source: Pintrest
What is **CORAL**

- Dynamically typed programming language
- Cross compatible with Python
- Optional static typing enforced by the compiler and runtime environment
- Type inference and optimization based on static typing
- Types: int, char, float, boolean, strings, lists
- First class functions
- No classes (no time)
- Compile and runtime exceptions
Implementation
Architectural Design

source.cl → Scanner → Parser → Semant → Code Generation → coral.native

- LLC
- executable
Coral v Python

- Coral is a smaller version of Python with extended support for typing.
- Coral uses the **same syntax** as Python, allowing for cross compatibility.
- The difference between Coral and Python is our **optimization and safety**.
Comparison to Python

Wall-time on simple programs allows comparison between Coral and Python. For a program like this:

```python
x = 100000000
count = 0

while x > 0:
    count += 1
    x -= 1

print(count)
```

performance is about 40 times faster (.4 seconds to 23.4 seconds wall time).
Key Features
Syntax & Grammar

- Coral strictly follows the current Python 3.7 syntax, and any valid Coral program can also be run and compiled by an up-to-date Python 3.7 interpreter.
- Coral supports for loops, while loops, for loops, if and else statements, first-class functions, all in a strictly Pythonic syntax.
- Some valid programs include:

```python
def gcd(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a

def max(arr):
    max_value = 0
    for val in arr:
        if val > max_value:
            max_value = val
    return max_value

def foo(x):
    return x + 5

def apply(f, value):
    return f(value)

x = 352  # this is a comment
y = 245
z = gcd(x, y)
arr = [1, 2, 3]
out = max(arr)
apply(foo, 5)  # returns 10
```
Coral supports **optional type annotations** as supported by Python 3.7, which can be attached to variable assignments and function declarations.

While these labels are only cosmetic in Python, they are fully enforced in Coral, either at compile time (if possible) or at runtime. A program will generally not compile (or in rare cases will terminate at runtime) if these type annotations are violated.

```python
def gcd(a : int, b : int) -> int:
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a

x : int = 352  # this is a comment
y : int = 245
z : int = gcd(x, y)
```

```python
def apply(foo : func, b):
    return foo(b)

def bar(x):
    return x

print(apply(bar, 3))
```
Type Inference

- Coral supports gradual/partial type-inference built on top of the optional typing system. This is a sort of **bottom-up type inference** based on identifying literals and propagating these types up through the tree.
- Even programs with no annotations can be **fully type-inferred**. The type inference system does its best to infer whatever is possible.

Welcome to the Coral programming language!

```python
>>> def foo(x, y):
...     z = x * y + 4 * 50 - x
...     while z < 50:
...         z += 1
...     return z
...
>>> z = foo(3, 4)
>>> print(z)
>>>
>>> def sum(a, b):
...     return a + b
...
>>> def one():
...     return 1
...
>>> def do_wild_things(f, a, b):
...     return (f(a, b) + f(a, b)) * f(a, b)
...
>>> z = do_wild_things(sum, 2 * one(), 4)
>>> print(z)
72
>>> type(z)
int
```
Compile Time Exceptions

- Uses type inference to determine types of functions and variables at compile time which allows both optimization and the enforcement of type annotations. Coral cannot be fully type inferred while retaining all the type flexibility of Python, but many common errors can be captured by the Coral compiler.

- At compile time, Coral checks for:
  - Invalid assignments (to explicitly typed variables): global and local, formal args, function returns
  - Invalid argument and return types (for functions and operators)

- For example:

```python
>>> def foo() -> int:
...    return "hello"
...
STypeException: invalid return type
```

```python
>>> def add(x: int[]):
...    sum = 0
...    for i in x:
...        sum += i
...    return sum
...
```

```python
>>> print(add([1, 2, 3]))
6
>>> print(add([1.0, 2.0, 3.0]))
STypeException: invalid type assigned to x
```
Runtime Exceptions

- Only has runtime checks when type isn't inferrable. Prevents violations of type annotations.
- Coral checks for:
  - Invalid assignments (to explicitly typed variables): global and local, formal args, function returns
  - Invalid argument types (for operators)
  - Initialization: can't use null objects
  - List bounds

```python
def dynamic():
    if x == 3:
        return 3
    else:
        return "hello"

x = 3
print(dynamic() * dynamic())

x = 4
print(dynamic() * dynamic())
```

```bash
$ Jacobs-MacBook-Pro-2:Coral JAustin$ .coral.native -r llvm-test.cl
9
RuntimeError: unsupported operand type(s) for binary *
```
Optimization

- Optimization is done in cases where there are **immutable Objects** and all of the Objects have **known types** through the type inference system.
- In programs which can be optimized, the code generation is similar to **MicroC** and therefore programs can run “as fast as C”. This optimization is integrated into the compilation, and can be performed only where possible, while seamlessly transitioning back to a dynamic Python-style runtime model.

Statistics for optimized code:

- For fully optimized code, LLVM loc count drops by at least 1000 lines, **reducing binary sizes by tens of kilobytes**.
- Runtime **performance increases by as much as 100x** for code like gcd or code involving frequent heap allocations in Python (like counting while loops).
Optimization Examples

```python
def gcd(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a

print(gcd(13, 334232512))

if True:
    x = 23.4
else:
    x = 5
print(x)
```

GCD function with dynamic objects created. Runtime is 10 seconds for Python and .2 seconds for Coral. No explicit type annotations.

```python
def count(x):
    sum = 0
    for i in range(x):
        if i / 20 < 5:
            sum += i
    return sum

print(count(50000))
```

For-loop based function traditionally expensive in Python. Does not terminate in reasonable time in Python. Runs in .75 seconds in Coral

```python
def foo(x: str) -> int:
    count = 0
    for char in x:
        print(char)
        if char == 'c':
            count += 1
    return count

foo("hello")
```

For-loop iteration over chars. Partial type inference for sub-operations even though full code cannot be optimized because of lists.
Testing
Test Suite

- Sample program output compared to *.out file.
- Checks the following file types: stest-*, sfail-*, and test-*, fail-* for semant tests and llvm/runtime tests respectively.
- Done by each member as feature implemented. Generally one new test for each new feature or commit.
- Over 100 tests in the final repository.
DEMO TIME
Thank you & Happy Holidays

Source: Pintrest