Seaflow

Language for Reactive Programing

Rohan Arora, Junyang Jin, Ho Sanlok Lee, Sarah Seidman

OOP vs Reactive Programming

OOP

- Application state is defined by stateful objects
- Developers defines functions that make state transitions

Reactive Programming

- Application is defined by streams of data and how each stream react to one another
- Each relation is independent and causes no side effect
- Application logics are highly decoupled

Seaflow Design

Reactive Programming language that features

- Simple, C-like syntax
- Immutability
- Data stream, “Observable”
- First-class functions
- Basic arrays and structs

```
int a = 0;

int $b = 0;
```

Dew Point Calculator

```
float $relative_humidity = 10.0;
float $temperature       = 20.0;
char  $temp_type         = ‘c’;

float fahrenheit_converter (float temp, char type) {
    float adjusted = if (type == ‘c’) temp else (temp - 32) / 1.8;
    return adjusted;
}

float $t = combine(fahrenheit_converter, $temperature, $temp_type);

subscribe(print, $t - (100 - $relative_humidity) / 5);
```

Features

Types, Operators, and Syntax

	int	float	char
Operators	+, -, *, /, ==, !=, >, >=, <, <=, &&, ||	+, -, *, /, ==, !=, >, >=, <, <=	+, -, *, /, ==, !=, >, >=, <, <=
Syntax	int i = 42;	float f = 12.7;	char c = ‘s’;
Built-in functions	printi(i);	printf(f);	printc(c);

If Statements

```
int a = 5;

int b = if (a < 10) 1 else 0;
```

If Statements

```
int a = 5;

int b = if (a < 10) 1 else 0;

char c = if (a > 10) foo() else bar();

printc(c);
```

```
char foo() {
      printi(0);
      return ‘a’;
}

char bar() {
      printi(42);
      return ‘b’;
}
```

```
42
b
```

Implicit Conversion

```
int a = 9;

float b = 26.2;

float c = a + b;

float d = b * a;

int d = if(a > b) 1 else 0;
```

Arrays

```
int[] a = [1,2,3,4];
int b = a.length;
int[] copy = a;

float f = [1.2, 3.4, 5.6][0];
int len = [1.2, 3.4, 5.6].length;

char[] hi = "hello " + "world!";

char foo(char[] c, int a) {
     return c[a];
}
foo(hi, 0);
```

```
prints(hi);
```

Structs

```
struct Num {
     int x;
     int y;
     char[] name;
};


int sum(struct Num p) {
     return p.x + p.y;
}

struct Num n = {30, 12, "seaflow"};

int z = sum(n);
```

```
struct Num a = {1,2, "hello"};
struct Num b = {3,4 "world!"};

struct Num[] nums = [a, b];
```

Structs

```
struct Birthday {
     int day;
     char[] month;
};

struct Deathday {
     int d;
     char[] m;
};

int when(struct Birthday b) {
     return b.day;
}
```

```
struct Deathday z = {25, "april"};
int d = when(z);

int e = when({7, "november"});

struct Birthday b = z;
```

Higher-Order Functions

```
int apply((int)->(int) func, int x) {
    return func(x);
}
```

```
apply((int x)->{return x + 1;}, 0);
```

Functions: a real first-class type!

```
(int)->(int) mult = (int y) -> {
     return y * 100;
};

(int)->(int) div = (int y) -> {
     return y / 100;
};

(int)->(int)[] arr = [mult, div];
int product = (arr[0])(1); /* 100 */
```

```
struct Foo {
     int x;
     (int)->(int) func;
};

struct Foo baz = { 15, div };

int quotient = (baz.func)(100); /* 1 */
```

Observables

```
int $a = 1;

subscribe(printi, $a);

$a = 2;
$a = 3;
```

Observables Operations

```
int $a = 1;

int $b1 = map(increment, $a);

int $b2 = $a + 1;
```

Observables Operations

```
int $a = 5;
float $b = 100.0;

float $c1 = combine(add, $a, $b);

float $c2 = $a + $b;
```

Observables Operations

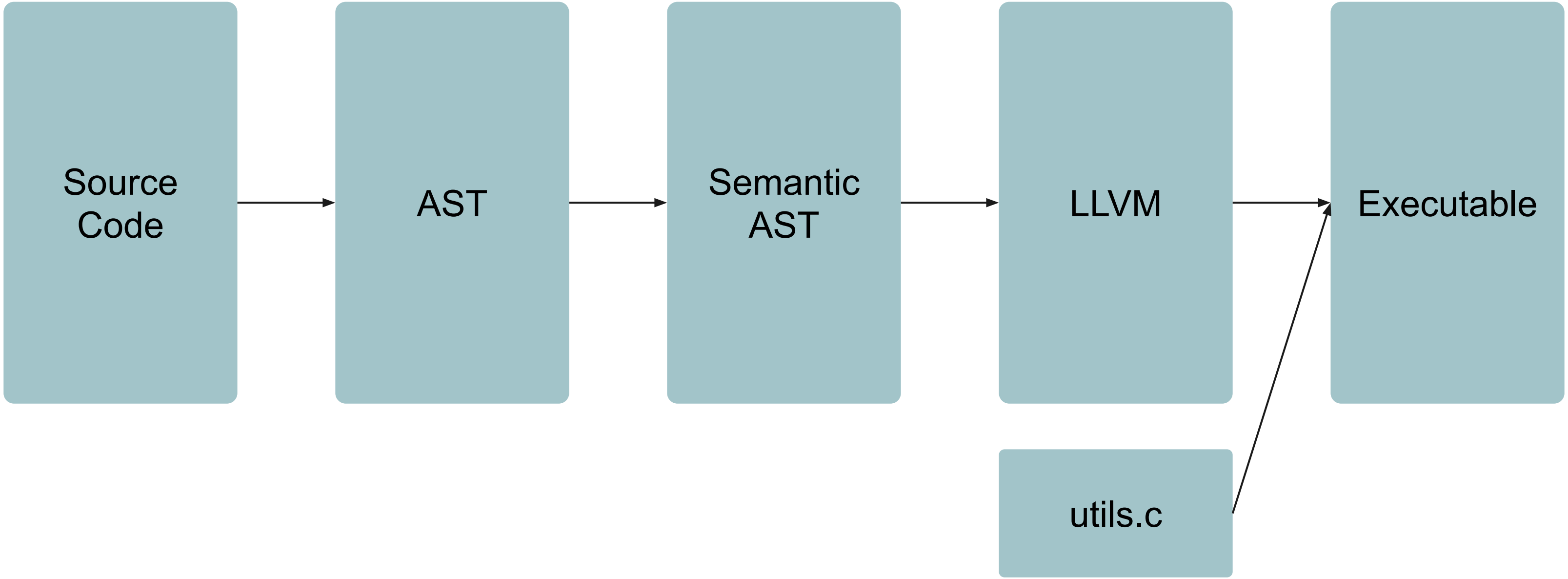
```
int $a = 5;
int $b = $a + 1;

complete($a);
```

Observables Chaining

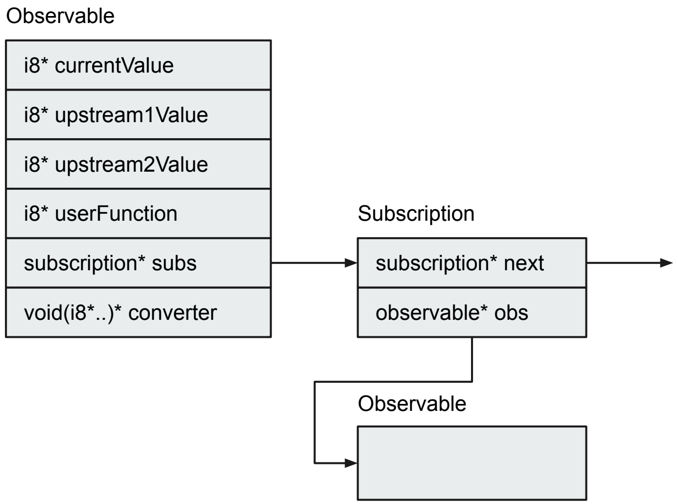
```
int $a = 5;
int $b = 7;
float $c = 0.5;

int $d = $a + 7 * $b;
float $e = $d * $c;
int $f = 10;

subscribe(printf, $e + $f);
```

Backend

Compiler Architecture



Observable



Observable

i8* currentValue

i8* upstream1Value

i8* upstream2Value

i8* userFunction

subscription* subs

void(i8*..)* converter

Subscription

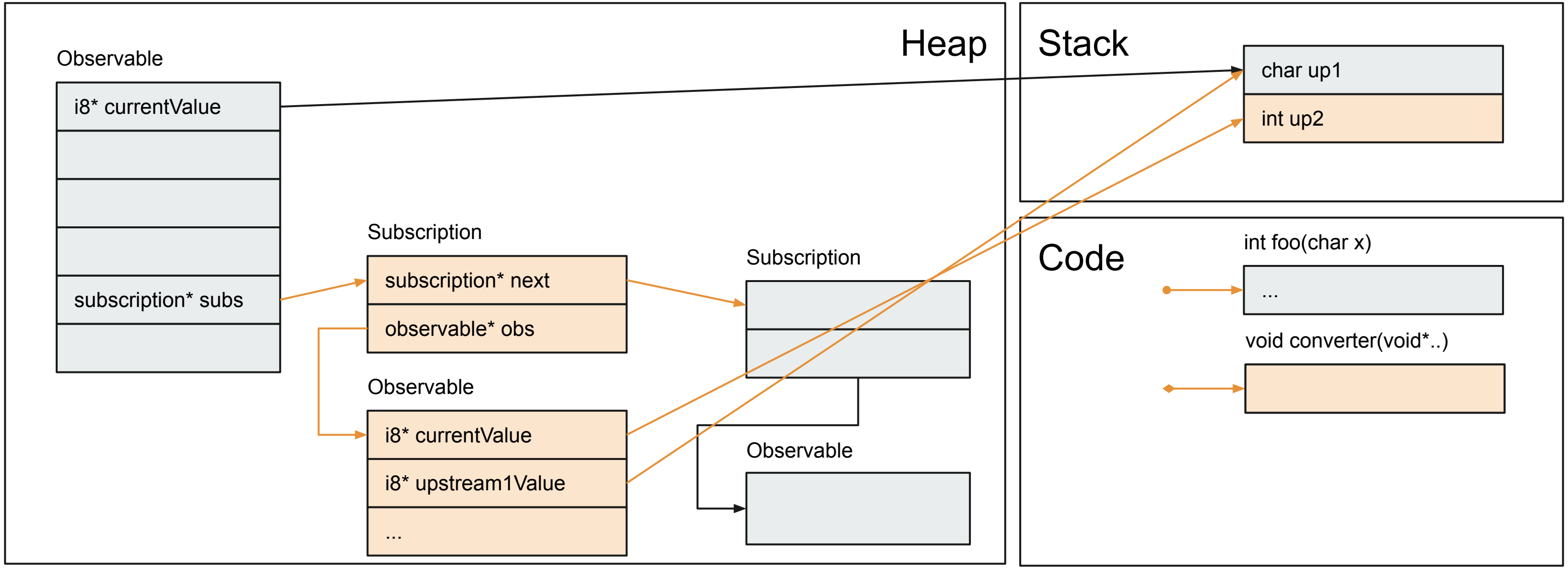
subscription* next

observable* obs

Observable

- Current value in the Stack
- Upstream values in the Stack
- User defined function
- Linked-list of child observables
- Type converter defined at compile time

Subscription

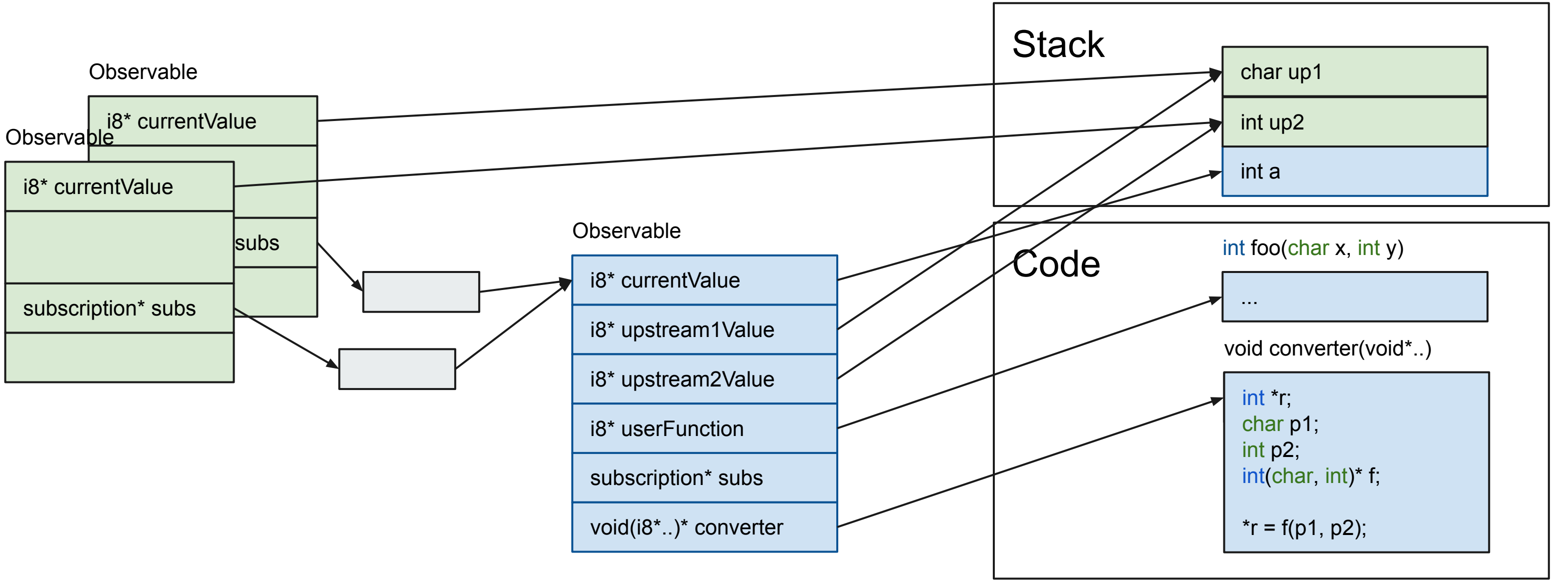


Propagation

```
function onNext (obs):

  for subscription in obs.subscriptions:
    child := subscription.obs

    child.curr = child.func(obs.curr)

    onNext(child)
```

Depth-first traversal on downstreams

Type Conversion



Demo