Claims

randomvald0069

May 2019

1 Frontend features

- Error messages
- val keyword (autoderived type, const keyword. Instead of var)
- Lambdas
- Functions as types
- Class field reference capturing with lambdas
- Classes
- Generics (classes)
- Generic type constraining (classes)
- Class extensions
- Class extensions with generic bound types
- Class constructors
- Character type
- Writes for chars, new lines and any value
- Garbage collection invokation
- Garbage collection debugging info
- Void type for lambda returns
- Expressions without any variable to bind the value to
- Downcasting
- Shorthand lambda invoking (a.b.c(arg);)
- Weeding

2 Backend features

- Code generation for all features
- Garbage collection (stop & copy)
- Garbage collection for lambda closures (this was difficult :()
- Peephole optimization
- Constant folding
- Function sandboxing (evaluate function return value that does not require input from arguments or upper scopes)
- Liveness analysis
- Register allocation by graph coloring
- Anonymous function generation
- Class field capturing
- Support for all the frontend features
- Runtime error checks (except out of memory, since that doesn't exactly make sense with a garbage collector)

We will show some of the more complex features first and lastly show valid code snippets, that we would show off if the possibility arose (like now). Given the time to implement a standard library, we would like to think that a functional, const only declarative programming approach would be very possible (any maybe the best use of the language), with this featureset, while still being fast and having low memory overhead.
When running the compiler in turbo mode, please use -x, this disables runtime checks.

sandboxing

The following code:

```
func f(): int
    var a: int;
    a = 22;
    return a;
end f


write 55;
write f();
```

Produces the code:

```
        mov $55, %rdx

        pushq %rsi
        pushq %rdi
        movq %rdx, %rsi
        movq $intprint, %rdi
        movq $0, %rax
        call printf
        popq %rdi
        popq %rsi

        mov $22, %rdx

        pushq %rsi
        pushq %rdi
        movq %rdx, %rsi
        movq $intprint, %rdi
        movq $0, %rax
        call printf
        popq %rdi
        popq %rsi
```

Instead of explaining everything, we would like let code examples from our language speak for themselves.

kitty/class.kitty

```
class ClassA {
    val h = 5;
};

class ClassC[A : ClassA] {
    val g = 2;
    var t: A;
};
class ClassB[A : ClassC, B] {
    var a: B;
    var b: A;

    val f = (z: A, transformer: (A) -> B):B -> {
        write z.g;
        write z.t.h;
        return transformer(z);
    };
};


var a: class ClassB[class ClassC[class ClassA], class ClassA];
var no: class ClassB[class ClassC[class ClassA], class ClassA];
```

kitty/bigprogram.kitty

```
class Optional[A] {
    constructor (value: A) {
        internalValue = value;
        hasValue = true;
    };

    var internalValue: A;
    val hasValue = false;

    val forEach = (apply: (A) -> void): void -> {
        if (hasValue == true) then {
            apply(internalValue);
        }
    };
};
```

```
class IntBox {
    constructor (toTake: int) {
        constValue = toTake;
    };
    val constValue = 0;
};

func c(): ((IntBox) -> void) -> void
    var internalOpt: class Optional[IntBox];
    var internalIntBox: class IntBox;
    allocate internalIntBox(13);
    allocate internalOpt(internalIntBox);

    return internalOpt.forEach;
end c

func b(): ((IntBox) -> void) -> void
    return c();
end b

func a(): ((IntBox) -> void) -> void
    return b();
end a

val showwer = a();

var opt: class Optional[IntBox];
var boxedInteger: class IntBox;

allocate boxedInteger(42);
allocate opt(boxedInteger);
opt.forEach((value: IntBox): void -> {
    write value.constValue;
});
gc;
opt.forEach((value: IntBox): void -> {
    write value.constValue;
});
opt.internalValue.constValue = 99;
opt.forEach((value: IntBox): void -> {
    write value.constValue;
});

showwer((ib: IntBox): void -> {
    write ib.constValue;
});
```

kitty/chartest.kitty

```
class String {
    constructor (initialSize: int) {
        stringSize = 0;
        allocate internal of length initialSize;
    };
    var stringSize: int;
    var internal: array of char;

    val size = (): int -> {
        return stringSize;
    };

    val getByIndex = (idx: int): char -> {
        return internal[idx];
    };

    val forEach = (f: (char) -> void): void -> {
        var i: int;
        i = 0;

        while i < stringSize do {
            f(internal[i]);
            i = i + 1;
        }
    };

    val print = (): void -> {
        var i: int;
        i = 0;

        while i < stringSize do {
            write internal[i];
            i = i + 1;
        }
    };

    val append = (c : char): void -> {
        internal[stringSize] = c;
        stringSize = stringSize + 1;
    };
};
```

```
var s: class String;
var s2: class String;

allocate s(30);
allocate s2(30);

s.append('h');
s.append('e');
s.append('l');
s.append('l');
s.append('o');

s2.append(' ');
s2.append('w');
s2.append('o');
s2.append('r');
s2.append('l');
s2.append('d');
s2.append('!');

s.print();
write nl;

s2.print();
write nl;

s2.forEach((c: char): void -> {
    s.append(c);
});

s.print();
write nl;
```

kitty/gctest2

```
# This should be moved in phase 2 of gc
class B {
    var mutableField: int;
    val set = (newValue: int): void -> {
        mutableField = newValue;
    };
    val get = (): int -> {
        return mutableField;
    };
    val print = (): void -> {
        write mutableField;
    };
};

# This should be moved in phase 1 of gc since its binded by the stack
class A {
    var refToB: class B;
    val deepSet = (newValue: int): void -> {
        refToB.mutableField = newValue;
    };
    val deepGet = (): int -> {
        return refToB.mutableField;
    };
    val deepPrint = (): void -> {
        write refToB.mutableField;
    };
};

class C {
    var refToA: class A;
};

var a: class A;
var c: class C;
allocate a;
allocate a.refToB;

allocate c;
allocate c.refToA;
allocate c.refToA.refToB;

a.deepSet(434);

gc;

allocate c;
allocate c.refToA;

c.refToA = a;
c.refToA.deepPrint();
```