Higher Kinded Types and Lifetimes

Felix Bowman

May 16, 2017

This Project

- The main focus was the implementation of the Type Checker
- Language Design considerations
- Tests!

Types

Types are for more than just simple errors

- Polymorphism
- Memory Safety

Parametric Polymorphism

Reduces code duplication, parametrizing similar code. Generics in Java are an example:

```
class List<T> {
    public T getAt(int index) {
        // get object of type T
    }

    public boolean add(T e) {
        // add an element of type T to the list
    }
}
```

```
class List {
    public Object getAt(int index) {
        // get any sort of object!
        // who knows what it will be.
    }

    public boolean add(Object e) {
        // add an object to list
    }
}
```

This can make a language much nicer to use, and plays nicely with lots of functional concepts.

Higher Kinded Types

Allow parametrization with type constructors as well as concrete types.

```
interface Mappable<F<A>> {
    F<B> map(Function<A, B> f);
}
```

(Note: not real Java) This is just one example, but lots of constructs in programming languages make good use of this idea.

Memory Management

It’s pretty useful to be able to store and reference areas of memory. Heap memory allocation and deallocation is problematic though.

- Manual memory management by the programmer is a notorious source of bugs
- Garbage collection is slow

Resource Acquisition Is Initialization, or RAII, ties dynamically allocated heap memory to the lexical lifetime of an object. Memory is freed when a variable goes out of scope.

- If RAIIed heap memory cannot be aliased then lots of the problems associated with manual dynamic allocation (dangling references, memory leaks, double free) are solved. So only allow one mutable reference to heap memory at any one point.
- Immutable references can still be created, as long as they do not live longer then the memory that they reference.

This idea can be enforced by a **Borrow Checker** and **Lifetime Checker** at compile time as part of semantic analysis.

Examples

```
{
    let x = Thing::new();       // Allocate some heap memory
    let y = x;                  // Alias the value in x
    println("()", x);           // x has moved into y, so can
                                // no longer be referenced.
                                // This is a compile time error.
}
```

```
fn foo(val1: Bar, val2: Bar) -> (u32, Bar, Bar) {
    // do something with val1 and val2.
    // pass back ownership, so they can still be used by the caller.
    (123, val1, val2)   // This is pretty clunky.
}

// & denotes a borrow value. Borrows are immutable by default.
fn foo(val1: &Bar, val2: &Bar) -> u32 {
    // do something with val1 and val2, but cannot mutate them.
    123
}
```

Language Design

Extension of the Lambda Calculus, with higher kinded types and dynamic memory allocation.

Implementation

- Haskell
- `Alex` for lexing, `Happy` for parsing
- `HUnit` for later language tests
- Tests are mostly langauge source code.

Phases are split into parser, lifetime checker, borrow checker, type checker with higher kinds, and evaluation in some cases

Questions?