CS 301 Final Paper

First-Class Functions and Closures in IC

Katherine Kjeer

May 16, 2016

1 Introduction

This paper describes an extension to the IC compiler: the addition of first-class functions and closures to the IC language and the support of these features in the IC compiler. We discuss the background of first-class functions and closures and the motivations for adding them to IC. We then give an overview of the design and implementation of these features in the IC compiler. Finally, we describe the evaluation process and discuss the results of evaluating the compiler extension.

2 Background

First-class functions and closures are important concepts in functional programming [1]. In addition, they allow programs to have a greater degree of flexibility and abstraction - a function that takes another function as an argument allows the outer function to delegate specific behavior to the inner function [2]. Closures are represented with an environment defining the variables that the closure can access, together with the code of the underlying function.

3 Design

Syntax

A function f can be declared as follows:

```
function f(T_0 e_0, ..., T_n e_n): T_r = {
    //body of f
}
```

Anonymous functions can be defined as follows:

```
(T_0 e_0, ..., T_n e_n): T_r => {
    //body of anonymous function
}
```

Functions can be declared and initialized using function types:

```
<int, int> => string f;
f = (int x, int y): string => {
    //body of f
}
```

Methods and higher-order functions can take functions as arguments: for example,

```
void m(<int, int> => string f) {
    string message = f(1, 2);
}
```

Function Types

Function types consist of the product of their argument types, combined with their return type:

$$\frac{}{E \vdash (e_0 : T_0, e_1 : T_1, ..., e_n : T_n) \to e_r : T_r \; \mathbf{:} \; T_0 \times T_1 \times ... \times T_n \to T_r} \tag{1}$$

High-Level Implementation Process

1. Parse IC programs with the additional syntax for first-class function types, expressions, and calls, and resolve and type-check the resulting AST
2. Transform the AST into a new AST where each function type is converted to a template class type, each function expression is converted to a closure class instance, and each function call is converted to a call to the closure instance's `call` method

3. Resolve, type-check, optimize, and generate code for the transformed AST

4 Implementation

The source-to-source AST translation is done by traversing the AST, creating template and closure classes, and replacing function types with template class types, function expressions with closure class instances, and function calls calls to the closure's `create` method. The translation returns a new AST that is used in the remainder of the compiler.

Template Classes

For each function type in the original IC program, a template class is generated for the type. The template class contains no fields and no `create` method, but it does contain a minimal `call` method, which returns the default value for the function's return type (`0`, `true`, "", or `null`, or no return value for functions with a return type of `Unit`). Each function type is then replaced by the corresponding template class type. For example, the following declaration:

```
<int, boolean> => string f;
```

would be replaced by:

```
PARAMS__int_boolean__RETURN__string_Template f;
```

Closure Classes

For each function expression in the original IC program, a closure class is created. The closure class' fields contain the fields of the enclosing class, the parameters to the enclosing method, the local variables of the enclosing method that are in scope at the function expression, the parameters to any enclosing functions, and the local variables of any enclosing functions that are in scope at the function expression. Each closure class extends the template class that corresponds to its function type. For example, a function expression with type `<int> => boolean` extends the template class `PARAMS__int__RETURN__boolean_Template`, and overrides the template class' `call` method. The overriden `call` method takes the same parameters as the original function, and when called, executes the body of the original function. In addition, each closure class has a `create` method, which takes as arguments all the necessary variables to set the fields of the closure class (fields, parameters, and local

variables of the enclosing class, method, and functions), sets the closure's fields, and returns a new closure instance. Each function expression in the original program is replaced by a call to the appropriate closure class' `create` method.

Function Calls

For each function call in the orginal IC program, an instance of the appropriate closure class is made using the closure class' `create` method, and a virtual call is made to the closure instance's `call` method, with the same parameters that were passed to the original function call. For example, the following function call:

```
f{1, true};
```

would be replaced by:

```
new f_Closure().create{...}.call{1, true};
```

where the arguments to `create` are the fields, parameters, and local variables of `f`'s enclosing class, method, and functions.

5 Evaluation

The extension adds a language feature to IC, and does not focus on performance. Thus, the evaluation metrics are the accuracy of the translated programs and the ease of writing IC programs with first-class functions. The extension is evaluated by writing IC programs that use first-class functions and closures, and comparing those programs' output to their expected behavior. The test programs used to evaluate the extension are located in the test/projectTest folder of the compiler repository (https://goo.gl/Kh6Yho). The folder contains a variety of small programs to test parts of the language feature (function declaration and initialization, passing functions as parameters, calling functions, etc.). In addition, several larger programs test the feature in a more complex setting:

1. **Map and Filter:** This program implements the standard `map` and `filter` functions separately for `int`, `string`, and `boolean` arrays (since IC does not have a generic type system). The program tests the `map` and `filter` functions on all three types of arrays.
2. **Finite Groups:** This program represents finite (integer) groups as an array of integers and a function `<int, int> => int` operator. The `GroupInt` methods determine

whether the provided array and function form a group by checking the four group axioms. The `GroupIntDriver` class creates and tests a representation of the groups $\mathbb{Z}/5\mathbb{Z}$ and $\mathbb{Z}/8\mathbb{Z}$, and the non-group consisting of an array of random integers and a operation that returns a random integer. (This representation can easily be reproduced for `string` groups, etc.).

Each test program successfully type-checks and is translated to a reasonable AST and TAC representation without apparent errors. The test programs also run and produce the expected output for each program. For the larger programs **Map and Filter** and **Finite Groups**, the implementation could be expressed naturally using the first-class function language feature.

6 Conclusion

This extension added first-class functions and closures to IC using closure conversion in a source-to-source translation. The implementation allows non-trivial programs to be written using first-class functions, and the programs run and behave as expected. There are a few areas for future work:

1. With the current implementation, first-class functions do not have access to the methods of their enclosing class. For example, if a function `f` is defined in class `A`, and class `A` also defines method `m`, then `m` cannot be called within the body of `f`.

2. Several programs that use first-class functions generate segfaults when run after any optimizations (even when the optimizations make no changes to the TAC list). The optimizer test programs all run correctly, as do numerous other test cases not designed to test the optimizer.

References

[1] Jason Swartz. *Learning Scala: Practical Functional Programming for the JVM*. O'Reilly Media, Sebastopol, Califorina, 2014.

[2] Luke VanderHart and Stuart Sierra. *Practical Clojure*. Apress, New York, New York, 2010.