CS4201 P2

Intermediate Representations

Student ID: 200007413

November, 2023

Contents

1 Introduction

1.1 Aims and Objectives

The primary aim for this practical was to develop my understanding of late-stage compiler steps in the conversion from a higher-order defined language to a first-order defining language [1].

The main objective for this practical was to create a partial compiler which takes in a source language syntax tree and first defunctionalizes it to ensure function calls are always made with the correct number of arguments.

1.2 Main Achievements

The code for this submission successfully compiles valid source language syntax trees as defined by the specification into valid Java code. This involves defunctionalization, conversion to ANF, and then translation to a Java class which can be written to a file and compiled.

1.3 Report Structure

In section 2, a documentation of the process of developing the source code for this submission. This will include the design and implementation of defunctionalization, conversion to A-normal form (administrative normal form), and then translation of A-Normal form syntax trees to a valid Java class definition.

Then, section 3 will outline and justify the testing methodology which was conducted alongside and following implementation. Included will be figures displaying the cabal test results.

[1] It is important to note that whilst Java includes some higher-order functionality such as being able to store functions as variables, my understanding of the specification that the resulting Java should be first-order.

Finally, section 4 will include an evaluation of the final submission with regards to the aims and objectives for the practical. A reflection of the main challenges and achievements made will follow, and then a final reflection on what was learned, and further changes and additions that could be made to improve the quality of the submission.

2 Design and Implementation

2.1 Code Structure

There are two Haskell files which make up the code for this practical. `Defun.hs` contains example syntax trees representing valid source language programs which each consist of function definitions and a main expression. The file also contains the defunctionalization functionality which is accessed through the `defun` function. This ensure that each function call in the main expression and defined functions are correctly applied to the number of arguments required in the associated function definition.

`ANF.hs` imports this file, and includes functionality for converting the defunctionalized programs to A-Normal form, by extracting complex expressions into their own variables. The file also includes the translation functions for converting these ANF program syntax trees to Java.

2.2 Java Representations

Here I will discuss the design decisions and implementation process for this practical. Much of the design decisions in defunctionalization and ANF conversion are determined by the design decisions for the final Java representations of different data structures.

Therefore, it seems reasonable to start with these decisions, and the overall plan for converting expressions from the source syntax trees to java.

2.2.1 Variable/Value Representation

The first design decision was how variables and values should be represented. This is important as there are different expression types in the source language (constructors, integers, variables, etc.) which can all appear as parameters and return values from functions. However, from the example syntax trees, these are not specified in list of function arguments.

The solution implemented for this problem was to use the Java catch-all type `Object` for variables and values. When a particular type is required, it is possible to cast the variable/value to that type where required (the specification notes that it is reasonable to assume the source language program has already been type-checked).

2.2.2 Defunctionalization

The next design decision was with regards to defunctionalization, and how partially applied functions are represented and applied where required. As the source language is higher-order, it is possible for function calls to be under-applied, meaning not enough arguments are given, or over-applied meaning too many arguments are given. Java is first-order, and so functions must always be called with the correct number of arguments. [1]

In the source language, the result of under-application is an intermediary function which can be called with the missing arguments which will then evaluate the function. As functions can themselves return functions, over-application calls the function with the first set of arguments required by the function, and then applies the extra arguments to the returned function.

As we have specified that all arguments and return values from functions must be of the Object type, we then must decide on a way of representing under-applied functions as objects. For this, a new Java class `Funs` was created, which holds a reference to the function that will be called when fully-applied, it also holds an array of the arguments applied so far, and a helper integer value which represents the number of arguments the function needs to be fully-applied.

All instances of Classes in java are children of the overarching Object type, and so can be passed as arguments to, and returned from functions.

```java
class Funs {
    Class<?> c = Reference.class;
    String fn;
    int fn_args;
    Object[] args;

    // Create a Funs instance
    Funs(String fn, int fn_args, Object[] args) {
        this.fn = fn;
        this.fn_args = fn_args;
        this.args = args;
    }
}
```

Figure 1: The constructor class definition injected into the final java code output.

As these `Funs` objects are representations of functions, a method for calling them with the missing arguments is required. This is the apply function, and takes as parameters the `Funs` function representation, and the parameters to apply to it.

The apply function adds the list of new arguments to the current list of applied arguments. If still not enough arguments are given, then an updated `Funs` object is returned with the combined list of arguments. If the function is now fully-applied, then the function stored by the object can be called with the given arguments, and result returned. If too many arguments are now applied, then the correct number of arguments can be extracted, and the function called with those, and then the apply function called again with the extra arguments, if the function returned a `Fun` object. If the result of the call is not a `Funs` object, then the source language program was invalid (as it is attempting to call a non-function).

To call a function in Java by it's name, the function `Class.getDeclaredMethod` can be used. For this, the name of the function, as well as the number of arguments and the parameter types are required. As the `Funs` object can represent functions with different numbers of parameters, it is not easy to specify the correct number of arguments in `getDeclaredMethod` programmatically in Java.

One solution to this problem is to generate different apply functions for each function defined in a program, with each apply function specifying the number of arguments as given in the function definition. However, this would mean that the effective number of functions is doubled, as each function requires it's own apply function.

```java
public static Object apply(Funs fun, Object[] args) {

    // Apply the new args to the fun parameter list
    fun.args = Stream.concat(Arrays.stream(fun.args),
        Arrays.stream(args)).toArray();

    try {

        // Fetch the method
        Method method = fun.c.getDeclaredMethod(fun.fn, Object[].class);

        // Under-application - return the updated Funs object
        if (fun.args.length < fun.fn_args) {
            return fun;

        // Over-application - eval function and apply extra arguments to
    result
        } else if (fun.args.length > fun.fn_args) {
            Object[] extra_args = Arrays.copyOfRange(fun.args,
                fun.fn_args, fun.args.length);
            fun.args = Arrays.copyOfRange(fun.args,0,fun.fn_args);
            return apply((Funs) method.invoke(null,
                Arrays.copyOfRange(fun.args,0,fun.fn_args)),
                    extra_args);

        // Fully-applied - return the evaluation result
        } else {
            return method.invoke(null, (Object) fun.args);
        }

    } catch (Exception e ) {
        e.printStackTrace();
        return null;
    }
}
```

Figure 2: Source code for the apply function

The code in Figure 2 shows a novel solution developed for this submission which is injected into the final Java class. This function accepts any `Funs` object, and an array of parameters to apply, avoiding the need for generating new functions for each function that would need applying.

To work, the parameters of the functions had to be modified from the arguments themselves, to a single generic Object[] array, the start of each function can then resolve

the arguments by indexing this array. The result of this change is that all functions are parameterised by the same single Object[].class type parameter.

The `getDeclaredMethod` function can then just needs the function name to find the function. The apply method can then check whether the application has made the function fully, under, or over-applied. If under-applied, the updated partially applied function object can be returned. If over-applied, the extra arguments can be extracted, and the function called with the correct number of arguments. This result must be a function representation, and so can be applied again using the apply function with the extra arguments. If fully-applied, then the function can be called with the updated argument list.

2.2.3 Calling Functions

Another design challenge faced was with the names given for function calls. As we have seen in the previous section, through return values and parameters, functions can be stored as variables as `Funs` objects. However, this means that `Call` statements be made to function names, or to these `Funs` objects.

To prevent the need to distinguish in the first place, all function calls of either function variables or functions directly can be converted into `Funs` objects wrapped in the `apply` function with the given arguments. This means that all function calls go through the apply function.

2.2.4 Constructors

```
Con "Cons" [(Val 3), (Con "Cons" [(Var "x"), (Con "Nil" [])])]
```

Figure 3: Example of a constructor in the source language representing a list with two elements: integer 3 and the variable x.

The source language includes lists and pairs implemented as constructor `Con` expressions, with string representing the constructor type, and a list where the head is a value, and the tail another `Con` expression. A string value of ”Nil” implies that the list is empty (see. Figure 3).

```
class Con {
    String type;
    Object x;
    Con xs;
    public Con(String type, Object x, Con xs) {
        this.type = type;
        this.x = x;
        this.xs = xs;
    }
}
```

Figure 4: The constructor class definition injected into the final java code output.

Java represents lists as arrays, and not constructors in the same way. Therefore, I decided to design a new constructor class `Con` which stored the type string, and then a head variable, and tail constructor variable (see. Figure 4).

2.3 Administrative Normal Form

The next part of the practical was to convert the defunctionalized syntax trees to administrative normal form (ANF), which extracts sub-expressions into `let` expressions to make the evaluation order explicit.

This process is required for all expressions which combine two or more expressions together (Cons, Call, Funs, BinOp), and so we can convert the base defunctionalized `Expr` syntax tree to a new `ANFExpr` syntax tree, where the compound expressions are replaced by variable names which refer to variables created in let expressions.

This requires generating unique names for the new variables created in ANF conversion to ensure that variables are not overwritten or re-initialized. One approach to solving this problem would be to create a marker where a new variable is required, and map the expression to a new variable name generated by the number of variables created so far.

However, a simpler approach I took was to generate names based upon their position in the syntax tree. As the syntax tree is traversed, each expression adds a unique marker for that expression to the name string. When a new variable is required, it just uses that name.

This is useful as we then do not need to keep track of the number of defined variables during ANF conversion or java translation. We can just keep track of the state of the syntax tree up till the current point, and then use the string as the identifier. In this submission, an identifier ”ANF” is added to the start of variables in the main function in hopes of avoiding reserved names and argument names.

2.4 Defunctionalization of Source Syntax Tree

Much of the design for defunctionalization here relies upon the design decisions made above for the Java translations and ANF representations. However, the defunctionalization begins with processing the initial source language syntax trees.

Two new `Expr` entry's were added. These are not valid source language expressions, and so wherever they appear, we know are created by defunctionalization. To unify the representation of functions, we can covert the function called by each Call expression to a Funs object and application. This is because we don't know if function calls are calling Funs objects as variables, or directly referring to a defined function. We can check the given name against the list of defined functions, and either create a Funs object based on the function name and wrap in an apply function call with the given arguments, or just wrap the Funs object in an apply call with the given arguments.

3 Testing

The testing framework Hspec was used which enabled unit testing of each function individually (see. Listing 1), as well as together to ensure the desired functionality was maintained throughout each part of the compilation process. Unfortunately, time limitations mean that the testing is not as thorough as I would have liked.

```
    Test suite spec: RUNNING...

ANF
  Unit Test ANF Expr Conversion
    converts Var to AVar [✓]
    converts Val to AVal [✓]
    extracts two sides of BinOp to correct variable names [✓]
    extracts Call arguments into let expressions correctly [✓]
    extracts value of Let expression correctly [✓]
```

```
    names ANF vars in Let body uniquely [✓]
    extracts the premise of an If statement into a let expression [✓]
    names ANF vars in if conclusion and alternative correctly [✓]
Defun
  DefunExpr Unit Test
    Does not change Vars [✓]
    Does not change Vals [✓]
    Converts function calls to defined functions to applications on Funs [✓]

Finished in 0.0039 seconds
11 examples, 0 failures
Test suite spec: PASS
```

Listing 1: Test results for the intermediate representations implementation

4 Evaluation

4.1 Aims and Objectives

Personally, I feel like I have greatly developed in the areas specified by the aims for this practical. I have a much greater understanding of the motivation for defunctionalization and ANF, and how the practicalities of these translations greatly depend upon the functionality of the defining language.

Overall, this submission meets all of the objectives of this practical. It allows a source language source tree to be converted into a compile-able and run-able Java class with function definitions, and a main function. The defunctionalization ensures that all function calls are made to the correct number of arguments, and function arguments and parameters return functions as an object representation. All expressions are successfully converted to ANF which ensures that the evaluation order is made explicit,

There are also extensions from the base requirements in the specification: a novel approach to the application to reduce the redundancy with having multiple apply functions is implemented. The main function's return value has also been extended to be able to return constructors.

4.2 Further Improvements

Given more time, there are features and issues I would have addressed given more time or another attempt. However, these are mostly small quirks, and code-organization issues.

The names of functions in the source syntax tree are maintained in the resulting Java code. Whilst this makes debugging the resulting Java easier (as Java errors provide stack traces with the function names), but means that all reserved terms should be avoided to prevent compilation errors. This could be solved relatively easily by generating new names for functions. One method for solving this would be to use the index of each function, and generate a name based off of that. Alternatively, the names for each function could be prepended with a string such as `ANFF_`, which is unlikely to conflict with any existing Java names.

Also, the structure of the exprToJava could be refactored to be more readable. The nextJavaLine function works to determine whether there needs to be a return or variable declaration added, but this makes the order of execution, as sometimes the function is called, and sometimes not, depending on the expression being converted.

Unifying expression evaluations to always use this function first, and where not required, just passing back to the exprToJava function would make the order of operation simply just the passing back and forth between the two functions for each expression.

References

[1] John C Reynolds. “Definitional interpreters for higher-order programming languages”. In: *Proceedings of the ACM annual conference-Volume 2*. 1972, pp. 717–740.