CS4201 - Practical 2 - Intermediate Representations

210018092

20th November 2024

Contents

Overview

The aim of this practical was to implement part of a compiler/translator for a small expression language, by converting to increasingly simple intermediate representations, then finally generating code in a target language (in this case, Java). There are three main steps to this process:

1. Defunctionalise the program, so that all function calls are defined top level functions, with the correct number of arguments.
2. Convert the defunctionalised program to ANF, so that the operands to all function calls, operators, and scrutinees of case and if expressions are variable names.
3. Generate Java code from the ANF. There should be one method for each function in the ANF (which will be the original functions, plus the APPLY function generated by defunctionalisation), and a main method which evaluates the main expression.

I have implemented the above steps in Haskell, in the `ANF.hs` file, which takes a `Defun` program, defunctionalises it, converts it to ANF, and generates executable Java code. In addition to this I have made my solution modular to both help with readability and to ensure that additional language features are handled with minimal or no changes to the existing code. I also decided to go beyond the practical specification and implement a test suite to ensure the correctness of my implementation to a greater extent than the provided test cases. This fulfills both the basic and extended requirements of the specification and beyond.

Design and Implementation

The implementation of this practical was structured into three distinct phases: defunctionalisation, conversion to Administrative Normal Form (ANF), and Java code generation. Each phase required specific design decisions to ensure correctness, maintainability, and seamless transition between the intermediate representations. This section delves into the key design considerations, challenges, and solutions implemented during the project.

2.1 Defunctionalisation

Defunctionalisation was the first phase of the transformation pipeline. The primary challenge in this phase was ensuring that all function calls were converted to top-level functions with the exact number of arguments. This required handling three distinct scenarios: perfectly applied functions, partially applied functions, and over-applied functions.

For partial application, a key design decision was to introduce constructor tags to represent the intermediate state of a function application. For example, a function `f(x, y, z)` partially applied with arguments `x` and `y` would be represented by a constructor `f_2(x, y)`. These tags serve as a compact and systematic way to track the number of arguments already applied, allowing the program to distinguish between different states of application. This approach simplified the transformation logic while maintaining the flexibility to handle arbitrarily complex applications.

Over-application presented an additional complexity, as it required splitting the excess arguments into successive calls to a special `APPLY` function. The design of `APPLY` was particularly interesting. It dynamically evaluated the remaining arguments and either completed the function application or returned a new partially applied function. By introducing this mechanism, the implementation achieved a balance between correctness and extensibility, ensuring that even cases with excessive arguments were handled gracefully.

The defunctionalisation process also needed to analyze the arity of all functions in the program. This was implemented using a helper function that mapped function names

to their respective argument counts. This mapping ensured that all transformations respected the expected input structure of each function.

2.2 Conversion to ANF

Administrative Normal Form (ANF) requires that every operand in a function call, binary operation, or conditional be a variable. The conversion to ANF involved restructuring each expression into a series of let-bindings, where intermediate results were assigned to fresh variables.

A significant design decision here was the mechanism for generating unique variable names. A helper function, `fresh`, was implemented to produce unique variable names by appending numeric suffixes to a base name. This ensured that variable names did not collide, even in deeply nested or recursive expressions. By using this systematic naming approach, the implementation avoided bugs related to scope and variable shadowing.

Handling case expressions in ANF posed another interesting challenge. Each constructor alternative in a case expression required extracting its arguments into distinct variables. To achieve this, the transformation introduced local bindings for constructor arguments, ensuring they were accessible in the body of each alternative. This not only preserved the semantics of the original expression but also simplified the subsequent translation to Java code.

Binary operations and conditionals were similarly transformed into ANF by isolating their operands. For instance, an expression like `x + y` was split into two let-bindings for the values of `x` and `y`, followed by a third let-binding for their sum. This systematic decomposition ensured that all expressions adhered to the strict variable-centric structure required by ANF.

2.3 Java Code Generation

The final phase involved translating the ANF representation into executable Java code. A major design decision here was to represent all values uniformly using Java's `Object` type. While this introduced some type-checking overhead, it provided the flexibility to handle both primitive values (like integers) and complex data structures (like constructors).

Constructors in the source language were mapped to a custom Java class, `Constructor`, with fields for a tag (representing the constructor name) and an array of arguments. This design allowed case expressions to be translated into Java `switch` statements, where the

constructor tag served as the discriminant. The arguments of each constructor were extracted into local variables, ensuring they could be used in the body of the corresponding case alternative.

A particularly noteworthy decision was how to handle nested expressions during Java code generation. To preserve the semantics of the ANF representation, each expression was translated recursively, with intermediate results stored in local variables or anonymous inner classes. For example, let-bindings in ANF were mapped to anonymous inner classes in Java, encapsulating the binding and ensuring proper scoping of variables. This approach, while unconventional, allowed for a direct and faithful translation of ANF to Java.

Function calls in Java were another area of interest. Top-level functions were translated into static Java methods, while the `APPLY` function was implemented as a generic method capable of handling partial application. The use of static methods ensured efficient function invocation while maintaining the modularity of the generated code.

2.4 Challenges and Solutions

One of the most intriguing challenges arose in the interaction between defunctionalisation and ANF conversion. Specifically, partially applied functions needed to be represented as constructor tags during defunctionalisation, but their ANF transformation required careful handling to avoid losing their state. This was resolved by ensuring that constructor tags introduced during defunctionalisation were treated as first-class values during ANF conversion.

Another challenge was the translation of nested function calls. For example, an expression like `f(g(x))` required transforming `g(x)` into a let-binding, followed by passing the resulting variable as an argument to `f`. This nested restructuring was implemented recursively, with each layer of the call stack generating new bindings for intermediate results. The recursive approach not only maintained the integrity of the original expression but also ensured that the generated Java code remained readable and efficient.

Lastly, the design of the `APPLY` function was a highlight of the project. Its ability to dynamically handle varying argument counts, combined with the use of constructor tags for partially applied functions, demonstrated the power of intermediate representations. This design ensured that the system could gracefully handle edge cases, such as over-application or nested partial applications, without requiring ad hoc solutions.

Testing

All tests were run on a lab machine. To run the tests, run the following commands in the `code` subdirectory:

```
$ ghc -main-is Test.mainTests -o Test Test.hs ANF.hs Defun.hs
$ ./Test
```

This will run all the tests and output the results to the console. As you can see from the terminal output below, all tests pass. If you are interested in the exact programs being run by each test, then please look at the `Test.hs` file in the `code` subdirectory. Please note I have copied the textual output of the terminal directly instead of taking a screenshot for accessibility reasons, but you can run the tests yourself to verify the results.

```
****@pc8-025-l:~/pldi-p2/code $ ./Test
Running test suite for Defun.hs and ANF.hs...
Running test: Double a number
PASS: Expected = 6, Returned = 6
Running test: Factorial of 5
PASS: Expected = 120, Returned = 120
Running test: Extract first element of a pair
PASS: Expected = 1, Returned = 1
Running test: Sum elements of a predefined list
PASS: Expected = 3, Returned = 3
Running test: Sum after mapping double over a list
PASS: Expected = 6, Returned = 6
Running test: Sum empty list (Nil)
PASS: Expected = 0, Returned = 0
Running test: Factorial of 0 (base case)
PASS: Expected = 1, Returned = 1
Running test: Over application
PASS: Expected = 6, Returned = 6
Running test: Under application and then apply
PASS: Expected = 4, Returned = 4
Running test: Empty definitions, valid program (1 + 1)
PASS: Expected = 2, Returned = 2
Running test: Add two numbers
PASS: Expected = 7, Returned = 7
Running test: Add zero
PASS: Expected = 10, Returned = 10
Running test: Add negative numbers
PASS: Expected = -8, Returned = -8
```

```
Running test: Complex chaining: add -> map -> sum
PASS: Expected = 0, Returned = 0
Passed 14 out of 14 tests.
```

Evaluation

The implementation of the practical demonstrates the transformation of a high-level expression language into executable Java code. While the objectives were successfully achieved, the process presented several challenges that required creative solutions. This section discusses these challenges, the strategies employed to address them, and the limitations that remain.

4.1 Challenges and Solutions

One of the most significant challenges was handling partial and over-application of functions during the defunctionalisation phase. Partial application, where a function is called with fewer arguments than its arity, had to be represented in a way that could carry the state of the partially applied arguments. Over-application, where more arguments are passed than expected, required similar attention to dynamically manage the additional arguments. The solution involved introducing a specialized `APPLY` function, which could handle these cases uniformly by encapsulating partially applied functions in constructors. For example, if a function was partially applied, a constructor object was created to hold the applied arguments, while over-applied functions were processed iteratively using `APPLY`.

Another notable challenge was ensuring variable name uniqueness during the conversion to Administrative Normal Form (ANF). In ANF, intermediate expressions must be assigned to unique variables, and collisions could lead to incorrect behavior or compilation errors in the generated Java code. To address this, a `fresh` function dynamically generated unique variable names based on the current program state, ensuring that no two variables shared the same name. This systematic approach avoided errors and allowed for seamless integration of complex nested expressions into the ANF representation.

The translation of case expressions into Java posed another significant obstacle. In the high-level expression language, `case` expressions operate on constructors, allowing dynamic pattern matching. Mapping this to Java required a design that encapsulated constructors with a tag and arguments, enabling `switch`-like behavior in Java. The solution involved defining a `Constructor` class in the generated Java code, with fields for a tag (to

identify the constructor type) and an array of arguments (to hold the constructor's values). During code generation, case expressions were converted to Java `switch` blocks, dynamically extracting constructor arguments as needed.

Finally, the recursive nature of some language features, such as functions operating on lists, presented a challenge in both correctness and efficiency. For instance, translating the `map` function required careful nesting of function calls to maintain correctness in both defunctionalisation and ANF stages. While this was successfully handled, it required intricate management of argument passing and scoping.

4.2 Limitations

Despite the successes, the implementation has several limitations. The generated Java code, while functional, is not optimized for readability or efficiency. The reliance on a generic `Object` type for all values adds a level of indirection, making the code harder to interpret and potentially introducing runtime overhead. Although this approach simplifies type handling in Java, it sacrifices the strong typing guarantees available in the original Haskell representation.

Another limitation lies in the recursive functions' behavior in Java. Deeply nested recursive calls can lead to stack overflow errors, as Java's runtime does not optimize tail recursion. This contrasts with Haskell's ability to handle such scenarios more gracefully due to its lazy evaluation model. Addressing this would require additional work to transform recursive functions into iterative equivalents during code generation, which was beyond the scope of this practical.

Additionally, while the implementation handles partial application effectively, it does so at the cost of introducing intermediate constructors for partially applied functions. This approach, while conceptually elegant, adds complexity to the generated Java code and increases runtime overhead when functions are applied repeatedly. A more efficient approach could involve precomputing closures or directly translating partial applications to inline methods.

Finally, error handling in the generated Java code is minimal. The high-level language assumes type correctness and valid inputs, but the generated Java does not perform robust checks for invalid states, such as mismatched constructor tags in `case` expressions. Adding such checks would enhance robustness but would also increase the size and complexity of the generated code.

4.3 Reflection and Future Work

Overall, the implementation successfully achieves the goals of the practical. The use of intermediate representations, such as ANF, demonstrates a clear pipeline from a high-level language to executable Java code. However, future work could focus on improving the readability and performance of the generated code. This could include optimizations for recursive functions, more efficient handling of partial applications, and enhanced error handling in the Java output. Furthermore, incorporating static typing into the generated code could align more closely with Java's strengths, reducing runtime overhead and improving maintainability.

The practical provided valuable insights into compiler design, particularly the importance of well-defined intermediate representations. By addressing the challenges and recognizing the limitations, this implementation highlights both the complexity and elegance of transforming abstract syntax trees into executable code.

Conclusion

In this practical, I have implemented part of a compiler/translator for a small expression language, by converting it to increasingly simple intermediate representations, then generating executable Java code from the ANF. My solution handles all five of the provided test programs as well as additional test cases. In addition to this, I have made my solution particularly modular and scalable to ensure that additional language features are handled with minimal or no changes to the existing code. I have gone beyond the practical specification by implementing a test suite to ensure the correctness of my implementation to a greater extent than the provided test cases. This fulfills both the basic and extended requirements of the specification and beyond.