University of Amsterdam

Assignment 5

Code Transformation and Optimisation

March 13, 2021

Students:
René Kok
13671146

Aram Mutlu
13574116

Lecturer:
Dhr. dr. C.U. Grelck

Course:
Compiler Construction

Course code:
5062COMP6Y

1 Introduction

This report provides information regarding the fifth assignment "Code Transformation and Optimisation" of the course Compiler Construction by Dhr. dr. C.U. Grelck. The goal of this assignment is to transform the given code into a Static Single Assignment Form (SSA), apply the loop unswitching optimisation and devise a formal compilation scheme that systematically eliminates all occurrences of while-loops.

2 Code Transformation and Optimisation

Consider the following CiviC code fragment:

```
i = 0;
while (i < n) {
    j = 0;
    while (j < m) {
        if (i < j) {
            val = val + i;
        } else if (j == i) {
            val = val - 1;
        } else {
            val = val + j;
        }
        j = j + 1;
    }
    i = i + 1;
}
```

Listing 1: CiviC code fragment

2.1 Static Single Assignment Form

Transform the above code into Static Single Assignment Form (SSA).

```
i_0 = 0;
p_0 = i_0 < n_0;
while (phi(p_0, p_1)) {
    j_0 = 0;
    i_1 = phi(i_0, i_2);
    y_0 = j_0 < m_0;

    while (phi(y_0, y_1)) {
        j_1 = phi(j_0, j_2);

        if (i_1 < j_1 ) {
            val_2 = phi(val_0, val_7);
            val_1 = val_2 + i_1;
        } else if ( j_1 == i_1 ) {
            val_4 = phi(val_0, val_7);
            val_3 = val_4 - 1;
        } else {
            val_6 = phi(val_0, val_7);
            val_5 = val_6 + j_1;
        }

        val_7 = phi(val_1, val_3, val_5);

        j_2 = j_1 + 1;
        y_1 = j_2 < m_0;
    }
    i_2 = i_1 + 1;
    p_1 = i_2 < n_0;
}
```

Listing 2: Static Single Assignment Form (SSA)

2.2 Machine-Independent Optimisation

Apply the loop unswitching optimisation to the (original) code above.

```
i = 0;
while (i < n ) {
    j = 0;

    while (j < m && i < j) {
        val = val + i ;
        j = j + 1;
    }

    if (j < m && j == i) {
        val = val - 1;
        j = j + 1;
    }

    while (j < m) {
        val = val + i ;
        j = j + 1;
    }

    i = i + 1;
}
```

Listing 3: Loop unswitching optimisation

2.3 Compilation Schemes

Devise a formal compilation scheme that systematically eliminates all occurrences of while-loops in the body of a CiviC function definition and replaces them by semantically equivalent code without while-loops.
To replace while-loops with semantically equivalent code, we need to know what our possibilities are. An option is to use a never ending for-loop with a break-if statement, but this is not supported in the CiviC language. An other option is to use recursive functions with a return-if statement (this is supported by the CiviC language). Last option is to use do-while-loops which is also supported by the CiviC language. We will replace the while-loops in a body of a CiviC function with do-while-loops with the following steps:

1. Check with pattern matching for while-loops in the code.
2. Recursively loop until no next pattern is found
3. Replace while-loop by do-while-loop:

```
C [[ while ( Expr ) Block Rest ]]
=> CX [[ { do Block while ( Expr ) } Rest ]]

Block => {[Statement]*}
       |  Statement
```