Final Fanatic Facility
A Well-Organized Modern C Compiler

Junru Shao 邵俊儒[*]

ACM Honored Class
Zhiyuan College
Shanghai Jiao Tong University

2015 年 5 月 31 日

Abstract

This is a report for my implementation of a C language compiler, which generates simple C code from MIPS assembly code. The compiler is implemented in Java and supports major feathers of C. Implementation of Static Single Assignment (SSA), interference graph coloring register allocation, and superior design of interface which leads to its portability is the three main outstanding points of my compiler, which distinguish itself from other compilers. The first section of the report contains detailed explaination of symbol table design, abstract syntax tree design. Section 2 introduces comprehensively my intermediate representations. Section 3 gives a general view of my optimization, mainly SSA-based optimization, while register allocation is also included. Section 4 list the usage of FFF, my compiler, as required by TAs. The last section draws a conclusion of my compiler.

[*]Class ID: F1324004. Student ID:5130309028.

Contents

1 Symbol Table Design, Lexer, Parser, Semantic Check

1.1 Parser and Variadic Functions

FFF makes good use of the existing parser, ANTLR 4.5, eliminating the need to reinvent the wheel. In order to support complicated types, I rewrote the grammar given in the course home page. This stage is somehow long, trivial and boring, as well does not differ much from others who also supports complicated type analysis.

The highlight is that FFF supports variadic functions, who has indefinite number of arguments. In fact, FFF uses internal library StdLib.c, to define the function int printf(char *, ...) . The function is hand-written by myself and is listed below.

```
int printf(char *format, ...) {
    va_list ap;
    char *prev = format;
    int arg, tmp, len;
    va_start(ap, format);
    for ( ; *format; ++format) {
        if (*format == '%') {
            *format = 0;
            ___yzgysjr_lib_putstring(prev);
            *format = '%';
            ++format;
            if (*format == 'd')
                ___yzgysjr_lib_putint(va_arg(ap, int));
            else if (*format == 'c')
                putchar(va_arg(ap, char));
            else if (*format == 's')
                ___yzgysjr_lib_putstring(va_arg(ap, char *));
            else {
                len = *(++format)-'0';
                ++format;
                arg = va_arg(ap, int);
                if (arg < 0) {
                    arg = -arg;
                    --len;
                    putchar('-');
                }
                for (tmp = arg; tmp; tmp /= 10)
                    --len;
                for (; len > 0; --len)
                    putchar('0');
                if (arg)
```

```
                    ___yzgysjr_lib_putint(arg);
                }
                prev = format + 1;
            }
        }
    ___yzgysjr_lib_putstring(prev);
}
```

My implementation of va_list, va_start, va_arg is identical to the standard one, which uses define expansion.

```
#define _INTSIZEOF(n) ((sizeof(n)+sizeof(int)-1)&~(sizeof(int)-1))
#define va_start(ap, v) (ap = (va_list)&v + _INTSIZEOF(v))
#define va_arg(ap, t) (*(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)))
typedef char *va_list;
```

1.2 Symbol Table Entries

Symbol table is carefully designed in FFF. The compiler communicate with symbol table from beginning of lexing to the end of translating to MIPS assembly.

Every entry in the function table has the following fields.

```
public class SymbolTableEntry {
    public int uId;
    public String name;
    public int scope;
    public Tokens type;
    public Object ref;
    public Object info;
}
```

FFF uses *uId*,short for universal identifier, to represent a symbol. Field *name* is simple the name of the symbol, empty or null if the symbol is anonymous. Field *scope* is the nested depth of the symbol. The next three fields are relatively important and reflects the philosophy of FFF Compiler.

Note that I used the word “universal”, every symbol, even virtual registers, are stored in symbol table. Details are shown in the following table.

Enum Tokens type	Object ref	Object info
VARIABLE	type	initializer
TYPEDEF_NAME	type	null
STRUCT_OR_UNION	reference to the type	declaration status
STRING_CONSTANT	the string	null
TEMPORARY_REGISTER	null	null

Explanation for declaration status: The operations in symbol table could be considered to has two main steps:

Declaration Declare that some symbol exists.

Definition Define clearly and completely.

Additionally, environment in FFF contains two symbol tables, handling structs and other variables separately.

The most important thing is that **a function is also a considered as a variable**. This design benefits a lot in the functional programming, although FFF only support simple higher-order functions.

1.3 Complicated Type Analyser

FFF maintains a stack to analyse complicated types, for more details, see Compiler2015.Parser.TypeAnalyser.java. The following code is from Manfan Yin, a good person who helped me a lot. FFF can resolve it easily and generate correct MIPS code.

```
struct A {
        int x, y;
        struct B {
                int i, j;
        } b;
        struct A *next;
};

/* a function that returns a pointer to function */
int (*func(int flag, int (*f)(), int (*g)()))() {
        if (flag) return f;
        else return g;
}

int main() {
        struct A a;
        /* the follow types are distinctly different */
        int a0[10][20]; /* two-dimensional array */
        int (*a1)[20]; /* a pointer to array */
        int *a2[20]; /* an array of pointers */
        int **a3; /* pointer to pointer */
        /* pointer to a function */
        int (*f)(), (*h)();
        /* function declaration, not a variable */
        int (*g(int ***e[10]))();
        /* complex type casting is also supported */
        f = (int (*)())(0x12345678);
        f = func(1, f, main); /* f */
        h = func(0, f, main); /* main */
        /* 0 1 */
        printf("%d␣%d\n", f == main, h == main);
}
```

1.4 Abstract Syntax Tree Design

Humor: There is an old saying that an AST who is not an IR is not a good AST, an AST who is not different from CST, is not different from salted fish.

While parsing, all variables, declarations, definitions and types, are stored in the AST. A typical CST or AST may contain declarations, types, and initializers. With the support of symbol table, any declaration is redundant to show in AST, so is definition and types. For convinience, I also convert initializer to a list of position-value pairs, where position indicates the subscript index of variable in an array the initializer initializes, and value is the expression to initialize the variable.

In all, AST in FFF only contains Statements. Below is the hierarchy of FFF's AST superclass.

```
Statement extends ASTNode
    ExpressionStatement
        (omitted)
    BreakStatement extends Statement
    CompoundStatement extends Statement
    ContinueStatement extends Statement
    ForStatement extends Statement
    IfStatement extends Statement
    ReturnStatement extends Statement
    WhileStatement extends Statement
```

Note that the AST is actually an Intermediate Representation, we could do interpretation, optimization, or generate assembly code directly from the AST with no difficulty.

The construction of AST is a perfect example of communication with symbol table.

2 Intermediate Representation

2.1 Intermediate Representation Tree

As mentioned above, AST in FFF is in fact a IR tree. This is the first version of my IR. I only do constant expression elimination on the tree, and immediately convert it into the second form of IR: control flow graph. Especially, generating stack machine code is a direct translation, which means generating code for JVM is easy. Due to the limited time, I did not write this part of bonus.

2.2 Control Flow Graph

A control flow graph (CFG) in is a representation, using graph notation, of all paths that might be traversed through a program during its execution.

Each vertex in CFG should have at most one conditional jump instruction, and the instruction must be placed at the end of the vertex. Thus, every vertex in CFG has at most two outcome edges. This is the second IR in FFF Compiler. To simplify the problem, FFF Compiler use *unconditionalGoto* and *branchIfFalse* to represent two outcome edges, and the criterion of conditional branch is save in the virtual register called *branchRegister*.

2.3 Linear IR Instruction Embedded in CFG

In the internal of vertex of CFG, there are linear instructions. I design several kinds of IR instructions, mostly for convenience of SSA construction, optimization and assembly translation. Here is the hierarchy:

```
ThreeAddressInstruction extends IRInstruction
    (omitted)
TwoAddressInstruction extends IRInstruction
    (omitted)
Call extends IRInstruction
Def extends IRInstruction
FetchReturn extends IRInstruction
Move extends IRInstruction
Nop extends IRInstruction
NopForBranch extends IRInstruction
PhiFunction extends IRInstruction
PushStack extends IRInstruction
ReadArray extends IRInstruction
SetReturn extends IRInstruction
WriteArray extends IRInstruction
```

Note: Def is used for global variables and parameters passed in in SSA construction. PhiFunction, NopForBranch is used in SSA optimization.

In IR, we also should use virtual registers, I design the super class IRRegister, and four kinds of IRRegister. The first kind is VirtualRegister, the usual one. The second kind is ImmediateValue, in convenience to represent immediate value. The third kind is array register, a combination of VirtualRegister and ImmediateValue, which might be used to represent memory access (In IR of FFF, only ReadArray and WriteArray could access memory).

2.4 Static Single Assignment Form

SSA is a property of IR, which requires that each variable is assigned exactly once, and every variable is defined before it is used. That is the reason why I use Def as an IRInstruction.

SSA is a modern technic in the field of compiler. I am the only one in our class who fully implemented SSA and optimizations based on SSA. I do not tend to describe what is SSA and how to construct and destruct SSA, which of these are easy to be found in any modern books about compiler.

An industrial compiler should guarantee its compiler's theoretical time complexity. Thus FFF uses **Lengauer-Tarjan Algorithm** to build dominator tree. After calculating dominance frontiers, FFF implements the **Fully Pruned Minimal SSA** insertion technic. These two algorithms are the best known algorithm in this field.

3 Alias Analysis, Optimization and Register Allocation

3.1 Alias Analysis

FFF does not tend to spill all the variables that there exists some pointer refering to it. On the other hand, it perform some operation on the AST and symbol table, changing a variable to an array containing only this variable, force it to access memory. Redundant memory access could be eliminated applying SSA on the whole memory, consider the memory as a single big variable.

For example, consider the following code fragment:

```
int a, *b;
b = &a;
a = 1;
```

It will be converted to the following:

```
int a[1], *b;
b = a;
a[0] = 1;
```

3.2 Global Dead Code Elimination

In SSA, dead code elimination is rather easy, consider the simplified def-use chain. One def has no use can be easily eliminated.

However, before constructing SSA, I performs naive dead code elimination, using LiveOut set. Detailed explanation could be seen on any compiler books.

3.3 Global Copy Folding while Constructing SSA

There exists a little trick for SSA construction that copy folding (copy propagation) could be done during the renaming stage, because a variable is defined only through its dominators or ϕ function. Therefore, during the depth-first walking in dominator tree, FFF maintains the renaming stack, if $x \leftarrow \text{Copy}(y)$, it pushes y to x's stack. The pseudocode is presented below.

```
function rename(X)
    for each statement A in block X do
```

```
        if A is not phi function statement
            for variable v in use(A)
                replace v with Top(Stack(v))
            end for
        end if
        for each variable v in def(A) do
            if A is CopyStatement v = vr
                PushStack(vr, Stack(v))
                Set A as Nop
            else
                get new name for v as v_new
                PushStack(v_new, Stack(v))
                replace v with v_new
            end if
        end for
    end for
    for each block Y in succ(X) do
        replace parameter v of phi function in Y with v_new
    end for
    for each block Y in childOfDominatorTree(X) do
        rename(Y)
    end for
    Pop all variables pushed while processing X
end function
```

3.4 Global Common Subexpression Elimination: with Copy Propagation, Constant Propagation

After SSA construction, we could do common subexpression elimination, like copy folding in SSA construction, we can do copy propagation and constant propagation along with subexpression elimination, the approach is easy: walking on the dominator tree, maintain two tables, one for expression and the other for copy.

3.5 Interference Graph Coloring Register Allocation

Linear scan register allocation is boring for me. In order to test the performance of graph coloring, I use a naive approach, color the graph bottom-up.

First, we need to construct the interference graph. We do a data-flow analysis and find interferences from the last instruction of a block to the first one, maintaining a set LiveNow representing the variables live at the current instruction.

```
for each block b do
```

```
    LiveNow = LiveOut(b)
    for each operation op do
        for each x in LiveNow do
            add interference edge (x, def(op))
        end for
        remove def(op) from LiveNow
        add use(op) to LiveNow
    end for
end for
```

After interference graph is built up, FFF colors the graph intuitively and tries to minimize the spill cost. In fact, SSA cut the live ranges of variables automatically, which result in the good performance of this interference graph coloring.

3.6 Performance Outline

I did not optimizations other than the ones described above, but the running speed is rather fast.

4 Usage of Compiler

4.1 Instruction to Compile

```
$CCHK < input.c > assem.s
```

4.2 Dumping Information

Note that do not combine arguments together, my argument parser only parse the first recognisable argument.

4.2.1 Dumping AST with Symbol Table

```
$CCHK -emit-ast < input.c
```

4.2.2 Dumping Basic Control Flow Graph

```
$CCHK -emit-ast < input.c
```

4.2.3 Dumping Control Flow Graph in SSA Form

```
$CCHK -emit-ssa < input.c
```

4.2.4 Dumping Control Flow Graph after Optimization Based on SSA

```
$CCHK -emit-optimized-ssa < input.c
```

4.2.5 Dumping Control Flow Graph after SSA destroyed

```
$CCHK -emit-optimized-cfg < input.c
```

4.3 Using Pretty Printer

```
$CCHK -printer=kr < input.c # for ms style pretty printer
$CCHK -printer=ms < input.c # for k&r style pretty printer
```

4.4 Pointers to Functions

The compiler automatically support pointer to functions.

4.5 Higher-Order Functions

The compiler automatically support higher-order functions.

4.6 Comprehensive Examples for All Bonus I Wrote

This is a program of quicksort modified from the given testcases.

5 Conclusion

Final Fanatic Facility (FFF) is a satisfying compiler that truly based on SSA form. This is the only compiler that really uses SSA in our class.

6 Thanks

First, I should thank all of our TAs for their hard work, and providing us kindly this chance to change our limit.

I would like to thank Manfan Yin for his generous help.

I would like to thank Jian Weng for his accompanying.