L3 Compiler Internals Guide

Mike De Rosa and Peter Lee
15-745 Optimizing Compilers

Spring 2006

1 Introduction

This is a quick guide to the internals of the L3 compilers for the Spring 2006 incarnation of *15-745 Optimizing Compilers*. This guide does not provide comprehensive documentation for the compilers, but rather a few starting points to help you in your assigned tasks.

There are three versions of the compiler. All three versions compile the same language, called L3. L3 is a safe, simple, C-like language. It is not intended to be a practical language for writing real-world programs, but instead an object of study for compiler optimizations. The three compilers, which are each written in a different programming language, are designed to provide a relatively small and simple "sandbox" for experimenting with fairly realistic optimization problems. So, while the L3 compilers are not serious research tools (like MachSUIF do provide most of what you need to reach the current horizon of contemporary research in compiler design. As such, L3 is an excellent "jumping off" point for anyone needing to gain background for research in the area.

The three L3 compilers are written in Java, Objective CAML, and Standard ML. All three compilers use the standard C preprocessor (`cpp`) to handle macro definitions, include files, etc., and produce target files containing the ASCII text of x86 assembly code. The x86 code is written in the AT&T syntax, for assembly by the GNU assembler (normally accessed via the `gcc` program). Note that the AT&T assembly syntax is different than the standard syntax defined by Intel. In particular, in the AT&T assembly syntax, operands are placed in left-to-right order (this is the *opposite* of the Intel

syntax), and operand length specifications are attached as suffixes to the operands instead of the operator. So, for example, the instruction to copy a 32-bit word from the third stack slot into the `%eax` register is written in GNU syntax like this:

```
movl  8(%ebp), %eax
```

whereas in the Intel syntax it is written as follows:

```
mov   eax, word[ebp+8]
```

See the *L3 Reference* document for more details about the L3 language and how to build and use the compilers.

2 Intermediate Representation

The 15-745 course is concerned only with analysis and optimization, and as such we will not describe the front end of the compiler that performs parsing, typechecking, and translation into an intermediate representation (IR). The only exception to this is the description of the top level control of the compiler, which determines exactly what phases get executed and in what order. These things will be described in the next sections. Here, we will focus on the intermediate representations.

The L3 compilers use, mainly, a tree form IR. The IR has constructors for representing statements (representing code to be executed for effect) and expressions (code to be executed for effect and value). In addition to statements and expressions, there are tree forms for basic arithmetic, relational, and logical operators, as well as values, which include words, labels, temps, and strings. (Strings are used only for assembly comments.)

We now describe the basic structure of the tree IR, followed by the implementation-specific details for each of the L3 compilers. We present each statement and expression construct in Backus-Naur Form, along with an informal description of the semantics.

Statement constructors

stm ::= COMMENT string. This has no effect, but generally results in a comment being inserted into the target assembly file. Useful for debugging purposes.

stm ::= MOVE exp$_d$ exp$_s$. Copy the value of **exp**$_s$ into **exp**$_d$.

stm ::= EXP exp. Evaluate **exp** for effect.

stm ::= JUMP label. Transfer control to the statement with label **label**.

stm ::= CJUMP relop exp$_1$ exp$_2$ label$_t$ label$_f$. Compute **exp**$_1$ **relop exp**$_2$. If true, then transfer control to label **label**$_t$, else **label**$_f$.

stm ::= SEQ stm stm. Execute the statements in sequence.

stm ::= LABEL label. Mark this statement as a potential jump destination.

stm ::= NOP. Do nothing.

Expression constructors

exp ::= CONST word. Defines a constant with a given 32-bit value (including boolean constants).

exp ::= NAME label. Defines a label value.

exp ::= TEMP temp. Refers to the value of the pseudo-register, **temp**.

exp ::= BINOP binop exp exp. The value of the specified binary operator, applied to the two operand expressions.

exp ::= MEM exp. Evaluates **exp** as a heap-memory address. In r-value contexts, refers to the contents of the heap-memory cell. In l-value contexts, refers to the address of the cell.

exp ::= CALL symbol exp*. Evaluates the argument expressions and then calls the named function, returning its return value.

exp ::= ESEQ stm exp. Executes the statment for effect, and then evaluates the expression, returning its value.

Version-specific details

Java verion. The IR is implemented in package `edu.cmu.cs.l3.tree`. The IR has two base classes, `IRExpression` and `IRStatement`. In addition to the base IR, there is also an expression-list construct, which can take a number of expressions in sequence, execute them, and return the last value. As a utility class, there is `IRPrint`, which pretty-prints any IR that it is given.

All IR classes, with the exception of `IRPrint`, are visitable, using the visitor interface defined in `edu.cmu.cs.l3.general.Visitable`.

The abstract syntax is translated to the IR and canonicalized in the class `edu.cmu.cs.l3.translate.Translator`.

Objective CAML version. The IR is implemented in the module `Ir`, defined in the OCaml source file, `ir.ml`. In addition to the statement and expression constructors described above, the OCaml version also defines two constructors, **PHI** and **INVARIANT**, though neither are used. The `Ir` module also provides a number of pretty-printing functions, include `pexp` and `pstmt`, which convert expressions and statements into strings.

Abstract syntax trees are translated into IR form by the `Translate` module, and then linearized by the module `Canon`.

Standard ML version. The IR is implemented in module `Tree`, defined in the SML source file, `trans/tree.sml`. The `Tree` module also provides a number of pretty-printing functions, include `pp_exp` and `pp_stmt`, which convert expressions and statements into strings.

Abstract syntax trees are translated into IR form by the `Translate` module, defined in `trans/trans.fun`, and then linearized by the module `Canon`, defined in `flow/canon.sml`.

3 Inserting Optimization Phases

Most of the optimizations in this course will be performed on the IR. However, some optimizations may work best on the tree form of the IR, whereas others may work best on the linearized version that is produced by canonicalization. For some purposes, you may find it desirable to do even further processing to achieve a 3-address form of the IR.

The Standard ML and OCaml versions of the compiler perform canonicalization in a module called `Canon`. So, one can intercept the IR form either before or after `Canon` is used. In the Java version, the relevant class is `translate.Translator`, which creates the Tree IR and then canonicalizes it.

Java version. The `translate.Translator` class contains a method called `translate`, which invokes the front-end operations of parsing and typechecking, then translates the resulting abstract syntax tree into tree IR form, and finally canonicalizes it. Code stored in IR form is organized into *fragments*, of which there are three kinds: string, global, and code. String fragments contain constant strings that will be inserted data segment of the assembly file. Global fragments contain global assembly directives. Finally, a code fragment encapsulates a single procedure and its associated IR.

The `translate.Translator` method can be modified to invoke a new IR optimization phase, either before or after canonicalization.

Objective CAML version. The `Main` module, defined in `main.ml`, contains a function called `main` that determines how the various phases of the compiler are invoked. In `main` you will find an invocation of the `Translate.trans_program` function, resulting in a list of pairs, each pair containing a function name and its tree IR form. This is followed immediately by calls to `Canon.linearize`, which produces linearized IR, `Canon.basicBlocks`, which forms basic blocks, and finally `Canon.traceSchedule`, which arranges the order of blocks and fixes up conditional brances. Note that there are hooks for SSA form, though none of this is implemented in the version of the compiler we have provided to you.

The `Main.main` function can be modified to invoke a new IR optimization phase, at any stage of its processing.

Standard ML version. The `Top` module, defined in `top/top.sml`, contains a function called `main` that determines how the various phases of the compiler are invoked. In `main` you will find an invocation of the `Trans.trans` function, resulting in a list of *fragments*, defined in the module `Frame`. A fragment is either an IR form of a procedure, its compiled assembly code, a constant declaration, or an external label. This is followed immediately by calls to `Semant.linearize`, which produces a linearized IR, and then `Semant.canonicalize`, which forms basic blocks and arranges the order of blocks.

The `Top.main` function can be modified to invoke a new IR optimization phase, at any stage of its processing.

4 Internal Assembly Representation

Some of the key optimizations we will consider take place on the assembly representation of the target code. We will provide documentation on this at a future date.