Lecture Notes on
Intermediate Representation

15-411: Compiler Design
Frank Pfenning

Lecture 10
September 23, 2010

1 Introduction

In this lecture we discuss the "middle end" of the compiler. After the source has been parsed we obtain an abstract syntax tree, on which we carry out various static analyses to see if the program is well-formed. In the *L2* language, this consists of checking that every finite control flow path ends in a return statement, that every variable is initialized before its use along every control flow path, and that `break` and `continue` statements occur only inside loops. In later languages, type-checking will be an important additional task.

After we have constructed and checked the abstract syntax tree, we transform the program through several forms of intermediate representation on the way to abstract symbolic assembly and finally actual x86-64 assembly form. How many intermediate representations and their precise form depends on the context: the complexity and form of the language, to what extent the compiler is engineered to be retargetable to different machine architectures, and what kinds of optimizations are important for the implementation. Some of the most well-understood intermediate forms are intermediate representation trees (IR trees), static single-assignment form (SSA), quads and triples. Quads (that is, three-address instructions) and triples (two-address instructions) are closer to the back end of the compiler and you will probably want to use one of them, maybe both. In this lecture we focus on IR trees.

2 Abstract Syntax Trees

We describe abstract syntax trees in a BNF form (Backus-Naur Form) which was originally designed for describing grammars. An abstract syntax tree is the output of parsing and is formed by removing immaterial information from the parse tree (e.g., tokens that are not important in the tree structure) and transforming into a more canonical form. Here we use BNF to describe the recursive structure of the abstract syntax trees.

$$\begin{array}{lcll}
\text{Expressions} & e & ::= & n \mid x \mid e_1 \oplus e_2 \mid e_1 \oslash e_2 \mid e_1 \texttt{\&\&} e_2 \mid e_1 \texttt{||} e_2 \mid f(e_1, \ldots, e_n) \\
\text{Statements} & s & ::= & \mathsf{assign}(x, e) \mid \mathsf{if}(e, s_1, s_2) \\
 & & & \mid \mathsf{while}(e, s) \mid \mathsf{break} \mid \mathsf{continue} \\
 & & & \mid \mathsf{return}(e) \mid \mathsf{nop} \mid \mathsf{seq}(s_1, s_2)
\end{array}$$

We use n for constants, x for variables, $\oplus$ for effect-free operators, $\oslash$ for potentially effectful operators (such as division, which could raise an exception), `&&` and `||` for logical "and" and "or", respectively. The latter have the meaning as in C, always returning either 0 or 1, and short-circuit evaluation if the left-hand side is false (for `&&`) or true (for `||`).

The break and continue statements must occur inside a while loop and also have the semantics of C: break jumps to the first statement after the current loop, and continue skips the remaining statements in the body of the loop and jumps directly to the testing of the exit condition.

3 IR Trees

In the translation to IR trees we want to achieve several goals. One is to isolate potentially effectful expressions, making their order of execution explicit. This simplifies instruction selection and also means that the remaining pure expressions can be optimized much more effectively. Another goal is to make the control flow explicit in the form of conditional or unconditional branches, which is closer to the assembly language target and allows us to apply standard program analyses based on an explicit control flow graph. The treatment in the textbook achieves this [App98, Chapters 7 and 8] but it does so in a somewhat complicated manner using tree transformations that would not be motivated for our language.

We describe the IR through *pure expressions* p and *commands* c. Programs are just sequences of commands; typically these would be the bodies of function definitions. An empty sequence of commands is denoted by "$\cdot$", and we write $r_1; r_2$ for the concatenation of two sequences of commands.

$$
\begin{array}{llcl}
\text{Pure Expressions} & p & ::= & n \mid x \mid p_1 \oplus p_2 \\
\text{Commands} & c & ::= & x \leftarrow p \\
 & & \mid & x \leftarrow p_1 \oslash p_2 \\
 & & \mid & x \leftarrow f(p_1, \ldots, p_n) \\
 & & \mid & \mathsf{if}\ (p)\ \mathsf{goto}\ l \\
 & & \mid & \mathsf{goto}\ l \\
 & & \mid & l : \\
 & & \mid & \mathsf{return}(p) \\
\text{Programs} & r & ::= & c_1; \ldots; c_n
\end{array}
$$

Pure expressions are a subset of all expressions that do not have any (side)effects. We choose an IR tree representation in which potentially effectful operations and function calls can only appear at the top-level of assignments. The logical operators are no longer present and must be eliminated in the translation in favor of conditionals. These transformations help optimizations and analysis. Function calls only take pure arguments, which guarantees the left-to-right evaluation order prescribed in the C0 language semantics. Since function calls may have effects, we also lift function calls to the command level rather than embedded inside expression evaluation.

4 Translating Expressions

The first idea may be to translate abstract syntax expressions to pure expressions, but this does not quite work because potentially effectful expressions have to be turned into commands, and commands are not permitted inside pure expressions. Returning just a command, or sequence of commands, is also insufficient because we somehow need to refer to the result of the translation as a pure expression so we can use it, for example, in a conditional jump or return command.

A solution is to translate from an expression e to a pair consisting of a sequence of instructions r and a pure expression p. After executing r, the value of p will the value of e (assuming the computation does not abort). We write

$$\mathsf{tr}(e) = \langle \check{e}, \hat{e} \rangle$$

where $\check{e}$ is a sequence of commands r that we need to write down to compute the effects of e and $\hat{e}$ is a pure expression p that we can use to compute the value of e back up. Here are the first three clauses in the definition of $\mathsf{tr}(e)$:

$$\begin{array}{lcl} \mathsf{tr}(n) & = & \langle \cdot, n \rangle \\ \mathsf{tr}(x) & = & \langle \cdot, x \rangle \\ \mathsf{tr}(e_1 \oplus e_2) & = & \langle (\check{e}_1; \check{e}_2), \hat{e}_1 \oplus \hat{e}_2 \rangle \end{array}$$

Constants and variables translate to themselves. If we have a pure operation $e_1 \oplus e_2$ it is possible that the subexpressions have effects, so we concatenate the command sequences for these to expressions $\check{e}_1$ and $\check{e}_2$. Now $\hat{e}_1$ and $\hat{e}_2$ are pure expressions referring to the values of e_1 and e_2, respectively, so we can combine them with a pure operation to get a pure expression representing the result.

We can see that the translation of any pure expression p yields an empty sequence of commands followed by the same pure expression p, that is, $\mathsf{tr}(p) = \langle \cdot, p \rangle$. Effectful operations and function calls require us to introduce some commands and a fresh temporary variable to refer to the value resulting from the operation or call.

$$\begin{array}{lcll} \mathsf{tr}(e_1 \oslash e_2) & = & \langle (\check{e}_1; \check{e}_2; t \leftarrow \hat{e}_1 \oslash \hat{e}_2), t \rangle & (t \text{ new}) \\ \mathsf{tr}(f(e_1, \ldots, e_n)) & = & \langle (\check{e}_1; \ldots; \check{e}_n; t \leftarrow f(\hat{e}_1, \ldots, \hat{e}_n)), t \rangle & (t \text{ new}) \end{array}$$

In this and other cases of the translation we need to make sure that new labels are created as targets for conditional jumps, just like we need to create new temporary variables.

Finally, a possible translation of one of the logical operators; the second one is left as an exercise. Note that $\check{e}_2$ is executed only if $\hat{e}_1$ is true.

$$\begin{array}{lcll} \mathsf{tr}(e_1 \,\&\&\, e_2) & = & \langle (\check{e}_1; & \\ & & \quad \mathsf{if}\ (!\hat{e}_1)\ \mathsf{goto}\ l_1; & \\ & & \quad \check{e}_2; & \\ & & \quad \mathsf{if}\ (!\hat{e}_2)\ \mathsf{goto}\ l_1; & \\ & & \quad t \leftarrow 1; & \\ & & \quad \mathsf{goto}\ l_2; & \\ & & \quad l_1 : ; t \leftarrow 0; & \\ & & \quad l_2 :), & \\ & & t \rangle & (t, l_1, l_2 \text{ new}) \end{array}$$

5 Translating Statements

Translating statements is in some ways simpler, because we only need to return a sequence of instructions. It is slightly more complicated in other ways, because inside loops we need to track the targets for break and continue statements. So the statement translation takes three arguments: the statement to translate, and two optional labels. We elide these labels for simplicity: they are absent on the top-level and passed down in recursive calls and change when entering a while loop. We write $\mathsf{tr}(s) = \check{s}$, where $\check{s}$ is a sequence of commands r.

Assigments and conditionals are simple, given the translation of expression from the previous section.

$$
\begin{array}{rcll}
\mathsf{tr}(\mathsf{assign}(x, e)) & = & \check{e}; & \\
 & & x \leftarrow \hat{e} & \\
\mathsf{tr}(\mathsf{if}(e, s_1, s_2)) & = & \check{e}; & \\
 & & \mathsf{if}\ (\hat{e})\ \mathsf{goto}\ l_1; & \\
 & & \check{s}_2; & \\
 & & \mathsf{goto}\ l_2; & \\
 & & l_1 : ; \check{s}_1; & \\
 & & l_2 : & (l_1, l_2 \text{ new})
\end{array}
$$

There are several plausible translation for a while loop. Due to the way many instantiations of the target architecture do branch prediction, it is efficient for the conditional branch in the loop to go backwards.

$$
\begin{array}{rcl}
\mathsf{tr}(\mathsf{while}(e, s)) & = & \mathsf{goto}\ l_2; \\
 & & l_1 : ; \check{s}; \\
 & & l_2 : ; \check{e}; \\
 & & \mathsf{if}\ (\hat{e})\ \mathsf{goto}\ l_1; \\
 & & l_3 :
\end{array}
$$

During the recursive translation of s, the break label is set to l_3 and the continue label is set to l_2.

The jump into the middle of the loop can interfere with optimizations, so it is often beneficial to replicate the test before loop entry, especially in the common case when the code for $\check{e}$ and $\hat{e}$ is small.

$$
\begin{array}{rcl}
\mathsf{tr}(\mathsf{while}(e, s)) & = & \check{e}; \\
 & & \mathsf{if}\ (!\hat{e})\ \mathsf{goto}\ l_3; \\
 & & l_1 : ; \check{s}; \\
 & & l_2 : ; \check{e}; \\
 & & \mathsf{if}\ (\hat{e})\ \mathsf{goto}\ l_1; \\
 & & l_3 :
\end{array}
$$

The remaining cases are simple: break and continue statements jump unconditionally to their corresponding labels; the cases for return, nop and seq are below.

$$\begin{array}{lcl}
\mathsf{tr}(\mathsf{return}(e)) & = & \check{e};\\
 & & \mathsf{return}(\hat{e})\\
\mathsf{tr}(\mathsf{nop}) & = & \cdot\\
\mathsf{tr}(\mathsf{seq}(s_1, s_2)) & = & \check{s}_1;\\
 & & \check{s}_2
\end{array}$$

6 Ambiguity in Language Specification

The C standard explicity leaves the order of evaluation of expressions unspecified [KR88, p. 200]:

> *The precedence and associativity of operators is fully specified, but the order of evaluation of expressions is, with certain exceptions, undefined, even if the subexpressions involve side effects.*

At first, this may seem like a virtue: by leaving evaluation order unspecified, the compiler can freely optimize expressions without running afoul the specification. The flip side of this coin is that programs are almost by definition not portable. They may check and execute just fine with a certain compiler, but subtly or catastrophically break when a compiler is updated, or the program is compiled with a different compiler.

A possible reply to this argument is that a program whose proper execution depends on the order of evaluation is simply wrong, and the programmer should not be surprised if it breaks. The flaw in this argument is that dependence on evaluation order may be a very subtle property, and neither language definition nor compiler give much help in identifying such flaws in a program. No amount of testing with a single compiler can uncover such problems, because often the code *will* execute correctly under the decision made for this compiler. It may even be that all available compilers at the time the code is written may agree, say, evaluating expressions from left to right, but the code could break in a future version.

Therefore I strongly believe that language specifications should be entirely unambiguous. In this course, this is also important because we want to hold all compilers to the same standard of correctness. This is also why the behavior of division by 0 and division overflow, namely an exception, is fully specified. It is not acceptable for an expression such as `(1/0)*0` to be "optimized" to `0`. Instead, it must raise an exception.

The translation to intermediate code presented here therefore must make sure that any ;̆potentially effectful expressions are indeed evaluated from left to right. Careful inspection of the translation will reveal this to be the case. On the resulting pure expressions, many valid optimizations can still be applied which would otherwise be impossible, such as commutativity, associativity, or distributivity, all of which hold for modular arithmetic.

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.