Lecture Notes on Intermediate Representation

15-411: Compiler Design
Frank Pfenning*

Lecture 10
September 29, 2016

1 Introduction

In this lecture we discuss the "middle end" of the compiler. After the source has been parsed and elaborated 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 type-checking, 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. For more specific information you may refer to the Lab 2 specification.

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 Elaboration

The language that we originally parse, because it is derived from C and designed to be compatible with C whenever possible, has many complications, quirks, and

*with contributions by André Platzer, Rob Simmons, and Jan Hoffmann

forms of syntactic sugar. One examples is the `for` statement, which is easier to describe in terms of its components - an optional initialization, a loop guard, and loop body, and an "afterthought" that is run after the loop body. As a starting example, this `for` loop could be elaborated into a `while` loop:

```
// Before elaboration
for (int x = 4; x < 30; x++) { y = y + x }

// After elaboration
{
  int x = 4;
  while (x < 30) { y = y + x; x += 1 }
}
```

The extra scope is necessary: if the next statement after this `for` loop is a declaration of `x`, we have to add the scope or we'd turn the well-formed program into a program that declares `x` twice. This brings up another use of elaboration: making the scope of variables more explicit. Consider this example:

```
int i = 4;
int j = i + 2;
return i + j;
```

To make it clearer that the declaration of `j` is in the scope of the declaration of `i`, we could require all variable declarations to stand on their own at the beginning of a basic block:

```
{ int i;
  i = 4;
  { int j;
    j = i + 2;
    return i + j;
  }
}
```

This is getting awkward, though: the surface structure of C-like languages just doesn't support make the scoping of individual declarations clear. Rather than continuing to perform manipulations on surface syntax, we introduce the BNF of an *elaborated* abstract syntax. Rather than being tied to the structure of the language's context-free grammar, the elaborated language tries to capture the meaning of the language constructs. In particular, $\mathsf{declare}(x, \tau, s)$ means that the variable x is declared (only) within the statement s.

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

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

Given our full elaborated language, we can say that a program like this one, which has a `for` loop:

```
for (int x = 4; x < 30; x++) { y = y + x; }
```

should be elaborated to this abstract syntax:

$$\begin{array}{l}\mathsf{declare}(x, \mathsf{int}, \mathsf{seq}(\mathsf{assign}(x, 4),\\ \qquad\qquad \mathsf{while}(x < 30,\\ \qquad\qquad\quad \mathsf{seq}(\mathsf{assign}(y, y + x), \mathsf{assign}(x, x + 1)))))\end{array}$$

A complete description of elaboration can be given in the form of inference rules. When we turn the `surface syntax` given by the parser into elaborated abstract syntax, we write `surface` $\rightsquigarrow$ elaborated. Here is a rule for elaborating a `for` statement where the initialization includes a declaration directly into our elaborated abstract syntax tree.

$$\frac{\begin{array}{l}\texttt{<tp>} \rightsquigarrow \tau\\ \texttt{<ident>} \rightsquigarrow x\\ \texttt{<exp1>} \rightsquigarrow e_1\\ \texttt{<exp2>} \rightsquigarrow e_2\\ \texttt{<stmt1>} \rightsquigarrow s_1\\ \texttt{<stmt2>} \rightsquigarrow s_2\end{array}}{\begin{array}{l}\texttt{for (<tp> <ident> = <exp1>; <exp2>; <stmt1>) <stmt2>}\\ \qquad \rightsquigarrow \mathsf{declare}(x, \tau, \mathsf{while}(e_1, \mathsf{seq}(s_2, s_1)))\end{array}}$$

We read this rule as being one case of a large case analysis, with one branch for every pattern in the surface syntax. When we want to translate a `for` loop that matches the pattern at the bottom, we have to do six things: elaborate the type, elaborate the identifier, elaborate both expressions, elaborate the afterthought, and then elaborate the loop body. Depending on how you set your compiler up, the first four operations might be trivial: you might not have any elaboration to do on expressions, types, or identifiers. However, it will be necessary to make a recursive call to the elaboration procedure in our running example, because we have to elaborate `x++` $\rightsquigarrow \mathsf{assign}(x, x + 1)$. We will also, in general, have to make sure any nested `for` loops in `<stmt2>`, the body of our `for` loop, are elaborated.

In summary, after we parse our program into a surface (or concrete) representation, we frequently want to transform that syntax into a abstract (or elaborated)

representation. The elaborated abstract syntax trees, our first intermediate language, makes it easier to do typechecking and later transformations. Especially if we do type checking on the elaborated syntax, doing *too much* elaboration can be a problem: it may make our error messages less readable. However, some elaboration is an important part of most languages.

Both the surface and elaboration representations may be referred to as *abstract syntax trees*. In the remainder of this lecture and course, we will ignore the surface representations and refer to the elaborated form as the abstract syntax tree, or AST.

3 IR Trees

Our next translation will be to a new intermediate representation form called *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 r 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.

Pure Expressions	p	$::=$	$n \mid x \mid p_1 \oplus p_2$
Commands	c	$::=$	$x \leftarrow p$
		$\mid$	$x \leftarrow p_1 \oslash p_2$
		$\mid$	$x \leftarrow f(p_1, \ldots, p_n)$
		$\mid$	$\mathsf{if}\ (p_1\ ?\ p_2)\ \mathsf{then}\ l_t\ \mathsf{else}\ l_f$
		$\mid$	$\mathsf{goto}\ l$
		$\mid$	$l:$
		$\mid$	$\mathsf{return}(p)$
Programs	r	$::=$	$c_1 \ ; \ \ldots \ ; \ c_n$

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 embedding them 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 commands 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 sub-expressions 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.

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

We postpone the translation of boolean expressions e_1 ? e_2, !e, e_1 && e_2 and e_1 || e_2 to Section 6.

5 Translating Statements

Translating statements is in some ways simpler, because we only need to return a sequence of commands. It is slightly more complicated in other ways, since we have to manage control flow via jumps and conditional branches. We write $\mathsf{tr}(s) = \check{s}$, where $\check{s}$ is a sequence of commands r.

Assignments and conditionals are simple, given the translation of expression from the previous section, as are return, nop and seq.

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

Conditionals require labels and jumps. Below is a first attempt. We combine labels with the following statement (where there is one) to make it easier to read.

$$\begin{array}{lcll} \mathsf{tr}(\mathsf{if}(e,s_1,s_2)) & = & & \check{e}\ ; \\ & & & \mathsf{if}\ (\hat{e}\ \mathsf{!=}\ 0)\ \mathsf{then}\ l_1\ \mathsf{else}\ l_2\ ; \\ & & l_1 \ : & \check{s}_1\ ; \\ & & & \mathsf{goto}\ l_3\ ; \\ & & l_2 \ : & \check{s}_2\ ; \\ & & l_3 \ : & & (l_1, l_2, l_3 \text{ fresh}) \end{array}$$

We can unify the presentation a bit more by inserting a redundant jump and combining a few commands involving control on the same line.

$$\begin{array}{lcll} \mathsf{tr}(\mathsf{if}(e,s_1,s_2)) & = & & \check{e}\ ; \\ & & & \mathsf{if}\ (\hat{e}\ \mathsf{!=}\ 0)\ \mathsf{then}\ l_1\ \mathsf{else}\ l_2\ ; \\ & & l_1 \ : & \check{s}_1\ ;\ \mathsf{goto}\ l_3\ ; \\ & & l_2 \ : & \check{s}_2\ ;\ \mathsf{goto}\ l_3\ ; \\ & & l_3 \ : & & (l_1, l_2, l_3 \text{ fresh}) \end{array}$$

This modification also puts the code into *basic blocks*, allowing us to divide our code into segments that begin with labels and end with a conditional or unconditional jump (or a return).

The remaining awkwardness in this code comes from having to compute e to a boolean value and then checking this against 0. While this is correct, it does not lead to particularly efficient machine code. We will present an improved translation in the next section.

Here is a similarly straightforward translation for while.

$$\begin{array}{lcll} \mathsf{tr}(\mathsf{while}(e,s)) & = & l_1 \ : & \check{e};\ \mathsf{if}\ (\hat{e}\ \mathsf{!=}\ 0)\ \mathsf{then}\ l_2\ \mathsf{else}\ l_3\ ; \\ & & l_2 \ : & \check{s}\ ;\ \mathsf{goto}\ l_1; \\ & & l_3 \ : & & (l_1, l_2, l_3 \text{ fresh}) \end{array}$$

6 Translating Boolean Expressions

As indicated above, the code with the translations above does not take advantage of the way conditional branches work in x86 and x86-64, where we can compare two values and then branch based on the outcome of the comparison by testing condition flags. So we may look for ways to translation conditionals ($\mathsf{if}(e, s_1, s_2)$) and loops ($\mathsf{while}(e, s)$) into simpler code.

One insight is that we use booleans mostly so we can branch on them. So we define a new function

$$\mathsf{cp}(b, l, l') = r$$

where b is a boolean expression. The resulting command sequence r should jump to l if b is true and jump to l' if b is false. Boolean expressions here are comparisons, negation, logical *and*, and logical *or*. They can also be function calls returning booleans or constants 0 for false and 1 for true.

We define

$$\begin{array}{lcl}
\mathsf{cp}(e_1 \mathsf{\ ?\ } e_2, l, l') & = & \check{e}_1 \mathsf{\ ;\ } \check{e}_2 \mathsf{\ ;} \\
 & & \mathsf{if\ } (\hat{e}_1 \mathsf{\ ?\ } \hat{e}_2) \mathsf{\ then\ } l \mathsf{\ else\ } l' \\
\mathsf{cp}(\mathsf{!}e, l, l') & = & \mathsf{cp}(e, l', l) \\
\mathsf{cp}(e_1 \mathsf{\ \&\&\ } e_2, l, l') & = & \mathsf{cp}(e_1, l_2, l') \mathsf{\ ;} \\
 & = & l_2 \ : \ \mathsf{cp}(e_2, l, l') \qquad (l_2 \text{ fresh}) \\
\mathsf{cp}(e_1 \mathsf{\ ||\ } e_2, l, l') & = & \textit{left to the reader} \\
 & & \\
\mathsf{cp}(0, l, l') & = & \mathsf{goto\ } l' \\
\mathsf{cp}(1, l, l') & = & \mathsf{goto\ } l \\
\mathsf{cp}(e, l, l') & = & \check{e} \mathsf{\ ;} \\
 & & \mathsf{if\ } (\hat{e} \mathsf{\ !=\ } 0) \mathsf{\ then\ } l \mathsf{\ else\ } l'
\end{array}$$

The last form should only be necessary if $e = x$ or $e = f(e_1, \ldots, f_n)$.

This is then used in the translation of statements in a straightforward way

$$\begin{array}{lcl}
\mathsf{tr}(\mathsf{if}(b, s_1, s_2)) & = & \mathsf{cp}(b, l_1, l_2) \\
 & & l_1 \ : \ \mathsf{tr}(s_1) \mathsf{\ ;\ goto\ } l_3 \\
 & & l_2 \ : \ \mathsf{tr}(s_2) \mathsf{\ ;\ goto\ } l_3 \\
 & & l_3 \ : \qquad (l_1, l_2, l_3 \text{ fresh})
\end{array}$$

We leave while loops using the cp translation to the reader.

We still have to define how to compile an expression that happens to be boolean (for example, as part of return statement).

$$\begin{array}{lcl}
\mathsf{tr}(e) & = \langle & \mathsf{cp}(e, l_1, l_2) \mathsf{\ ;} \\
 & & l_1 \ : \ t \leftarrow 1 \mathsf{\ ;\ goto\ } l_3 \\
 & & l_2 \ : \ t \leftarrow 0 \mathsf{\ ;\ goto\ } l_3 \\
 & & l_3 \ : \\
 & , \ t \ \rangle & \qquad (l_1, l_2, l_3, t \text{ fresh})
\end{array}$$

7 Extended Basic Blocks

As discussed, the translation we've discussed so far translates code into a *basic block* form. One of the benefits of basic blocks is that they can be treated as a single independent unit in many analyses. Constant propagation is an obvious example: a definition at the beginning of a basic block will necessarily be the only reaching definition for the remainder of that block. (Unless, of course, another definition of the same temp is reached, and *that* definition is the only one that reaches further.)

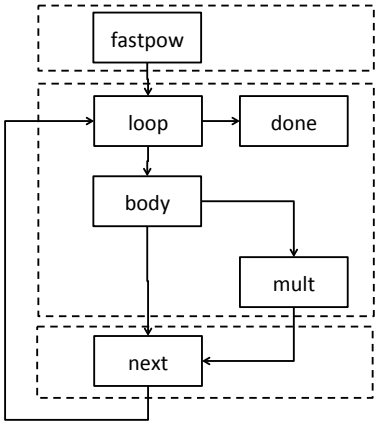
However, the transformation to basic blocks sometimes results in us having a large number of labels and jumps, which can obscure the structure of the program to some degree:

```
int fastpow(int b, int e)
//@requires e >= 0;
{
  int r = 1;
  while (e > 0)
    //@loop_invariant e >= 0;
    // r * b^e remains invariant
    {
      if (e % 2 != 0)
        r = r * b;
      b = b * b;
      e = e / 2;
    }
  return r;
}
```

```
fastpow(b,e):
  r <- 1
  goto loop
loop:
  if (e > 0) then body else done
body:
  t0 <- e % 2
  if (t0 == 0) then mult else next
mult:
  r <- r * b
  goto next
next:
  b <- b * b
  e <- e / 2
  goto loop
done:
  return r
```

An *extended basic block* is a collection of basic blocks with one label at the beginning (that may be the target of multiple jumps) and internal labels, each of which is the target of only one internal jump and no external jumps. In the example above, we could observe that `loop` and `next` are the only blocks which are the target of multiple jumps. Therefore, there are three extended basic blocks: the function's entry is one extended basic block, the four blocks beginning with `loop`, `body`, `mult`, and `done` comprise another basic block, and `next` is the third extended basic block.

This extended basic block structure can be seen in the control flow graph of the program, a graph with one node for every label and an edge for every possible jump. The extended basic blocks are depicted with dashed lines in this control flow graph:



Another way of thinking about extended basic blocks as yet another intermediate language, where instead of basic blocks being sequences of commands ending in a jump, they are *trees* of commands that branch at conditional statements and have unconditional jumps (`goto` or `return`) as their leaves.

```
fastpow(b,e):
  r <- 1
  goto loop
loop:
  if (e > 0)
  then
    t <- e % 2       // loop
    if (t0 == 0)
       r <- r * b    // mult
       goto next
    else
       goto next
  else
    return r         // done
next:
  b <- b * b
  e <- e /2
  goto loop
```

Representing extended basic blocks in this form recovers some of the original program's branching structure while still allowing optimizations like constant propagation to proceed eagerly within a basic block.

8 Ambiguity in Language Specification

The C standard explicitly 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.

9 Translating C0 to C

At this point in time, the `cc0` compiler for C0 performs lexing, parsing, and static semantic checks and then generates corresponding C code. This translation has to take care of protecting the C0 code against the undefined or unspecified behavior

of certain expressions in C. We list here some of them and the compiler's approach to accommodating them.

- Undefined behavior of certain divisions, shifts, and memory accesses. These are handled by protecting the corresponding operations in C with tests and reliably raising the required exceptions.
- Undefined behavior of overflow of signed integer arithmetic. This is currently handled using the `-fwrapv` flag for `gcc` which requires two's complement integer arithmetic for signed quantities. It was previously handled by declaring C variables as unsigned (for which modular arithmetic is specified) and casting them to corresponding signed quantities before comparisons.
- Unspecified evaluation order. This is handled by a similar translation as shown this lecture, isolating potentially effectful expressions in sequences of assignment statements. This fixes evaluation order since evaluation order of a sequence statements is guaranteed in C even if it is not for expressions.
- Unspecified size of `int` and related integral types. This is currently handled by checking, before invoking the generated binary, that `int` does indeed have 32 bits. At a previous point in time it was handled more portably by translating C0's `int` type to C's `int32_t`.

Questions

1. In the section on abstract syntax trees it looks like we have defined a language instead of an abstract syntax tree. Why not just have people program directly in the elaborated language? What can be represented in the surface representation but not the elaborated AST? What can be represented in the elaborated AST but not the surface syntax?
2. Which choices of $i, j, k, l \in \{1, 2\}$ make the following translation valid?

$$\mathsf{tr}(e_1 + e_2) = \langle (\check{e}_i; \check{e}_j), \hat{e}_k + \hat{e}_l \rangle$$

3. You can make your translation more uniform by requiring all translations to put their results into temp variables using commands, as we did in the lecture on instruction selection. Discuss the difference.
4. Extend the elaboration to hand `break` and `continue` for while loops under their C semantics.
5. Does each basic block in the intermediate representation for C0 have at most 2 predecessors?

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.