15 Building a Runnable Program

15.2.1 GCC and LLVM

Traditionally, all machine-independent code improvement in `gcc` was based on RTL. Over time it became clear that the IF had become an obstacle to further improvements in the compiler, and that a higher-level form was needed. GIMPLE was introduced to meet that need. Since `gcc` v.4.9 (2014), GENERIC has been used for semantic analysis and, in a few cases, for certain language-specific code improvement. As its final task, each front end converts the program from GENERIC into GIMPLE. Depending on the requested level of code improvement, the "middle end" may perform over 140 phases of code improvement and transformation on the GIMPLE representation, after which it converts to RTL and performs as many as 70 additional phases before handing the result to the back end for target code generation.

Both GIMPLE and RTL are meant to be kept in memory across compiler phases, rather than being written to a file. Both IFs have a human-readable external format, which the compiler can write and (partially) read, but this format is not needed by the compiler: the internal version is much better suited for automatic manipulation.

GIMPLE

EXAMPLE 15.19
GCD program in GIMPLE

The GIMPLE code generated by a `gcc` front end is essentially a distillation of GENERIC, with many of the most complex (and often language-specific) features "lowered" into a smaller, common set of tree node types. As a simple example, consider the `gcd` program of Example 1.20:

```
int main () {
    int i = getint();
    int j = getint();
    while (i != j) {
        if (i > j) i = i - j;
        else j = j - i;
    }
    putint(i);
}
```

Figure C-15.11 illustrates the "high GIMPLE" produced by `gcc`'s C front end when given this program as input. If we compare this GIMPLE code to Figure 15.2, which loosely[1] resembles GENERIC, we see at least two significant differences. First, all of the nodes that comprise a subroutine appear on a single list, with control flow represented by explicit `gotos` and by `true` and `false` branches for conditions. Second, both conditions and assignments have been designed to capture an embedded binary expression, allowing us in many cases to collapse a small subtree into a GIMPLE single node.

Over the course of its many phases, the `gcc` middle end will make many additional changes to this code, not only to improve its quality but also to further lower its level of abstraction. This "flattening" of the tree makes it easier to translate into RTL.

Perhaps the most significant transformation of GIMPLE is the conversion to *static single assignment (SSA) form*. As noted in the main text and explored more fully in Section C-17.4.1, SSA conversion facilitates subsequent code transformations by introducing extra variable names into the program in such a way that nothing is ever written in more than one place.

RTL

RTL is loosely based on the S-expressions of Lisp. Each RTL expression consists of an operator or expression type and a sequence of operands. In its external form, these are represented by a parenthesized list in which the element immediately inside the left parenthesis is the operator. Each such list is then embedded in a wrapper that points to predecessor and successor expressions in linear order. Internally, RTL expressions are represented by C structs and pointers. This pointer-rich structure constitutes the interface among the compiler's many back-end phases. There are several dozen expression types, including constants, references to values in memory or registers, arithmetic and logical operations, comparisons, bit-field manipulations, type conversions, and stores to memory or registers.

The body of a subroutine consists of a sequence of RTL expressions. Each expression in the sequence is called an `insn` (instruction). Each `insn` begins with one of six special codes:

insn: an "ordinary" RTL expression.

jump_insn: an expression that may transfer control to a label.

call_insn: an expression that may make a subroutine call.

code_label: a possible target of a jump.

barrier: an indication that the previous `insn` always jumps away. Control will never "fall through" to here.

---

1 Unlike the informal notation of Figure 15.2, GENERIC and GIMPLE make no distinction between syntax tree nodes and symbol table nodes. In effect, the symbol table is merged into the syntax tree.

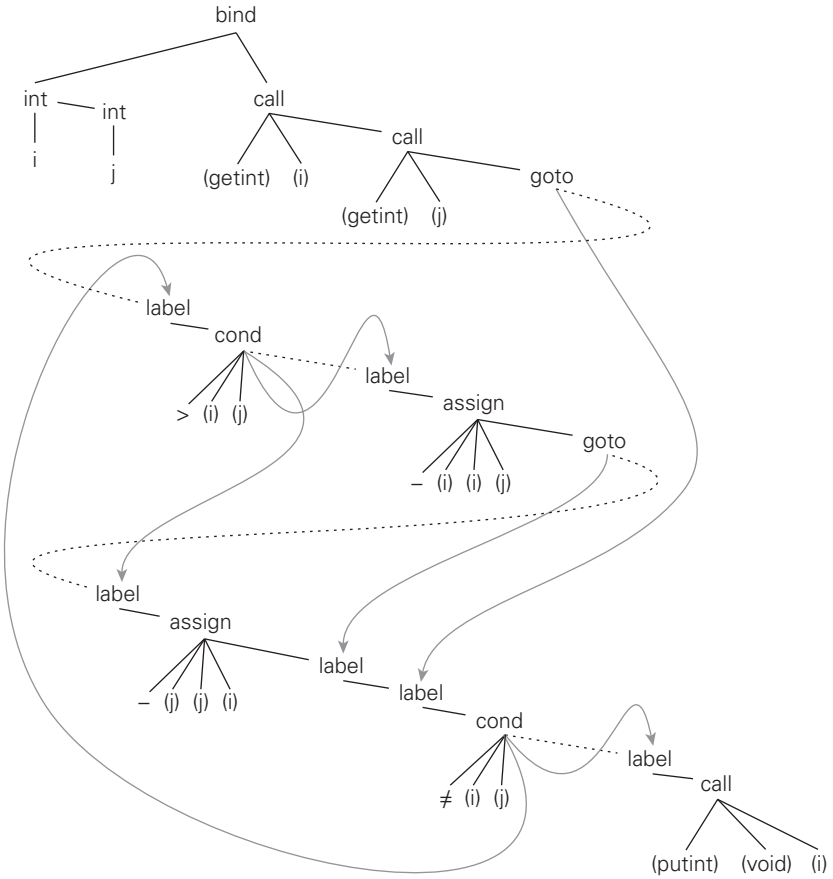

Figure 15.11 Simplified GIMPLE for the `gcd` program. The left child of the `bind` node holds local symbol table information; references to this information—and to global functions getint and putint—are indicated in the rest of the figure with parenthesized names. The first child of a call node names the function, the second the place to assign the return value, and the rest the arguments. An assign node with children ⟨op, a, b, c⟩ represents the assignment a := b op c. In each condition node, the first three children are a comparison operator and its operands; the last two are pointers to the subtrees for the outcomes true and false.

note: a pure annotation. There are nine different kinds of these, to identify the tops and bottoms of loops, scopes, subroutines, and so on.

The sequence is not always completely linear; `insns` are sometimes collected into pairs or triples that correspond to target machine instructions with delay slots. Over a dozen different kinds of (non-*note*) annotations can be attached to an individual `insn`, to identify side effects, specify target machine instructions or registers, keep track of the points at which values are defined and used, automatically increment

or decrement registers that are used to iterate over an array, and so on. `Insns` may also refer to various dynamically allocated structures, including the symbol table.

EXAMPLE 15.20
An RTL `insn` sequence

A simplified `insn` sequence for the code of Example C-15.19 appears in Figure C-15.12. The three leading numbers in each `insn` represent the `insn`'s unique id and those of its predecessor and successor, respectively. The fourth, when present, identifies the `insn`'s basic block. Fields for the various `insn` annotations are not shown. The `:SI` *mode specifier* on a memory or register reference indicates access to a single (4-byte) integer; `:DI` and `:QI` modes correspond to double (8-byte) and quarter (1-byte) integers.

A full explanation of the RTL notation is beyond what we can cover here. As an example, however, `insn` 26 loads the memory location found 4 bytes back from the frame pointer (namely, `i`) into virtual register 64. The following `insn`, 27, sets the memory location found 8 bytes back from the frame pointer (namely, `j`) to the result of subtracting register 64 from that same memory location. In parallel (as a side effect), `insn` 27 also "clobbers" (overwrites) the contents of virtual condition code register 17. ■

In order to generate target code, the back end matches `insns` against patterns stored in a semiformal description of the target machine. Both this description and the routines that manipulate the machine-dependent parts of an `insn` are segregated into a relatively small number of separately compiled files. As a result, much of the compiler back end is machine independent, and need not actually be modified when porting to a new machine.

`Clang` AST format

EXAMPLE 15.21
GCD program as a `clang` AST

As noted in the main text, the `clang` front end for LLVM employs a fairly conventional high-level AST format, not unlike `gcc`'s GENERIC. Figure C-15.13 shows a simplified version of the tree for our `gcd` program. The resemblance to Figure 15.2 is immediately apparent. Unlike the GIMPLE code of Figure reffig-high-gimple, with its explicit `gotos`, the `clang` AST nodes encode high-level control structures explicitly. Each CompoundStmt node has one child for every statement in the block; a CallExpr node has one child for the function to be called and one for each argument. Symbol table information is implicit in the declaration nodes of the tree—in this case, FunctionDecl and VarDecl. The cinit annotation on a VarDecl node indicates the presence of a child node to specify the initial value. ■

LLVM IR

LLVM's IR is central to the design of the compiler suite; it has been carefully crafted to represent programs in almost any language, to be easily mapped to almost any modern processor, and to facilitate the full range of modern code improvement techniques. As explained in system documentation (at *llvm.org*), it can be represented, equivalently, as data structures in memory, as compact binary "bitcode," or as human-readable pseudo-assembly notation. Programs are normally passed from one compiler phase to the next as in-memory data structures, but they can also be exported to—or imported from—bitcode files. The pseudo-assembly notation is mainly intended for compiler debugging and development.

```
(insn 5 2 6 2 (set (reg:QI 0 ax) (const_int 0)))
(call_insn 6 5 7 2 (set (reg:SI 0 ax) (call (mem:QI (symbol_ref:DI ("getint"))) (const_int 0))))
(insn 7 6 8 2 (set (reg:SI 60) (reg:SI 0 ax)))
(insn 8 7 9 2 (set (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -4))) (reg:SI 60)))
(insn 9 8 10 2 (set (reg:QI 0 ax) (const_int 0)))
(call_insn 10 9 11 2 (set (reg:SI 0 ax) (call (mem:QI (symbol_ref:DI ("getint"))) (const_int 0))))
(insn 11 10 12 2 (set (reg:SI 61) (reg:SI 0 ax)))
(insn 12 11 13 2 (set (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -8))) (reg:SI 61)))
(jump_insn 13 12 14 2 (set (pc) (label_ref 28)))
(barrier 14 13 30)
(code_label 30 14 15 4 4 "")
(insn 16 15 17 4 (set (reg:SI 62) (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -4)))))
(insn 17 16 18 4 (set (reg:CCGC 17)
                (compare:CCGC (reg:SI 62) (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -8))))))
(jump_insn 18 17 19 4 (set (pc) (if_then_else (le (reg:CCGC 17) (const_int 0)) (label_ref 24) (pc))))
(insn 20 19 21 5 (set (reg:SI 63) (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -8)))))
(insn 21 20 22 5 (parallel [
    (set (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -4)))
         (minus:SI (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -4))) (reg:SI 63)))
    (clobber (reg:CC 17))
]))
(jump_insn 22 21 23 5 (set (pc) (label_ref 28)))
(barrier 23 22 24)
(code_label 24 23 25 6 3 "")
(insn 26 25 27 6 (set (reg:SI 64) (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -4)))))
(insn 27 26 28 6 (parallel [
    (set (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -8)))
         (minus:SI (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -8))) (reg:SI 64)))
    (clobber (reg:CC 17))
]))
(code_label 28 27 29 7 2 "")
(insn 31 29 32 7 (set (reg:SI 65) (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -4)))))
(insn 32 31 33 7 (set (reg:CCZ 17)
                    (compare:CCZ (reg:SI 65) (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -8))))))
(jump_insn 33 32 34 7 (set (pc) (if_then_else (ne (reg:CCZ 17) (const_int 0)) (label_ref 30) (pc))))
(insn 35 34 36 8 (set (reg:SI 66) (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -4)))))
(insn 36 35 37 8 (set (reg:SI 5 di) (reg:SI 66)))
(call_insn 37 36 40 8 (call (mem:QI (symbol_ref:DI ("putint"))) (const_int 0)))
(insn 40 37 41 8 (clobber (reg/i:SI 0 ax)))
(insn 41 40 39 8 (clobber (reg:SI 59 [ <retval> ])))
(insn 39 41 42 8 (set (reg/i:SI 0 ax) (reg:SI 59 [ <retval> ])))
(insn 42 39 0 8 (use (reg/i:SI 0 ax)))
```

Figure 15.12 Simplified textual RTL for the `gcd` program. Most annotations (more than half the original length) have been elided here. Register 54 is the frame pointer. Local variable `i` is at offset −4. Local variable `j` is at offset −8.

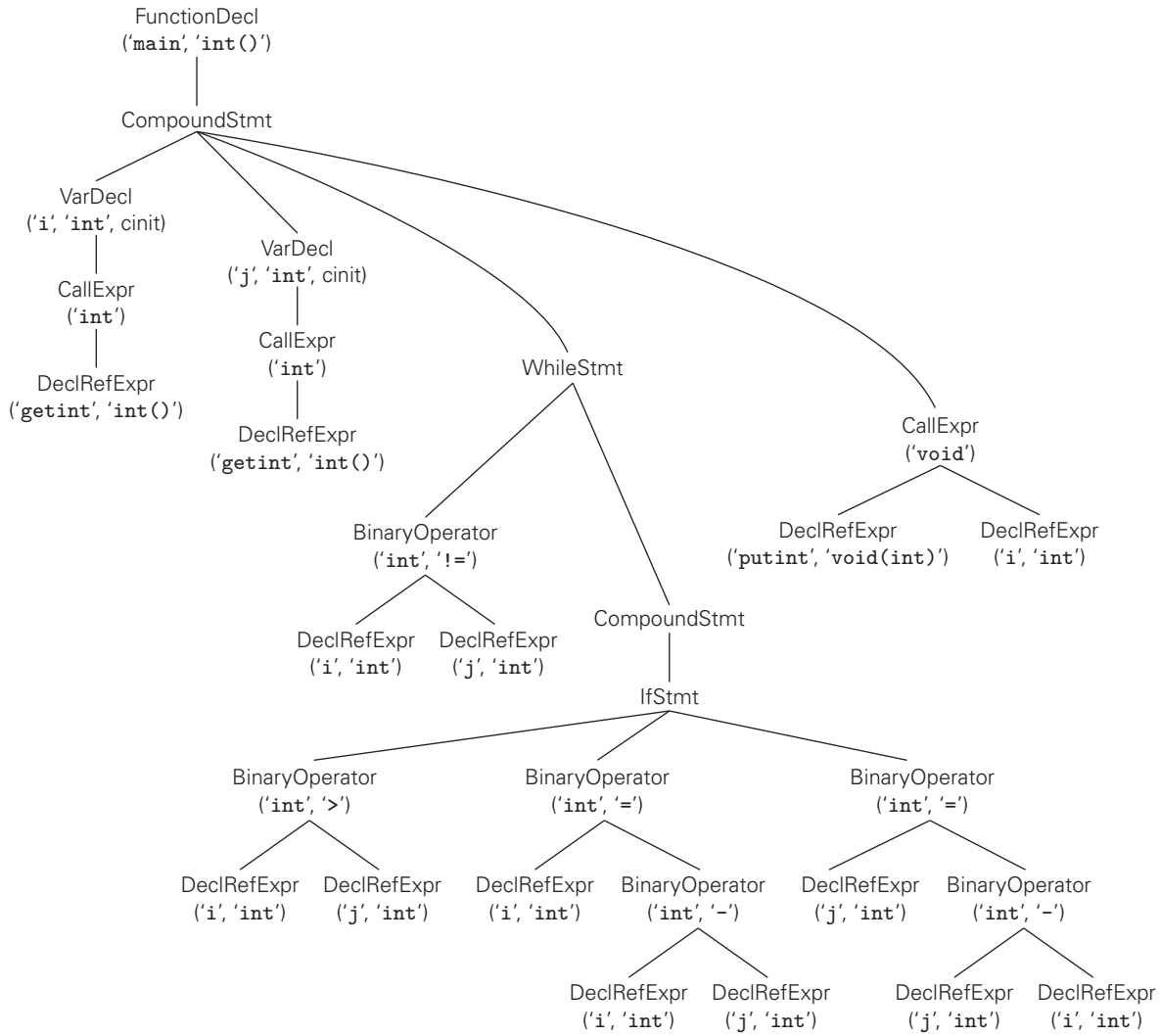

Figure 15.13 Simplified `clang` AST for the `gcd` program. Nodes that implicitly dereference variables (lvalues) to obtain their contents (rvalues) have been elided.

EXAMPLE 15.22

LLVM IR for the GCD program.

LLVM IR for our `gcd` program appears in Figure C-15.14. Unlike the RTL of Figure C-15.12, which was generated at the low, default level of optimization, the code here was generated with a `-O3` command line switch. As a result, variables `i` and `j` are kept in (virtual) registers throughout the computation, rather than being repeatedly read from and written to memory. Because they remain in memory, SSA form requires that we assign the proper incoming values into new virtual registers whenever control paths merge. Specifically, at the top of the loop (label `4`), a `phi` instruction assigns virtual register `%5` (the current location of `j`) the value from

```
define i32 @main() local_unnamed_addr #0 {
  %1 = tail call i32 (...) @getint() #2
  %2 = tail call i32 (...) @getint() #2
  %3 = icmp eq i32 %1, %2
  br i1 %3, label %13, label %4
4:                                ; preds = %0, %4
  %5 = phi i32 [ %11, %4 ], [ %2, %0 ]
  %6 = phi i32 [ %9, %4 ], [ %1, %0 ]
  %7 = icmp slt i32 %5, %6
  %8 = select i1 %7, i32 %5, i32 0
  %9 = sub nsw i32 %6, %8
  %10 = select i1 %7, i32 0, i32 %6
  %11 = sub nsw i32 %5, %10
  %12 = icmp eq i32 %9, %11
  br i1 %12, label %13, label %4, !llvm.loop !6
13:                               ; preds = %4, %0
  %14 = phi i32 [ %1, %0 ], [ %9, %4 ]
  tail call void @putint(i32 noundef %14) #2
  ret i32 0
}
```

Figure 15.14 Textual LLVM IR for function `main` in the `gcd` program, generated at optimization level `-O3`. Comments begin with a semicolon; metadata begins with an exclamation point. (Function calls are labeled `tail` not because they are actually tail recursive, but because they do not violate any of the rules that would prevent reuse of the stack frame if they *were* tail recursive.)

register %2 if control has entered from the header block, and from register %11 if control has come around from the bottom of the loop. A second `phi` instruction makes a similar choice for `i` at the top of the loop, and a third for `i` at the beginning of the footer. Among other things, the fact that every virtual register has only one assignment in the code means that instructions can safely be reordered so long as operands are always computed before they are used. We never have to worry about "overwriting" a virtual register before its final use, or about writing values to it out of order.

In addition to promoting `i` and `j` to registers, `-O3` optimization causes the compiler to use *predication* (Section C-5.3.2), rather than branching, for the `if`… `then` … `else` in the loop. Specifically, the `icmp` instruction at the top of the loop assigns the outcome of the `if` comparison into virtual register %7. The `select` instructions then use this register to chose whether to place a zero or one of `j` and `i`, respectively, into registers %8 and %10. Finally, the `sub` instructions subtract these values from `i` and `j`, effectively updating one and leaving the other unchanged. ■

✓ CHECK YOUR UNDERSTANDING

24. Characterize GIMPLE, RTL, `clang` AST format, LLVM IR, Java bytecode, and Common Intermediate Language as high-, medium-, or low-level intermediate forms.

25. Name three languages (other than C) for which there exist `gcc` front ends.
26. Name three languages (other than C) for which there exist LLVM front ends but not `gcc` front ends.
27. What is the internal IF of `gcc`'s front ends?
28. Give brief descriptions of GIMPLE and RTL. How do they differ? Why was GIMPLE introduced?
29. Compare RTL and LLVM IR. In what ways does the latter more closely resemble typical assembly language?

15 Building a Runnable Program

15.7 Dynamic Linking

To be amenable to dynamic linking, a library must either (1) be located at the same address in every program that uses it, or (2) have no relocatable words in its code segment, so that the content of the segment does not depend on its address. The first approach is straightforward but restrictive: it generally requires that we assign a unique address to every sharable library; otherwise we run the risk that some newly created program will want to use two libraries that have been given overlapping address ranges. In Unix System V R3, which took the unique-address approach, shared libraries could only be installed by the system administrator. This requirement tended to limit the use of dynamic linking to a relatively small number of popular libraries. The second approach, in which a shared library can be linked at any address, requires the generation of *position-independent code*. It allows users to employ dynamic linking whenever they want, without administrator intervention.

The cost of user-managed dynamic linking is that executable programs are no longer self-contained. They depend for correct execution on the availability of appropriate dynamic libraries at execution time. If different programs are built with different expectations of (which versions of) which libraries will be available, conflicts can arise. On Microsoft platforms, where dynamic libraries have names ending in `.dll`, compatibility problems are sometimes referred to as "DLL hell." The frequency and severity of the problem can be minimized with good software engineering practice. In particular, a *package management system* may maintain a database of dependences between programs and libraries, and among the libraries themselves. If installer programs use the database correctly, problems will be detected at install time, when they can reasonably be addressed, rather than at the arbitrarily delayed point at which a program first attempts to use an incompatible or missing library.

15.7.1 Position-Independent Code

A code segment that contains no relocatable words is said to constitute *position-independent code* (PIC). To generate PIC, the compiler must observe the following rules:

1. Use PC-relative addressing, rather than jumps to absolute addresses, for all internal branches.
2. Similarly, avoid absolute references to statically allocated data, by using displacement addressing with respect to some standard base register. If the code and data segments are guaranteed to lie at a known offset from one another, then the program counter can be used for this purpose. Otherwise, the caller must initialize some other base register as part of the entry point's calling sequence.
3. Use an extra level of indirection for every control transfer out of the PIC segment, and for every load or store of static memory outside the corresponding data segment. The indirection allows the (non-PIC) target address to be kept in the data segment, which is private to each program instance.

EXAMPLE 15.23
PIC under x86/Linux

Exact details vary among processors, vendors, and operating systems. Conventions for `gcc` on recent versions of x86 Linux are illustrated in Figure C-15.15. Each code segment is accompanied by a *linkage table*—known in Linux as the segment's *global offset table* (GOT). This table lists the locations of all code and data whose addresses were not statically determined. All processes that use the same library share a single copy of the library's code segment, but each process has its own copy of the library's GOT. Both the code segment and the GOT can lie at different locations in the address spaces of different processes, but the *offset* between the two must always be the same.

Like the main program, each shared library is typically composed of multiple compilation units, joined together by a *static linker*, which resolves internal references. Resolution of references from the main program into shared libraries—or among the libraries themselves—is delayed until load time or run time, and is the job of the *dynamic linker*. By construction, shared libraries never make references back into the main program.

Libraries are permitted to have (process-private) data as well as code, but the total amount of such data is assumed (in Linux, at least) to be small enough that the data can be statically linked without wasting significant space. Each process therefore has a single data segment (shown in the figure at the lower left), containing the data of the main program and of all the libraries it may call, directly or indirectly. (Extensions to delay the linking of library data are considered in Exercise C-15.14.)

Focusing for the moment on the dashed arrows of the figure (and ignoring the dotted arrows), a read of `X` or `Y` in `main` can use a statically resolved address. A read of `X` or `Y` in `foo` uses PC-relative addressing to find the appropriate slot in `foo`'s GOT, and then loads `X` or `Y` indirectly. Similar indirection is required for subroutine calls into dynamically linked libraries. To avoid duplication of the indirection code, the compiler incorporates a (shared, read-only) `procedure`

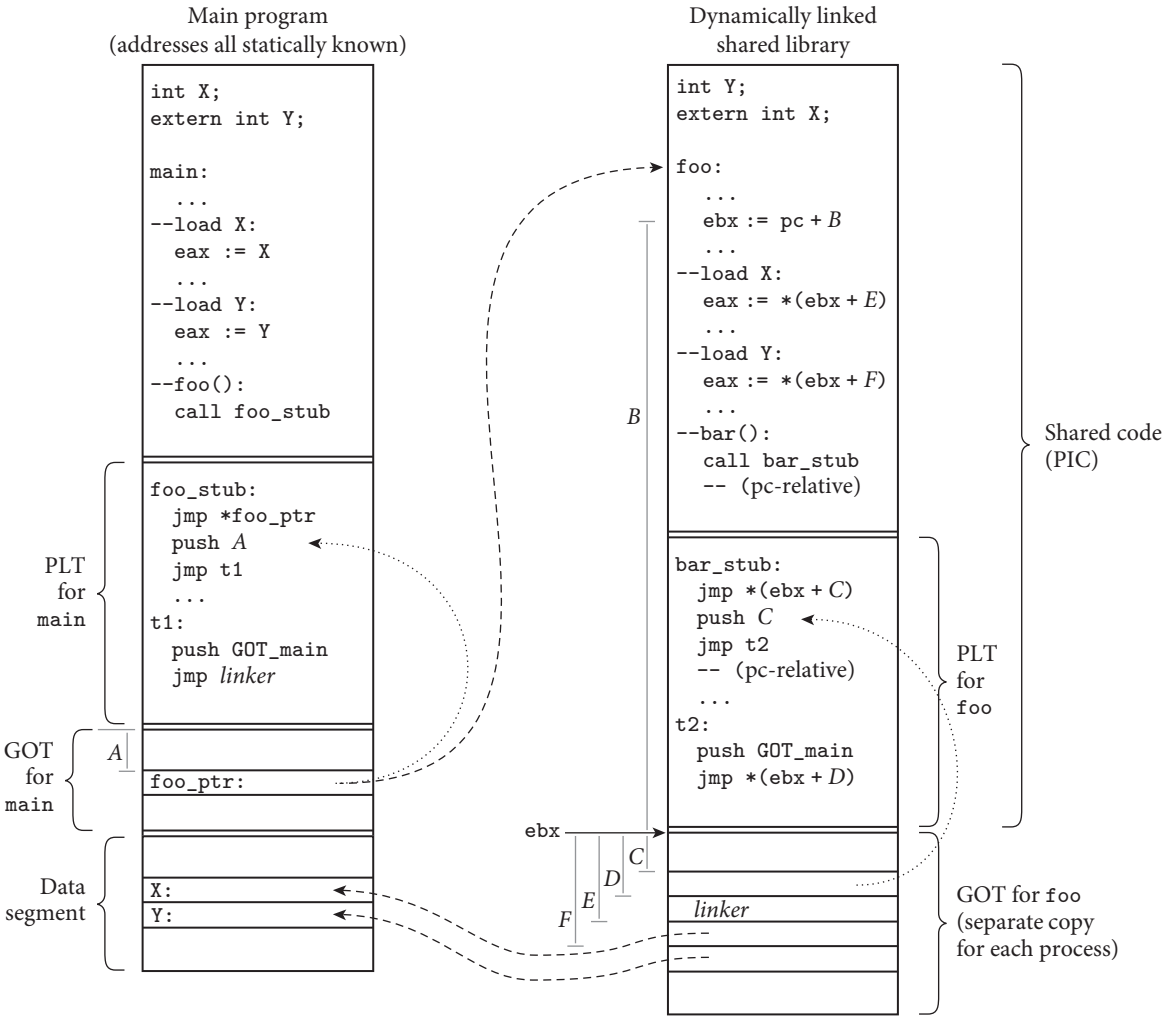

Figure 15.15 A dynamically linked shared library. Calls to `foo` and `bar` are made indirectly, using an address stored in the global offset tables (GOTs) of `main` and `foo`, respectively. Similarly, references to global variables `X` and `Y`, when made from `foo`, must employ a level of indirection. Resolved values are shown with dashed lines; initial values to support lazy linking (Section C-15.7.2) are shown with dotted lines. In the prologue of `foo`, register `ebx` is set to point to `foo`'s GOT, using `pc`-relative arithmetic.

`linkage table` (PLT) in each code segment. To effect a call to `foo`, `main` calls a *stub* routine, here named `foo_stub`. This, in turn, performs an indirect jump to the address of `foo` found in `main`'s GOT. Inside `foo`, the call to `bar` is only slightly more complicated: the compiler must use PC-relative addressing to find the appropriate slot in `foo`'s GOT.

EXAMPLE 15.24
PC-relative addressing on the x86

Most machines—including the x86—can perform branches and calls using PC-relative addressing. In our Linux example (Figure C-15.15), the machine-language encoding of `call bar_stub` in library `foo` will specify the offset between the call instruction and the `bar_stub` location in `foo`'s PLT.

Many machines can also use PC-relative addressing in load and store instructions. On the x86-64, for example, the load of `X` in `foo` could say `rax := *(rip + G)`, where *G* is the offset from the load instruction to `X`'s entry in `foo`'s GOT (on the x86-64, `rip` [instruction pointer register] is the name of the program counter). Unfortunately, the x86-32 does not support PC-relative addressing for loads and stores. To compensate, each PIC code segment on x86-32 Linux defines the following tiny subroutine:

```
get_pc:
    ebx := *esp     -- load location referred to by esp
    ret                 -- i.e., the return address -- into ebx
```

Given this definition, the pseudo-instruction `ebx := pc + B` in Figure C-15.15 can be implemented as

```
    call get_pc
    ebx += B
```

after which `ebx` can be used as the base for displacement addressing within `foo`'s GOT. ■

15.7.2 Fully Dynamic (Lazy) Linking

If all or most of the symbols exported by a shared library were always referenced by the parent program, it might make sense to link the library in its entirety at load time. When the program began running, its GOTs would then appear as suggested by the dashed arrows in Figure C-15.15. Certain systems indeed work in this fashion. In any given execution of a program, however, there may be references to libraries that are not actually used, because the input data never cause execution to follow the code path(s) on which the references appear. If these "potentially unnecessary" references are numerous, we may avoid a significant amount of work by linking the library *lazily* on demand. Moreover even in a program that uses all its symbols, incremental lazy linking may improve the system's interactive responsiveness by allowing programs to begin execution faster. A language system that allows the dynamic creation or discovery of program components (e.g., as in Common Lisp or Java) must also use lazy linking to delay the resolution of external references in dynamically compiled components.

EXAMPLE 15.25
Dynamic linking in Linux on the x86

When a Linux program first starts running, the data entries in its GOTs are indeed initialized as previously discussed; all addresses are known, because data locations are statically assigned. Code entries in the GOTs, however, point back into the corresponding PLTs, as suggested by the dotted arrows in Figure C-15.15.

Now consider what happens when `main` calls `foo_stub`. The `foo_ptr` entry in `main`'s GOT points to the second instruction of `foo_stub`—immediately after the indirect jump. That jump, in other words, ends up targeting the very next instruction, as if it had not happened at all. The next instruction, for its part, pushes onto the stack the offset of `foo`'s entry in `main`'s GOT. It then jumps (using PC-relative addressing) to a special entry in the PLT. This entry in turn pushes the address of `main`'s GOT and jumps to the dynamic linker, whose address is statically known. The dynamic linker consults symbol table information found in `main`'s executable file. Specifically, it looks up the GOT address and offset that were passed to it on the stack and discovers that they correspond to `foo`. It chooses a place for `foo` in the process's address space, creates a (process-specific) `foo` GOT at the appropriate offset (using symbol table information from `foo`'s own object file), initializes any data locations in that GOT to point to appropriate locations in the process's data segment, and initializes code locations in the GOT to point to the second instructions of the corresponding entries in `foo`'s PLT.

Now that `foo` has been given a location in the process's address space, the dynamic linker can modify `foo`'s entry in `main`'s PLT so that subsequent calls from `main` to `foo` will skip the linking step, and instead follow the single indirection suggested by the dashed arrow in Figure C-15.15. Last of all, the linker pops its arguments from the stack (leaving the return address pushed by `main` in its original call to `foo_stub`) and branches directly to `foo`. When `foo` completes, it will return to the correct address in `main`.

If and when `foo` calls `bar`, a similar series of events will take place. The principal difference is that both the body of `foo` and the stubs in its PLT must use PC-relative addressing to access entries in `foo`'s GOT. ■

✓ CHECK YOUR UNDERSTANDING

30. Explain the addressing challenge faced by dynamic linking systems.
31. What is *position-independent code*? What is it good for? What special precautions must a compiler follow in order to produce it?
32. Explain the need for PC-relative addressing in position-independent code. How is it accomplished on the x86-32?
33. What is the purpose of a *linkage table*?
34. What is *lazy* dynamic linking? What is its purpose? How does it work?

15 Building a Runnable Program

15.9 Exercises

15.12 Compare and contrast GIMPLE with the notation we have been using for abstract grammars (Section 4.1).

15.13 PC-relative branches on many processors are limited in range—they can only target locations within 2^k bytes of the current PC, for some k less than the wordsize of the machine. Explain how to generate position-independent code that needs to branch farther than this.

15.14 We have noted that Linux creates a single data segment containing all the static data of libraries that might be called (directly or indirectly) by a given program. The space required for this segment is usually not a problem: most libraries have little static data—often none at all. Suppose this were not the case. If we wanted to perform dynamic linking for modules with large amounts of per-module static data, how could we extend Linux's dynamic linking mechanisms to perform fully dynamic (lazy) linking not only of code, but also of data?

15.15 In Example C-9.72 we described how the GNU Ada Translator (`gnat`) for the x86 uses dynamically generated code to represent a subroutine closure. Explain how a similar technique could be used to simplify the mechanism of Figure C-15.15, if we were willing to modify code segments at run time.

15 Building a Runnable Program

15.10 Explorations

15.21 Find the on-line documentation for `gcc`, which explains both GIMPLE and RTL, and enumerates command-line flags that will cause the compiler to dump its intermediate forms to standard output. (Version 4.8.4 of the compiler supports 26 such flags for GIMPLE and 67 for RTL.) Using appropriate flags and a small but nontrivial input program, arrange for the compiler to dump several versions of both GIMPLE and RTL. Study the output and describe how it has been changed by the intervening code improvement phases.

15.22 Find out how linking works under your favorite non-Linux system. Can code be dynamically linked? Can (nonprivileged) users create shared libraries? How does the loader or dynamic linker determine which libraries a program will need? How does it locate their object code? If your compiler can generate both position-independent and non-position-independent code, how do the two compare in size and run-time efficiency?

15.23 Learn about *pointer swizzling* [Wil92], originally developed to run programs on machines with insufficient virtual address space. Explain its connection to dynamic linking.

15.24 Learn about ASIS, the Ada Semantic Interface Specification. How does it improve on tools based on the earlier Diana notation? How does it work in `gnat`?

15.25 Learn about MLIR, the Multi-Level Intermediate Representation being developed as an extension to the LLVM framework. In contrast to LLVM IR, MLIR aims to represent programs at multiple levels of abstraction, and in particular to facilitate high-level optimization for GPUs, tensor units, and other specialized accelerators. Explore the origins of MLIR in Google's TPU project. What exactly does MLIR enable that traditional medium- and low-level IFs do not?