Functions

This chapter studies the compilation of a subset of Python in which only top-level
function definitions are allowed. This kind of function appears in the C programming
language, and it serves as an important stepping-stone to implementing lexically
scoped functions in the form of lambda abstractions, which is the topic of chapter 9.

8.1 The $L_{\text{Fun}}$ Language

The concrete syntax and abstract syntax for function definitions and function appli-
cation are shown in figures 8.1 and 8.2, with which we define the $L_{\text{Fun}}$ language.
Programs in $L_{\text{Fun}}$ begin with zero or more function definitions. The function names
from these definitions are in scope for the entire program, including all the function
definitions, and therefore the ordering of function definitions does not matter. The
abstract syntax for function parameters in figure 8.2 is a list of pairs, each of which
consists of a parameter name and its type. This design differs from Python’s ast
module, which has a more complex structure for function parameters to handle
keyword parameters, defaults, and so on. The type checker in type_check_Lfun
converts the complex Python abstract syntax into the simpler syntax shown in
figure 8.2. The fourth and sixth parameters of the FunctionDef constructor are
for decorators and a type comment, neither of which are used by our compiler. We
recommend replacing them with None in the shrink pass. The concrete syntax for
function application is $\text{exp}(\text{exp}, \ldots)$, where the first expression must evaluate to a
function and the remaining expressions are the arguments. The abstract syntax for
function application is $\text{Call}(\text{exp}, \text{exp}^*)$.

Functions are first-class in the sense that a function pointer is data and can be
stored in memory or passed as a parameter to another function. Thus, there is a
function type, written

$$\text{Callable}[\text{type}_1, \ldots, \text{type}_n, \text{rtype}]$$

for a function whose $n$ parameters have the types $\text{type}_1$ through $\text{type}_n$ and whose
return type is $\text{rtype}$. The main limitation of these functions (with respect to Python
functions) is that they are not lexically scoped. That is, the only external entities
Figure 8.1
The concrete syntax of $L_{\text{Fun}}$, extending $L_{\text{While}}$ (figure 6.1).

that can be referenced from inside a function body are other globally defined functions. The syntax of $L_{\text{Fun}}$ prevents function definitions from being nested inside each other.

The program shown in figure 8.3 is a representative (though slightly artificial) example of defining and using functions in $L_{\text{Fun}}$. We define a function twice that applies some other function $f$ twice to an integer and returns the result. We also define a function inc. The program applies twice to inc and 40. The result is 42, which we print.

An $L_{\text{Fun}}$ function designed to be executed for its side-effects need not return a value. This is specified in the function’s type by giving a return type of None and in the function’s code by using a return statement without an expression, or by “falling off the end” of the function without executing an explicit return statement at all. In the abstract syntax, the None type is represented by the VoidType() constructor; the name “void” is taken from C-like languages. There are no values of type void. If a function $f$ has void return type, calls to $f$ are not permitted in contexts that expect a value; in particular the results of such calls cannot be assigned to variables, passed as arguments to other functions, or used as explicit return values in return statements. In effect, such functions calls can only be used as expressions that appear in statement context. On the other hand, it is legal to use a function that returns a non-void value in a statement context; the returned value is just ignored.

The definitional interpreter for $L_{\text{Fun}}$ is shown in figure 8.4. The case for the Module AST is responsible for setting up the mutual recursion between the top-level function definitions. We create a dictionary named env and fill it in by mapping each function name to a new Function value, each of which stores a reference to the env. (We define the class Function for this purpose.) To interpret a function call,
we match the result of the function expression to obtain a function value. We then extend the function's environment with the mapping of parameters to argument values. Finally, we interpret the body of the function in this extended environment.

The type checker for $\mathcal{L}_{\text{Fun}}$ is shown in figures 8.5 and 8.6. (We omit some complications, including the code that parses function parameters into the simpler abstract syntax.) Similarly to the interpreter, the case for the Module AST is responsible for setting up the mutual recursion between the top-level function definitions. We begin by create a mapping $\text{env}$ from every function name to its type. We then type
class InterpLfun(InterpLwhile):
    def apply_fun(self, fun, args):
        match fun:
            case Function(name, xs, body, env):
                new_env = {x: v for (x, v) in env.items()}
                for (x, arg) in zip(xs, args):
                    new_env[x] = arg
                return self.interp_stmts(body, new_env)
            case _:
                raise Exception('apply_fun: unexpected: ' + repr(fun))

def interp_exp(self, e, env):
    match e:
        case Call(Name(f), args) if f in builtin_functions:
            return super().interp_exp(e, env)
        case Call(func, args):
            f = self.interp_exp(func, env)
            vs = [self.interp_exp(arg, env) for arg in args]
            return self.apply_fun(f, vs)
        case _:
            return super().interp_exp(e, env)

def interp_stmt(self, s, env, cont):
    match s:
        case Return(None):
            return None
        case Return(value):
            return self.interp_exp(value, env)
        case FunctionDef(name, params, bod, dl, returns, comment):
            if isinstance(params, ast.arguments):
                ps = [p.arg for p in params.args]
            else:
                ps = [x for (x, t) in params]
            env[name] = Function(name, ps, bod, env)
            return self.interp_stmts(cont, env)
        case _:
            return super().interp_stmt(s, env, cont)

def interp(self, p):
    match p:
        case Module(ss):
            env = {}
            self.interp_stmts(ss, env)
            if 'main' in env.keys():
                self.apply_fun(env['main'], [])
        case _:
            raise Exception('interp: unexpected ' + repr(p))
check the program using this mapping. In the case for function call, we match the
type of the function expression to a function type and check that the types of the
argument expressions are equal to the function’s parameter types. The type of the
call as a whole is the return type from the function type.
```python
def type_check_exp(self, e, env):
    match e:
        case Call(Name(f), args) if f in builtin_functions:
            return super().type_check_exp(e, env)
        case Call(func, args):
            func_t = self.type_check_exp(func, env)
            args_t = [self.type_check_exp(arg, env) for arg in args]
            match func_t:
                case FunctionType(params_t, return_t):
                    for (arg_t, param_t) in zip(args_t, params_t):
                        self.check_type_equal(param_t, arg_t, e)
                    return return_t
                case _:
                    raise Exception('type_check_exp: in call, unexpected ' + \
                                    'repr(func_t)')
        case _:
            return super().type_check_exp(e, env)

def type_check_stmts(self, ss, env):
    if len(ss) == 0:
        return VoidType()
    match ss[0]:
        case Assign([Name(id)], value):
            if self.type_check_exp(value, env) == VoidType():
                raise Exception('type_check_stmts: attempt to assign ' + \
                                'void-typed expression ' + repr(value))
            return super().type_check_stmts(ss, env) # recheck
        case FunctionDef(name, params, body, dl, returns, comment):
            new_env = {x: t for (x, t) in env.items()}
            unique_names = {x for x, _ in params}
            if len(unique_names) != len(params):
                raise Exception('type_check: duplicate parameter name ' + \
                                'in function ' + repr(name))
            for x, t in params:
                if t == VoidType():
                    raise Exception('type_check: argument type for ' + \
                                    repr(x) + ' declared as void')
            for x, t in params:
                new_env[x] = t
            rt = self.type_check_stmts(body, new_env)
            self.check_type_equal(new_returns, rt, ss[0])
            return self.type_check_stmts(ss[1:], env)
        case Return(None):
            return VoidType()
        case Return(value):
            t = self.type_check_exp(value, env)
            if t == VoidType():
                raise Exception('type_check_stmts: attempt to return ' + \
                                'void-typed expression ' + repr(value))
            return t
        case _:
            return super().type_check_stmts(ss, env)
```

**Figure 8.5**
Type checker for the \(\mathcal{L}_{\text{Fun}}\) language (part 1).
def type_check(self, p):
    match p:
        case Module(body):
            env = {}
            for s in body:
                match s:
                    case FunctionDef(name, params, bod, dl, returns, comment):
                        params_t = [t for (x, t) in params]
                        if name in env:
                            raise Exception('type_check: ' \
                                + 'duplicate function name ' + name)
                        env[name] = FunctionType(params_t,
                            self.parse_type_annot(returns))
                        self.type_check_stmts(body, env)
                    case _:
                        raise Exception('type_check: unexpected ' + repr(p))

Figure 8.6
Type checker for the $\mathcal{L}_{\text{Fun}}$ language (part 2).
8.2 Functions in x86

The x86 architecture provides a few features to support the implementation of functions. We have already seen that there are labels in x86 so that one can refer to the location of an instruction, as is needed for jump instructions. Labels can also be used to mark the beginning of the instructions for a function. Going further, we can obtain the address of a label by using the leaq instruction. For example, the following puts the address of the inc label into the rbx register:

\[
\text{leaq inc(\%rip), } \%\text{rbx}
\]

The addressing mode inc(\%rip) essentially stands for the address of the inc function using a special instruction-pointer-relative addressing mode of the x86-64 processor. In particular, the assembler computes the distance \( d \) between the address of inc and where the rip would be at that moment and then changes the inc(\%rip) argument to \( d(\%\text{rip}) \), which at runtime will compute the address of inc.

In section 2.2 we used the callq instruction to jump to functions whose locations were given by a label, such as read_int. This will continue to work in this chapter when we call a function directly by its name (such as twice in the example of figure 8.3). But to support first-class functions (such as \( f \) in the same figure), we need to be able to enter a function by jumping to an address held in a register; that is, we use indirect function calls. The x86 syntax for this is a callq instruction that requires an asterisk before the register name.

\[
\text{callq } \star\%\text{rbx}
\]

8.2.1 Calling Conventions

The callq instruction provides partial support for implementing functions: it pushes the return address on the stack and it jumps to the target. However, callq does not handle

1. parameter passing,
2. pushing frames on the procedure call stack and popping them off, or
3. determining how registers are shared by different functions.

Regarding parameter passing, recall that the x86-64 calling convention for Unix-based systems uses the following six registers to pass arguments to a function, in the given order:

\[
\text{rdi rsi rdx rcx r8 r9}
\]

If there are more than six arguments, then the calling convention mandates using space on the frame of the caller for the rest of the arguments. Specifically, the caller should arrange for argument 7 to be placed on the stack immediately above where the return address will go during the call, argument 8 immediately above that, and so on. The return value of the function is stored in register rax.
Regarding frames and the procedure call stack, recall from section 2.2 that the stack grows down and each function call uses a chunk of space on the stack called a frame. The caller sets the stack pointer, register `rsp`, to the last data item in its frame. The callee must not change anything in the caller’s frame, that is, anything that is at or above the stack pointer. The callee is free to use locations that are below the stack pointer.

Regarding the sharing of registers between different functions, recall from section 4.1 that the registers are divided into two groups, the caller-saved registers and the callee-saved registers. The caller should assume that all the caller-saved registers are overwritten with arbitrary values by the callee. For that reason we recommend in section 4.1 that variables that are live during a function call should not be assigned to caller-saved registers.

On the flip side, if the callee wants to use a callee-saved register, the callee must save the contents of those registers on their stack frame and then put them back prior to returning to the caller. For that reason we recommend in section 4.1 that if the register allocator assigns a variable to a callee-saved register, then the prelude of the `main` function must save that register to the stack and the conclusion of `main` must restore it. This recommendation now generalizes to all functions.

Recall that the base pointer, register `rbp`, is used as a point of reference within a frame, so that each local variable can be accessed at a fixed (negative) offset from the base pointer (section 2.2). Furthermore, if more than six arguments are passed, the seventh and succeeding ones can also be accessed at fixed (positive) offsets from the base pointer. Figure 8.7 shows the layout of the caller and callee frames.

### 8.3 Shrink `L_{Fun}`

The shrink pass performs a minor modification to ease the later passes. This pass introduces an explicit `main` function that gobbles up all the top-level statements of the module.

\[
\text{Module}(\text{def } \ldots \text{stmt } \ldots) \\
\Rightarrow \text{Module}(\text{def } \ldots \text{mainDef})
\]

where `mainDef` is

\[
\text{FunctionDef}('\text{main}', [], \text{int}, \text{None}, \text{stmt } \ldots \text{Return}(\text{Constant}(0)), \text{None})
\]

### 8.4 Reveal Functions and the `L_{FunRef}` Language

The syntax of `L_{Fun}` is inconvenient for purposes of compilation in that it conflates the use of function names and local variables. This is a problem because we need to compile the use of a function name differently from the use of a local variable. In particular, we use `leaq` to convert the function name (a label in x86) to an address in a register, unless the function name appears directly in the caller position of a call expression. Thus, we create a new pass that changes function references from
### Chapter 8

#### Caller View

<table>
<thead>
<tr>
<th>Contents</th>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>return address</td>
<td>Caller</td>
</tr>
<tr>
<td>old rbp</td>
<td></td>
</tr>
<tr>
<td>callee-saved 1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>callee-saved j</td>
<td></td>
</tr>
<tr>
<td>local variable 1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>local variable k</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

#### Callee View

<table>
<thead>
<tr>
<th>Contents</th>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument 6 + n</td>
<td></td>
</tr>
<tr>
<td>argument 7</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>return address</td>
<td>Callee</td>
</tr>
<tr>
<td>old rbp</td>
<td></td>
</tr>
<tr>
<td>callee-saved 1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>callee-saved n</td>
<td></td>
</tr>
<tr>
<td>local variable 1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>local variable m</td>
<td></td>
</tr>
</tbody>
</table>

#### Figure 8.7

Memory layout of caller and callee frames, where %rsp is the stack pointer just before the call is performed.

Name$(f)$ to FunRef$(f)$. This pass is named `reveal_functions` and the output language is $L_{FunRef}$.

The `reveal_functions` pass should come before the `remove_complex_operands` pass because function references should sometimes be categorized as complex expressions.

### 8.5 Remove Complex Operands

The primary decisions to make for this pass are whether to classify FunRef and Call as either atomic or complex expressions. Recall that an atomic expression ends up as an immediate argument of an x86 instruction. Function application translates to a sequence of instructions, so Call must be classified as a complex expression. On the other hand, the arguments of Call should be atomic expressions. Regarding FunRef, as discussed previously, the function label often needs to be converted to an address using the `leaq` instruction. In such cases, even though FunRef seems rather simple, it needs to be classified as a complex expression so that we generate an assignment statement with a left-hand side that can serve as the target of the `leaq`. However, FunRef’s that appear in the caller position of calls will turn into
direct calls, so we should not pull them out as complex subexpressions (since that would make it harder to recognize this case when doing instruction selection).

The output of this pass, $L_{\text{FunRef}}$ (figure 8.8), extends $L_{\text{While}}$ (figure 6.6) with $\text{FunRef}$ and $\text{Call}$ in the grammar for expressions and augments programs to include a list of function definitions. Also, $L_{\text{FunRef}}$ adds $\text{Return}$ to the grammar for statements.

### 8.6 Explicate Control and the $C_{\text{Fun}}$ Language

Figure 8.9 defines the abstract syntax for $C_{\text{Fun}}$, the output of $\text{explicate\_control}$. The auxiliary functions for assignment and effect-only contexts need to handle $\text{Call}$ and $\text{FunRef}$ and the function for predicate context should be updated for $\text{Call}$ but not $\text{FunRef}$. (A $\text{FunRef}$ cannot be a Boolean.) The code for handling statements needs to be extended to handle returns. We recommend defining a new auxiliary function for processing function definitions. The top-level $\text{explicate\_control}$ function can then apply this new auxiliary function to all the $L_{\text{FunRef}}$ function definitions. Function bodies that “fall off the end” should acquire a terminating $\text{Return(None)}$. 

---

**Figure 8.8**

$L_{\text{FunRef}}$ is $L_{\text{FunRef}}$ in monadic normal form.
Figure 8.9
The abstract syntax of $C_{\text{Fun}}$, extending $C_{\text{If}}$ (figure 5.8).
The output of select instructions is a program in the \texttt{x86\textsubscript{Def} callq+} language; the definition of its concrete syntax is shown in figure 8.10, and the definition of its abstract syntax is shown in figure 8.11. We discuss the new instructions as needed in this section.

An assignment of a function reference to a variable becomes a load-effective-address instruction as follows, where \textit{lhs}' is the translation of \textit{lhs} from \textit{atm} in \textit{C\textsubscript{Fun}} to \textit{arg} in \textit{x86\textsubscript{Var} Def}. The \texttt{FunRef} becomes a \texttt{Global} AST node, whose concrete syntax is instruction-pointer-relative addressing.

\[
\textit{lhs} = \texttt{FunRef}(f); \quad \Rightarrow \quad \texttt{leaq} f(\%\text{rip}), \textit{lhs}'
\]
Regarding function definitions, we reuse the `FunctionDef` constructor from \( \mathcal{L}_{\text{Fun}} \) and \( \mathcal{C}_{\text{Fun}} \) in the abstract syntax of \( x86^{\text{Def}}_{\text{calq}} \). But we need to remove the parameters and instead perform parameter passing using the conventions discussed in section 8.2. That is, the first six arguments are passed in registers, and any additional arguments are pushed on the stack. We recommend turning the parameters into local variables and generating instructions at the beginning of the function to move from the argument-passing registers (section 8.2.1) or the stack to these local variables.

\[
\text{FunctionDef}(f, [(x_1, T_1), \ldots], B, \_, T_r, \_)
\]
\[
\Rightarrow
\text{FunctionDef}(f, [], B', \_, T_r, \_)
\]

The basic blocks \( B' \) are the same as \( B \) except that the start block is modified to add the instructions for moving from the argument registers and stack slots to the parameter variables. So the start block of \( B \) shown on the left of the following is changed to the code on the right:

```
f_{\text{start}}:
  \text{movq } \%rdi, x_1
  \text{movq } \%rsi, x_2
\text{start:}
  \text{...}
  \text{movq } 16(\%rbp), x_7
  \text{movq } 24(\%rbp), x_8
  \text{...}
  \text{instr}_1
  \text{...}
  \text{instr}_n
```

Recall that we use the label `start` for the initial block of a program, and in section 2.5 we recommend labeling the conclusion of the program with `conclusion`, so that `Return(Arg)` can be compiled to an assignment to `rax` followed by a jump to `conclusion`. With the addition of function definitions, there is a start block and conclusion for each function, but their labels need to be unique. We recommend prepending the function’s name to `start` and `conclusion`, respectively, to obtain unique labels.

By changing the parameters to local variables, we are giving the register allocator control over which registers or stack locations to use for them. If you implement the move-biasing challenge (section 4.7), the register allocator will try to assign the parameter variables to the corresponding argument register, in which case the `patch_instructions` pass will remove the `movq` instruction. This happens in the example translation given in figure 8.13 in section 8.11, in the `add` function. Also, note that the register allocator will perform liveness analysis on this sequence of move instructions and build the interference graph. So, for example, \( x_1 \) will be marked as interfering with \( rsi \), and that will prevent the mapping of \( x_1 \) to \( rsi \), which is good because otherwise the first `movq` would overwrite the argument in \( rsi \) that is needed for \( x_2 \).
Next, consider the compilation of function calls. In the mirror image of the handling of parameters in function definitions, the first six arguments are moved to the argument-passing registers, and any subsequent arguments are pushed onto the stack (in reverse order). To keep $\text{rsp}$ properly 16-byte aligned at the call, we recommend pushing an extra pseudo-argument on the stack if the number of real stack arguments is not divisible by 2. In the general case, the function is not given as a label, but its address is produced by the argument $\text{arg}_0$. So, we translate the call into an indirect function call. After the call, we must adjust $\%\text{rsp}$ to remove any arguments that were passed on the stack. The return value from the function is stored in $\text{rax}$, so it needs to be moved into the $\text{lhs}$. If $n$ is an even number greater than 6, the translation looks like this:

$$\text{lhs} = \text{Call}(\text{arg}_0, \text{arg}_1, \text{arg}_2 \ldots \text{arg}_n)$$

$$\Rightarrow$$

movq $\text{arg}_1$, %rdi
movq $\text{arg}_2$, %rsi

movq $\text{arg}_6$, %r9
pushq $\text{arg}_{n-6}$

pushq $\text{arg}_{n-5}$
... callq *$\text{arg}_0$
addq $8(n-6)$, %rsp
movq %rax, $\text{lhs}$

The $\text{IndirectCallq}$ AST node includes an integer for the arity of the function, that is, the number of parameters. That information is useful in the $\text{uncover\_live}$ pass for determining which argument-passing registers are potentially read during the call.

In the special case where $\text{arg}_0$ is a built-in function or a $\text{FunRef}$, we should use a slightly more efficient direct call in place of the indirect call.

### 8.8 Register Allocation

The addition of functions requires some changes to all three aspects of register allocation, which we discuss in the following subsections.

#### 8.8.1 Liveness Analysis

The $\text{IndirectCallq}$ instruction should be treated like $\text{Callq}$ regarding its written locations $W$, in that they should include all the caller-saved registers. Recall that the reason for that is to force variables that are live across a function call to be assigned to callee-saved registers or to be spilled to the stack.

Regarding the set of read locations $R$, the arity field of $\text{IndirectCallq}$ determines how many of the argument-passing registers should be considered as read by this instruction. Also, the target field of $\text{IndirectCallq}$ should be included in the set of read locations $R$. 
8.8.2 Build Interference Graph
With the addition of function definitions, we compute a separate interference graph for each function (not just one for the whole program).

8.8.3 Allocate Registers
The primary change to the allocate_registers pass is adding an auxiliary function for handling definitions (the def nonterminal shown in figure 8.11) with one case for function definitions. The logic is the same as described in chapter 4 except that now register allocation is performed many times, once for each function definition, instead of just once for the whole program.

8.9 Patch Instructions
In patch_instructions, you should deal with the x86 idiosyncrasy that the destination argument of leaq must be a register.

8.10 Prelude and Conclusion
We now must generate a prelude and conclusion for each function definition. This code is similar to the prelude and conclusion generated for the main function presented in chapter 7. To review, the prelude of every function should carry out the following steps:

1. Push rbp to the stack and set rbp to current stack pointer.
2. Push to the stack all the callee-saved registers that were used for register allocation.
3. Move the stack pointer rsp down to make room for the regular spills (aligned to 16 bytes).
4. Jump to the start block.

The conclusion of every function should do the following:

1. Move the stack pointer back up past the regular spills.
2. Restore the callee-saved registers by popping them from the stack.
3. Restore rbp by popping it from the stack.
4. Return to the caller with the retq instruction.

The output of this pass is x86_callq*, which differs from x86_defcallq* in that there is no longer an AST node for function definitions. Instead, a program is just an association list of basic blocks, as in x86. So we have the following grammar rule:

\[
\text{x86}\text{callq}^* ::= \text{X86Program}((\text{label . block} \ldots ))
\]

Figure 8.12 gives an overview of the passes for compiling LFun to x86.
Figure 8.12
Diagram of the passes for $\mathcal{L}_{\text{Fun}}$, a language with functions.

Exercise 8.1 Expand your compiler to handle $\mathcal{L}_{\text{Fun}}$ as outlined in this chapter. Create eight new programs that use functions including examples that pass functions and return functions from other functions, recursive functions, etc. Test your compiler on these new programs and all your previously created test programs.

8.11 An Example Translation

Figure 8.13 shows an example translation of a simple function in $\mathcal{L}_{\text{Fun}}$ to x86. The figure includes the results of explicate_control and select_instructions.
Figure 8.13
Example compilation of a simple function to x86.