8 Functions

Specialized version of this chapter for use at PSU, Winter 2024.

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.

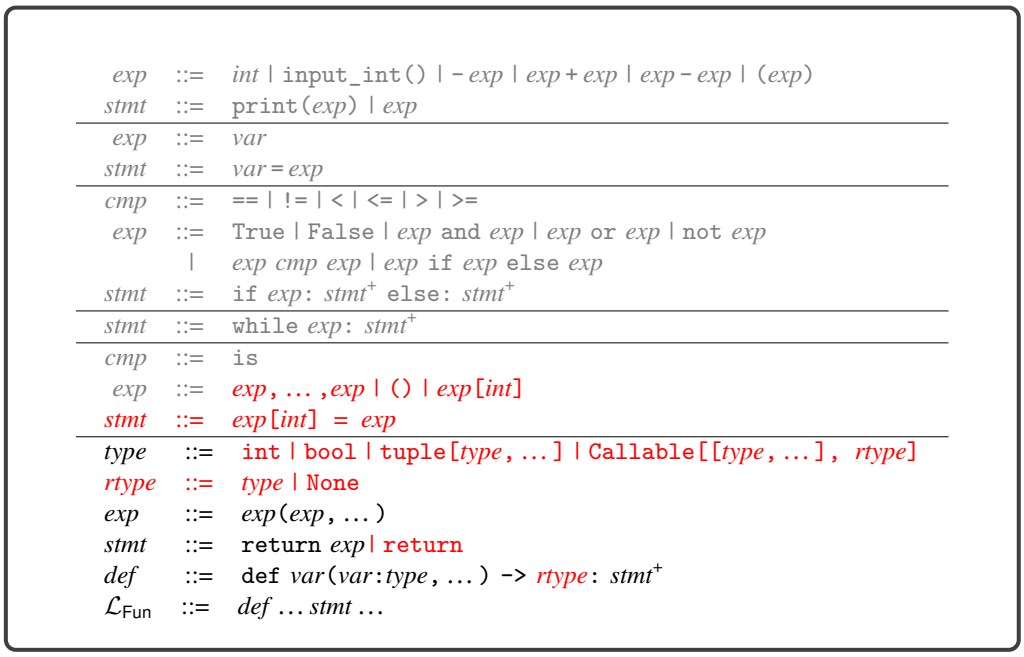
8.1 The $\mathcal{L}_{\mathsf{Fun}}$ Language

The concrete syntax and abstract syntax for function definitions and function application are shown in figures 8.1 and 8.2, with which we define the $\mathcal{L}_{\mathsf{Fun}}$ language. Programs in $\mathcal{L}_{\mathsf{Fun}}$ 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 *exp*`(`*exp*`, ...)`, where the first expression must evaluate to a function and the remaining expressions are the arguments. The abstract syntax for function application is `Call(`*exp*`,` *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

`Callable[[`$type_1$`,`$\cdots$`,`$type_n$`],` $type_R$`]`

for a function whose n parameters have the types $type_1$ through $type_n$ and whose return type is $type_R$. The main limitation of these functions (with respect to Python functions) is that they are not lexically scoped. That is, the only external entities



exp	::=	*int* | `input_int()` | `-` *exp* | *exp* `+` *exp* | *exp* `-` *exp* | `(`*exp*`)`
stmt	::=	`print(`*exp*`)` | *exp*
exp	::=	*var*
stmt	::=	*var* `=` *exp*
cmp	::=	`==` | `!=` | `<` | `<=` | `>` | `>=`
exp	::=	`True` | `False` | *exp* `and` *exp* | *exp* `or` *exp* | `not` *exp*
	|	*exp cmp exp* | *exp* `if` *exp* `else` *exp*
stmt	::=	`if` *exp*`:` *stmt*$^+$ `else:` *stmt*$^+$
stmt	::=	`while` *exp*`:` *stmt*$^+$
cmp	::=	`is`
exp	::=	*exp*`,` ... `,`*exp* | `()` | *exp*`[`*int*`]`
stmt	::=	*exp*`[`*int*`]` `=` *exp*
type	::=	`int` | `bool` | `tuple[`*type*`,` ...`]` | `Callable[[`*type*`,` ...`],` *rtype*`]`
rtype	::=	*type* | `None`
exp	::=	*exp*`(`*exp*`,` ...`)`
stmt	::=	`return` *exp* | `return`
def	::=	`def` *var*`(`*var*`:`*type*`,` ...`)` `->` *rtype*`:` *stmt*$^+$
$\mathcal{L}_{\mathsf{Fun}}$	::=	*def* ... *stmt* ...

Figure 8.1
The concrete syntax of $\mathcal{L}_{\mathsf{Fun}}$, extending $\mathcal{L}_{\mathsf{Tup}}$ (figure 7.1).

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

The program shown in figure 8.3 is a representative example of defining and using functions in $\mathcal{L}_{\mathsf{Fun}}$. We define a function `map` that applies some other function `f` to both elements of a tuple and returns a new tuple containing the results. We also define a function `inc`. The program applies `map` to `inc` and `(0, 41)`. The result is `(1, 42)`, from which we return `42`.

An $\mathcal{L}_{\mathsf{Fun}}$ 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 $\mathcal{L}_{\mathsf{Fun}}$ is shown in figure 8.4. The case for the `Module` AST is responsible for setting up the mutual recursion between the top-level

```
exp    ::= Constant(int) | Call(Name('input_int'),[])
         | UnaryOp(USub(),exp) | BinOp(exp,Add(),exp)
         | BinOp(exp,Sub(),exp)
stmt   ::= Expr(Call(Name('print'),[exp])) | Expr(exp)
-----------------------------------------------------
exp    ::= Name(var)
stmt   ::= Assign([Name(var)], exp)
-----------------------------------------------------
boolop ::= And() | Or()
cmp    ::= Eq() | NotEq() | Lt() | LtE() | Gt() | GtE()
bool   ::= True | False
exp    ::= Constant(bool) | BoolOp(boolop,[exp,exp])
         | UnaryOp(Not(),exp) | Compare(exp,[cmp],[exp])
         | IfExp(exp,exp,exp)
stmt   ::= If(exp, stmt+, stmt+)
-----------------------------------------------------
stmt   ::= While(exp, stmt+, [])
-----------------------------------------------------
cmp    ::= Is()
exp    ::= Tuple(exp*,Load()) | Subscript(exp,Constant(int),Load())
stmt   ::= Assign([Subscript(exp,Constant(int),Store())], exp)
-----------------------------------------------------
type   ::= IntType() | BoolType() | TupleType(type*)
         | FuncType(type*, rtype)
rtype  ::= type | VoidType()
exp    ::= Call(exp, exp*)
stmt   ::= Return(exp) | Return(None)
params ::= (var,type)*
def    ::= FunctionDef(var, params, stmt+, None, rtype, None)
L_Fun  ::= Module([def ... stmt ...])
```

Figure 8.2
The abstract syntax of $\mathcal{L}_{\mathsf{Fun}}$, extending $\mathcal{L}_{\mathsf{Tup}}$ (figure 7.2).

```
def map(f : Callable[[int], int], v : tuple[int,int]) -> tuple[int,int]:
  return f(v[0]), f(v[1])

def inc(x : int) -> int:
  return x + 1

print(map(inc, (0, 41))[1])
```

Figure 8.3
Example of using functions in $\mathcal{L}_{\mathsf{Fun}}$.

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 $\mathcal{L}_{\mathsf{Fun}}$ ~~is shown in figure 8.5~~ can be found in on-line materials. ~~(We omit 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 creating a mapping `env` from every function name to its type. We then type check the program using this mapping. To check a function definition, we copy and extend the `env` with the parameters of the function. We then type check the body of the function and obtain the actual return type `rt`, which is either the type of the expression in a `return` statement or the `VoidType` if the `return` statement carries no expression or if control reaches the end of the function without a `return` statement. (If there are multiple `return` statements, the types of their expressions must agree.) Finally, we check that the actual return type `rt` is equal to the declared return type `returns`. To check a 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.

```
class InterpLfun(InterpLtup):
  def apply_fun(self, fun, args, e):
    match fun:
      case Function(name, xs, body, env):
        new_env = env.copy().update(zip(xs, args))
        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('input_int'), []):
        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, e)
      case _:
        return super().interp_exp(e, env)

  def interp_stmt(self, s, env, cont):
    match s:
      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'], [], None)
      case _:
        raise Exception('interp: unexpected ' + repr(p))
```

Figure 8.4
Interpreter for the $\mathcal{L}_{\mathsf{Fun}}$ language.

Figure 8.5
Type checker for the $\mathcal{L}_{\mathsf{Fun}}$ language. Omitted: See actual file for our slightly different version.

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:

```
leaq inc(%rip), %rbx
```

Recall from section 7.6 that `inc(%rip)` is an example of instruction-pointer-relative addressing.

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 `map` in the example of figure 8.3). But to support first-class functions (such as `inc` 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.

```
callq *%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:

```
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.

Recall that we store variables of tuple type on the root stack. So, the prelude of a function needs to move the root stack pointer `r15` up according to the number of variables of tuple type and the conclusion needs to move the root stack pointer back down. Also, the prelude must initialize to `0` this frame's slots in the root stack to signal to the garbage collector that those slots do not yet contain a valid pointer. Otherwise the garbage collector will interpret the garbage bits in those slots as memory addresses and try to traverse them, causing serious mayhem!

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 8.6 shows the layout of the caller and callee frames.

8.2.2 ~~Efficient Tail Calls~~

8.3 Shrink $\mathcal{L}_{\text{Fun}}$

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.

`Module(`*def* ... *stmt* ... `)`
⇒ `Module(`*def* ... *mainDef*`)`

where *mainDef* is

`FunctionDef('main', [],` *stmt* ... `Return(Constant(0)), None, IntType(), None)`

Shrink must also be extended to handle expressions in `return` statements.

8.4 Reveal Functions and the $\mathcal{L}_{\text{FunRef}}$ Language

The syntax of $\mathcal{L}_{\text{Fun}}$ 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

Caller View	Callee View	Contents	Frame
8(%rbp)		return address	
0(%rbp)		old rbp	
-8(%rbp)		callee-saved 1	Caller
...		...	
$-8j$(%rbp)		callee-saved j	
$-8(j+1)$(%rbp)		local variable 1	
...		...	
$-8(j+k)$(%rbp)		local variable k	
$8n-8$(%rsp)	$8n+8$(%rbp)	argument $6+n$	
	...	...	
0(%rsp)	16(%rbp)	argument 7	
	8(%rbp)	return address	
	0(%rbp)	old rbp	
	-8(%rbp)	callee-saved 1	Callee
	...	...	
	$-8n$(%rbp)	callee-saved n	
	$-8(n+1)$(%rbp)	local variable 1	
	...	...	
	$-8(n+m)$(%rbp)	local variable m	

Figure 8.6
Memory layout of caller and callee frames, where %rsp is the stack pointer just before the call is performed.

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. Thus, we create a new pass that changes function references from `Name(`f`)` to `FunRef(`f`)`. This pass is named `reveal_functions` and the output language is $\mathcal{L}_{\text{FunRef}}$.

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

8.5 ~~Limit Functions~~

8.6 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. So should the argument of a `Return`. Regarding `FunRef`, as discussed previously, the function

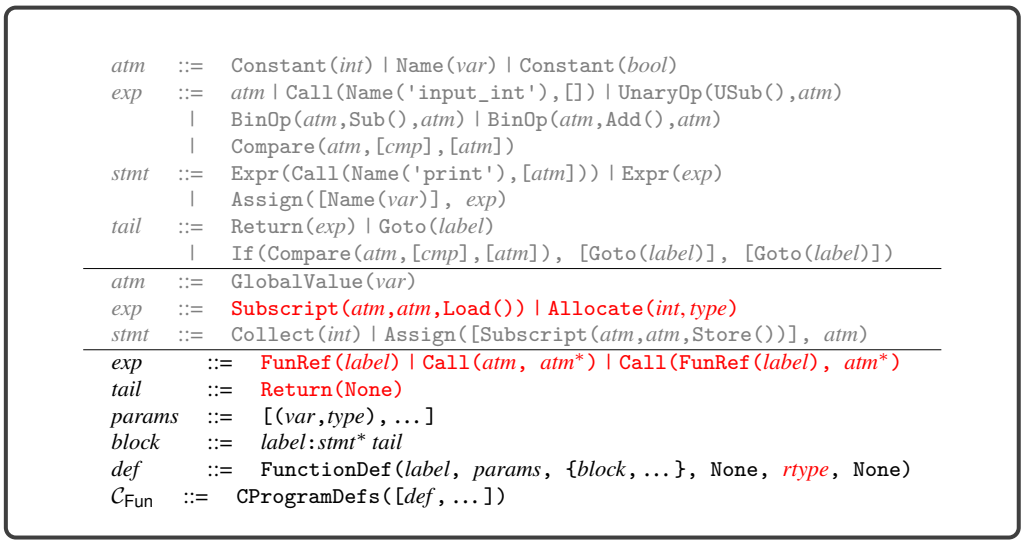
Figure 8.7
$\mathcal{L}^{mon}_{\text{FunRef}}$ is $\mathcal{L}_{\text{FunRef}}$ in monadic normal form.

label needs to be converted to an address using the `leaq` instruction. Thus, 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`.

The output of this pass, $\mathcal{L}^{mon}_{\text{FunRef}}$ (figure 8.7), extends $\mathcal{L}^{mon}_{\text{Alloc}}$ (figure 7.11) with `FunRef` and `Call` in the grammar for expressions and augments programs to include a list of function definitions. Also, $\mathcal{L}^{mon}_{\text{FunRef}}$ adds `Return` to the grammar for statements.

8.7 Explicate Control and the $\mathcal{C}_{\text{Fun}}$ Language

Figure 8.8 defines the abstract syntax for $\mathcal{C}_{\text{Fun}}$, the output of `explicate_control`. The auxiliary functions for assignment should be updated with cases for `Call` and `FunRef` and the function for predicate context should be updated for `Call` but not `FunRef`. (A `FunRef` cannot be a Boolean.) The code for handling statements needs to be extended to handle `return`s. Also, function bodies that "fall off the end" should acquire a terminating `Return(None)`. ~~In assignment and predicate contexts, Call remains Call, whereas in tail position Call becomes Ta~~
We recommend defining a new auxiliary function for processing function definitions. This code is similar to the case for `Program` in $\mathcal{L}_{\text{Tup}}$. The top-level `explicate_control` function that handles the `ProgramDefs` form of $\mathcal{L}_{\text{Fun}}$ can then apply this new function to all the function definitions.



```
atm    ::=  Constant(int) | Name(var) | Constant(bool)
exp    ::=  atm | Call(Name('input_int'),[]) | UnaryOp(USub(),atm)
         |  BinOp(atm,Sub(),atm) | BinOp(atm,Add(),atm)
         |  Compare(atm,[cmp],[atm])
stmt   ::=  Expr(Call(Name('print'),[atm])) | Expr(exp)
         |  Assign([Name(var)], exp)
tail   ::=  Return(exp) | Goto(label)
         |  If(Compare(atm,[cmp],[atm]), [Goto(label)], [Goto(label)])
-----------------------------------------------------------------------
atm    ::=  GlobalValue(var)
exp    ::=  Subscript(atm,atm,Load()) | Allocate(int,type)
stmt   ::=  Collect(int) | Assign([Subscript(atm,atm,Store())], atm)
-----------------------------------------------------------------------
exp      ::=  FunRef(label) | Call(atm, atm*) | Call(FunRef(label), atm*)
tail     ::=  Return(None)
params   ::=  [(var,type), ...]
block    ::=  label:stmt* tail
def      ::=  FunctionDef(label, params, {block, ...}, None, rtype, None)
C_Fun    ::=  CProgramDefs([def, ...])
```

Figure 8.8
The abstract syntax of $\mathcal{C}_{\mathsf{Fun}}$, extending $\mathcal{C}_{\mathsf{Tup}}$ (figure 7.12).

~~The translation of Return statements requires a new auxiliary function to handle expressions in~~

```
reg    ::= rsp | rbp | rax | rbx | rcx | rdx | rsi | rdi |
           r8 | r9 | r10 | r11 | r12 | r13 | r14 | r15
arg    ::= $int | %reg | int(%reg)
instr  ::= addq arg,arg | subq arg,arg | negq arg | movq arg,arg |
           pushq arg | popq arg | callq label | retq | jmp label |
           label: instr
-------------------------------------------------------------------
bytereg ::= ah | al | bh | bl | ch | cl | dh | dl
arg     ::= %bytereg
cc      ::= e | ne | l | le | g | ge
instr   ::= xorq arg, arg | cmpq arg, arg | setcc arg | movzbq arg, arg
          | jcc label
-------------------------------------------------------------------
arg    ::= label(%rip)
-------------------------------------------------------------------
instr  ::= callq *arg | leaq arg, %reg
block  ::= instr+
def    ::= .globl label (label: block)*
x86Def_callq*  ::= def*
```

Figure 8.9
The concrete syntax of $\text{x86}^{\text{Def}}_{\text{callq}*}$ (extends $\text{x86}_{\text{Global}}$ of figure 7.13).

```
arg       ::= Constant(int) | Reg(reg) | Deref(reg,int) | ByteReg(reg)
            | Global(label) | FunRef(label)
instr     ::= IndirectCallq(arg, int) | Instr('leaq',[arg,Reg(reg)])
block     ::= label: instr*
def       ::= (label,{block, ...})
x86Def_callq*  ::= X86ProgramDefs([def, ...])
```

Figure 8.10
The abstract syntax of $\text{x86}^{\text{Def}}_{\text{callq}*}$ (extends $\text{x86}_{\text{Global}}$ of figure 7.14).

8.8 Select Instructions and the $\text{x86}^{\text{Def}}_{\text{callq}*}$ Language

The output of select instructions is a program in the $\text{x86}^{\text{Def}}_{\text{callq}*}$ language; the definition of its concrete syntax is shown in figure 8.9, and the definition of its abstract syntax is shown in figure 8.10. ~~We use the align directive on the labels of function definitions to make sure~~ 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 *lhs′* is the translation of *lhs* from *atm* in $\mathcal{C}_{\text{Fun}}$ to *arg* in $\text{x86}^{\text{Var,Def}}_{\text{callq}*}$. The `FunRef` becomes a `Global` AST node, whose concrete syntax is instruction-pointer-relative addressing.

lhs `= FunRef(`*f* *n*`);` $\Rightarrow$ `leaq` *f*`(%rip),` *lhs′*

Regarding function definitions, 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.

```
FunctionDef(f, [(x1,T1), ...], B, _, Tr, _)
⇒
FunctionDef(f, [], B′, _, int, _)
```

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 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:

```
start:                    ⇒   f_start:
  instr1                        movq %rdi, x1
  ...                           movq %rsi, x2
  instrn                        ...
                                movq 16(%rbp), x7
                                movq 24(%rbp), x8
                                ...
                                instr1
                                ...
                                instrn
```

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 8.12 in section 8.12, 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 `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. Note that in the general case the function is not given as a label, but its address is produced by the argument arg_0. So, we translate the call into an indirect function call. The return value from the function is stored in `rax`, so it needs to be moved into the *lhs*. If n is an even number greater than 6, the translation looks like this:

lhs `= Call(`arg_0`,` arg_1 $arg_2 \ldots arg_n$`)`
⇒
`movq` arg_1`, %rdi`
`movq` arg_2`, %rsi`
⋮
`movq` arg_6`, %r9`
`pushq` arg_n
⋮
`pushq` arg_7
`callq *`arg_0
`addq $8(`$n-6$`), %rsp`
`movq %rax,` *lhs*

The `IndirectCallq` AST node includes an integer for the arity of the function, that is, the number of parameters. That information is useful in the `uncover_live` pass for determining which argument-passing registers are potentially read during the call.

In the special case where arg_0 is a built-in function or a `FunRef`, we should use a slightly more efficient direct call in place of the indirect call.

~~For tail calls, the parameter passing is the same as non-tail calls: generate instructions to move t~~

8.9 Register Allocation

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

8.9.1 Liveness Analysis

The `IndirectCallq` instruction should be treated like `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 fields of `TailJmp` and `IndirectCallq` determine how many of the argument-passing registers should be considered as read by those instructions. Also, the target field of `TailJmp` and `IndirectCallq` should be included in the set of read locations R.

8.9.2 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).

Recall that in section 7.7 we discussed the need to spill tuple-typed variables that are live during a call to `collect`, the garbage collector. With the addition of functions to our language, we need to revisit this issue. Functions that perform allocation contain calls to the collector. Thus, we should not only spill a tuple-typed variable when it is live during a call to `collect`, but we should spill the variable if it is live during a call to any user-defined function. Thus, in the `build_interference` pass, we recommend adding interference edges between call-live tuple-typed variables and the callee-saved registers (in addition to creating edges between call-live variables and the caller-saved registers).

8.9.3 Allocate Registers

The primary change to the `allocate_registers` pass is adding an auxiliary function for handling definitions (the *def* nonterminal shown in figure 8.10) 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.

8.10 Patch Instructions

In `patch_instructions`, you should deal with the x86 idiosyncrasy that the destination argument of `leaq` must be a register. ~~Additionally, you should ensure that the argument of T~~

8.11 Generate Prelude and Conclusion

~~Now that register allocation is complete, we can translate the TailJmp into a sequence of instructio~~

Regarding function definitions, we generate a prelude and conclusion for each one. 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. Move the root stack pointer `r15` up by the size of the root-stack frame for this function, which depends on the number of spilled tuple-typed variables.
5. Initialize to zero all new entries in the root-stack frame.
6. Jump to the start block.

The prelude of the `main` function has an additional task: call the `initialize` function to set up the garbage collector, and then move the value of the global

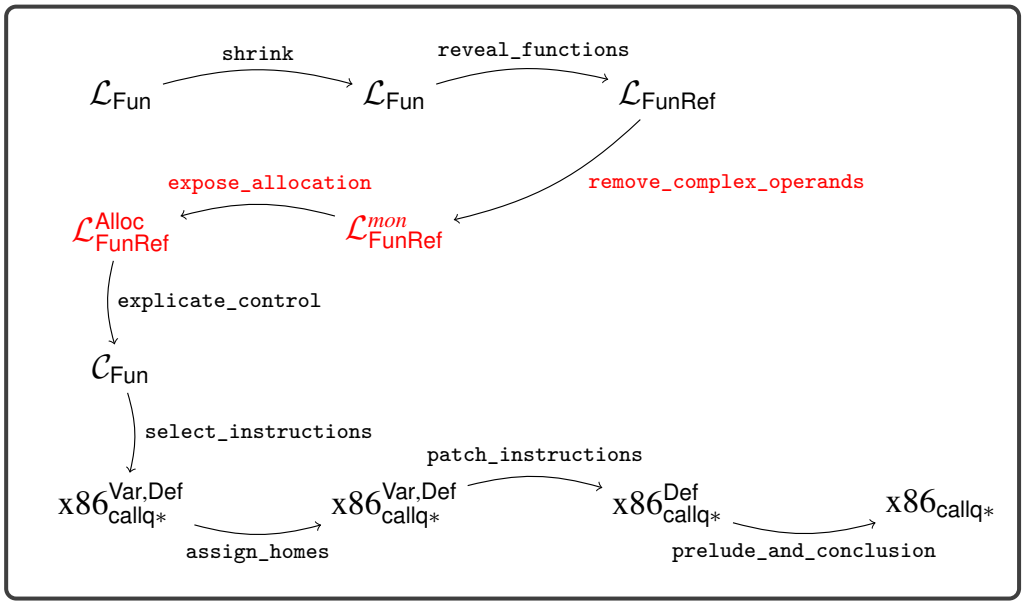


Figure 8.11
Diagram of the passes for $\mathcal{L}_{\text{Fun}}$, a language with functions.

`rootstack_begin` in `r15`. This initialization should happen before step 4, which depends on `r15`.

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. Move the root stack pointer back down by the size of the root-stack frame for this function.
4. Restore `rbp` by popping it from the stack.
5. Return to the caller with the `retq` instruction.

The output of this pass is $\text{x86}_{\text{callq}*}$, which differs from $\text{x86}^{\text{Def}}_{\text{callq}*}$ in that there is no longer an AST node for function definitions. Instead, a program is just a dictionary of basic blocks, as in $\text{x86}_{\text{Global}}$. So we have the following grammar rule:

$$\text{x86}_{\text{callq}*} ::= \texttt{X86Program(\{}\mathit{label}\texttt{:}\ \mathit{instr}^{*}\texttt{, ...\}})$$

Figure 8.11 gives an overview of the passes for compiling $\mathcal{L}_{\text{Fun}}$ to x86.

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, functions that create tuples, and functions that make tail calls. Test your compiler on these new programs and all your previously created test programs.

8.12 An Example Translation

Figure 8.12 shows an example translation of a simple function in $\mathcal{L}_{\mathsf{Fun}}$ to x86. The figure includes the results of `explicate_control` and `select_instructions`.

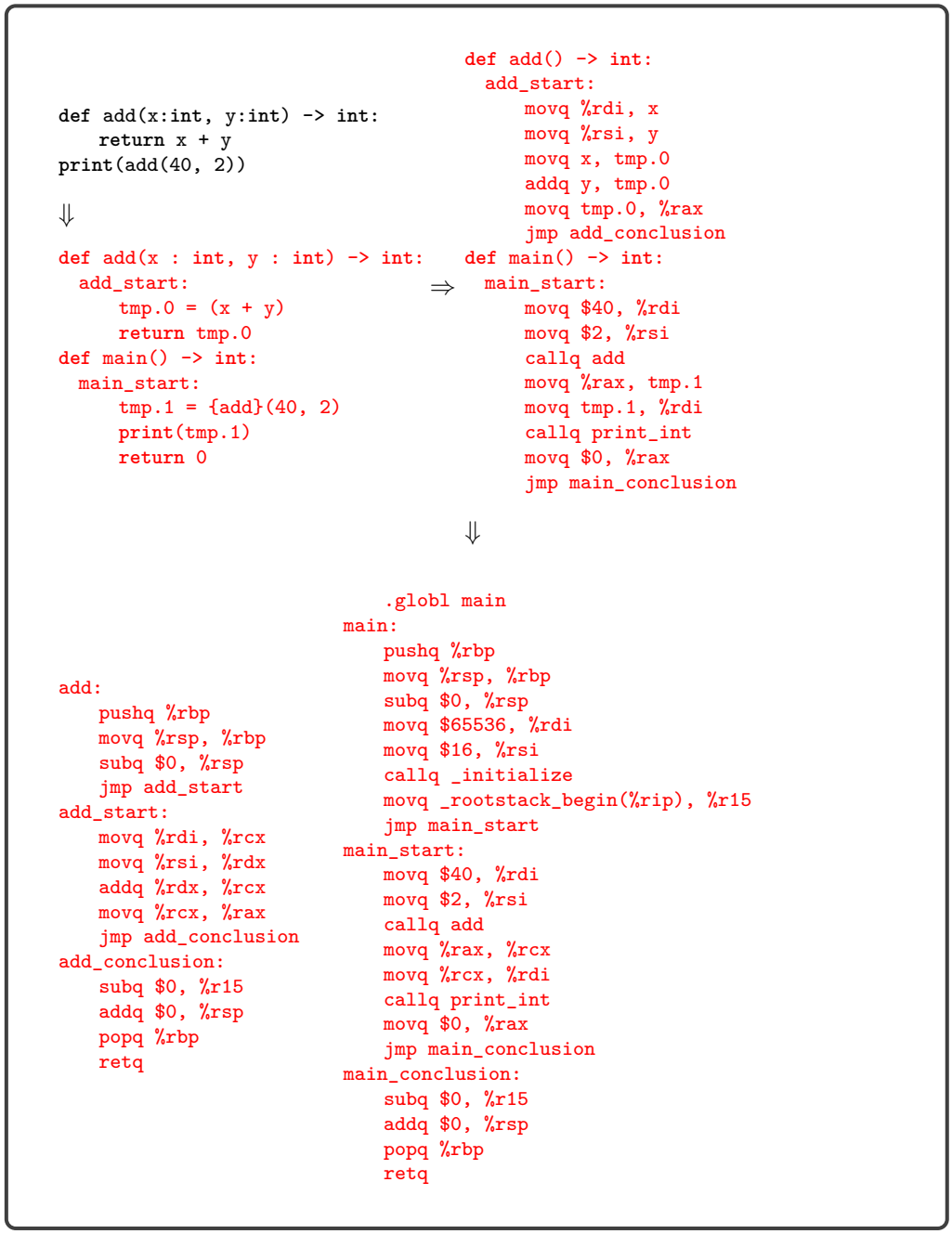


Figure 8.12
Example compilation of a simple function to x86.