Essentials of Compilation
The Incremental, Nano-Pass Approach

Jeremy G. Siek
Indiana University

with contributions from:
Carl Factora
Andre Kuhlenschmidt
Ryan R. Newton
Ryan Scott
Cameron Swords
Michael M. Vitousek
Michael Vollmer

January 3, 2021

ii

This book is dedicated to the programming language wonks at Indiana University.

Contents

List of Figures

Preface

The tradition of compiler writing at Indiana University goes back to research and courses on programming languages by Professor Daniel Friedman in the 1970's and 1980's. Friedman conducted research on lazy evaluation [41] in the context of Lisp [73] and then studied continuations [33] and macros [65] in the context of the Scheme [95], a dialect of Lisp. One of the students of those courses, Kent Dybvig, went on to build Chez Scheme [31], a production-quality and efficient compiler for Scheme. After completing his Ph.D. at the University of North Carolina, he returned to teach at Indiana University. Throughout the 1990's and 2000's, Professor Dybvig continued development of Chez Scheme and taught the compiler course.

The compiler course evolved to incorporate novel pedagogical ideas while also including elements of effective real-world compilers. One of Friedman's ideas was to split the compiler into many small "passes" so that the code for each pass would be easy to understood in isolation. In contrast, most compilers of the time were organized into only a few monolithic passes for reasons of compile-time efficiency. Another idea, called "the game", was to test the code generated by each pass on interpreters for each intermediate language, thereby helping to pinpoint errors in individual passes. Dybvig, with later help from his students Dipanwita Sarkar and Andrew Keep, developed infrastructure to support this approach and evolved the course, first to use smaller micro-passes and then into even smaller nano-passes [84, 58]. I was a student in this compiler course in the early 2000's as part of my Ph.D. studies at Indiana University. Needless to say, I enjoyed the course immensely!

During that time, another graduate student named Abdulaziz Ghuloum observed that the front-to-back organization of the course made it difficult for students to understand the rationale for the compiler design. Ghuloum proposed an incremental approach in which the students start by implementing a complete compiler for a very small subset of the language. In each subsequent stage they add a feature to the language and then add or

modify passes to handle the new feature [44]. In this way, the students see how the language features motivate aspects of the compiler design.

After graduating from Indiana University in 2005, I went on to teach at the University of Colorado. I adapted the nano-pass and incremental approaches to compiling a subset of the Python language [89]. I very much enjoyed teaching the course organized in this way, and even better, many of the students learned a lot and got excited about compilers.

I returned to Indiana University in 2013. In my absence the compiler course had switched from the front-to-back organization to a back-to-front [32]. While that organization also works well, I prefer the incremental approach and started porting and adapting the structure of the Colorado course back into the land of Scheme. In the meantime Indiana University had moved on from Scheme to Racket [38], so the course is now about compiling a subset of Racket (and Typed Racket) to the x86 assembly language.

This is the textbook for the incremental version of the compiler course at Indiana University (Spring 2016 - present). With this book I hope to make the Indiana compiler course available to people that have not had the chance to study compilers at Indiana University.

Prerequisites

The material in this book is challenging but rewarding. It is meant to prepare students for a lifelong career in programming languages.

The book uses the Racket language both for the implementation of the compiler and for the language that is compiled, so a student should be proficient with Racket or Scheme prior to reading this book. There are many excellent resources for learning Scheme and Racket [30, 1, 40, 34, 35, 39].

It is helpful but not necessary for the student to have prior exposure to the x86 assembly language [53], as one might obtain from a computer systems course [17]. This book introduces the parts of x86-64 assembly language that are needed. We follow the System V calling conventions [16, 72], which means that the assembly code that we generate will work properly with our runtime system (written in C) when it is compiled using the GNU C compiler (`gcc`) on the Linux and MacOS operating systems. (Minor adjustments are needed for MacOS which we note as they arise.) The GNU C compiler, when running on the Microsoft Windows operating system, follows the Microsoft x64 calling convention [74, 75]. So the assembly code that we generate will *not* work properly with our runtime system on Windows. One option to consider for using a Windows computer is to run a virtual machine

with Linux as the guest operating system.

Acknowledgments

Many people have contributed to the ideas, techniques, and organization of this book and have taught courses based on it. Many of the compiler design decisions in this book are drawn from the assignment descriptions of [32]. We also would like to thank John Clements, Bor-Yuh Evan Chang, Daniel P. Friedman, Ronald Garcia, Abdulaziz Ghuloum, Jay McCarthy, Nate Nystrom, Dipanwita Sarkar, Oscar Waddell, and Michael Wollowski.

Jeremy G. Siek
`http://homes.soic.indiana.edu/jsiek`

1

Preliminaries

In this chapter we review the basic tools that are needed to implement a compiler. Programs are typically input by a programmer as text, i.e., a sequence of characters. The program-as-text representation is called *concrete syntax*. We use concrete syntax to concisely write down and talk about programs. Inside the compiler, we use *abstract syntax trees* (ASTs) to represent programs in a way that efficiently supports the operations that the compiler needs to perform. The translation from concrete syntax to abstract syntax is a process called *parsing* [2]. We do not cover the theory and implementation of parsing in this book. A parser is provided in the supporting materials for translating from concrete to abstract syntax.

ASTs can be represented in many different ways inside the compiler, depending on the programming language used to write the compiler. We use Racket's `struct` feature to represent ASTs (Section 1.1). We use grammars to define the abstract syntax of programming languages (Section 1.2) and pattern matching to inspect individual nodes in an AST (Section 1.3). We use recursive functions to construct and deconstruct ASTs (Section 1.4). This chapter provides an brief introduction to these ideas.

1.1   Abstract Syntax Trees and Racket Structures

Compilers use abstract syntax trees to represent programs because they often need to ask questions like: for a given part of a program, what kind of language feature is it? What are its sub-parts? Consider the program on the left and its AST on the right. This program is an addition operation and it has two sub-parts, a read operation and a negation. The negation has another sub-part, the integer constant `8`. By using a tree to represent

the program, we can easily follow the links to go from one part of a program to its sub-parts.

```
(+ (read) (- 8))
```
                                                                          (1.1)
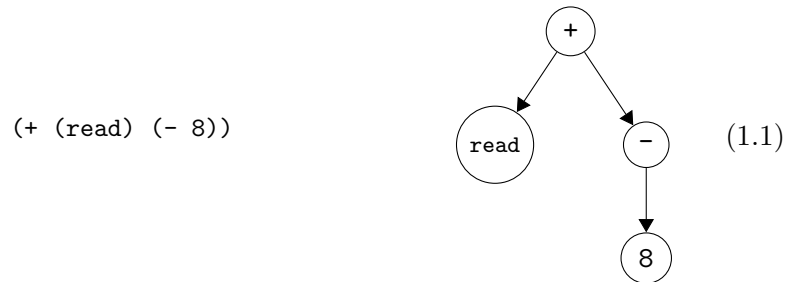
We use the standard terminology for trees to describe ASTs: each circle above is called a *node*. The arrows connect a node to its *children* (which are also nodes). The top-most node is the *root*. Every node except for the root has a *parent* (the node it is the child of). If a node has no children, it is a *leaf* node. Otherwise it is an *internal* node.

We define a Racket `struct` for each kind of node. For this chapter we require just two kinds of nodes: one for integer constants and one for primitive operations. The following is the `struct` definition for integer constants.

```
(struct Int (value))
```

An integer node includes just one thing: the integer value. To create a AST node for the integer 8, we write (`Int 8`).

```
(define eight (Int 8))
```

We say that the value created by (`Int 8`) is an *instance* of the `Int` structure. The following is the `struct` definition for primitives operations.

```
(struct Prim (op args))
```

A primitive operation node includes an operator symbol `op` and a list of children `args`. For example, to create an AST that negates the number 8, we write (`Prim '- (list eight)`).

```
(define neg-eight (Prim '- (list eight)))
```

Primitive operations may have zero or more children. The `read` operator has zero children:

```
(define rd (Prim 'read '()))
```

whereas the addition operator has two children:

```
(define ast1.1 (Prim '+ (list rd neg-eight)))
```

We have made a design choice regarding the `Prim` structure. Instead of using one structure for many different operations (`read`, `+`, and `-`), we could have instead defined a structure for each operation, as follows.

```
(struct Read ())
(struct Add (left right))
(struct Neg (value))
```

The reason we choose to use just one structure is that in many parts of the compiler the code for the different primitive operators is the same, so we might as well just write that code once, which is enabled by using a single structure.

When compiling a program such as (1.1), we need to know that the operation associated with the root node is addition and we need to be able to access its two children. Racket provides pattern matching to support these kinds of queries, as we see in Section 1.3.

In this book, we often write down the concrete syntax of a program even when we really have in mind the AST because the concrete syntax is more concise. We recommend that, in your mind, you always think of programs as abstract syntax trees.

1.2 Grammars

A programming language can be thought of as a *set* of programs. The set is typically infinite (one can always create larger and larger programs), so one cannot simply describe a language by listing all of the programs in the language. Instead we write down a set of rules, a *grammar*, for building programs. Grammars are often used to define the concrete syntax of a language, but they can also be used to describe the abstract syntax. We write our rules in a variant of Backus-Naur Form (BNF) [9, 64]. As an example, we describe a small language, named $R_{\texttt{Int}}$, that consists of integers and arithmetic operations.

The first grammar rule for the abstract syntax of $R_{\texttt{Int}}$ says that an instance of the `Int` structure is an expression:

$$exp ::= (\texttt{Int}\ int) \tag{1.2}$$

Each rule has a left-hand-side and a right-hand-side. The way to read a rule is that if you have an AST node that matches the right-hand-side, then you can categorize it according to the left-hand-side. A name such as *exp* that is defined by the grammar rules is a *non-terminal*. The name *int* is a also

a non-terminal, but instead of defining it with a grammar rule, we define it with the following explanation. We make the simplifying design decision that all of the languages in this book only handle machine-representable integers. On most modern machines this corresponds to integers represented with 64-bits, i.e., the in range -2^{63} to $2^{63} - 1$. We restrict this range further to match the Racket `fixnum` datatype, which allows 63-bit integers on a 64-bit machine. So an *int* is a sequence of decimals (0 to 9), possibly starting with $-$ (for negative integers), such that the sequence of decimals represent an integer in range -2^{62} to $2^{62} - 1$.

The second grammar rule is the `read` operation that receives an input integer from the user of the program.

$$exp ::= (\texttt{Prim read ()}) \tag{1.3}$$

The third rule says that, given an *exp* node, the negation of that node is also an *exp*.

$$exp ::= (\texttt{Prim - (}exp\texttt{))} \tag{1.4}$$

Symbols in typewriter font such as `-` and `read` are *terminal* symbols and must literally appear in the program for the rule to be applicable.

We can apply these rules to categorize the ASTs that are in the $R_{\texttt{Int}}$ language. For example, by rule (1.2) (`Int 8`) is an *exp*, then by rule (1.4) the following AST is an *exp*.

$$\texttt{(Prim '- (list (Int 8)))} \qquad\qquad \tag{1.5}$$

The next grammar rule is for addition expressions:

$$exp ::= (\texttt{Prim + (}exp\ exp\texttt{))} \tag{1.6}$$

We can now justify that the AST (1.1) is an *exp* in $R_{\texttt{Int}}$. We know that (`Prim 'read '()`) is an *exp* by rule (1.3) and we have already categorized (`Prim '- (list (Int 8))`) as an *exp*, so we apply rule (1.6) to show that

`(Prim '+ (list (Prim 'read '()) (Prim '- (list (Int 8)))))`

is an *exp* in the $R_{\texttt{Int}}$ language.

If you have an AST for which the above rules do not apply, then the AST is not in $R_{\texttt{Int}}$. For example, the program (`- (read) (+ 8)`) is not in $R_{\texttt{Int}}$ because there are no rules for `+` with only one argument, nor for `-`

$$\begin{array}{rcl} \textit{exp} & ::= & \textit{int} \mid (\texttt{read}) \mid (\texttt{- } \textit{exp}) \mid (\texttt{+ } \textit{exp} \ \textit{exp}) \\ R_{\texttt{Int}} & ::= & \textit{exp} \end{array}$$

Figure 1.1: The concrete syntax of $R_{\texttt{Int}}$.

$$\begin{array}{rcl} \textit{exp} & ::= & (\texttt{Int } \textit{int}) \mid (\texttt{Prim read ()}) \mid (\texttt{Prim - } (\textit{exp})) \\ & \mid & (\texttt{Prim + } (\textit{exp} \ \textit{exp})) \\ R_{\texttt{Int}} & ::= & (\texttt{Program '() } \textit{exp}) \end{array}$$

Figure 1.2: The abstract syntax of $R_{\texttt{Int}}$.

with two arguments. Whenever we define a language with a grammar, the language only includes those programs that are justified by the rules.

The last grammar rule for $R_{\texttt{Int}}$ states that there is a `Program` node to mark the top of the whole program:

$$R_{\texttt{Int}} ::= (\texttt{Program '() } \textit{exp})$$

The `Program` structure is defined as follows

```
(struct Program (info body))
```

where `body` is an expression. In later chapters, the `info` part will be used to store auxiliary information but for now it is just the empty list.

It is common to have many grammar rules with the same left-hand side but different right-hand sides, such as the rules for *exp* in the grammar of $R_{\texttt{Int}}$. As a short-hand, a vertical bar can be used to combine several right-hand-sides into a single rule.

We collect all of the grammar rules for the abstract syntax of $R_{\texttt{Int}}$ in Figure 1.2. The concrete syntax for $R_{\texttt{Int}}$ is defined in Figure 1.1.

The `read-program` function provided in `utilities.rkt` of the support materials reads a program in from a file (the sequence of characters in the concrete syntax of Racket) and parses it into an abstract syntax tree. See the description of `read-program` in Appendix 12.2 for more details.

1.3 Pattern Matching

As mentioned in Section 1.1, compilers often need to access the parts of an AST node. Racket provides the `match` form to access the parts of a structure. Consider the following example and the output on the right.

```
(match ast1.1
  [(Prim op (list child1 child2))
    (print op)])                                    '+
```

In the above example, the `match` form takes an AST (1.1) and binds its parts
to the three pattern variables `op`, `child1`, and `child2`, and then prints out
the operator. In general, a match clause consists of a *pattern* and a *body*.
Patterns are recursively defined to be either a pattern variable, a structure
name followed by a pattern for each of the structure's arguments, or an S-
expression (symbols, lists, etc.). (See Chapter 12 of The Racket Guide[1] and
Chapter 9 of The Racket Reference[2] for a complete description of `match`.)
The body of a match clause may contain arbitrary Racket code. The pattern
variables can be used in the scope of the body, such as `op` in (`print op`).

A `match` form may contain several clauses, as in the following function
`leaf?` that recognizes when an $R_{\texttt{Int}}$ node is a leaf in the AST. The `match`
proceeds through the clauses in order, checking whether the pattern can
match the input AST. The body of the first clause that matches is executed.
The output of `leaf?` for several ASTs is shown on the right.

```
(define (leaf? arith)
  (match arith
    [(Int n) #t]
    [(Prim 'read '()) #t]
    [(Prim '- (list e1)) #f]
    [(Prim '+ (list e1 e2)) #f]))

(leaf? (Prim 'read '()))                         #t
(leaf? (Prim '- (list (Int 8))))                 #f
(leaf? (Int 8))                                  #t
```

When writing a `match`, we refer to the grammar definition to identify
which non-terminal we are expecting to match against, then we make sure
that 1) we have one clause for each alternative of that non-terminal and 2)
that the pattern in each clause corresponds to the corresponding right-hand
side of a grammar rule. For the `match` in the `leaf?` function, we refer to the
grammar for $R_{\texttt{Int}}$ in Figure 1.2. The *exp* non-terminal has 4 alternatives,
so the `match` has 4 clauses. The pattern in each clause corresponds to the
right-hand side of a grammar rule. For example, the pattern (`Prim '+
(list e1 e2)`) corresponds to the right-hand side (`Prim + (`*exp exp*`)`).

---

[1]`https://docs.racket-lang.org/guide/match.html`
[2]`https://docs.racket-lang.org/reference/match.html`

When translating from grammars to patterns, replace non-terminals such as *exp* with pattern variables of your choice (e.g. `e1` and `e2`).

1.4 Recursive Functions

Programs are inherently recursive. For example, an $R_{\texttt{Int}}$ expression is often made of smaller expressions. Thus, the natural way to process an entire program is with a recursive function. As a first example of such a recursive function, we define `exp?` below, which takes an arbitrary value and determines whether or not it is an $R_{\texttt{Int}}$ expression. We say that a function is defined by *structural recursion* when it is defined using a sequence of match clauses that correspond to a grammar, and the body of each clause makes a recursive call on each child node.[3] Below we also define a second function, named `Rint?`, that determines whether an AST is an $R_{\texttt{Int}}$ program. In general we can expect to write one recursive function to handle each non-terminal in a grammar.

```
(define (exp? ast)
  (match ast
    [(Int n) #t]
    [(Prim 'read '()) #t]
    [(Prim '- (list e)) (exp? e)]
    [(Prim '+ (list e1 e2))
      (and (exp? e1) (exp? e2))]
    [else #f]))

(define (Rint? ast)
  (match ast
    [(Program '() e) (exp? e)]
    [else #f]))

(Rint? (Program '() ast1.1)                     #t
(Rint? (Program '()                             #f
       (Prim '- (list (Prim 'read '())
                     (Prim '+ (list (Num 8)))))))))
```

You may be tempted to merge the two functions into one, like this:

---

[3]This principle of structuring code according to the data definition is advocated in the book *How to Design Programs* `http://www.ccs.neu.edu/home/matthias/HtDP2e/`.

```
(define (Rint? ast)
  (match ast
    [(Int n) #t]
    [(Prim 'read '()) #t]
    [(Prim '- (list e)) (Rint? e)]
    [(Prim '+ (list e1 e2)) (and (Rint? e1) (Rint? e2))]
    [(Program '() e) (Rint? e)]
    [else #f]))
```

Sometimes such a trick will save a few lines of code, especially when it comes to the `Program` wrapper. Yet this style is generally *not* recommended because it can get you into trouble. For example, the above function is subtly wrong: `(Rint? (Program '() (Program '() (Int 3))))` returns true when it should return false.

1.5   Interpreters

In general, the intended behavior of a program is defined by the specification of the language. For example, the Scheme language is defined in the report by [92]. The Racket language is defined in its reference manual [38]. In this book we use interpreters to specify each language that we consider. An interpreter that is designated as the definition of a language is called a *definitional interpreter* [82]. We warm up by creating a definitional interpreter for the R_{Int} language, which serves as a second example of structural recursion. The `interp-Rint` function is defined in Figure 1.3. The body of the function is a match on the input program followed by a call to the `interp-exp` helper function, which in turn has one match clause per grammar rule for R_{Int} expressions.

Let us consider the result of interpreting a few R_{Int} programs. The following program adds two integers.

```
(+ 10 32)
```

The result is `42`, the answer to life, the universe, and everything: `42`![4]. We wrote the above program in concrete syntax whereas the parsed abstract syntax is:

```
(Program '() (Prim '+ (list (Int 10) (Int 32))))
```

The next example demonstrates that expressions may be nested within each other, in this case nesting several additions and negations.

---

[4] *The Hitchhiker's Guide to the Galaxy* by Douglas Adams.

```
(define (interp-exp e)
  (match e
    [(Int n) n]
    [(Prim 'read '())
     (define r (read))
     (cond [(fixnum? r) r]
           [else (error 'interp-exp "read expected an integer" r)])]
    [(Prim '- (list e))
     (define v (interp-exp e))
     (fx- 0 v)]
    [(Prim '+ (list e1 e2))
     (define v1 (interp-exp e1))
     (define v2 (interp-exp e2))
     (fx+ v1 v2)]))

(define (interp-Rint p)
  (match p
    [(Program '() e) (interp-exp e)]))
```

Figure 1.3: Interpreter for the $R_{\texttt{Int}}$ language.

```
(+ 10 (- (+ 12 20)))
```

What is the result of the above program?

As mentioned previously, the $R_{\texttt{Int}}$ language does not support arbitrarily-large integers, but only 63-bit integers, so we interpret the arithmetic operations of $R_{\texttt{Int}}$ using fixnum arithmetic in Racket. Suppose

$$n = 999999999999999999$$

which indeed fits in 63-bits. What happens when we run the following program in our interpreter?

```
(+ (+ (+ n n) (+ n n)) (+ (+ n n) (+ n n)))))
```

It produces an error:

```
fx+: result is not a fixnum
```

We establish the convention that if running the definitional interpreter on a program produces an error then the meaning of that program is *unspecified*, unless the error is a `trapped-error`. A compiler for the language is under no obligations regarding programs with unspecified behavior; it does not have to produce an executable, and if it does, that executable can do anything. On the other hand, if the error is a `trapped-error`, then the compiler must

produce an executable and it is required to report that an error occurred. To signal an error, exit with a return code of `255`. The interpreters in chapters 8 and 10 use `trapped-error`.

Moving on to the last feature of the R_{Int} language, the `read` operation prompts the user of the program for an integer. Recall that program (1.1) performs a `read` and then subtracts `8`. So if we run

```
(interp-Rint (Program '() ast1.1))
```

and if the input is `50`, the result is `42`.

We include the `read` operation in R_{Int} so a clever student cannot implement a compiler for R_{Int} that simply runs the interpreter during compilation to obtain the output and then generates the trivial code to produce the output. (Yes, a clever student did this in the first instance of this course.)

The job of a compiler is to translate a program in one language into a program in another language so that the output program behaves the same way as the input program does. This idea is depicted in the following diagram. Suppose we have two languages, $\mathcal{L}_1$ and $\mathcal{L}_2$, and a definitional interpreter for each language. Given a compiler that translates from language $\mathcal{L}_1$ to $\mathcal{L}_2$ and given any program P_1 in $\mathcal{L}_1$, the compiler must translate it into some program P_2 such that interpreting P_1 and P_2 on their respective interpreters with same input i yields the same output o.

$$
\begin{array}{ccc}
P_1 & \xrightarrow{\text{compile}} & P_2 \\[2ex]
{\scriptstyle\text{interp-}\mathcal{L}_1(i)}\searrow & & \downarrow{\scriptstyle\text{interp-}\mathcal{L}_2(i)} \\[2ex]
 & o &
\end{array}
\qquad (1.7)
$$
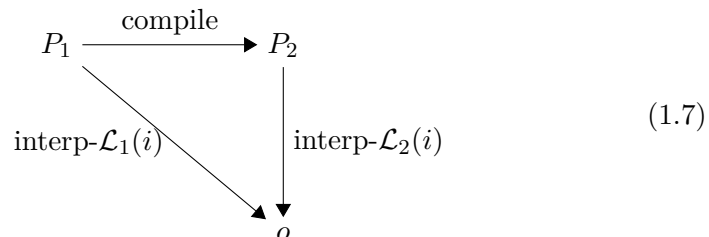
In the next section we see our first example of a compiler.

1.6 Example Compiler: a Partial Evaluator

In this section we consider a compiler that translates R_{Int} programs into R_{Int} programs that may be more efficient, that is, this compiler is an optimizer. This optimizer eagerly computes the parts of the program that do not depend on any inputs, a process known as *partial evaluation* [55]. For example, given the following program

```
(+ (read) (- (+ 5 3)))
```

```
(define (pe-neg r)
  (match r
    [(Int n) (Int (fx- 0 n))]
    [else (Prim '- (list r))]))

(define (pe-add r1 r2)
  (match* (r1 r2)
    [((Int n1) (Int n2)) (Int (fx+ n1 n2))]
    [(_ _) (Prim '+ (list r1 r2))]))

(define (pe-exp e)
  (match e
    [(Int n) (Int n)]
    [(Prim 'read '()) (Prim 'read '())]
    [(Prim '- (list e1)) (pe-neg (pe-exp e1))]
    [(Prim '+ (list e1 e2)) (pe-add (pe-exp e1) (pe-exp e2))]))

(define (pe-Rint p)
  (match p
    [(Program '() e) (Program '() (pe-exp e))]))
```

Figure 1.4: A partial evaluator for $R_{\texttt{Int}}$.

our compiler will translate it into the program

```
(+ (read) -8)
```

Figure 1.4 gives the code for a simple partial evaluator for the $R_{\texttt{Int}}$ language. The output of the partial evaluator is an $R_{\texttt{Int}}$ program. In Figure 1.4, the structural recursion over *exp* is captured in the `pe-exp` function whereas the code for partially evaluating the negation and addition operations is factored into two separate helper functions: `pe-neg` and `pe-add`. The input to these helper functions is the output of partially evaluating the children.

The `pe-neg` and `pe-add` functions check whether their arguments are integers and if they are, perform the appropriate arithmetic. Otherwise, they create an AST node for the arithmetic operation.

To gain some confidence that the partial evaluator is correct, we can test whether it produces programs that get the same result as the input programs. That is, we can test whether it satisfies Diagram 1.7. The following code runs the partial evaluator on several examples and tests the output program. The `parse-program` and `assert` functions are defined in

Appendix 12.2.

```
(define (test-pe p)
  (assert "testing pe-Rint"
    (equal? (interp-Rint p) (interp-Rint (pe-Rint p)))))

(test-pe (parse-program `(program () (+ 10 (- (+ 5 3))))))
(test-pe (parse-program `(program () (+ 1 (+ 3 1)))))
(test-pe (parse-program `(program () (- (+ 3 (- 5))))))
```

2

Integers and Variables

This chapter is about compiling a subset of Racket to x86-64 assembly code [53]. The subset, named $R_{\mathtt{Var}}$, includes integer arithmetic and local variable binding. We often refer to x86-64 simply as x86. The chapter begins with a description of the $R_{\mathtt{Var}}$ language (Section 2.1) followed by an introduction to of x86 assembly (Section 2.2). The x86 assembly language is large so we discuss only the instructions needed for compiling $R_{\mathtt{Var}}$. We introduce more x86 instructions in later chapters. After introducing $R_{\mathtt{Var}}$ and x86, we reflect on their differences and come up with a plan to break down the translation from $R_{\mathtt{Var}}$ to x86 into a handful of steps (Section 2.3). The rest of the sections in this chapter give detailed hints regarding each step (Sections 2.4 through 2.9). We hope to give enough hints that the well-prepared reader, together with a few friends, can implement a compiler from $R_{\mathtt{Var}}$ to x86 in a couple weeks. To give the reader a feeling for the scale of this first compiler, the instructor solution for the $R_{\mathtt{Var}}$ compiler is approximately 500 lines of code.

2.1   The $R_{\mathtt{Var}}$ Language

The $R_{\mathtt{Var}}$ language extends the $R_{\mathtt{Int}}$ language with variable definitions. The concrete syntax of the $R_{\mathtt{Var}}$ language is defined by the grammar in Figure 2.1 and the abstract syntax is defined in Figure 2.2. The non-terminal *var* may be any Racket identifier. As in $R_{\mathtt{Int}}$, `read` is a nullary operator, `-` is a unary operator, and `+` is a binary operator. Similar to $R_{\mathtt{Int}}$, the abstract syntax of $R_{\mathtt{Var}}$ includes the `Program` struct to mark the top of the program. Despite the simplicity of the $R_{\mathtt{Var}}$ language, it is rich enough to exhibit several compilation techniques.

$$
\begin{array}{rcl}
exp & ::= & int \mid (\texttt{read}) \mid (\texttt{-}\ exp) \mid (\texttt{+}\ exp\ exp) \\
    & \mid & var \mid (\texttt{let}\ ([var\ exp])\ exp) \\
R_{\texttt{Var}} & ::= & exp
\end{array}
$$

Figure 2.1: The concrete syntax of $R_{\texttt{Var}}$.

$$
\begin{array}{rcl}
exp & ::= & (\texttt{Int}\ int) \mid (\texttt{Prim}\ \texttt{read}\ ()) \\
    & \mid & (\texttt{Prim}\ \texttt{-}\ (exp)) \mid (\texttt{Prim}\ \texttt{+}\ (exp\ exp)) \\
    & \mid & (\texttt{Var}\ var) \mid (\texttt{Let}\ var\ exp\ exp) \\
R_{\texttt{Var}} & ::= & (\texttt{Program}\ \texttt{'()}\ exp)
\end{array}
$$

Figure 2.2: The abstract syntax of $R_{\texttt{Var}}$.

Let us dive further into the syntax and semantics of the $R_{\texttt{Var}}$ language. The `let` feature defines a variable for use within its body and initializes the variable with the value of an expression. The abstract syntax for `let` is defined in Figure 2.2. The concrete syntax for `let` is

```
(let ([var exp]) exp)
```

For example, the following program initializes x to 32 and then evaluates the body (+ 10 x), producing 42.

```
(let ([x (+ 12 20)]) (+ 10 x))
```

When there are multiple `let`'s for the same variable, the closest enclosing `let` is used. That is, variable definitions overshadow prior definitions. Consider the following program with two `let`'s that define variables named x. Can you figure out the result?

```
(let ([x 32]) (+ (let ([x 10]) x) x))
```

For the purposes of depicting which variable uses correspond to which definitions, the following shows the x's annotated with subscripts to distinguish them. Double check that your answer for the above is the same as your answer for this annotated version of the program.

```
(let ([x₁ 32]) (+ (let ([x₂ 10]) x₂) x₁))
```

The initializing expression is always evaluated before the body of the `let`, so in the following, the `read` for x is performed before the `read` for y. Given the input 52 then 10, the following produces 42 (not -42).

```
(let ([x (read)]) (let ([y (read)]) (+ x (- y))))
```

2.1.1 Extensible Interpreters via Method Overriding

To prepare for discussing the interpreter for R_{Var}, we need to explain why we choose to implement the interpreter using object-oriented programming, that is, as a collection of methods inside of a class. Throughout this book we define many interpreters, one for each of the languages that we study. Because each language builds on the prior one, there is a lot of commonality between their interpreters. We want to write down those common parts just once instead of many times. A naive approach would be to have, for example, the interpreter for R_{If} handle all of the new features in that language and then have a default case that dispatches to the interpreter for R_{Var}. The following code sketches this idea.

```
(define (interp-Rvar e)
  (match e
    [(Prim '- (list e))
     (fx- 0 (interp-Rvar e))]
    ...))
```

```
(define (interp-Rif e)
  (match e
    [(If cnd thn els)
     (match (interp-Rif cnd)
       [#t (interp-Rif thn)]
       [#f (interp-Rif els)])]
    ...
    [else (interp-Rvar e)]))
```

The problem with this approach is that it does not handle situations in which an R_{If} feature, like If, is nested inside an R_{Var} feature, like the - operator, as in the following program.

```
(Prim '- (list (If (Bool #t) (Int 42) (Int 0))))
```

If we invoke `interp-Rif` on this program, it dispatches to `interp-Rvar` to handle the - operator, but then it recurisvely calls `interp-Rvar` again on the argument of -, which is an If. But there is no case for If in `interp-Rvar`, so we get an error!

To make our interpreters extensible we need something called *open recursion*, where the tying of the recursive knot is delayed to when the functions are composed. Object-oriented languages provide open recursion with the late-binding of overridden methods. The following code sketches this idea for interpreting R_{Var} and R_{If} using the `class` feature of Racket. We define one class for each language and define a method for interpreting expressions inside each class. The class for R_{If} inherits from the class for R_{Var} and the method `interp-exp` in R_{If} overrides the `interp-exp` in R_{Var}. Note that the default case of `interp-exp` in R_{If} uses `super` to invoke `interp-exp`, and because R_{If} inherits from R_{Var}, that dispatches to the `interp-exp` in R_{Var}.

```
                                    (define interp-Rif-class
                                     (class interp-Rvar-class
    (define interp-Rvar-class         (define/override (interp-exp e)
      (class object%                    (match e
        (define/public (interp-exp e)     [(If cnd thn els)
          (match e                         (match (interp-exp cnd)
            [(Prim '- (list e))              [#t (interp-exp thn)]
             (fx- 0 (interp-exp e))]         [#f (interp-exp els)])]
            ...))                         ...
        ...))                             [else (super interp-exp e)]))
                                        ...
                                      ))
```

Getting back to the troublesome example, repeated here:

```
(define e0 (Prim '- (list (If (Bool #t) (Int 42) (Int 0)))))
```

We can invoke the `interp-exp` method for $R_{\texttt{If}}$ on this expression by creating an object of the $R_{\texttt{If}}$ class and sending it the `interp-exp` method with the argument `e0`.

```
(send (new interp-Rif-class) interp-exp e0)
```

The default case of `interp-exp` in $R_{\texttt{If}}$ handles it by dispatching to the `interp-exp` method in $R_{\texttt{Var}}$, which handles the - operator. But then for the recursive method call, it dispatches back to `interp-exp` in $R_{\texttt{If}}$, where the `If` is handled correctly. Thus, method overriding gives us the open recursion that we need to implement our interpreters in an extensible way.

2.1.2   Definitional Interpreter for $R_{\mathtt{Var}}$

Having justified the use of classes and methods to implement interpreters, we turn to the definitional interpreter for $R_{\mathtt{Var}}$ in Figure 2.3. It is similar to the interpreter for $R_{\mathtt{Int}}$ but adds two new `match` cases for variables and `let`. For `let` we need a way to communicate the value bound to a variable to all the uses of the variable. To accomplish this, we maintain a mapping from variables to values. Throughout the compiler we often need to map variables to information about them. We refer to these mappings as *environments*.[1] For simplicity, we use an association list (alist) to represent the environment. The sidebar to the right gives a brief introduction to alists and the `racket/dict` package. The `interp-exp` function takes the current environment, `env`, as an extra parameter. When the interpreter encounters a variable, it finds the corresponding value using the `dict-ref` function. When the interpreter encounters a `Let`, it evaluates the initializing expression, extends the environment with the result value bound to the variable, using `dict-set`, then evaluates the body of the `Let`.

> **Association Lists as Dictionaries**
>
> An *association list* (alist) is a list of key-value pairs. For example, we can map people to their ages with an alist.
>
> ```
> (define ages
>   '((jane . 25) (sam . 24) (kate . 45)))
> ```
>
> The *dictionary* interface is for mapping keys to values. Every alist implements this interface. The package `racket/dict` provides many functions for working with dictionaries. Here are a few of them:
>
> **(dict-ref** *dict key***)** returns the value associated with the given *key*.
>
> **(dict-set** *dict key val***)** returns a new dictionary that maps *key* to *val* but otherwise is the same as *dict*.
>
> **(in-dict** *dict***)** returns the sequence of keys and values in *dict*. For example, the following creates a new alist in which the ages are incremented.
> ```
> (for/list ([(k v) (in-dict ages)])
>   (cons k (add1 v)))
> ```

The goal for this chapter is to implement a compiler that translates any program P_1 written in the $R_{\mathtt{Var}}$ language into an x86 assembly program P_2 such that P_2 exhibits the same behavior when run on a computer as the P_1 program interpreted by `interp-Rvar`. That is, they output the same integer n. We depict this correctness criteria in the following diagram.

$$P_1 \xrightarrow{\ \text{compile}\ } P_2$$

interp-Rvar   interp-x86int

$$n$$

---
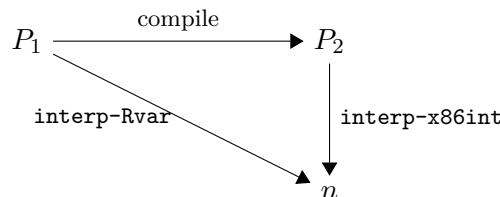
[1] Another common term for environment in the compiler literature is *symbol table*.

```
(define interp-Rvar-class
  (class object%
    (super-new)

    (define/public ((interp-exp env) e)
      (match e
        [(Int n) n]
        [(Prim 'read '())
         (define r (read))
         (cond [(fixnum? r) r]
               [else (error 'interp-exp "expected an integer" r)])]
        [(Prim '- (list e)) (fx- 0 ((interp-exp env) e))]
        [(Prim '+ (list e1 e2))
         (fx+ ((interp-exp env) e1) ((interp-exp env) e2))]
        [(Var x) (dict-ref env x)]
        [(Let x e body)
         (define new-env (dict-set env x ((interp-exp env) e)))
         ((interp-exp new-env) body)]))

    (define/public (interp-program p)
      (match p
        [(Program '() e) ((interp-exp '()) e)]))
  ))

(define (interp-Rvar p)
  (send (new interp-Rvar-class) interp-program p))
```

Figure 2.3: Interpreter for the R_{Var} language.

$$
\begin{array}{lll}
reg & ::= & \texttt{rsp} \mid \texttt{rbp} \mid \texttt{rax} \mid \texttt{rbx} \mid \texttt{rcx} \mid \texttt{rdx} \mid \texttt{rsi} \mid \texttt{rdi} \mid \\
& & \texttt{r8} \mid \texttt{r9} \mid \texttt{r10} \mid \texttt{r11} \mid \texttt{r12} \mid \texttt{r13} \mid \texttt{r14} \mid \texttt{r15} \\
arg & ::= & \texttt{\$}int \mid \texttt{\%}reg \mid int\texttt{(\%}reg\texttt{)} \\
instr & ::= & \texttt{addq}\ arg, arg \mid \texttt{subq}\ arg, arg \mid \texttt{negq}\ arg \mid \texttt{movq}\ arg, arg \mid \\
& & \texttt{callq}\ label \mid \texttt{pushq}\ arg \mid \texttt{popq}\ arg \mid \texttt{retq} \mid \texttt{jmp}\ label \\
& & label\texttt{:}\ instr \\
\texttt{x86}_{\texttt{Int}} & ::= & \texttt{.globl main} \\
& & \texttt{main:}\ instr \dots
\end{array}
$$

Figure 2.4: The syntax of the x86_Int assembly language (AT&T syntax).

In the next section we introduce the x86_Int subset of x86 that suffices for compiling $R_{\texttt{Var}}$.

2.2 The x86_Int Assembly Language

Figure 2.4 defines the concrete syntax for x86_Int. We use the AT&T syntax expected by the GNU assembler. A program begins with a `main` label followed by a sequence of instructions. The `globl` directive says that the `main` procedure is externally visible, which is necessary so that the operating system can call it. In the grammar, ellipses such as ... are used to indicate a sequence of items, e.g., *instr* ... is a sequence of instructions. An x86 program is stored in the computer's memory. For our purposes, the computer's memory is as a mapping of 64-bit addresses to 64-bit values. The computer has a *program counter* (PC) stored in the `rip` register that points to the address of the next instruction to be executed. For most instructions, the program counter is incremented after the instruction is executed, so it points to the next instruction in memory. Most x86 instructions take two operands, where each operand is either an integer constant (called *immediate value*), a *register*, or a memory location.

A register is a special kind of variable. Each one holds a 64-bit value; there are 16 general-purpose registers in the computer and their names are given in Figure 2.4. A register is written with a `%` followed by the register name, such as `%rax`.

An immediate value is written using the notation $\texttt{\$}n$ where n is an integer. An access to memory is specified using the syntax $n(\%r)$, which obtains the address stored in register r and then adds n bytes to the address. The resulting address is used to load or store to memory depending on whether

```
        .globl main
main:
        movq    $10, %rax
        addq    $32, %rax
        retq
```

Figure 2.5: An x86 program equivalent to (+ 10 32).

it occurs as a source or destination argument of an instruction.

An arithmetic instruction such as addq *s, d* reads from the source *s* and destination *d*, applies the arithmetic operation, then writes the result back to the destination *d*. The move instruction movq *s, d* reads from *s* and stores the result in *d*. The callq *label* instruction jumps to the procedure specified by the label and retq returns from a procedure to its caller. We discuss procedure calls in more detail later in this chapter and in Chapter 6. The instruction jmp *label* updates the program counter to the address of the instruction after the specified label.

Appendix 12.3 contains a quick-reference for all of the x86 instructions used in this book.

Figure 2.5 depicts an x86 program that is equivalent to (+ 10 32). The instruction movq $10, %rax puts 10 into register rax and then addq $32, %rax adds 32 to the 10 in rax and puts the result, 42, back into rax. The last instruction, retq, finishes the main function by returning the integer in rax to the operating system. The operating system interprets this integer as the program's exit code. By convention, an exit code of 0 indicates that a program completed successfully, and all other exit codes indicate various errors. Nevertheless, in this book we return the result of the program as the exit code.

The x86 assembly language varies in a couple ways depending on what operating system it is assembled in. The code examples shown here are correct on Linux and most Unix-like platforms, but when assembled on Mac OS X, labels like main must be prefixed with an underscore, as in _main.

We exhibit the use of memory for storing intermediate results in the next example. Figure 2.6 lists an x86 program that is equivalent to (+ 52 (- 10)). This program uses a region of memory called the *procedure call stack* (or *stack* for short). The stack consists of a separate *frame* for each procedure call. The memory layout for an individual frame is shown in Figure 2.7. The register rsp is called the *stack pointer* and points to the item at the top of the stack. The stack grows downward in memory, so we

```
start:
        movq    $10, -8(%rbp)
        negq    -8(%rbp)
        movq    -8(%rbp), %rax
        addq    $52, %rax
        jmp conclusion

        .globl main
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        jmp start
conclusion:
        addq    $16, %rsp
        popq    %rbp
        retq
```

Figure 2.6: An x86 program equivalent to `(+ 52 (- 10))`.

Position	Contents
8(`%rbp`)	return address
0(`%rbp`)	old `rbp`
-8(`%rbp`)	variable 1
-16(`%rbp`)	variable 2
. . .	. . .
0(`%rsp`)	variable n

Figure 2.7: Memory layout of a frame.

increase the size of the stack by subtracting from the stack pointer. In the context of a procedure call, the *return address* is the instruction after the call instruction on the caller side. The function call instruction, `callq`, pushes the return address onto the stack prior to jumping to the procedure. The register `rbp` is the *base pointer* and is used to access variables that are stored in the frame of the current procedure call. The base pointer of the caller is pushed onto the stack after the return address and then the base pointer is set to the location of the old base pointer. In Figure 2.7 we number the variables from 1 to n. Variable 1 is stored at address $-8(\texttt{\%rbp})$, variable 2 at $-16(\texttt{\%rbp})$, etc.

Getting back to the program in Figure 2.6, consider how control is trans-

ferred from the operating system to the `main` function. The operating system issues a `callq main` instruction which pushes its return address on the stack and then jumps to `main`. In x86-64, the stack pointer `rsp` must be divisible by 16 bytes prior to the execution of any `callq` instruction, so when control arrives at `main`, the `rsp` is 8 bytes out of alignment (because the `callq` pushed the return address). The first three instructions are the typical *prelude* for a procedure. The instruction `pushq %rbp` saves the base pointer for the caller onto the stack and subtracts 8 from the stack pointer. The second instruction `movq %rsp, %rbp` changes the base pointer so that it points the location of the old base pointer. The instruction `subq $16, %rsp` moves the stack pointer down to make enough room for storing variables. This program needs one variable (8 bytes) but we round up to 16 bytes so that `rsp` is 16-byte aligned and we're ready to make calls to other functions. The last instruction of the prelude is `jmp start`, which transfers control to the instructions that were generated from the Racket expression `(+ 52 (- 10))`.

The first instruction under the `start` label is `movq $10, -8(%rbp)`, which stores 10 in variable 1. The instruction `negq -8(%rbp)` changes variable 1 to −10. The next instruction moves the −10 from variable 1 into the `rax` register. Finally, `addq $52, %rax` adds 52 to the value in `rax`, updating its contents to 42.

The three instructions under the label `conclusion` are the typical *conclusion* of a procedure. The first two instructions restore the `rsp` and `rbp` registers to the state they were in at the beginning of the procedure. The instruction `addq $16, %rsp` moves the stack pointer back to point at the old base pointer. Then `popq %rbp` returns the old base pointer to `rbp` and adds 8 to the stack pointer. The last instruction, `retq`, jumps back to the procedure that called this one and adds 8 to the stack pointer.

The compiler needs a convenient representation for manipulating x86 programs, so we define an abstract syntax for x86 in Figure 2.8. We refer to this language as x86$_{\text{Int}}$. The main difference compared to the concrete syntax of x86$_{\text{Int}}$ (Figure 2.4) is that labels are not allowed in front of every instructions. Instead instructions are grouped into *blocks* with a label associated with every block, which is why the `X86Program` struct includes an alist mapping labels to blocks. The reason for this organization becomes apparent in Chapter 4 when we introduce conditional branching. The `Block` structure includes an *info* field that is not needed for this chapter, but becomes useful in Chapter 3. For now, the *info* field should contain an empty list. Also, regarding the abstract syntax for `callq`, the `Callq` struct includes an integer for representing the arity of the function, i.e., the number of arguments,

reg	::=	`rsp` | `rbp` | `rax` | `rbx` | `rcx` | `rdx` | `rsi` | `rdi` |
		`r8` | `r9` | `r10` | `r11` | `r12` | `r13` | `r14` | `r15`
arg	::=	`(Imm` *int*`)` | `(Reg` *reg*`)` | `(Deref` *reg int*`)`
instr	::=	`(Instr addq (` *arg arg*`))` | `(Instr subq (` *arg arg*`))`
	|	`(Instr movq (` *arg arg*`))` | `(Instr negq (` *arg*`))`
	|	`(Callq` *label int*`)` | `(Retq)` | `(Pushq` *arg*`)` | `(Popq` *arg*`)` | `(Jmp` *label*`)`
block	::=	`(Block` *info* `(`*instr* `...))`
x86$_{\text{Int}}$	::=	`(X86Program` *info* `((`*label* `.` *block*`)`...`))`

Figure 2.8: The abstract syntax of x86$_{\text{Int}}$ assembly.

which is helpful to know during register allocation (Chapter 3).

2.3   Planning the trip to x86 via the C_{Var} language

To compile one language to another it helps to focus on the differences between the two languages because the compiler will need to bridge those differences. What are the differences between R_{Var} and x86 assembly? Here are some of the most important ones:

(a) x86 arithmetic instructions typically have two arguments and update the second argument in place. In contrast, R_{Var} arithmetic operations take two arguments and produce a new value. An x86 instruction may have at most one memory-accessing argument. Furthermore, some instructions place special restrictions on their arguments.

(b) An argument of an R_{Var} operator can be a deeply-nested expression, whereas x86 instructions restrict their arguments to be integers constants, registers, and memory locations.

(c) The order of execution in x86 is explicit in the syntax: a sequence of instructions and jumps to labeled positions, whereas in R_{Var} the order of evaluation is a left-to-right depth-first traversal of the abstract syntax tree.

(d) A program in R_{Var} can have any number of variables whereas x86 has 16 registers and the procedure calls stack.

(e) Variables in R_{Var} can overshadow other variables with the same name. In x86, registers have unique names and memory locations have unique addresses.

We ease the challenge of compiling from $R_{\tt Var}$ to x86 by breaking down the problem into several steps, dealing with the above differences one at a time. Each of these steps is called a *pass* of the compiler. This terminology comes from the way each step passes over the AST of the program. We begin by sketching how we might implement each pass, and give them names. We then figure out an ordering of the passes and the input/output language for each pass. The very first pass has $R_{\tt Var}$ as its input language and the last pass has x86$_{\tt Int}$ as its output language. In between we can choose whichever language is most convenient for expressing the output of each pass, whether that be $R_{\tt Var}$, x86$_{\tt Int}$, or new *intermediate languages* of our own design. Finally, to implement each pass we write one recursive function per non-terminal in the grammar of the input language of the pass.

select-instructions handles the difference between $R_{\tt Var}$ operations and x86 instructions. This pass converts each $R_{\tt Var}$ operation to a short sequence of instructions that accomplishes the same task.

remove-complex-opera* ensures that each subexpression of a primitive operation is a variable or integer, that is, an *atomic* expression. We refer to non-atomic expressions as *complex*. This pass introduces temporary variables to hold the results of complex subexpressions.[2]

explicate-control makes the execution order of the program explicit. It convert the abstract syntax tree representation into a control-flow graph in which each node contains a sequence of statements and the edges between nodes say which nodes contain jumps to other nodes.

assign-homes replaces the variables in $R_{\tt Var}$ with registers or stack locations in x86.

uniquify deals with the shadowing of variables by renaming every variable to a unique name.

The next question is: in what order should we apply these passes? This question can be challenging because it is difficult to know ahead of time which orderings will be better (easier to implement, produce more efficient code, etc.) so oftentimes trial-and-error is involved. Nevertheless, we can try to plan ahead and make educated choices regarding the ordering.

What should be the ordering of **explicate-control** with respect to **uniquify**? The **uniquify** pass should come first because **explicate-control**

---

[2]The subexpressions of an operation are often called operators and operands which explains the presence of **opera*** in the name of this pass.
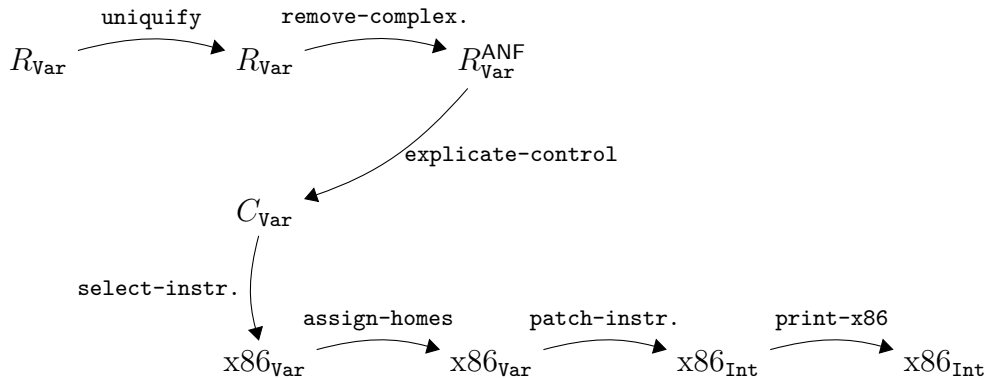
Figure 2.9: Diagram of the passes for compiling $R_{\mathtt{Var}}$.

changes all the `let`-bound variables to become local variables whose scope is the entire program, which would confuse variables with the same name. We place `remove-complex-opera*` before `explicate-control` because the later removes the `let` form, but it is convenient to use `let` in the output of `remove-complex-opera*`. The ordering of `uniquify` with respect to `remove-complex-opera*` does not matter so we arbitrarily choose `uniquify` to come first.

Last, we consider `select-instructions` and `assign-homes`. These two passes are intertwined. In Chapter 6 we learn that, in x86, registers are used for passing arguments to functions and it is preferable to assign parameters to their corresponding registers. On the other hand, by selecting instructions first we may run into a dead end in `assign-homes`. Recall that only one argument of an x86 instruction may be a memory access but `assign-homes` might fail to assign even one of them to a register. A sophisticated approach is to iteratively repeat the two passes until a solution is found. However, to reduce implementation complexity we recommend a simpler approach in which `select-instructions` comes first, followed by the `assign-homes`, then a third pass named `patch-instructions` that uses a reserved register to fix outstanding problems.

Figure 2.9 presents the ordering of the compiler passes and identifies the input and output language of each pass. The last pass, `print-x86`, converts from the abstract syntax of $\mathtt{x86_{Int}}$ to the concrete syntax. In the following two sections we discuss the $C_{\mathtt{Var}}$ intermediate language and the $\mathtt{x86_{Var}}$ dialect of x86. The remainder of this chapter gives hints regarding the implementation of each of the compiler passes in Figure 2.9.

$$
\begin{array}{lll}
atm & ::= & (\text{Int } int) \mid (\text{Var } var) \\
exp & ::= & atm \mid (\text{Prim read ()}) \mid (\text{Prim - } (atm)) \\
    & \mid   & (\text{Prim + } (atm\ atm)) \\
stmt & ::= & (\text{Assign (Var } var)\ exp) \\
tail & ::= & (\text{Return } exp) \mid (\text{Seq } stmt\ tail) \\
C_{\text{Var}} & ::= & (\text{CProgram } info\ ((label\ .\ tail)\dots)) \\
\end{array}
$$

Figure 2.10: The abstract syntax of the C_{Var} intermediate language.

2.3.1 The C_{Var} Intermediate Language

The output of `explicate-control` is similar to the C language [61] in that it has separate syntactic categories for expressions and statements, so we name it C_{Var}. The abstract syntax for C_{Var} is defined in Figure 2.10. (The concrete syntax for C_{Var} is in the Appendix, Figure 12.2.) The C_{Var} language supports the same operators as R_{Var} but the arguments of operators are restricted to atomic expressions. Instead of `let` expressions, C_{Var} has assignment statements which can be executed in sequence using the `Seq` form. A sequence of statements always ends with `Return`, a guarantee that is baked into the grammar rules for *tail*. The naming of this non-terminal comes from the term *tail position*, which refers to an expression that is the last one to execute within a function.

A C_{Var} program consists of a control-flow graph represented as an alist mapping labels to tails. This is more general than necessary for the present chapter, as we do not yet introduce `goto` for jumping to labels, but it saves us from having to change the syntax in Chapter 4. For now there will be just one label, `start`, and the whole program is its tail. The *info* field of the `CProgram` form, after the `explicate-control` pass, contains a mapping from the symbol `locals` to a list of variables, that is, a list of all the variables used in the program. At the start of the program, these variables are uninitialized; they become initialized on their first assignment.

The definitional interpreter for C_{Var} is in the support code for this book, in the file `interp-Cvar.rkt`. The support code is in a `github` repository at the following URL:

> https://github.com/IUCompilerCourse/public-student-support-code

2.3.2 The x86$_{\text{Var}}$ dialect

The x86$_{\text{Var}}$ language is the output of the pass `select-instructions`. It extends x86$_{\text{Int}}$ with an unbounded number of program-scope variables and

removes the restrictions regarding instruction arguments.

2.4 Uniquify Variables

The `uniquify` pass compiles R_{Var} programs into R_{Var} programs in which
every `let` binds a unique variable name. For example, the `uniquify` pass
should translate the program on the left into the program on the right.

```
(let ([x 32])                          (let ([x.1 32])
  (+ (let ([x 10]) x) x))    ⇒           (+ (let ([x.2 10]) x.2) x.1))
```

The following is another example translation, this time of a program with a
`let` nested inside the initializing expression of another `let`.

```
(let ([x (let ([x 4])                  (let ([x.2 (let ([x.1 4])
          (+ x 1))])        ⇒                    (+ x.1 1))])
  (+ x 2))                               (+ x.2 2))
```

We recommend implementing `uniquify` by creating a structurally recur-
sive function named `uniquify-exp` that mostly just copies an expression.
However, when encountering a `let`, it should generate a unique name for
the variable and associate the old name with the new name in an alist.[3] The
`uniquify-exp` function needs to access this alist when it gets to a variable
reference, so we add a parameter to `uniquify-exp` for the alist.

The skeleton of the `uniquify-exp` function is shown in Figure 2.11. The
function is curried so that it is convenient to partially apply it to an alist and
then apply it to different expressions, as in the last case for primitive opera-
tions in Figure 2.11. The `for/list` form of Racket is useful for transforming
each element of a list to produce a new list.

Exercise 1. Complete the `uniquify` pass by filling in the blanks in Fig-
ure 2.11, that is, implement the cases for variables and for the `let` form in
the file `compiler.rkt` in the support code.

Exercise 2. Create five R_{Var} programs that exercise the most interest-
ing parts of the `uniquify` pass, that is, the programs should include `let`
forms, variables, and variables that overshadow each other. The five pro-
grams should be placed in the subdirectory named `tests` and the file names
should start with `var_test_` followed by a unique integer and end with the
file extension `.rkt`. The `run-tests.rkt` script in the support code checks
whether the output programs produce the same result as the input pro-
grams. The script uses the `interp-tests` function (Appendix 12.2) from

---

[3]The Racket function `gensym` is handy for generating unique variable names.

```
(define (uniquify-exp env)
  (lambda (e)
    (match e
      [(Var x) ___]
      [(Int n) (Int n)]
      [(Let x e body) ___]
      [(Prim op es)
       (Prim op (for/list ([e es]) ((uniquify-exp env) e)))])))

(define (uniquify p)
  (match p
    [(Program '() e) (Program '() ((uniquify-exp '()) e))]))
```

Figure 2.11: Skeleton for the `uniquify` pass.

`utilities.rkt` to test your `uniquify` pass on the example programs. The `passes` parameter of `interp-tests` is a list that should have one entry for each pass in your compiler. For now, define `passes` to contain just one entry for `uniquify` as follows.

```
(define passes
  (list (list "uniquify" uniquify interp-Rvar type-check-Rvar)))
```

Run the `run-tests.rkt` script in the support code to check whether the output programs produce the same result as the input programs.

2.5   Remove Complex Operands

The `remove-complex-opera*` pass compiles R_{Var} programs into R_{Var} programs in which the arguments of operations are atomic expressions. Put another way, this pass removes complex operands, such as the expression `(- 10)` in the program below. This is accomplished by introducing a new `let`-bound variable, binding the complex operand to the new variable, and then using the new variable in place of the complex operand, as shown in the output of `remove-complex-opera*` on the right.

```
(+ 52 (- 10))
```
$\Rightarrow$
```
(let ([tmp.1 (- 10)])
  (+ 52 tmp.1))
```

Figure 2.12 presents the grammar for the output of this pass, the language $R_{\text{Var}}^{\text{ANF}}$. The only difference is that operator arguments are required to be atomic expressions. In the literature, this is called *administrative normal form*, or ANF for short [26, 37].

$$
\begin{array}{rcl}
atm & ::= & (\texttt{Int } int) \mid (\texttt{Var } var) \\
exp & ::= & atm \mid (\texttt{Prim read ()}) \\
    & \mid & (\texttt{Prim - } (atm)) \mid (\texttt{Prim + } (atm\ atm)) \\
    & \mid & (\texttt{Let } var\ exp\ exp) \\
R_1^\dagger & ::= & (\texttt{Program '() } exp)
\end{array}
$$

Figure 2.12: $R_{\texttt{Var}}^{\textsf{ANF}}$ is $R_{\texttt{Var}}$ in administrative normal form (ANF).

We recommend implementing this pass with two mutually recursive functions, `rco-atom` and `rco-exp`. The idea is to apply `rco-atom` to subexpressions that are required to be atomic and to apply `rco-exp` to subexpressions that can be atomic or complex (see Figure 2.12). Both functions take an $R_{\texttt{Var}}$ expression as input. The `rco-exp` function returns an expression. The `rco-atom` function returns two things: an atomic expression and alist mapping temporary variables to complex subexpressions. You can return multiple things from a function using Racket's `values` form and you can receive multiple things from a function call using the `define-values` form. If you are not familiar with these features, review the Racket documentation. Also, the `for/lists` form is useful for applying a function to each element of a list, in the case where the function returns multiple values.

The following shows the output of `rco-atom` on the expression `(- 10)` (using concrete syntax to be concise).

```
                                           tmp.1
(- 10)                            ⇒        ((tmp.1 . (- 10)))
```

Take special care of programs such as the following one that binds a variable to an atomic expression. You should leave such variable bindings unchanged, as shown in to the program on the right

```
(let ([a 42])                             (let ([a 42])
  (let ([b a])                 ⇒            (let ([b a])
    b))                                       b))
```

A careless implementation of `rco-exp` and `rco-atom` might produce the following output with unnecessary temporary variables.

```
(let ([tmp.1 42])
  (let ([a tmp.1])
    (let ([tmp.2 a])
      (let ([b tmp.2])
        b))))
```

Exercise 3. Implement the `remove-complex-opera*` function in `compiler.rkt`. Create three new R_{Int} programs that exercise the interesting code in the `remove-complex-opera*` pass (Following the same file name guidelines as before.). In the `run-tests.rkt` script, add the following entry to the list of `passes` and then run the script to test your compiler.

`(list "remove-complex" remove-complex-opera* interp-Rvar type-check-Rvar)`

While debugging your compiler, it is often useful to see the intermediate programs that are output from each pass. To print the intermeidate programs, place the following before the call to `interp-tests` in `run-tests.rkt`.

`(debug-level 1)`

2.6   Explicate Control

The `explicate-control` pass compiles R_{Var} programs into C_{Var} programs that make the order of execution explicit in their syntax. For now this amounts to flattening `let` constructs into a sequence of assignment statements. For example, consider the following R_{Var} program.

```
(let ([y (let ([x 20])
          (+ x (let ([x 22]) x)))])
  y)
```

The output of the previous pass and of `explicate-control` is shown below. Recall that the right-hand-side of a `let` executes before its body, so the order of evaluation for this program is to assign 20 to `x.1`, 22 to `x.2`, and `(+ x.1 x.2)` to `y`, then return `y`. Indeed, the output of `explicate-control` makes this ordering explicit.

```
(let ([y (let ([x.1 20])             start:
          (let ([x.2 22])              x.1 = 20;
            (+ x.1 x.2)))])     ⇒      x.2 = 22;
  y)                                   y = (+ x.1 x.2);
                                       return y;
```

The organization of this pass depends on the notion of tail position that we have alluded to earlier. Formally, *tail position* in the context of R_{Var} is defined recursively by the following two rules.

1. In (`Program ()` e), expression e is in tail position.

2. If (`Let` x e_1 e_2) is in tail position, then so is e_2.

```
(define (explicate-tail e)
  (match e
    [(Var x) ___]
    [(Int n) (Return (Int n))]
    [(Let x rhs body) ___]
    [(Prim op es) ___]
    [else (error "explicate-tail unhandled case" e)]))

(define (explicate-assign e x cont)
  (match e
    [(Var x) ___]
    [(Int n) (Seq (Assign (Var x) (Int n)) cont)]
    [(Let y rhs body) ___]
    [(Prim op es) ___]
    [else (error "explicate-assign unhandled case" e)]))

(define (explicate-control p)
  (match p
    [(Program info body) ___]))
```

Figure 2.13: Skeleton for the `explicate-control` pass.

We recommend implementing `explicate-control` using two mutually recursive functions, `explicate-tail` and `explicate-assign`, as suggested in the skeleton code in Figure 2.13. The `explicate-tail` function should be applied to expressions in tail position whereas the `explicate-assign` should be applied to expressions that occur on the right-hand-side of a `let`. The `explicate-tail` function takes an *exp* in R_{Var} as input and produces a *tail* in C_{Var} (see Figure 2.10). The `explicate-assign` function takes an *exp* in R_{Var}, the variable that it is to be assigned to, and a *tail* in C_{Var} for the code that will come after the assignment. The `explicate-assign` function returns a *tail* in C_{Var}.

The `explicate-assign` function is in accumulator-passing style in that the `cont` parameter is used for accumulating the output. The reader might be tempted to instead organize `explicate-assign` in a more direct fashion, without the `cont` parameter and perhaps using `append` to combine statements. We warn against that alternative because the accumulator-passing style is key to how we generate high-quality code for conditional expressions in Chapter 4.

Exercise 4. Implement the `explicate-control` function in `compiler.rkt`.

Create three new $R_{\tt Int}$ programs that exercise the code in `explicate-control`. In the `run-tests.rkt` script, add the following entry to the list of `passes` and then run the script to test your compiler.

```
(list "explicate control" explicate-control interp-Cvar type-check-Cvar)
```

2.7   Select Instructions

In the `select-instructions` pass we begin the work of translating from $C_{\tt Var}$ to x86$_{\tt Var}$. The target language of this pass is a variant of x86 that still uses variables, so we add an AST node of the form (`Var` *var*) to the *arg* non-terminal of the x86$_{\tt Int}$ abstract syntax (Figure 2.8). We recommend implementing the `select-instructions` with three auxiliary functions, one for each of the non-terminals of $C_{\tt Var}$: *atm*, *stmt*, and *tail*.

The cases for *atm* are straightforward, variables stay the same and integer constants are changed to immediates: (`Int` *n*) changes to (`Imm` *n*).

Next we consider the cases for *stmt*, starting with arithmetic operations. For example, consider the addition operation. We can use the `addq` instruction, but it performs an in-place update. So we could move arg_1 into the left-hand side *var* and then add arg_2 to *var*.

$$var \;\texttt{=}\; \texttt{(+}\; arg_1\;\; arg_2 \texttt{);} \qquad\Rightarrow\qquad \begin{array}{l}\texttt{movq}\; arg_1 \texttt{,}\;\; var \\ \texttt{addq}\; arg_2 \texttt{,}\;\; var\end{array}$$

There are also cases that require special care to avoid generating needlessly complicated code. For example, if one of the arguments of the addition is the same variable as the left-hand side of the assignment, then there is no need for the extra move instruction. The assignment statement can be translated into a single `addq` instruction as follows.

$$var \;\texttt{=}\; \texttt{(+}\; arg_1\;\; var \texttt{);} \qquad\Rightarrow\qquad \texttt{addq}\; arg_1 \texttt{,}\;\; var$$

The `read` operation does not have a direct counterpart in x86 assembly, so we provide this functionality with the function `read_int` in the file `runtime.c`, written in C [61]. In general, we refer to all of the functionality in this file as the *runtime system*, or simply the *runtime* for short. When compiling your generated x86 assembly code, you need to compile `runtime.c` to `runtime.o` (an "object file", using `gcc` option `-c`) and link it into the executable. For our purposes of code generation, all you need to do is translate an assignment of `read` into a call to the `read_int` function followed by a move from `rax` to the left-hand-side variable. (Recall that the return value of a function goes into `rax`.)

$$var = (\text{read}); \qquad \Rightarrow \qquad \begin{array}{l} \texttt{callq read\_int} \\ \texttt{movq \%rax, } var \end{array}$$

There are two cases for the *tail* non-terminal: `Return` and `Seq`. Regarding `Return`, we recommend treating it as an assignment to the `rax` register followed by a jump to the conclusion of the program (so the conclusion needs to be labeled). For (`Seq` *s t*), you can translate the statement *s* and tail *t* recursively and then append the resulting instructions.

Exercise 5. Implement the `select-instructions` pass in `compiler.rkt`. Create three new example programs that are designed to exercise all of the interesting cases in this pass. In the `run-tests.rkt` script, add the following entry to the list of `passes` and then run the script to test your compiler.

```
(list "instruction selection" select-instructions interp-pseudo-x86-0)
```

2.8   Assign Homes

The `assign-homes` pass compiles x86$_{\texttt{Var}}$ programs to x86$_{\texttt{Var}}$ programs that no longer use program variables. Thus, the `assign-homes` pass is responsible for placing all of the program variables in registers or on the stack. For runtime efficiency, it is better to place variables in registers, but as there are only 16 registers, some programs must necessarily resort to placing some variables on the stack. In this chapter we focus on the mechanics of placing variables on the stack. We study an algorithm for placing variables in registers in Chapter 3.

Consider again the following $R_{\texttt{Var}}$ program from Section 2.5.

```
(let ([a 42])
  (let ([b a])
    b))
```

The output of `select-instructions` is shown on the left and the output of `assign-homes` on the right. In this example, we assign variable `a` to stack location `-8(%rbp)` and variable `b` to location `-16(%rbp)`.

```
locals-types:
    a : Integer, b : Integer
start:                                      stack-space: 16
    movq $42, a                             start:
    movq a, b                 ⇒                 movq $42, -8(%rbp)
    movq b, %rax                                movq -8(%rbp), -16(%rbp)
    jmp conclusion                              movq -16(%rbp), %rax
                                                jmp conclusion
```

The `locals-types` entry in the *info* of the `X86Program` node is an al-
ist mapping all the variables in the program to their types (for now just
`Integer`). The `assign-homes` pass should replace all uses of those variables
with stack locations. As an aside, the `locals-types` entry is computed by
`type-check-Cvar` in the support code, which installs it in the *info* field of
the `CProgram` node, which should be propagated to the `X86Program` node.

In the process of assigning variables to stack locations, it is convenient
for you to compute and store the size of the frame (in bytes) in the *info*
field of the `X86Program` node, with the key `stack-space`, which is needed
later to generate the conclusion of the `main` procedure. The x86-64 standard
requires the frame size to be a multiple of 16 bytes.

Exercise 6. Implement the `assign-homes` pass in `compiler.rkt`, defining
auxiliary functions for the non-terminals *arg*, *instr*, and *block*. We rec-
ommend that the auxiliary functions take an extra parameter that is an
alist mapping variable names to homes (stack locations for now). In the
`run-tests.rkt` script, add the following entry to the list of `passes` and
then run the script to test your compiler.

```
(list "assign homes" assign-homes interp-x86-0)
```

2.9   Patch Instructions

The `patch-instructions` pass compiles from x86$_\text{Var}$ to x86$_\text{Int}$ by making
sure that each instruction adheres to the restriction that at most one argu-
ment of an instruction may be a memory reference.

We return to the following example.

```
(let ([a 42])
  (let ([b a])
    b))
```

The `assign-homes` pass produces the following output for this program.

```
stack-space: 16
start:
    movq $42, -8(%rbp)
    movq -8(%rbp), -16(%rbp)
    movq -16(%rbp), %rax
    jmp conclusion
```

The second `movq` instruction is problematic because both arguments are
stack locations. We suggest fixing this problem by moving from the source

location to the register `rax` and then from `rax` to the destination location, as follows.

```
movq -8(%rbp), %rax
movq %rax, -16(%rbp)
```

Exercise 7. Implement the `patch-instructions` pass in `compiler.rkt`. Create three new example programs that are designed to exercise all of the interesting cases in this pass. In the `run-tests.rkt` script, add the following entry to the list of `passes` and then run the script to test your compiler.

```
(list "patch instructions" patch-instructions interp-x86-0)
```

2.10   Print x86

The last step of the compiler from R_{Var} to x86 is to convert the x86$_{\text{Int}}$ AST (defined in Figure 2.8) to the string representation (defined in Figure 2.4). The Racket `format` and `string-append` functions are useful in this regard. The main work that this step needs to perform is to create the `main` function and the standard instructions for its prelude and conclusion, as shown in Figure 2.6 of Section 2.2. You will need to know the amount of space needed for the stack frame, which you can obtain from the `stack-space` entry in the *info* field of the `X86Program` node.

When running on Mac OS X, you compiler should prefix an underscore to labels like `main`. The Racket call (`system-type 'os`) is useful for determining which operating system the compiler is running on. It returns `'macosx`, `'unix`, or `'windows`.

Exercise 8. Implement the `print-x86` pass in `compiler.rkt`. In the `run-tests.rkt` script, add the following entry to the list of `passes` and then run the script to test your compiler.

```
(list "print x86" print-x86 #f)
```

Uncomment the call to the `compiler-tests` function (Appendix 12.2), which tests your complete compiler by executing the generated x86 code. Compile the provided `runtime.c` file to `runtime.o` using `gcc`. Run the script to test your compiler.

2.11   Challenge: Partial Evaluator for R_{Var}

This section describes optional challenge exercises that involve adapting and improving the partial evaluator for R_{Int} that was introduced in Section 1.6.

Exercise 9. Adapt the partial evaluator from Section 1.6 (Figure 1.4) so that it applies to $R_{\mathtt{Var}}$ programs instead of $R_{\mathtt{Int}}$ programs. Recall that $R_{\mathtt{Var}}$ adds `let` binding and variables to the $R_{\mathtt{Int}}$ language, so you will need to add cases for them in the `pe-exp` function. Once complete, add the partial evaluation pass to the front of your compiler and make sure that your compiler still passes all of the tests.

The next exercise builds on Exercise 9.

Exercise 10. Improve on the partial evaluator by replacing the `pe-neg` and `pe-add` auxiliary functions with functions that know more about arithmetic. For example, your partial evaluator should translate

$$(+ \ 1 \ (+ \ (\text{read}) \ 1)) \qquad \text{into} \qquad (+ \ 2 \ (\text{read}))$$

To accomplish this, the `pe-exp` function should produce output in the form of the *residual* non-terminal of the following grammar. The idea is that when processing an addition expression, we can always produce either 1) an integer constant, 2) and addition expression with an integer constant on the left-hand side but not the right-hand side, or 3) or an addition expression in which neither subexpression is a constant.

$$
\begin{array}{lll}
inert & ::= & var \mid (\text{read}) \mid (\text{-} \ var) \mid (\text{-} \ (\text{read})) \mid (+ \ inert \ inert) \\
& \mid & (\text{let} \ ([var \ inert]) \ inert) \\
residual & ::= & int \mid (+ \ int \ inert) \mid inert
\end{array}
$$

The `pe-add` and `pe-neg` functions may assume that their inputs are *residual* expressions and they should return *residual* expressions. Once the improvements are complete, make sure that your compiler still passes all of the tests. After all, fast code is useless if it produces incorrect results!

3

Register Allocation

In Chapter 2 we learned how to store variables on the stack. In this Chapter we learn how to improve the performance of the generated code by placing some variables into registers. The CPU can access a register in a single cycle, whereas accessing the stack can take 10s to 100s of cycles. The program in Figure 3.1 serves as a running example. The source program is on the left and the output of instruction selection is on the right. The program is almost in the x86 assembly language but it still uses variables.

The goal of register allocation is to fit as many variables into registers as possible. Some programs have more variables than registers so we cannot always map each variable to a different register. Fortunately, it is common for different variables to be needed during different periods of time during program execution, and in such cases several variables can be mapped to the same register. Consider variables x and z in Figure 3.1. After the variable x is moved to z it is no longer needed. Variable z, on the other hand, is used only after this point, so x and z could share the same register. The topic of Section 3.2 is how to compute where a variable is needed. Once we have that information, we compute which variables are needed at the same time, i.e., which ones *interfere* with each other, and represent this relation as an undirected graph whose vertices are variables and edges indicate when two variables interfere (Section 3.3). We then model register allocation as a graph coloring problem (Section 3.4).

If we run out of registers despite these efforts, we place the remaining variables on the stack, similar to what we did in Chapter 2. It is common to use the verb *spill* for assigning a variable to a stack location. The decision to spill a variable is handled as part of the graph coloring process (Section 3.4).

We make the simplifying assumption that each variable is assigned to one

After instruction selection:

```
locals-types:
    x : Integer, y : Integer,
    z : Integer, t : Integer,
    v : Integer, w : Integer
start:
    movq $1, v
    movq $42, w
    movq v, x
    addq $7, x
    movq x, y
    movq x, z
    addq w, z
    movq y, t
    negq t
    movq z, %rax
    addq t, %rax
    jmp conclusion
```

Example $R_{\tt Var}$ program:

```
(let ([v 1])
  (let ([w 42])
    (let ([x (+ v 7)])
      (let ([y x])
        (let ([z (+ x w)])
          (+ z (- y)))))))
```

Figure 3.1: A running example for register allocation.

location (a register or stack address). A more sophisticated approach is to assign a variable to one or more locations in different regions of the program. For example, if a variable is used many times in short sequence and then only used again after many other instructions, it could be more efficient to assign the variable to a register during the initial sequence and then move it to the stack for the rest of its lifetime. We refer the interested reader to Cooper and Torczon [23] for more information about that approach.

3.1   Registers and Calling Conventions

As we perform register allocation, we need to be aware of the *calling conventions* that govern how functions calls are performed in x86. Even though $R_{\tt Var}$ does not include programmer-defined functions, our generated code includes a `main` function that is called by the operating system and our generated code contains calls to the `read_int` function.

Function calls require coordination between two pieces of code that may be written by different programmers or generated by different compilers. Here we follow the System V calling conventions that are used by the GNU C compiler on Linux and MacOS [16, 72]. The calling conventions include rules about how functions share the use of registers. In particular, the caller

is responsible for freeing up some registers prior to the function call for use by the callee. These are called the *caller-saved registers* and they are

`rax rcx rdx rsi rdi r8 r9 r10 r11`

On the other hand, the callee is responsible for preserving the values of the *callee-saved registers*, which are

`rsp rbp rbx r12 r13 r14 r15`

We can think about this caller/callee convention from two points of view, the caller view and the callee view:

- The caller should assume that all the caller-saved registers get over-written with arbitrary values by the callee. On the other hand, the caller can safely assume that all the callee-saved registers contain the same values after the call that they did before the call.

- The callee can freely use any of the caller-saved registers. However, if the callee wants to use a callee-saved register, the callee must arrange to put the original value back in the register prior to returning to the caller. This can be accomplished by saving the value to the stack in the prelude of the function and restoring the value in the conclusion of the function.

In x86, registers are also used for passing arguments to a function and for the return value. In particular, the first six arguments to a function are passed in the following six registers, in this order.

`rdi rsi rdx rcx r8 r9`

If there are more than six arguments, then the convention is to use space on the frame of the caller for the rest of the arguments. However, in Chapter 6 we arrange to never need more than six arguments. For now, the only function we care about is `read_int` and it takes zero arguments. The register `rax` is used for the return value of a function.

The next question is how these calling conventions impact register allocation. Consider the $R_{\mathtt{Var}}$ program in Figure 3.2. We first analyze this example from the caller point of view and then from the callee point of view.

The program makes two calls to the `read` function. Also, the variable x is in-use during the second call to `read`, so we need to make sure that the value in x does not get accidentally wiped out by the call to `read`. One obvious approach is to save all the values in caller-saved registers to the stack prior to each function call, and restore them after each call. That way, if the register allocator chooses to assign x to a caller-saved register, its value

will be preserved across the call to `read`. However, saving and restoring to the stack is relatively slow. If `x` is not used many times, it may be better to assign `x` to a stack location in the first place. Or better yet, if we can arrange for `x` to be placed in a callee-saved register, then it won't need to be saved and restored during function calls.

The approach that we recommend for variables that are in-use during a function call is to either assign them to callee-saved registers or to spill them to the stack. On the other hand, for variables that are not in-use during a function call, we try the following alternatives in order 1) look for an available caller-saved register (to leave room for other variables in the callee-saved register), 2) look for a callee-saved register, and 3) spill the variable to the stack.

It is straightforward to implement this approach in a graph coloring register allocator. First, we know which variables are in-use during every function call because we compute that information for every instruction (Section 3.2). Second, when we build the interference graph (Section 3.3), we can place an edge between each of these variables and the caller-saved registers in the interference graph. This will prevent the graph coloring algorithm from assigning those variables to caller-saved registers.

Returning to the example in Figure 3.2, let us analyze the generated x86 code on the right-hand side, focusing on the `start` block. Notice that variable `x` is assigned to `rbx`, a callee-saved register. Thus, it is already in a safe place during the second call to `read_int`. Next, notice that variable `y` is assigned to `rcx`, a caller-saved register, because there are no function calls in the remainder of the block.

Next we analyze the example from the callee point of view, focusing on the prelude and conclusion of the `main` function. As usual the prelude begins with saving the `rbp` register to the stack and setting the `rbp` to the current stack pointer. We now know why it is necessary to save the `rbp`: it is a callee-saved register. The prelude then pushes `rbx` to the stack because 1) `rbx` is a callee-saved register and 2) `rbx` is assigned to a variable (`x`). The other callee-saved registers are not saved in the prelude because they are not used. The prelude subtracts 8 bytes from the `rsp` to make it 16-byte aligned and then jumps to the `start` block. Shifting attention to the `conclusion`, we see that `rbx` is restored from the stack with a `popq` instruction.

Generated x86 assembly:

```
start:
        callq   read_int
        movq    %rax, %rbx
        callq   read_int
        movq    %rax, %rcx
        addq    %rcx, %rbx
        movq    %rbx, %rax
        addq    $42, %rax
        jmp _conclusion


        .globl main
main:
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %rbx
        subq    $8, %rsp
        jmp start
conclusion:
        addq    $8, %rsp
        popq    %rbx
        popq    %rbp
        retq
```

Example $R_{\mathtt{Var}}$ program:

```
(let ([x (read)])
  (let ([y (read)])
    (+ (+ x y) 42)))
```

Figure 3.2: An example with function calls.

3.2   Liveness Analysis

The `uncover-live` pass performs *liveness analysis*, that is, it discovers
which variables are in-use in different regions of a program. A variable
or register is *live* at a program point if its current value is used at some
later point in the program. We refer to variables and registers collectively
as *locations*. Consider the following code fragment in which there are two
writes to b. Are a and b both live at the same time?

```
1   movq $5, a
2   movq $30, b
3   movq a, c
4   movq $10, b
5   addq b, c
```

The answer is no because a is live from line 1 to 3 and b is live from line 4 to
5. The integer written to b on line 2 is never used because it is overwritten
(line 4) before the next read (line 5).

The live locations can be computed by traversing the instruction sequence back to front (i.e., backwards in execution order). Let $I_1, \ldots, I_n$ be the instruction sequence. We write $L_{\mathsf{after}}(k)$ for the set of live locations after instruction I_k and $L_{\mathsf{before}}(k)$ for the set of live locations before instruction I_k. The live locations after an instruction are always the same as the live locations before the next instruction.

$$L_{\mathsf{after}}(k) = L_{\mathsf{before}}(k + 1) \qquad (3.1)$$

To start things off, there are no live locations after the last instruction, so

$$L_{\mathsf{after}}(n) = \emptyset \qquad (3.2)$$

> **The Racket Set Package**
>
> A *set* is an unordered collection of elements without duplicates.
>
> (**set** $v \ldots$) constructs a set containing the specified elements.
>
> (**set-union** $set_1\ set_2$) returns the union of the two sets.
>
> (**set-subtract** $set_1\ set_2$) returns the difference of the two sets.
>
> (**set-member?** $set\ v$) is element v in *set*?
>
> (**set-count** set) how many unique elements are in *set*?
>
> (**set->list** set) converts the set to a list.

We then apply the following rule repeatedly, traversing the instruction sequence back to front.

$$L_{\mathsf{before}}(k) = (L_{\mathsf{after}}(k) - W(k)) \cup R(k), \qquad (3.3)$$

where $W(k)$ are the locations written to by instruction I_k and $R(k)$ are the
locations read by instruction I_k.

```
1  movq $5, a
2  movq $30, b
3  movq a, c
4  movq $10, b
5  addq b, c
```

$$L_{\text{before}}(1) = \emptyset, L_{\text{after}}(1) = \{\texttt{a}\}$$
$$L_{\text{before}}(2) = \{\texttt{a}\}, L_{\text{after}}(2) = \{\texttt{a}\}$$
$$L_{\text{before}}(3) = \{\texttt{a}\}, L_{\text{after}}(2) = \{\texttt{c}\}$$
$$L_{\text{before}}(4) = \{\texttt{c}\}, L_{\text{after}}(4) = \{\texttt{b}, \texttt{c}\}$$
$$L_{\text{before}}(5) = \{\texttt{b}, \texttt{c}\}, L_{\text{after}}(5) = \emptyset$$

Figure 3.3: Example output of liveness analysis on a short example.

There is a special case for `jmp` instructions. The locations that are live before a `jmp` should be the locations in L_{before} at the target of the jump. So we recommend maintaining an alist named `label->live` that maps each label to the L_{before} for the first instruction in its block. For now the only `jmp` in a x86$_{\text{Var}}$ program is the one at the end, to the conclusion. (For example, see Figure 3.1.) The conclusion reads from `rax` and `rsp`, so the alist should map `conclusion` to the set $\{\texttt{rax}, \texttt{rsp}\}$.

Let us walk through the above example, applying these formulas starting with the instruction on line 5. We collect the answers in Figure 3.3. The L_{after} for the `addq b, c` instruction is $\emptyset$ because it is the last instruction (formula 3.2). The L_{before} for this instruction is $\{\texttt{b}, \texttt{c}\}$ because it reads from variables `b` and `c` (formula 3.3), that is

$$L_{\text{before}}(5) = (\emptyset - \{\texttt{c}\}) \cup \{\texttt{b}, \texttt{c}\} = \{\texttt{b}, \texttt{c}\}$$

Moving on the the instruction `movq $10, b` at line 4, we copy the live-before set from line 5 to be the live-after set for this instruction (formula 3.1).

$$L_{\text{after}}(4) = \{\texttt{b}, \texttt{c}\}$$

This move instruction writes to `b` and does not read from any variables, so we have the following live-before set (formula 3.3).

$$L_{\text{before}}(4) = (\{\texttt{b}, \texttt{c}\} - \{\texttt{b}\}) \cup \emptyset = \{\texttt{c}\}$$

The live-before for instruction `movq a, c` is $\{\texttt{a}\}$ because it writes to $\{\texttt{c}\}$ and reads from $\{\texttt{a}\}$ (formula 3.3). The live-before for `movq $30, b` is $\{\texttt{a}\}$ because it writes to a variable that is not live and does not read from a variable. Finally, the live-before for `movq $5, a` is $\emptyset$ because it writes to variable `a`.

Exercise 11. Perform liveness analysis on the running example in Figure 3.1, computing the live-before and live-after sets for each instruction. Compare your answers to the solution shown in Figure 3.4.

```
                           {rsp}
            movq $1, v
                           {v, rsp}
            movq $42, w
                           {v, w, rsp}
            movq v, x
                           {w, x, rsp}
            addq $7, x
                           {w, x, rsp}
            movq x, y
                           {w, x, y, rsp}
            movq x, z
                           {w, y, z, rsp}
            addq w, z
                           {y, z, rsp}
            movq y, t
                           {t, z, rsp}
            negq t
                           {t, z, rsp}
            movq z, %rax
                           {rax, t, rsp}
            addq t, %rax
                           {rax, rsp}
            jmp conclusion
```

Figure 3.4: The running example annotated with live-after sets.

Exercise 12. Implement the `uncover-live` pass. Store the sequence of live-after sets in the *info* field of the `Block` structure. We recommend creating an auxiliary function that takes a list of instructions and an initial live-after set (typically empty) and returns the list of live-after sets. We also recommend creating auxiliary functions to 1) compute the set of locations that appear in an *arg*, 2) compute the locations read by an instruction (the R function), and 3) the locations written by an instruction (the W function). The `callq` instruction should include all of the caller-saved registers in its write-set W because the calling convention says that those registers may be written to during the function call. Likewise, the `callq` instruction should include the appropriate argument-passing registers in its read-set R, depending on the arity of the function being called. (This is why the abstract syntax for `callq` includes the arity.)

3.3   Build the Interference Graph

Based on the liveness analysis, we know
where each location is live. However, dur-
ing register allocation, we need to answer
questions of the specific form: are locations
u and v live at the same time? (And there-
fore cannot be assigned to the same regis-
ter.) To make this question more efficient
to answer, we create an explicit data struc-
ture, an *interference graph*. An interference
graph is an undirected graph that has an
edge between two locations if they are live
at the same time, that is, if they interfere
with each other.

> **The Racket Graph Library**
>
> A *graph* is a collection of vertices and edges
> where each edge connects two vertices. A
> graph is *directed* if each edge points from
> a source to a target. Otherwise the graph
> is *undirected*.
>
> **(undirected-graph** *edges***)** constructs
>   a undirected graph from a list of
>   edges. Each edge is represented by
>   a list containing two vertices.
>
> **(add-vertex!** *graph vertex***)** inserts a ver-
>   tex into the graph.
>
> **(add-edge!** *graph source target***)** inserts
>   an edge between the two vertices
>   into the graph.
>
> **(in-neighbors** *graph vertex***)** returns   a
>   sequence of all the neighbors of the
>   given vertex.
>
> **(in-vertices** *graph***)** returns a sequence
>   of all the vertices in the graph.

An obvious way to compute the inter-
ference graph is to look at the set of live
location between each instruction and add
an edge to the graph for every pair of vari-
ables in the same set. This approach is less
than ideal for two reasons. First, it can be
expensive because it takes $O(n^2)$ time to
consider at every pair in a set of n live loca-
tions. Second, in the special case where two
locations hold the same value (because one
was assigned to the other), they can be live
at the same time without interfering with
each other.

A better way to compute the interference graph is to focus on writes [8].
The writes performed by an instruction must not overwrite something in
a live location. So for each instruction, we create an edge between the
locations being written to and the live locations. (Except that one should
not create self edges.) Note that for the `callq` instruction, we consider all
of the caller-saved registers as being written to, so an edge is added between
every live variable and every caller-saved register. For `movq`, we deal with the
above-mentioned special case by not adding an edge between a live variable
v and the destination if v matches the source. So we have the following two
rules.

1. If instruction I_k is a move such as `movq` s`,` d, then add the edge (d, v)

```
movq $1, v          v interferes with rsp,
movq $42, w         w interferes with v and rsp,
movq v, x           x interferes with w and rsp,
addq $7, x          x interferes with w and rsp,
movq x, y           y interferes with w and rsp but not x,
movq x, z           z interferes with w, y, and rsp,
addq w, z           z interferes with y and rsp,
movq y, t           t interferes with z and rsp,
negq t              t interferes with z and rsp,
movq z, %rax        rax interferes with t and rsp,
addq t, %rax        rax interferes with rsp.
jmp conclusion      no interference.
```

Figure 3.5: Interference results for the running example.
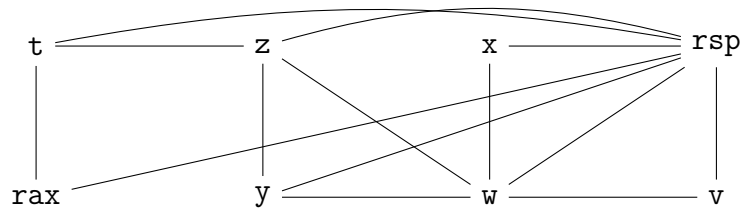
Figure 3.6: The interference graph of the example program.

for every $v \in L_{\mathsf{after}}(k)$ unless $v = d$ or $v = s$.

2. For any other instruction I_k, for every $d \in W(k)$ add an edge (d, v) for every $v \in L_{\mathsf{after}}(k)$ unless $v = d$.

Working from the top to bottom of Figure 3.4, we apply the above rules to each instruction. We highlight a few of the instructions. The first instruction is movq $1, v and the live-after set is $\{v, rsp\}$. Rule 1 applies, so v interferes with rsp. The fourth instruction is addq $7, x and the live-after set is $\{w, x, rsp\}$. Rule 2 applies so x interferes with w and rsp. The next instruction is movq x, y and the live-after set is $\{w, x, y, rsp\}$. Rule 1 applies, so y interferes with w and rsp but not x because x is the source of the move and therefore x and y hold the same value. Figure 3.5 lists the interference results for all of the instructions and the resulting interference graph is shown in Figure 3.6.

Exercise 13. Implement the compiler pass named `build-interference` according to the algorithm suggested above. We recommend using the `graph` package to create and inspect the interference graph. The output graph of this pass should be stored in the *info* field of the program, under the key `conflicts`.

3.4   Graph Coloring via Sudoku

We come to the main event, mapping variables to registers and stack locations. Variables that interfere with each other must be mapped to different locations. In terms of the interference graph, this means that adjacent vertices must be mapped to different locations. If we think of locations as colors, the register allocation problem becomes the graph coloring problem [11, 83].

The reader may be more familiar with the graph coloring problem than he or she realizes; the popular game of Sudoku is an instance of the graph coloring problem. The following describes how to build a graph out of an initial Sudoku board.

- There is one vertex in the graph for each Sudoku square.

- There is an edge between two vertices if the corresponding squares are in the same row, in the same column, or if the squares are in the same 3×3 region.

- Choose nine colors to correspond to the numbers 1 to 9.

- Based on the initial assignment of numbers to squares in the Sudoku board, assign the corresponding colors to the corresponding vertices in the graph.

If you can color the remaining vertices in the graph with the nine colors, then you have also solved the corresponding game of Sudoku. Figure 3.7 shows an initial Sudoku game board and the corresponding graph with colored vertices. We map the Sudoku number 1 to blue, 2 to yellow, and 3 to red. We only show edges for a sampling of the vertices (the colored ones) because showing edges for all of the vertices would make the graph unreadable.

It turns out that some techniques for playing Sudoku correspond to heuristics used in graph coloring algorithms. For example, one of the basic techniques for Sudoku is called Pencil Marks. The idea is to use a process of elimination to determine what numbers are no longer available for a square and write down those numbers in the square (writing very small).
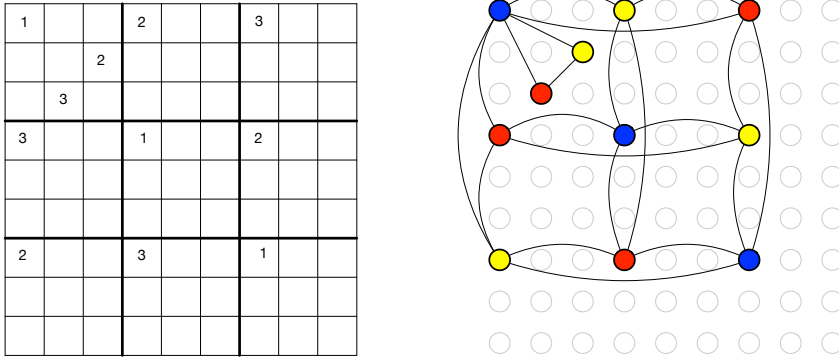
Figure 3.7: A Sudoku game board and the corresponding colored graph.

For example, if the number 1 is assigned to a square, then write the pencil mark 1 in all the squares in the same row, column, and region. The Pencil Marks technique corresponds to the notion of *saturation* due to [15]. The saturation of a vertex, in Sudoku terms, is the set of numbers that are no longer available. In graph terminology, we have the following definition:

$$\text{saturation}(u) = \{c \mid \exists v.v \in \text{neighbors}(u) \text{ and color}(v) = c\}$$

where neighbors(u) is the set of vertices that share an edge with u.

Using the Pencil Marks technique leads to a simple strategy for filling in numbers: if there is a square with only one possible number left, then choose that number! But what if there are no squares with only one possibility left? One brute-force approach is to try them all: choose the first one and if it ultimately leads to a solution, great. If not, backtrack and choose the next possibility. One good thing about Pencil Marks is that it reduces the degree of branching in the search tree. Nevertheless, backtracking can be horribly time consuming. One way to reduce the amount of backtracking is to use the most-constrained-first heuristic. That is, when choosing a square, always choose one with the fewest possibilities left (the vertex with the highest saturation). The idea is that choosing highly constrained squares earlier rather than later is better because later on there may not be any possibilities left in the highly saturated squares.

However, register allocation is easier than Sudoku because the register allocator can map variables to stack locations when the registers run out. Thus, it makes sense to replace backtracking with greedy search: make the

Algorithm: DSATUR
Input: a graph G
Output: an assignment color[v] for each vertex $v \in G$

$W \leftarrow \text{vertices}(G)$
while $W \neq \emptyset$ **do**
    pick a vertex u from W with the highest saturation,
        breaking ties randomly
    find the lowest color c that is not in $\{\text{color}[v] \ : \ v \in \text{adjacent}(u)\}$
    color[u] $\leftarrow c$
    $W \leftarrow W - \{u\}$

Figure 3.8: The saturation-based greedy graph coloring algorithm.

best choice at the time and keep going. We still wish to minimize the number of colors needed, so we use the most-constrained-first heuristic in the greedy search. Figure 3.8 gives the pseudo-code for a simple greedy algorithm for register allocation based on saturation and the most-constrained-first heuristic. It is roughly equivalent to the DSATUR algorithm [15, 43, 3]. Just as in Sudoku, the algorithm represents colors with integers. The integers 0 through $k-1$ correspond to the k registers that we use for register allocation. The integers k and larger correspond to stack locations. The registers that are not used for register allocation, such as `rax`, are assigned to negative integers. In particular, we assign -1 to `rax` and -2 to `rsp`.

With the DSATUR algorithm in hand, let us return to the running example and consider how to color the interference graph in Figure 3.6. We start by assigning the register nodes to their own color. For example, `rax` is assigned the color -1 and `rsp` is assigned -2. The variables are not yet colored, so they are annotated with a dash. We then update the saturation for vertices that are adjacent to a register, obtaining the following annotated graph. For example, the saturation for `t` is $\{-1, -2\}$ because it interferes with both `rax` and `rsp`.
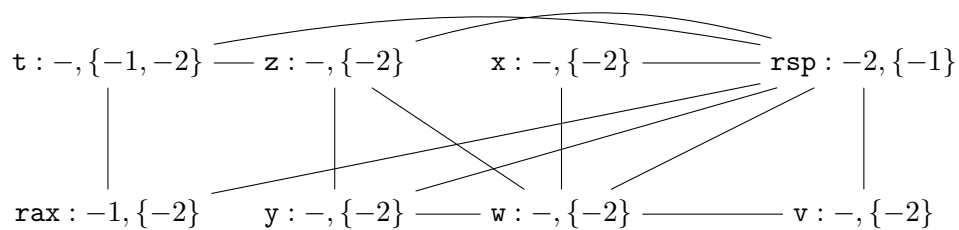
The algorithm says to select a maximally saturated vertex. So we pick `t` and color it with the first available integer, which is 0. We mark 0 as no longer available for `z`, `rax`, and `rsp` because they interfere with `t`.

$$\texttt{t}: 0, \{-1, -2\} - \texttt{z}: -, \{0, -2\} \qquad \texttt{x}: -, \{-2\} \longrightarrow \texttt{rsp}: -2, \{-1, 0\}$$

$$\texttt{rax}: -1, \{0, -2\} \qquad \texttt{y}: -, \{-2\} \longrightarrow \texttt{w}: -, \{-2\} \longrightarrow \texttt{v}: -, \{-2\}$$
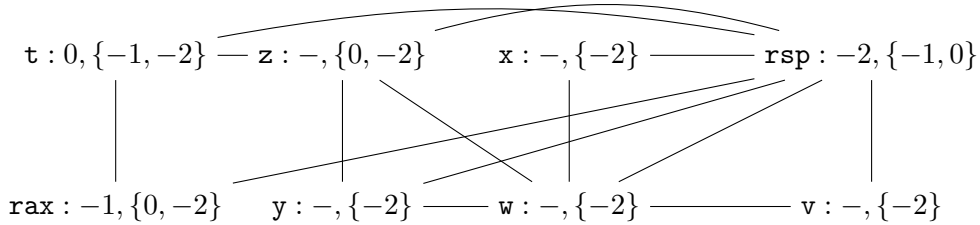
We repeat the process, selecting the next maximally saturated vertex, which is `z`, and color it with the first available number, which is 1. We add 1 to the saturation for the neighboring vertices `t`, `y`, `w`, and `rsp`.

$$\texttt{t}: 0, \{-1, 1, -2\} - \texttt{z}: 1, \{0, -2\} \qquad \texttt{x}: -, \{-2\} \longrightarrow \texttt{rsp}: -2, \{-1, 0, 1\}$$

$$\texttt{rax}: -1, \{0, -2\} \qquad \texttt{y}: -, \{1, -2\} \longrightarrow \texttt{w}: -, \{1, -2\} \longrightarrow \texttt{v}: -, \{-2\}$$
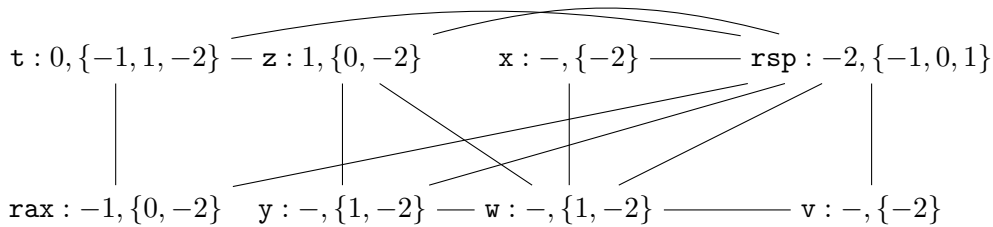
The most saturated vertices are now `w` and `y`. We color `w` with the first available color, which is 0.

$$\texttt{t}: 0, \{-1, 1, -2\} - \texttt{z}: 1, \{0, -2\} \qquad \texttt{x}: -, \{0, -2\} \longrightarrow \texttt{rsp}: -2, \{-1, 0, 1\}$$

$$\texttt{rax}: -1, \{0, -2\} \quad \texttt{y}: -, \{0, 1, -2\} - \texttt{w}: 0, \{1, -2\} \longrightarrow \texttt{v}: -, \{0, -2\}$$
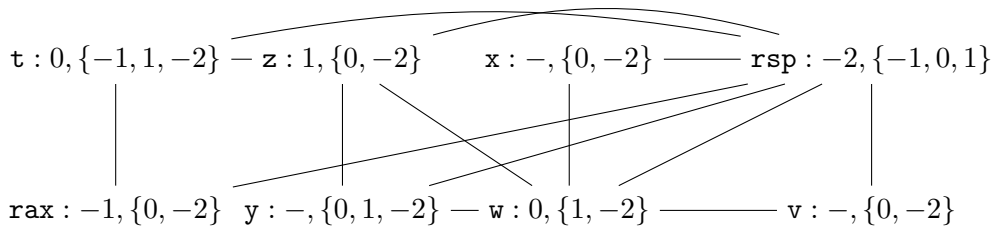
Vertex `y` is now the most highly saturated, so we color `y` with 2. We cannot choose 0 or 1 because those numbers are in `y`'s saturation set. Indeed, `y` interferes with `w` and `z`, whose colors are 0 and 1 respectively.

$$\texttt{t}: 0, \{-1, 1, -2\} - \texttt{z}: 1, \{0, 2, -2\} \qquad \texttt{x}: -, \{0, -2\} \longrightarrow \texttt{rsp}: -2, \{-1, 0, 1, 2\}$$

$$\texttt{rax}: -1, \{0, -2\} \quad \texttt{y}: 2, \{0, 1, -2\} - \texttt{w}: 0, \{1, 2, -2\} \longrightarrow \texttt{v}: -, \{0, -2\}$$
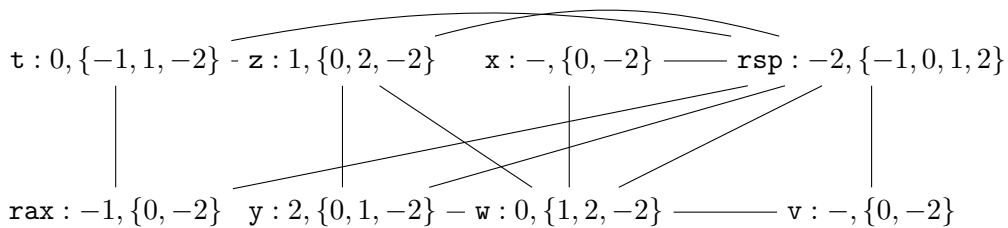
Now `x` and `v` are the most saturated, so we color `v` with 1.

$$\texttt{t}:0,\{-1,1,-2\}\text{ - }\texttt{z}:1,\{0,2,-2\}\quad\texttt{x}:-,\{0,-2\}\text{ —— }\texttt{rsp}:-2,\{-1,0,1,2\}$$

$$\texttt{rax}:-1,\{0,-2\}\quad\texttt{y}:2,\{0,1,-2\}-\texttt{w}:0,\{1,2,-2\}\text{ ——— }\texttt{v}:1,\{0,-2\}$$
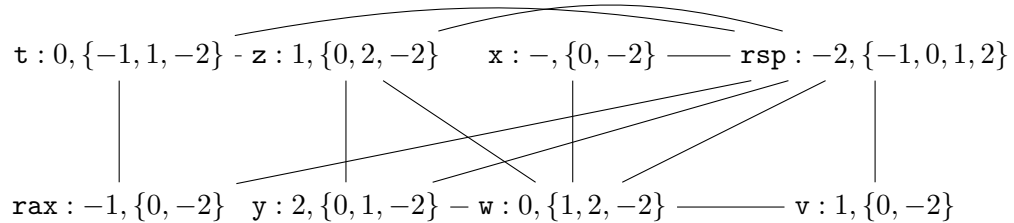
In the last step of the algorithm, we color `x` with 1.

$$\texttt{t}:0,\{-1,1,-2\}\text{ - }\texttt{z}:1,\{0,2,-2\}\quad\texttt{x}:1,\{0,-2\}\text{ —— }\texttt{rsp}:-2,\{-1,0,1,2\}$$

$$\texttt{rax}:-1,\{0,-2\}\quad\texttt{y}:2,\{0,1,-2\}-\texttt{w}:0,\{1,2,-2\}\text{ ——— }\texttt{v}:1,\{0,-2\}$$
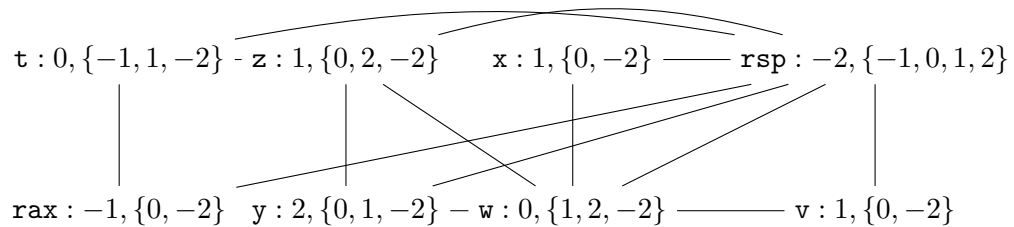
We recommend creating an auxiliary function named `color-graph` that takes an interference graph and a list of all the variables in the program. This function should return a mapping of variables to their colors (represented as natural numbers). By creating this helper function, you will be able to reuse it in Chapter 6 when we add support for functions.

To prioritize the processing of highly saturated nodes inside the `color-graph` function, we recommend using the priority queue data structure (see the side bar on the right). In addition, you will need to maintain a mapping from variables to their "handles" in the priority queue so that you can notify the priority queue when their saturation changes.

With the coloring complete, we finalize the assignment of variables to registers and stack locations. We map the first k colors

---

Priority Queue

A *priority queue* is a collection of items in which the removal of items is governed by priority. In a "min" queue, lower priority items are removed first. An implementation is in `priority_queue.rkt` of the support code.

`(make-pqueue cmp)` constructs an empty priority queue that uses the *cmp* predicate to determine whether its first argument has lower or equal priority to its second argument.

`(pqueue-count queue)` returns the number of items in the queue.

`(pqueue-push! queue item)` inserts the item into the queue and returns a handle for the item in the queue.

`(pqueue-pop! queue)` returns the item with the lowest priority.

`(pqueue-decrease-key! queue handle)` notifies the queue that the priority has decreased for the item associated with the given handle.

to the k registers and the rest of the colors to stack locations. Suppose for the moment that we have just one register to use for register allocation, `rcx`. Then we have the following map from colors to locations.

$$\{0 \mapsto \text{\%rcx}, \ 1 \mapsto \text{-8(\%rbp)}, \ 2 \mapsto \text{-16(\%rbp)}\}$$

Composing this mapping with the coloring, we arrive at the following assignment of variables to locations.

$$\{\text{v} \mapsto \text{-8(\%rbp)}, \text{w} \mapsto \text{\%rcx}, \text{x} \mapsto \text{-8(\%rbp)}, \text{y} \mapsto \text{-16(\%rbp)},$$
$$\text{z} \mapsto \text{-8(\%rbp)}, \text{t} \mapsto \text{\%rcx}\}$$

Adapt the code from the `assign-homes` pass (Section 2.8) to replace the variables with their assigned location. Applying the above assignment to our running example, on the left, yields the program on the right.

```
movq $1, v              movq $1, -8(%rbp)
movq $42, w             movq $42, %rcx
movq v, x               movq -8(%rbp), -8(%rbp)
addq $7, x              addq $7, -8(%rbp)
movq x, y               movq -8(%rbp), -16(%rbp)
movq x, z       ⇒       movq -8(%rbp), -8(%rbp)
addq w, z               addq %rcx, -8(%rbp)
movq y, t               movq -16(%rbp), %rcx
negq t                  negq %rcx
movq z, %rax            movq -8(%rbp), %rax
addq t, %rax            addq %rcx, %rax
jmp conclusion          jmp conclusion
```

Exercise 14. Implement the compiler pass `allocate-registers`. Create five programs that exercise all of the register allocation algorithm, including spilling variables to the stack. Replace `assign-homes` in the list of `passes` in the `run-tests.rkt` script with the three new passes: `uncover-live`, `build-interference`, and `allocate-registers`. Temporarily remove the `print-x86` pass from the list of passes and the call to `compiler-tests`. Run the script to test the register allocator.

3.5 Patch Instructions

The remaining step in the compilation to x86 is to ensure that the instructions have at most one argument that is a memory access. In the running

example, the instruction `movq -8(%rbp), -16(%rbp)` is problematic. The fix is to first move `-8(%rbp)` into `rax` and then move `rax` into `-16(%rbp)`. The two moves from `-8(%rbp)` to `-8(%rbp)` are also problematic, but they can be fixed by simply deleting them. In general, we recommend deleting all the trivial moves whose source and destination are the same location. The following is the output of `patch-instructions` on the running example.

```
movq $1, -8(%rbp)
movq $42, %rcx
movq -8(%rbp), -8(%rbp)
addq $7, -8(%rbp)
movq -8(%rbp), -16(%rbp)                    movq $1, -8(%rbp)
movq -8(%rbp), -8(%rbp)                      movq $42, %rcx
addq %rcx, -8(%rbp)                          addq $7, -8(%rbp)
movq -16(%rbp), %rcx                         movq -8(%rbp), %rax
negq %rcx                          ⇒         movq %rax, -16(%rbp)
movq -8(%rbp), %rax                          addq %rcx, -8(%rbp)
addq %rcx, %rax                              movq -16(%rbp), %rcx
jmp conclusion                               negq %rcx
                                             movq -8(%rbp), %rax
                                             addq %rcx, %rax
                                             jmp conclusion
```

Exercise 15. Implement the `patch-instructions` compiler pass. Insert it after `allocate-registers` in the list of `passes` in the `run-tests.rkt` script. Run the script to test the `patch-instructions` pass.

3.6   Print x86

Recall that the `print-x86` pass generates the prelude and conclusion instructions to satisfy the x86 calling conventions (Section 3.1). With the addition of the register allocator, the callee-saved registers used by the register allocator must be saved in the prelude and restored in the conclusion. In the `allocate-registers` pass, add an entry to the *info* of X86Program named `used-callee` that stores the set of callee-saved registers that were assigned to variables. The `print-x86` pass can then access this information to decide which callee-saved registers need to be saved and restored. When calculating the size of the frame to adjust the `rsp` in the prelude, make sure to take into account the space used for saving the callee-saved registers. Also, don't forget that the frame needs to be a multiple of 16 bytes!

An overview of all of the passes involved in register allocation is shown in Figure 3.9.

Exercise 16. Update the `print-x86` pass as described in this section. In the `run-tests.rkt` script, reinstate `print-x86` in the list of passes and the
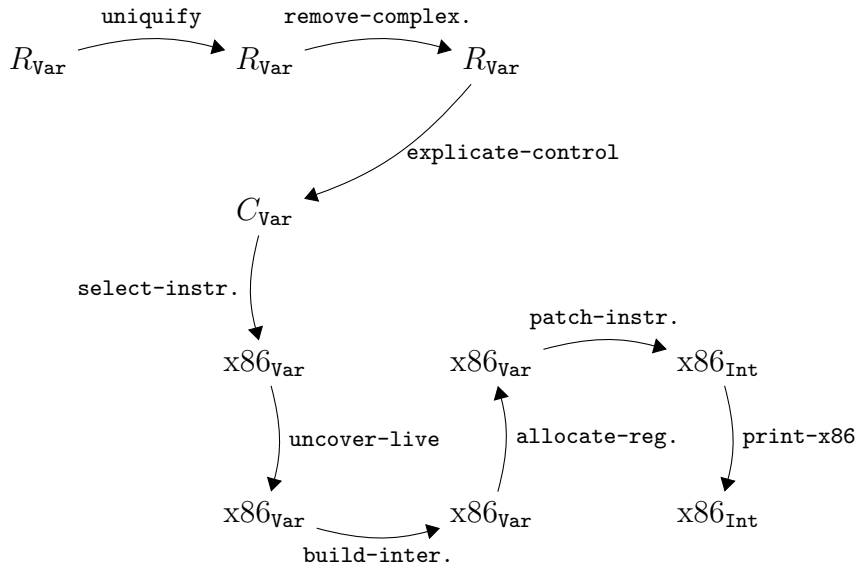
Figure 3.9: Diagram of the passes for $R_{\tt Var}$ with register allocation.

call to `compiler-tests`. Run the script to test the complete compiler for $R_{\tt Var}$ that performs register allocation.

3.7   Challenge: Move Biasing

This section describes an enhancement to the register allocator for students looking for an extra challenge or who have a deeper interest in register allocation.

To motivate the need for move biasing we return to the running example but this time use all of the general purpose registers. So we have the following mapping of color numbers to registers.

$$\{0 \mapsto \texttt{\%rcx},\ 1 \mapsto \texttt{\%rdx},\ 2 \mapsto \texttt{\%rsi}\}$$

Using the same assignment of variables to color numbers that was produced by the register allocator described in the last section, we get the following program.

```
movq $1, v                      movq $1, %rdx
movq $42, w                     movq $42, %rcx
movq v, x                       movq %rdx, %rdx
addq $7, x                      addq $7, %rdx
movq x, y                       movq %rdx, %rsi
movq x, z                       movq %rdx, %rdx
addq w, z            ⇒          addq %rcx, %rdx
movq y, t                       movq %rsi, %rcx
negq t                          negq %rcx
movq z, %rax                    movq %rdx, %rax
addq t, %rax                    addq %rcx, %rax
jmp conclusion                  jmp conclusion
```

In the above output code there are two `movq` instructions that can be re-moved because their source and target are the same. However, if we had put `t`, `v`, `x`, and `y` into the same register, we could instead remove three `movq` instructions. We can accomplish this by taking into account which variables appear in `movq` instructions with which other variables.

We say that two variables *p* and *q* are *move related* if they participate together in a `movq` instruction, that is, `movq p, q` or `movq q, p`. When the register allocator chooses a color for a variable, it should prefer a color that has already been used for a move-related variable (assuming that they do not interfere). Of course, this preference should not override the preference for registers over stack locations. This preference should be used as a tie breaker when choosing between registers or when choosing between stack locations.

We recommend representing the move relationships in a graph, similar to how we represented interference. The following is the *move graph* for our running example.
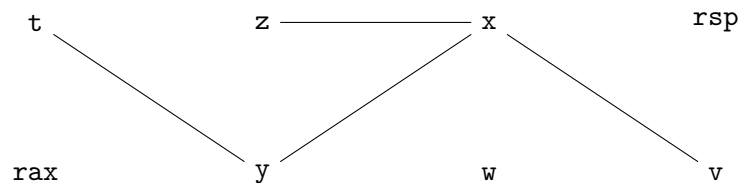
Now we replay the graph coloring, pausing to see the coloring of `y`. Recall

the following configuration. The most saturated vertices were w and y.

$$\texttt{t}:0,\{1,-2\} \;—\; \texttt{z}:1,\{0,-2\} \qquad \texttt{x}:-,\{-2\}\;\;\texttt{rsp}:-2,\{-1,0,1,2\}$$

$$\texttt{rax}:-1,\{0,-2\} \quad \texttt{y}:-,\{1,-2\} \;—\; \texttt{w}:-,\{1,-2\} \;——\; \texttt{v}:-,\{-2\}$$
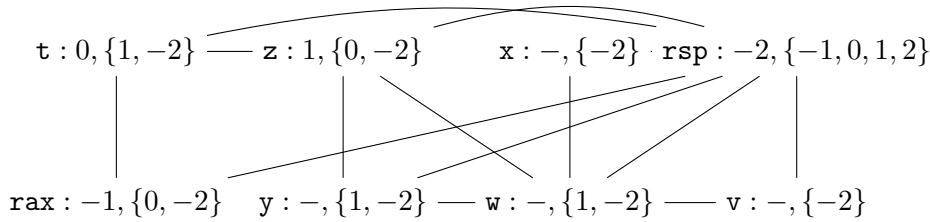
Last time we chose to color w with 0. But this time we see that w is not move related to any vertex, but y is move related to t. So we choose to color y the same color as t, 0.

$$\texttt{t}:0,\{1,-2\} \;—\; \texttt{z}:1,\{0,-2\} \qquad \texttt{x}:-,\{-2\}\;\cdot\;\texttt{rsp}:-2,\{-1,0,1,2\}$$

$$\texttt{rax}:-1,\{0,-2\} \quad \texttt{y}:0,\{1,-2\} \;—\; \texttt{w}:-,\{0,1,-2\} \;—\; \texttt{v}:-,\{-2\}$$
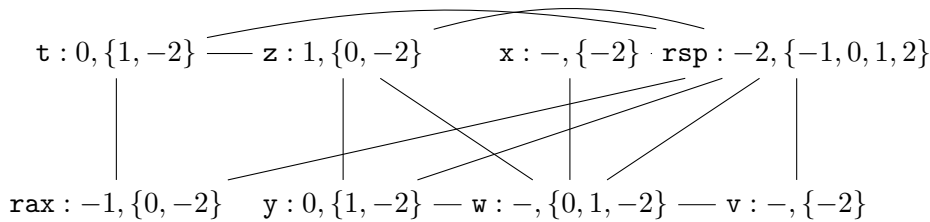
Now w is the most saturated, so we color it 2.

$$\texttt{t}:0,\{1,-2\} \;—\; \texttt{z}:1,\{0,2,-2\} \qquad \texttt{x}:-,\{2,-2\}\texttt{-rsp}:-2,\{-1,0,1,2\}$$

$$\texttt{rax}:-1,\{0,-2\} \quad \texttt{y}:0,\{1,2,-2\} \;—\; \texttt{w}:2,\{0,1,-2\} \;—\; \texttt{v}:-,\{2,-2\}$$
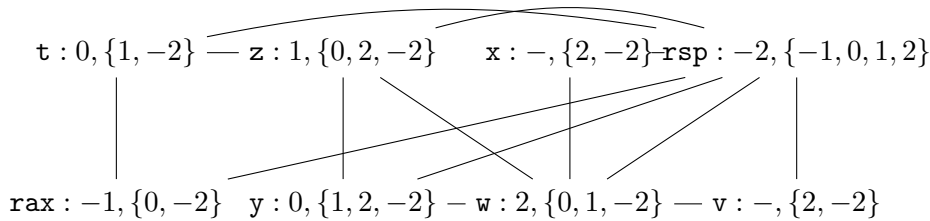
At this point, vertices x and v are most saturated, but x is move related to y and z, so we color x to 0 to match y. Finally, we color v to 0.

$$\texttt{t}:0,\{1,-2\} \;—\; \texttt{z}:1,\{0,2,-2\} \qquad \texttt{x}:0,\{2,-2\}\texttt{-rsp}:-2,\{-1,0,1,2\}$$

$$\texttt{rax}:-1,\{0,-2\} \quad \texttt{y}:0,\{1,2,-2\} \;—\; \texttt{w}:2,\{0,1,-2\} \;—\; \texttt{v}:0,\{2,-2\}$$
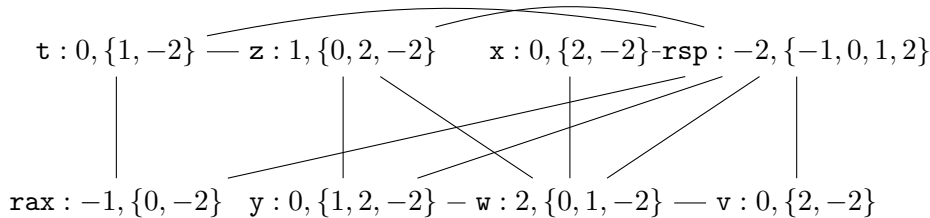
So we have the following assignment of variables to registers.

$$\{\texttt{v} \mapsto \texttt{\%rcx},\; \texttt{w} \mapsto \texttt{\%rsi},\; \texttt{x} \mapsto \texttt{\%rcx},\; \texttt{y} \mapsto \texttt{\%rcx},\; \texttt{z} \mapsto \texttt{\%rdx},\; \texttt{t} \mapsto \texttt{\%rcx}\}$$

We apply this register assignment to the running example, on the left, to obtain the code in the middle. The `patch-instructions` then removes the three trivial moves to obtain the code on the right.

```
movq $1, v              movq $1, %rcx
movq $42, w             movq $42, %rsi
movq v, x               movq %rcx, %rcx          movq $1, %rcx
addq $7, x              addq $7, %rcx            movq $42, %rsi
movq x, y               movq %rcx, %rcx          addq $7, %rcx
movq x, z               movq %rcx, %rdx          movq %rcx, %rdx
addq w, z         ⇒     addq %rsi, %rdx    ⇒     addq %rsi, %rdx
movq y, t               movq %rcx, %rcx          negq %rcx
negq t                  negq %rcx                movq %rdx, %rax
movq z, %rax            movq %rdx, %rax          addq %rcx, %rax
addq t, %rax            addq %rcx, %rax          jmp conclusion
jmp conclusion          jmp conclusion
```

Exercise 17. Change your implementation of `allocate-registers` to take move biasing into account. Create two new tests that include at least one opportunity for move biasing and visually inspect the output x86 programs to make sure that your move biasing is working properly. Make sure that your compiler still passes all of the tests.

Figure 3.10 shows the x86 code generated for the running example (Figure 3.1) with register allocation and move biasing. To demonstrate both the use of registers and the stack, we have limited the register allocator to use just two registers: `rbx` and `rcx`. In the prelude of the `main` function, we push `rbx` onto the stack because it is a callee-saved register and it was assigned to variable by the register allocator. We subtract 8 from the `rsp` at the end of the prelude to reserve space for the one spilled variable. After that subtraction, the `rsp` is aligned to 16 bytes.

Moving on the the `start` block, we see how the registers were allocated. Variables v, x, and y were assigned to `rbx` and variable z was assigned to `rcx`. Variable w was spilled to the stack location `-16(%rbp)`. Recall that the prelude saved the callee-save register `rbx` onto the stack. The spilled variables must be placed lower on the stack than the saved callee-save registers, so in this case w is placed at `-16(%rbp)`.

In the `conclusion`, we undo the work that was done in the prelude. We move the stack pointer up by 8 bytes (the room for spilled variables), then we pop the old values of `rbx` and `rbp` (callee-saved registers), and finish with `retq` to return control to the operating system.

```
start:
        movq    $1, %rbx
        movq    $42, -16(%rbp)
        addq    $7, %rbx
        movq    %rbx, %rcx
        addq    -16(%rbp), %rcx
        negq    %rbx
        movq    %rcx, %rax
        addq    %rbx, %rax
        jmp conclusion

        .globl main
main:
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %rbx
        subq    $8, %rsp
        jmp start

conclusion:
        addq    $8, %rsp
        popq    %rbx
        popq    %rbp
        retq
```

Figure 3.10: The x86 output from the running example (Figure 3.1).

4

Booleans and Control Flow

The $R_{\texttt{Int}}$ and $R_{\texttt{Var}}$ languages only have a single kind of value, integers. In this chapter we add a second kind of value, the Booleans, to create the $R_{\texttt{If}}$ language. The Boolean values *true* and *false* are written #t and #f respectively in Racket. The $R_{\texttt{If}}$ language includes several operations that involve Booleans (and, not, eq?, <, etc.) and the conditional if expression. With the addition of if, programs can have non-trivial control flow which impacts explicate-control and liveness analysis. Also, because we now have two kinds of values, we need to handle programs that apply an operation to the wrong kind of value, such as (not 1).

There are two language design options for such situations. One option is to signal an error and the other is to provide a wider interpretation of the operation. The Racket language uses a mixture of these two options, depending on the operation and the kind of value. For example, the result of (not 1) in Racket is #f because Racket treats non-zero integers as if they were #t. On the other hand, (car 1) results in a run-time error in Racket because car expects a pair.

Typed Racket makes similar design choices as Racket, except much of the error detection happens at compile time instead of run time. Typed Racket accepts and runs (not 1), producing #f. But in the case of (car 1), Typed Racket reports a compile-time error because Typed Racket expects the type of the argument to be of the form (Listof T) or (Pairof T1 T2).

The $R_{\texttt{If}}$ language performs type checking during compilation like Typed Racket. In Chapter 8 we study the alternative choice, that is, a dynamically typed language like Racket. The $R_{\texttt{If}}$ language is a subset of Typed Racket; for some operations we are more restrictive, for example, rejecting (not 1).

This chapter is organized as follows. We begin by defining the syntax

and interpreter for the $R_{\texttt{If}}$ language (Section 4.1). We then introduce the idea of type checking and build a type checker for $R_{\texttt{If}}$ (Section 4.2). To compile $R_{\texttt{If}}$ we need to enlarge the intermediate language $C_{\texttt{Var}}$ into $C_{\texttt{If}}$ (Section 4.3) and $x86_{\texttt{Int}}$ into $x86_{\texttt{If}}$ (Section 4.4). The remaining sections of this chapter discuss how our compiler passes change to accommodate Booleans and conditional control flow. There is one new pass, named `shrink`, that translates some operators into others, thereby reducing the number of operators that need to be handled in later passes. The largest changes occur in `explicate-control`, to translate `if` expressions into control-flow graphs (Section 4.8). Regarding register allocation, the liveness analysis now has multiple basic blocks to process and there is the interesting question of how to handle conditional jumps.

4.1   The $R_{\texttt{If}}$ Language

The concrete syntax of the $R_{\texttt{If}}$ language is defined in Figure 4.1 and the abstract syntax is defined in Figure 4.2. The $R_{\texttt{If}}$ language includes all of $R_{\texttt{Var}}$ (shown in gray), the Boolean literals `#t` and `#f`, and the conditional `if` expression. We expand the operators to include

1. subtraction on integers,

2. the logical operators `and`, `or` and `not`,

3. the `eq?` operation for comparing two integers or two Booleans, and

4. the `<`, `<=`, `>`, and `>=` operations for comparing integers.

We reorganize the abstract syntax for the primitive operations in Figure 4.2, using only one grammar rule for all of them. This means that the grammar no longer checks whether the arity of an operators matches the number of arguments. That responsibility is moved to the type checker for $R_{\texttt{If}}$, which we introduce in Section 4.2.

Figure 4.3 defines the interpreter for $R_{\texttt{If}}$, which inherits from the interpreter for $R_{\texttt{Var}}$ (Figure 2.3). The literals `#t` and `#f` evaluate to the corresponding Boolean values. The conditional expression (`if` *cnd thn els*) evaluates *cnd* and then either evaluates *thn* or *els* depending on whether *cnd* produced `#t` or `#f`. The logical operations `not` and `and` behave as you might expect, but note that the `and` operation is short-circuiting. That is, given the expression (`and` $e_1 \, e_2$), the expression e_2 is not evaluated if e_1 evaluates to `#f`.

$$
\begin{array}{lll}
bool & ::= & \texttt{\#t} \mid \texttt{\#f} \\
cmp & ::= & \texttt{eq?} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
exp & ::= & int \mid \texttt{(read)} \mid \texttt{(-}\ exp\texttt{)} \mid \texttt{(+}\ exp\ exp\texttt{)} \mid \texttt{(-}\ exp\ exp\texttt{)} \\
& \mid & var \mid \texttt{(let (}[var\ exp]\texttt{)}\ exp\texttt{)} \\
& \mid & bool \mid \texttt{(and}\ exp\ exp\texttt{)} \mid \texttt{(or}\ exp\ exp\texttt{)} \mid \texttt{(not}\ exp\texttt{)} \\
& \mid & \texttt{(}cmp\ exp\ exp\texttt{)} \mid \texttt{(if}\ exp\ exp\ exp\texttt{)} \\
R_{\texttt{If}} & ::= & exp
\end{array}
$$

Figure 4.1: The concrete syntax of $R_{\texttt{If}}$, extending $R_{\texttt{Var}}$ (Figure 2.1) with Booleans and conditionals.

$$
\begin{array}{lll}
bool & ::= & \texttt{\#t} \mid \texttt{\#f} \\
cmp & ::= & \texttt{eq?} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
op & ::= & cmp \mid \texttt{read} \mid \texttt{+} \mid \texttt{-} \mid \texttt{and} \mid \texttt{or} \mid \texttt{not} \\
exp & ::= & \texttt{(Int}\ int\texttt{)} \mid \texttt{(Var}\ var\texttt{)} \mid \texttt{(Let}\ var\ exp\ exp\texttt{)} \\
& \mid & \texttt{(Prim}\ op\ \texttt{(}exp\ldots\texttt{))} \\
& \mid & \texttt{(Bool}\ bool\texttt{)} \mid \texttt{(If}\ exp\ exp\ exp\texttt{)} \\
R_{\texttt{If}} & ::= & \texttt{(Program '()}\ exp\texttt{)}
\end{array}
$$

Figure 4.2: The abstract syntax of $R_{\texttt{If}}$.

With the increase in the number of primitive operations, the interpreter would become repetitive without some care. We refactor the case for `Prim`, moving the code that differs with each operation into the `interp-op` method shown in in Figure 4.4. We handle the `and` operation separately because of its short-circuiting behavior.

4.2 Type Checking $R_{\texttt{If}}$ Programs

It is helpful to think about type checking in two complementary ways. A type checker predicts the type of value that will be produced by each expression in the program. For $R_{\texttt{If}}$, we have just two types, `Integer` and `Boolean`. So a type checker should predict that

```
(+ 10 (- (+ 12 20)))
```

produces an `Integer` while

```
(and (not #f) #t)
```

produces a `Boolean`.

```
(define interp-Rif-class
  (class interp-Rvar-class
    (super-new)

    (define/public (interp-op op) ...)

    (define/override ((interp-exp env) e)
      (define recur (interp-exp env))
      (match e
        [(Bool b) b]
        [(If cnd thn els)
         (match (recur cnd)
           [#t (recur thn)]
           [#f (recur els)])]
        [(Prim 'and (list e1 e2))
         (match (recur e1)
           [#t (match (recur e2) [#t #t] [#f #f])]
           [#f #f])]
        [(Prim op args)
         (apply (interp-op op) (for/list ([e args]) (recur e)))]
        [else ((super interp-exp env) e)]))
  ))

(define (interp-Rif p)
  (send (new interp-Rif-class) interp-program p))
```

Figure 4.3: Interpreter for the $R_{\texttt{If}}$ language. (See Figure 4.4 for `interp-op`.)

```
(define/public (interp-op op)
  (match op
    ['+ fx+]
    ['- fx-]
    ['read read-fixnum]
    ['not (lambda (v) (match v [#t #f] [#f #t]))]
    ['or (lambda (v1 v2)
           (cond [(and (boolean? v1) (boolean? v2))
                  (or v1 v2)]))]
    ['eq? (lambda (v1 v2)
            (cond [(or (and (fixnum? v1) (fixnum? v2))
                       (and (boolean? v1) (boolean? v2))
                       (and (vector? v1) (vector? v2)))
                   (eq? v1 v2)]))]
    ['< (lambda (v1 v2)
          (cond [(and (fixnum? v1) (fixnum? v2))
                 (< v1 v2)]))]
    ['<= (lambda (v1 v2)
           (cond [(and (fixnum? v1) (fixnum? v2))
                  (<= v1 v2)]))]
    ['> (lambda (v1 v2)
          (cond [(and (fixnum? v1) (fixnum? v2))
                 (> v1 v2)]))]
    ['>= (lambda (v1 v2)
           (cond [(and (fixnum? v1) (fixnum? v2))
                  (>= v1 v2)]))]
    [else (error 'interp-op "unknown operator")]))
```

Figure 4.4: Interpreter for the primitive operators in the R_{If} language.

Another way to think about type checking is that it enforces a set of rules about which operators can be applied to which kinds of values. For example, our type checker for $R_{\tt If}$ signals an error for the below expression

```
(not (+ 10 (- (+ 12 20))))
```

The subexpression `(+ 10 (- (+ 12 20)))` has type `Integer` but the type checker enforces the rule that the argument of `not` must be a `Boolean`.

We implement type checking using classes and methods because they provide the open recursion needed to reuse code as we extend the type checker in later chapters, analogous to the use of classes and methods for the interpreters (Section 2.1.1).

We separate the type checker for the $R_{\tt Var}$ fragment into its own class, shown in Figure 4.5. The type checker for $R_{\tt If}$ is shown in Figure 4.6 and it inherits from the type checker for $R_{\tt Var}$. These type checkers are in the files `type-check-Rvar.rkt` and `type-check-Rif.rkt` of the support code. Each type checker is a structurally recursive function over the AST. Given an input expression `e`, the type checker either signals an error or returns an expression and its type (`Integer` or `Boolean`). It returns an expression because there are situations in which we want to change or update the expression.

Next we discuss the `match` cases in `type-check-exp` of Figure 4.5. The type of an integer constant is `Integer`. To handle variables, the type checker uses the environment `env` to map variables to types. Consider the case for `let`. We type check the initializing expression to obtain its type `T` and then associate type `T` with the variable `x` in the environment used to type check the body of the `let`. Thus, when the type checker encounters a use of variable `x`, it can find its type in the environment. Regarding primitive operators, we recursively analyze the arguments and then invoke `type-check-op` to check whether the argument types are allowed.

Several auxiliary methods are used in the type checker. The method `operator-types` defines a dictionary that maps the operator names to their parameter and return types. The `type-equal?` method determines whether two types are equal, which for now simply dispatches to `equal?` (deep equality). The `check-type-equal?` method triggers an error if the two types are not equal. The `type-check-op` method looks up the operator in the `operator-types` dictionary and then checks whether the argument types are equal to the parameter types. The result is the return type of the operator.

Next we discuss the type checker for $R_{\tt If}$ in Figure 4.6. The operator `eq?` requires the two arguments to have the same type. The type of a Boolean

```
(define type-check-Rvar-class
  (class object%
    (super-new)

    (define/public (operator-types)
      '((+ . ((Integer Integer) . Integer))
        (- . ((Integer) . Integer))
        (read . (() . Integer))))

    (define/public (type-equal? t1 t2) (equal? t1 t2))

    (define/public (check-type-equal? t1 t2 e)
      (unless (type-equal? t1 t2)
        (error 'type-check "~a != ~a\nin ~v" t1 t2 e)))

    (define/public (type-check-op op arg-types e)
      (match (dict-ref (operator-types) op)
        [`(,param-types . ,return-type)
         (for ([at arg-types] [pt param-types])
           (check-type-equal? at pt e))
         return-type]
        [else (error 'type-check-op "unrecognized ~a" op)]))

    (define/public (type-check-exp env)
      (lambda (e)
        (match e
          [(Int n) (values (Int n) 'Integer)]
          [(Var x) (values (Var x) (dict-ref env x))]
          [(Let x e body)
           (define-values (e^ Te) ((type-check-exp env) e))
           (define-values (b Tb) ((type-check-exp (dict-set env x Te)) body))
           (values (Let x e^ b) Tb)]
          [(Prim op es)
           (define-values (new-es ts)
             (for/lists (exprs types) ([e es]) ((type-check-exp env) e)))
           (values (Prim op new-es) (type-check-op op ts e))]
          [else (error 'type-check-exp "couldn't match" e)])))

    (define/public (type-check-program e)
      (match e
        [(Program info body)
         (define-values (body^ Tb) ((type-check-exp '()) body))
         (check-type-equal? Tb 'Integer body)
         (Program info body^)]
        [else (error 'type-check-Rvar "couldn't match ~a" e)]))
  ))

(define (type-check-Rvar p)
  (send (new type-check-Rvar-class) type-check-program p))
```

Figure 4.5: Type checker for the R_{Var} language.

```
(define type-check-Rif-class
  (class type-check-Rvar-class
    (super-new)
    (inherit check-type-equal?)

    (define/override (operator-types)
      (append '((- . ((Integer Integer) . Integer))
                (and . ((Boolean Boolean) . Boolean))
                (or . ((Boolean Boolean) . Boolean))
                (< . ((Integer Integer) . Boolean))
                (<= . ((Integer Integer) . Boolean))
                (> . ((Integer Integer) . Boolean))
                (>= . ((Integer Integer) . Boolean))
                (not . ((Boolean) . Boolean))
                )
              (super operator-types)))

    (define/override (type-check-exp env)
      (lambda (e)
        (match e
          [(Prim 'eq? (list e1 e2))
           (define-values (e1^ T1) ((type-check-exp env) e1))
           (define-values (e2^ T2) ((type-check-exp env) e2))
           (check-type-equal? T1 T2 e)
           (values (Prim 'eq? (list e1^ e2^)) 'Boolean)]
          [(Bool b) (values (Bool b) 'Boolean)]
          [(If cnd thn els)
           (define-values (cnd^ Tc) ((type-check-exp env) cnd))
           (define-values (thn^ Tt) ((type-check-exp env) thn))
           (define-values (els^ Te) ((type-check-exp env) els))
           (check-type-equal? Tc 'Boolean e)
           (check-type-equal? Tt Te e)
           (values (If cnd^ thn^ els^) Te)]
          [else ((super type-check-exp env) e)])))
    ))

(define (type-check-Rif p)
  (send (new type-check-Rif-class) type-check-program p))
```

Figure 4.6: Type checker for the $R_{\tt If}$ language.

$$
\begin{array}{lll}
atm & ::= & (\texttt{Int } int) \mid (\texttt{Var } var) \mid (\texttt{Bool } bool) \\
cmp & ::= & \texttt{eq?} \mid \texttt{<} \\
exp & ::= & atm \mid (\texttt{Prim read ()}) \\
& \mid & (\texttt{Prim - } (atm)) \mid (\texttt{Prim + } (atm\ atm)) \\
& \mid & (\texttt{Prim 'not } (atm)) \mid (\texttt{Prim '} cmp\ (atm\ atm)) \\
stmt & ::= & (\texttt{Assign (Var } var)\ exp) \\
tail & ::= & (\texttt{Return } exp) \mid (\texttt{Seq } stmt\ tail) \mid (\texttt{Goto } label) \\
& \mid & (\texttt{IfStmt (Prim } cmp\ (atm\ atm))\ (\texttt{Goto } label)\ (\texttt{Goto } label)) \\
C_\text{If} & ::= & (\texttt{CProgram } info\ ((label\ .\ tail)\dots)) \\
\end{array}
$$

Figure 4.7: The abstract syntax of C_If, an extension of C_Var (Figure 2.10).

constant is `Boolean`. The condition of an `if` must be of `Boolean` type and the two branches must have the same type. The `operator-types` function adds dictionary entries for the other new operators.

Exercise 18. Create 10 new test programs in R_If. Half of the programs should have a type error. For those programs, create an empty file with the same base name but with file extension `.tyerr`. For example, if the test `cond_test_14.rkt` is expected to error, then create an empty file named `cond_test_14.tyerr`. This indicates to `interp-tests` and `compiler-tests` that a type error is expected. The other half of the test programs should not have type errors.

In the `run-tests.rkt` script, change the second argument of `interp-tests` and `compiler-tests` to `type-check-Rif`, which causes the type checker to run prior to the compiler passes. Temporarily change the `passes` to an empty list and run the script, thereby checking that the new test programs either type check or not as intended.

4.3 The C_If Intermediate Language

Figure 4.7 defines the abstract syntax of the C_If intermediate language. (The concrete syntax is in the Appendix, Figure 12.3.) Compared to C_Var, the C_If language adds logical and comparison operators to the *exp* non-terminal and the literals `#t` and `#f` to the *arg* non-terminal.

Regarding control flow, C_If adds `goto` and `if` statements to the *tail* non-terminal. The condition of an `if` statement is a comparison operation and the branches are `goto` statements, making it straightforward to compile `if` statements to x86.

$$
\begin{array}{lll}
\textit{bytereg} & ::= & \texttt{ah} \mid \texttt{al} \mid \texttt{bh} \mid \texttt{bl} \mid \texttt{ch} \mid \texttt{cl} \mid \texttt{dh} \mid \texttt{dl} \\
\textit{arg} & ::= & \$\textit{int} \mid \%\textit{reg} \mid \textit{int}(\%\textit{reg}) \mid \%\textit{bytereg} \\
\textit{cc} & ::= & \texttt{e} \mid \texttt{l} \mid \texttt{le} \mid \texttt{g} \mid \texttt{ge} \\
\textit{instr} & ::= & \texttt{addq}\ \textit{arg},\textit{arg} \mid \texttt{subq}\ \textit{arg},\textit{arg} \mid \texttt{negq}\ \textit{arg} \mid \texttt{movq}\ \textit{arg},\textit{arg} \mid \\
& & \texttt{callq}\ \textit{label} \mid \texttt{pushq}\ \textit{arg} \mid \texttt{popq}\ \textit{arg} \mid \texttt{retq} \mid \texttt{jmp}\ \textit{label} \\
& & \textit{label}:\ \textit{instr} \mid \texttt{xorq}\ \textit{arg},\ \textit{arg} \mid \texttt{cmpq}\ \textit{arg},\ \textit{arg} \mid \\
& & \texttt{set}\textit{cc}\ \textit{arg} \mid \texttt{movzbq}\ \textit{arg},\ \textit{arg} \mid \texttt{j}\textit{cc}\ \textit{label} \\
\mathrm{x86_{If}} & ::= & \texttt{.globl main} \\
& & \texttt{main}:\ \textit{instr}\ldots
\end{array}
$$

Figure 4.8: The concrete syntax of $\mathrm{x86_{If}}$ (extends $\mathrm{x86_{Int}}$ of Figure 2.4).

4.4   The $\mathrm{x86_{If}}$ Language

To implement the new logical operations, the comparison operations, and the `if` expression, we need to delve further into the x86 language. Figures 4.8 and 4.9 define the concrete and abstract syntax for the $\mathrm{x86_{If}}$ subset of x86, which includes instructions for logical operations, comparisons, and conditional jumps.

   One challenge is that x86 does not provide an instruction that directly implements logical negation (`not` in R_{If} and C_{If}). However, the `xorq` instruction can be used to encode `not`. The `xorq` instruction takes two arguments, performs a pairwise exclusive-or (XOR) operation on each bit of its arguments, and writes the results into its second argument. Recall the truth table for exclusive-or:

$$
\begin{array}{c|cc}
 & 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 1 & 0
\end{array}
$$

For example, applying XOR to each bit of the binary numbers 0011 and 0101 yields 0110. Notice that in the row of the table for the bit 1, the result is the opposite of the second bit. Thus, the `not` operation can be implemented by `xorq` with 1 as the first argument:

$$
\textit{var}\ \texttt{=}\ (\texttt{not}\ \textit{arg})\texttt{;} \qquad \Rightarrow \qquad
\begin{array}{l}
\texttt{movq}\ \textit{arg},\textit{var} \\
\texttt{xorq \$1},\textit{var}
\end{array}
$$

   Next we consider the x86 instructions that are relevant for compiling the comparison operations. The `cmpq` instruction compares its two arguments

$$
\begin{array}{lll}
\textit{bytereg} & ::= & \texttt{ah} \mid \texttt{al} \mid \texttt{bh} \mid \texttt{bl} \mid \texttt{ch} \mid \texttt{cl} \mid \texttt{dh} \mid \texttt{dl} \\
\textit{arg} & ::= & (\texttt{Imm}\ \textit{int}) \mid (\texttt{Reg}\ \textit{reg}) \mid (\texttt{Deref}\ \textit{reg}\ \textit{int}) \mid (\texttt{ByteReg}\ \textit{bytereg}) \\
\textit{cc} & ::= & \texttt{e} \mid \texttt{l} \mid \texttt{le} \mid \texttt{g} \mid \texttt{ge} \\
\textit{instr} & ::= & (\texttt{Instr addq}\ (\ \textit{arg arg})) \mid (\texttt{Instr subq}\ (\ \textit{arg arg})) \\
& \mid & (\texttt{Instr 'movq}\ (\ \textit{arg arg})) \mid (\texttt{Instr negq}\ (\ \textit{arg})) \\
& \mid & (\texttt{Callq}\ \textit{label int}) \mid (\texttt{Retq}) \mid (\texttt{Pushq}\ \textit{arg}) \mid (\texttt{Popq}\ \textit{arg}) \mid (\texttt{Jmp}\ \textit{label}) \\
& \mid & (\texttt{Instr xorq}\ (\ \textit{arg arg})) \mid (\texttt{Instr cmpq}\ (\ \textit{arg arg})) \\
& \mid & (\texttt{Instr set}\ (\ \textit{cc arg})) \mid (\texttt{Instr movzbq}\ (\ \textit{arg arg})) \\
& \mid & (\texttt{JmpIf}\ \textit{cc label}) \\
\textit{block} & ::= & (\texttt{Block}\ \textit{info}\ (\textit{instr} \ldots)) \\
\text{x86}_\text{If} & ::= & (\texttt{X86Program}\ \textit{info}\ ((\textit{label}\ .\ \textit{block}) \ldots))
\end{array}
$$

Figure 4.9: The abstract syntax of x86$_\text{If}$ (extends x86$_\text{Int}$ of Figure 2.8).

to determine whether one argument is less than, equal, or greater than the other argument. The `cmpq` instruction is unusual regarding the order of its arguments and where the result is placed. The argument order is backwards: if you want to test whether $x < y$, then write `cmpq` y, x. The result of `cmpq` is placed in the special EFLAGS register. This register cannot be accessed directly but it can be queried by a number of instructions, including the `set` instruction. The instruction **set***cc d* puts a `1` or `0` into the destination d depending on whether the comparison comes out according to the condition code *cc* (`e` for equal, `l` for less, `le` for less-or-equal, `g` for greater, `ge` for greater-or-equal). The `set` instruction has an annoying quirk in that its destination argument must be single byte register, such as `al` (L for lower bits) or `ah` (H for higher bits), which are part of the `rax` register. Thankfully, the `movzbq` instruction can be used to move from a single byte register to a normal 64-bit register. The abstract syntax for the `set` instruction differs from the concrete syntax in that it separates the instruction name from the condition code.

The x86 instruction for conditional jump is relevant to the compilation of `if` expressions. The instruction **j***cc label* updates the program counter to point to the instruction after *label* depending on whether the result in the EFLAGS register matches the condition code *cc*, otherwise the jump instruction falls through to the next instruction. Like the abstract syntax for `set`, the abstract syntax for conditional jump separates the instruction name from the condition code. For example, `(JmpIf le foo)` corresponds to `jle foo`. Because the conditional jump instruction relies on the EFLAGS register, it is common for it to be immediately preceded by a `cmpq` instruction

to set the EFLAGS register.

4.5   Shrink the $R_{\mathtt{If}}$ Language

The $R_{\mathtt{If}}$ language includes several operators that are easily expressible with
other operators. For example, subtraction is expressible using addition and
negation.

$$(\texttt{-}\ e_1\ e_2)\quad \Rightarrow\quad (\texttt{+}\ e_1\ (\texttt{-}\ e_2))$$

Several of the comparison operations are expressible using less-than and
logical negation.

$$(\texttt{<=}\ e_1\ e_2)\quad \Rightarrow\quad (\texttt{let}\ (\texttt{[tmp.1}\ e_1\texttt{]})\ (\texttt{not}\ (\texttt{<}\ e_2\ \texttt{tmp.1})))$$

The `let` is needed in the above translation to ensure that expression e_1 is
evaluated before e_2.

By performing these translations in the front-end of the compiler, the
later passes of the compiler do not need to deal with these operators, making
the passes shorter.

Exercise 19. Implement the pass `shrink` to remove subtraction, `and`, `or`,
`<=`, `>`, and `>=` from the language by translating them to other constructs
in $R_{\mathtt{If}}$. Create six test programs that involve these operators. In the
`run-tests.rkt` script, add the following entry for `shrink` to the list of
passes (it should be the only pass at this point).

```
(list "shrink" shrink interp-Rif type-check-Rif)
```

This instructs `interp-tests` to run the intepreter `interp-Rif` and the type
checker `type-check-Rif` on the output of `shrink`. Run the script to test
your compiler on all the test programs.

4.6   Uniquify Variables

Add cases to `uniquify-exp` to handle Boolean constants and `if` expressions.

Exercise 20. Update the `uniquify-exp` for $R_{\mathtt{If}}$ and add the following entry
to the list of `passes` in the `run-tests.rkt` script.

```
(list "uniquify" uniquify interp-Rif type-check-Rif)
```

Run the script to test your compiler.

$$
\begin{aligned}
atm \quad &::= \quad (\texttt{Int } \mathit{int}) \mid (\texttt{Var } \mathit{var}) \mid (\texttt{Bool } \mathit{bool}) \\
exp \quad &::= \quad atm \mid (\texttt{Prim read } ()) \\
&\quad\mid \quad (\texttt{Prim - } (atm)) \mid (\texttt{Prim + } (atm\ atm)) \\
&\quad\mid \quad (\texttt{Let } \mathit{var}\ exp\ exp) \\
&\quad\mid \quad (\texttt{Prim not } (atm)) \\
&\quad\mid \quad (\texttt{Prim } \mathit{cmp}\ (atm\ atm)) \mid (\texttt{If } exp\ exp\ exp) \\
R_2^{\dagger} \quad &::= \quad (\texttt{Program } ()\ exp)
\end{aligned}
$$

Figure 4.10: $R_{\texttt{if}}^{\textsf{ANF}}$ is $R_{\texttt{If}}$ in administrative normal form (ANF).

4.7 Remove Complex Operands

The output language for this pass is $R_{\texttt{if}}^{\textsf{ANF}}$ (Figure 4.10), the administrative normal form of $R_{\texttt{If}}$. The `Bool` form is an atomic expressions but `If` is not. All three sub-expressions of an `If` are allowed to be complex expressions but the operands of `not` and the comparisons must be atoms.

Add cases for `Bool` and `If` to the `rco-exp` and `rco-atom` functions according to whether the output needs to be *exp* or *atm* as specified in the grammar for $R_{\texttt{if}}^{\textsf{ANF}}$. Regarding `If`, it is particularly important to **not** replace its condition with a temporary variable because that would interfere with the generation of high-quality output in the `explicate-control` pass.

Exercise 21. Add cases for Boolean constants and `if` to the `rco-atom` and `rco-exp` functions in `compiler.rkt`. Create three new $R_{\texttt{Int}}$ programs that exercise the interesting code in this pass. In the `run-tests.rkt` script, add the following entry to the list of `passes` and then run the script to test your compiler.

```
(list "remove-complex" remove-complex-opera* interp-Rif type-check-Rif)
```

4.8 Explicate Control

Recall that the purpose of `explicate-control` is to make the order of evaluation explicit in the syntax of the program. With the addition of `if` this get more interesting.

As a motivating example, consider the following program that has an `if` expression nested in the predicate of another `if`.

```
(let ([x (read)])
  (let ([y (read)])
    (if (if (< x 1) (eq? x 0) (eq? x 2))
        (+ y 2)
        (+ y 10))))
```

The naive way to compile `if` and the comparison would be to handle each of them in isolation, regardless of their context. Each comparison would be translated into a `cmpq` instruction followed by a couple instructions to move the result from the EFLAGS register into a general purpose register or stack location. Each `if` would be translated into a `cmpq` instruction followed by a conditional jump. The generated code for the inner `if` in the above example would be as follows.

```
...
cmpq $1, x          ;; (< x 1)
setl %al
movzbq %al, tmp
cmpq $1, tmp        ;; (if ...)
je then_branch_1
jmp else_branch_1
...
```

However, if we take context into account we can do better and reduce the use of `cmpq` instructions for accessing the EFLAG register.

Our goal will be compile `if` expressions so that the relevant comparison instruction appears directly before the conditional jump. For example, we want to generate the following code for the inner `if`.

```
...
cmpq $1, x
je then_branch_1
jmp else_branch_1
...
```

One way to achieve this is to reorganize the code at the level of R_{If}, pushing the outer `if` inside the inner one, yielding the following code.

```
(let ([x (read)])
  (let ([y (read)])
    (if (< x 1)
      (if (eq? x 0)
        (+ y 2)
        (+ y 10))
      (if (eq? x 2)
        (+ y 2)
        (+ y 10)))))
```

Unfortunately, this approach duplicates the two branches from the outer `if` and a compiler must never duplicate code!

We need a way to perform the above transformation but without duplicating code. That is, we need a way for different parts of a program to refer to the same piece of code. At the level of x86 assembly this is straightforward because we can label the code for each branch and insert jumps in all the places that need to execute the branch. In our intermediate language, we need to move away from abstract syntax *trees* and instead use *graphs*. In particular, we use a standard program representation called a *control flow graph* (CFG), due to Frances Elizabeth Allen [4]. Each vertex is a labeled sequence of code, called a *basic block*, and each edge represents a jump to another block. The `CProgram` construct of C_{Var} and C_{If} contains a control flow graph represented as an alist mapping labels to basic blocks. Each basic block is represented by the *tail* non-terminal.

Figure 4.11 shows the output of the `remove-complex-opera*` pass and then the `explicate-control` pass on the example program. We walk through the output program and then discuss the algorithm. Following the order of evaluation in the output of `remove-complex-opera*`, we first have two calls to `(read)` and then the comparison `(< x 1)` in the predicate of the inner `if`. In the output of `explicate-control`, in the block labeled `start`, is two assignment statements followed by a `if` statement that branches to `block40` or `block41`. The blocks associated with those labels contain the translations of the code `(eq? x 0)` and `(eq? x 2)`, respectively. In particular, we start `block40` with the comparison `(eq? x 0)` and then branch to `block38` or `block39`, the two branches of the outer `if`, i.e., `(+ y 2)` and `(+ y 10)`. The story for `block41` is similar.

Recall that in Section 2.6 we implement `explicate-control` for R_{Var} using two mutually recursive functions, `explicate-tail` and `explicate-assign`. The former function translates expressions in tail position whereas the later function translates expressions on the right-hand-side of a `let`. With the addition of `if` expression in R_{If} we have a new kind of position to deal with:

```
(let ([x (read)])                           start:
   (let ([y (read)])                             x = (read);
      (if (if (< x 1)                            y = (read);
              (eq? x 0)                          if (< x 1) goto block40;
              (eq? x 2))                         else goto block41;
          (+ y 2)                          block40:
          (+ y 10))))                           if (eq? x 0) goto block38;
                                                else goto block39;
              ⇓                        ⇒    block41:
                                                if (eq? x 2) goto block38;
(let ([x (read)])                               else goto block39;
   (let ([y (read)])                       block38:
      (if (if (< x 1)                           return (+ y 2);
              (eq? x 0)                    block39:
              (eq? x 2))                       return (+ y 10);
          (+ y 2)
          (+ y 10))))
```

Figure 4.11: Translation from $R_{\texttt{If}}$ to $C_{\texttt{If}}$ via the `explicate-control`.

the predicate position of the `if`. We need another function, `explicate-pred`, that takes an $R_{\texttt{If}}$ expression and two blocks for the then-branch and else-branch. The output of `explicate-pred` is a block. In the following paragraphs we discuss specific cases in the `explicate-pred` function as well as additions to the `explicate-tail` and `explicate-assign` functions.

The skeleton for the `explicate-pred` function is given in Figure 4.12. It has a case for every expression that can have type `Boolean`. We detail a few cases here and leave the rest for the reader. The input to this function is an expression and two blocks, `thn` and `els`, for the two branches of the enclosing `if`. Consider the case for Boolean constants in Figure 4.12. We perform a kind of partial evaluation and output either the `thn` or `els` branch depending on whether the constant is true or false. This case demonstrates that we sometimes discard the `thn` or `els` blocks that are input to `explicate-pred`.

The case for `if` in `explicate-pred` is particularly illuminating because it deals with the challenges we discussed above regarding nested `if` expressions (Figure 4.11). The `thn^` and `els^` branches of the `if` inherit their context from the current one, that is, predicate context. So you should recursively apply `explicate-pred` to the `thn^` and `els^` branches. For both of those recursive calls, pass `thn` and `els` as the extra parameters. Thus, `thn` and `els` may get used twice, once inside each recursive call. As discussed above, to avoid duplicating code, we need to add them to the control-flow graph so

```
(define (explicate-pred cnd thn els)
  (match cnd
    [(Var x) ___]
    [(Let x rhs body) ___]
    [(Prim 'not (list e)) ___]
    [(Prim op es) #:when (or (eq? op 'eq?) (eq? op '<))
     (IfStmt (Prim op arg*) (force (block->goto thn))
             (force (block->goto els)))]
    [(Bool b) (if b thn els)]
    [(If cnd^ thn^ els^) ___]
    [else (error "explicate-pred unhandled case" cnd)]))
```

Figure 4.12: Skeleton for the `explicate-pred` auxiliary function.

that we can instead refer to them by name and execute them with a `goto`. However, as we saw in the cases above for Boolean constants, the blocks `thn` and `els` may not get used at all and we don't want to prematurely add them to the control-flow graph if they end up being discarded.

The solution to this conundrum is to use *lazy evaluation*[41] to delay adding the blocks to the control-flow graph until the points where we know they will be used. Racket provides support for lazy evaluation with the `racket/promise` package. The expression (`delay` $e_1 \ldots e_n$) creates a *promise* in which the evaluation of the expressions is postponed. When (`force` p) is applied to a promise p for the first time, the expressions $e_1 \ldots e_n$ are evaluated and the result of e_n is cached in the promise and returned. If `force` is applied again to the same promise, then the cached result is returned. If `force` is applied to an argument that is not a promise, `force` simply returns the argument.

We use lazy evaluation for the input and output blocks of the functions `explicate-pred` and `explicate-assign` and for the output block of `explicate-tail`. So instead of taking and returning blocks, they take and return promises. Furthermore, when we come to a situation in which we a block might be used more than once, as in the case for `if` in `explicate-pred`, we transform the promise into a new promise that will add the block to the control-flow graph and return a `goto`. The following auxiliary function named `block->goto` accomplishes this task. It begins with `delay` to create a promise. When forced, this promise will force the original promise. If that returns a `goto` (because the block was already added to the control-flow graph), then we return the `goto`. Otherwise we add the block to the control-flow graph with another auxiliary function named `add-node`. That

function returns the label for the new block, which we use to create a `goto`.

```
(define (block->goto block)
  (delay
    (define b (force block))
    (match b
      [(Goto label) (Goto label)]
      [else (Goto (add-node b))])))
```

Returning to the discussion of `explicate-pred` (Figure 4.12), consider the case for comparison operators. This is one of the base cases of the recursive function so we translate the comparison to an `if` statement. We apply `block->goto` to `thn` and `els` to obtain two promises that will add then to the control-flow graph, which we can immediately `force` to obtain the two goto's that form the branches of the `if` statement.

The `explicate-tail` and `explicate-assign` functions need additional cases for Boolean constants and `if`. In the cases for `if`, the two branches inherit the current context, so in `explicate-tail` they are in tail position and in `explicate-assign` they are in assignment position. The `cont` parameter of `explicate-assign` is used in both recursive calls, so make sure to use `block->goto` on it.

The way in which the **shrink** pass transforms logical operations such as `and` and `or` can impact the quality of code generated by `explicate-control`. For example, consider the following program.

```
(if (and (eq? (read) 0) (eq? (read) 1))
    0
    42)
```

The `and` operation should transform into something that the `explicate-pred` function can still analyze and descend through to reach the underlying `eq?` conditions. Ideally, your `explicate-control` pass should generate code similar to the following for the above program.

```
start:
    tmp1 = (read);
    if (eq? tmp1 0) goto block40;
    else goto block39;
block40:
    tmp2 = (read);
    if (eq? tmp2 1) goto block38;
    else goto block39;
block38:
    return 0;
```

```
block39:
    return 42;
```

Exercise 22. Implement the pass `explicate-control` by adding the cases for Boolean constants and `if` to the `explicate-tail` and `explicate-assign`. Implement the auxiliary function `explicate-pred` for predicate contexts. Create test cases that exercise all of the new cases in the code for this pass. Add the following entry to the list of `passes` in `run-tests.rkt` and then run this script to test your compiler.

`(list "explicate-control" explicate-control interp-Cif type-check-Cif)`

4.9   Select Instructions

The `select-instructions` pass translate C_{If} to $x86_{\text{If}}^{\text{Var}}$. Recall that we implement this pass using three auxiliary functions, one for each of the nonterminals *atm*, *stmt*, and *tail*.

For *atm*, we have new cases for the Booleans. We take the usual approach of encoding them as integers, with true as 1 and false as 0.

$$\texttt{\#t} \Rightarrow \texttt{1} \qquad \texttt{\#f} \Rightarrow \texttt{0}$$

For *stmt*, we discuss a couple cases. The `not` operation can be implemented in terms of `xorq` as we discussed at the beginning of this section. Given an assignment *var* = (`not` *atm*);, if the left-hand side *var* is the same as *atm*, then just the `xorq` suffices.

$$var = (\texttt{not } var); \quad \Rightarrow \quad \texttt{xorq \$1, } var$$

Otherwise, a `movq` is needed to adapt to the update-in-place semantics of x86. Let *arg* be the result of translating *atm* to x86. Then we have

$$var = (\texttt{not } atm); \quad \Rightarrow \quad \begin{array}{l} \texttt{movq } arg, \ var \\ \texttt{xorq \$1, } var \end{array}$$

Next consider the cases for `eq?` and less-than comparison. Translating these operations to x86 is slightly involved due to the unusual nature of the `cmpq` instruction discussed above. We recommend translating an assignment from `eq?` into the following sequence of three instructions.

$$var = (\texttt{eq? } atm_1 \ atm_2); \quad \Rightarrow \quad \begin{array}{l} \texttt{cmpq } arg_2, \ arg_1 \\ \texttt{sete \%al} \\ \texttt{movzbq \%al, } var \end{array}$$

Regarding the *tail* non-terminal, we have two new cases: `goto` and `if` statements. Both are straightforward to translate to x86. A `goto` becomes a jump instruction.

$$\texttt{goto } \ell; \quad \Rightarrow \quad \texttt{jmp } \ell$$

An `if` statement becomes a compare instruction followed by a conditional jump (for the "then" branch) and the fall-through is to a regular jump (for the "else" branch).

$$
\begin{array}{lcl}
\texttt{if (eq? } atm_1 \ atm_2 \texttt{) goto } \ell_1; & & \texttt{cmpq } arg_2, \ arg_1 \\
\texttt{else goto } \ell_2; & \Rightarrow & \texttt{je } \ell_1 \\
& & \texttt{jmp } \ell_2
\end{array}
$$

Exercise 23. Expand your `select-instructions` pass to handle the new features of the $R_{\texttt{If}}$ language. Add the following entry to the list of `passes` in `run-tests.rkt`

```
(list "select-instructions" select-instructions interp-pseudo-x86-1)
```

Run the script to test your compiler on all the test programs.

4.10   Register Allocation

The changes required for $R_{\texttt{If}}$ affect liveness analysis, building the interference graph, and assigning homes, but the graph coloring algorithm itself does not change.

4.10.1   Liveness Analysis

Recall that for $R_{\texttt{Var}}$ we implemented liveness analysis for a single basic block (Section 3.2). With the addition of `if` expressions to $R_{\texttt{If}}$, `explicate-control` produces many basic blocks arranged in a control-flow graph. We recommend that you create a new auxiliary function named `uncover-live-CFG` that applies liveness analysis to a control-flow graph.

The first question we is: what order should we process the basic blocks in the control-flow graph? Recall that to perform liveness analysis on a basic block we need to know its live-after set. If a basic block has no successors (i.e. no out-edges in the control flow graph), then it has an empty live-after set and we can immediately apply liveness analysis to it. If a basic block has some successors, then we need to complete liveness analysis on those blocks first. In graph theory, a sequence of nodes is in *topological order* if each vertex comes before its successors. We need the opposite, so we can

transpose the graph before computing a topological order. Use the `tsort` and `transpose` functions of the Racket `graph` package to accomplish this. As an aside, a topological ordering is only guaranteed to exist if the graph does not contain any cycles. That is indeed the case for the control-flow graphs that we generate from R_{If} programs. However, in Chapter 9 we add loops to R_{While} and learn how to handle cycles in the control-flow graph.

You'll need to construct a directed graph to represent the control-flow graph. Do not use the `directed-graph` of the `graph` package because that only allows at most one edge between each pair of vertices, but a control-flow graph may have multiple edges between a pair of vertices. The `multigraph.rkt` file in the support code implements a graph representation that allows multiple edges between a pair of vertices.

The next question is how to analyze jump instructions. Recall that in Section 3.2 we maintain an alist named `label->live` that maps each label to the set of live locations at the beginning of its block. We use `label->live` to determine the live-before set for each (`Jmp` *label*) instruction. Now that we have many basic blocks, `label->live` needs to be updated as we process the blocks. In particular, after performing liveness analysis on a block, we take the live-before set of its first instruction and associate that with the block's label in the `label->live`.

In x86$_{\text{If}}^{\text{Var}}$ we also have the conditional jump (`JmpIf` *cc label*) to deal with. Liveness analysis for this instruction is particularly interesting because during compilation we do not know which way a conditional jump will go. So we do not know whether to use the live-before set for the following instruction or the live-before set for the *label*. However, there is no harm to the correctness of the compiler if we classify more locations as live than the ones that are truly live during a particular execution of the instruction. Thus, we can take the union of the live-before sets from the following instruction and from the mapping for *label* in `label->live`.

The auxiliary functions for computing the variables in an instruction's argument and for computing the variables read-from (R) or written-to (W) by an instruction need to be updated to handle the new kinds of arguments and instructions in x86$_{\text{If}}^{\text{Var}}$.

Exercise 24. Update the `uncover-live` pass and implement the `uncover-live-CFG` auxiliary function to apply liveness analysis to the control-flow graph. Add the following entry to the list of `passes` in the `run-tests.rkt` script.

```
(list "uncover-live" uncover-live interp-pseudo-x86-1)
```

4.10.2   Build the Interference Graph

Many of the new instructions in x86$_{\text{If}}^{\text{Var}}$ can be handled in the same way as the instructions in x86$_{\text{Var}}$. Thus, if your code was already quite general, it will not need to be changed to handle the new instructions. If you code is not general enough, we recommend that you change your code to be more general. For example, you can factor out the computing of the the read and write sets for each kind of instruction into two auxiliary functions.

Note that the `movzbq` instruction requires some special care, similar to the `movq` instruction. See rule number 1 in Section 3.3.

Exercise 25. Update the `build-interference` pass for x86$_{\text{If}}^{\text{Var}}$ and add the following entries to the list of `passes` in the `run-tests.rkt` script.

```
(list "build-interference" build-interference interp-pseudo-x86-1)
(list "allocate-registers" allocate-registers interp-x86-1)
```

Run the script to test your compiler on all the R_{If} test programs.

4.11   Patch Instructions

The second argument of the `cmpq` instruction must not be an immediate value (such as an integer). So if you are comparing two immediates, we recommend inserting a `movq` instruction to put the second argument in `rax`. Also, recall that instructions may have at most one memory reference. The second argument of the `movzbq` must be a register. There are no special restrictions on the jump instructions.

Exercise 26. Update `patch-instructions` pass for x86$_{\text{If}}^{\text{Var}}$. Add the following entry to the list of `passes` in `run-tests.rkt` and then run this script to test your compiler.

```
(list "patch-instructions" patch-instructions interp-x86-1)
```

Figure 4.13 lists all the passes needed for the compilation of R_{If}.

4.12   An Example Translation

Figure 4.14 shows a simple example program in R_{If} translated to x86, showing the results of `explicate-control`, `select-instructions`, and the final x86 assembly code.

$R_{\tt If} \xrightarrow{\text{type-check}} R_{\tt If} \xrightarrow{\text{shrink}} R_{\tt If} \xrightarrow{\text{uniquify}} R_{\tt If} \xrightarrow{\text{remove-complex.}} R_{\tt If}$
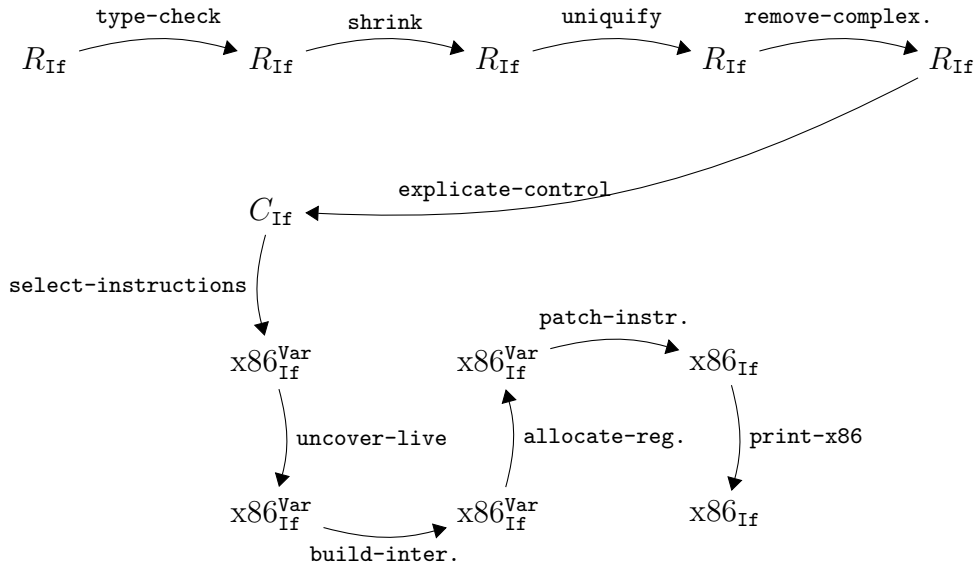
Figure 4.13: Diagram of the passes for $R_{\tt If}$, a language with conditionals.

4.13 Challenge: Remove Jumps

There is an opportunity for optimizing jumps that is apparent in the example of Figure 4.14. The `start` block ends with a jump to `block7953` and there are no other jumps to `block7953` in the rest of the program. In this situation we can avoid the runtime overhead of this jump by merging `block7953` into the preceding block, in this case the `start` block. Figure 4.15 shows the output of `select-instructions` on the left and the result of this optimization on the right.

Exercise 27. Implement a pass named `remove-jumps` that merges basic blocks into their preceding basic block, when there is only one preceding block. The pass should translate from $x86^{\tt Var}_{\tt If}$ to $x86^{\tt Var}_{\tt If}$. In the `run-tests.rkt` script, add the following entry to the list of `passes` between `allocate-registers` and `patch-instructions`.

```
(list "remove-jumps" remove-jumps interp-pseudo-x86-1)
```

Run this script to test your compiler. Check that `remove-jumps` accomplishes the goal of merging basic blocks on several test programs.

```
                                        start:
                                            callq   read_int
(if (eq? (read) 1) 42 0)                    movq    %rax, %rcx
                                            cmpq    $1, %rcx
⇓                                           je block7952
                                            jmp block7953
start:                                  block7953:
    tmp7951 = (read);                       movq    $0, %rax
    if (eq? tmp7951 1)                       jmp conclusion
        goto block7952;                 block7952:
    else                                    movq    $42, %rax
        goto block7953;                     jmp conclusion
block7952:
    return 42;                              .globl main
block7953:                              main:
    return 0;                               pushq   %rbp
                                ⇒           movq    %rsp, %rbp
⇓                                           pushq   %r13
                                            pushq   %r12
start:                                      pushq   %rbx
    callq read_int                          pushq   %r14
    movq %rax, tmp7951                       subq    $0, %rsp
    cmpq $1, tmp7951                         jmp start
    je block7952                        conclusion:
    jmp block7953                            addq    $0, %rsp
block7953:                                  popq    %r14
    movq $0, %rax                           popq    %rbx
    jmp conclusion                          popq    %r12
block7952:                                  popq    %r13
    movq $42, %rax                          popq    %rbp
    jmp conclusion                          retq
```

Figure 4.14: Example compilation of an `if` expression to x86.

```
start:                                      start:
    callq read_int                              callq read_int
    movq %rax, tmp7951                           movq %rax, tmp7951
    cmpq $1, tmp7951                             cmpq $1, tmp7951
    je block7952                                je block7952
    jmp block7953                               je block7952
block7953:                  ⇒                   movq $0, %rax
    movq $0, %rax                               jmp conclusion
    jmp conclusion                          block7952:
block7952:                                      movq $42, %rax
    movq $42, %rax                              jmp conclusion
    jmp conclusion
```

Figure 4.15: Merging basic blocks by removing unnecessary jumps.

5

Tuples and Garbage Collection

In this chapter we study the implementation of mutable tuples (called "vectors" in Racket). This language feature is the first to use the computer's *heap* because the lifetime of a Racket tuple is indefinite, that is, a tuple lives forever from the programmer's viewpoint. Of course, from an implementer's viewpoint, it is important to reclaim the space associated with a tuple when it is no longer needed, which is why we also study *garbage collection garbage collection* techniques in this chapter.

Section 5.1 introduces the $R_{\texttt{Vec}}$ language including its interpreter and type checker. The $R_{\texttt{Vec}}$ language extends the $R_{\texttt{If}}$ language of Chapter 4 with vectors and Racket's `void` value. The reason for including the later is that the `vector-set!` operation returns a value of type `Void`[1].

Section 5.2 describes a garbage collection algorithm based on copying live objects back and forth between two halves of the heap. The garbage collector requires coordination with the compiler so that it can see all of the *root* pointers, that is, pointers in registers or on the procedure call stack.

Sections 5.4 through 5.9 discuss all the necessary changes and additions to the compiler passes, including a new compiler pass named `expose-allocation`.

---

[1]Racket's `Void` type corresponds to what is called the `Unit` type in the programming languages literature. Racket's `Void` type is inhabited by a single value `void` which corresponds to `unit` or `()` in the literature [81].

$$
\begin{array}{rcl}
\textit{type} & ::= & \texttt{Integer} \mid \texttt{Boolean} \mid (\texttt{Vector } \textit{type} \dots) \mid \texttt{Void} \\
\textit{exp} & ::= & \textit{int} \mid (\texttt{read}) \mid (\texttt{- } \textit{exp}) \mid (\texttt{+ } \textit{exp exp}) \mid (\texttt{- } \textit{exp exp}) \\
& \mid & \textit{var} \mid (\texttt{let } ([\textit{var exp}]) \textit{ exp}) \\
& \mid & \texttt{\#t} \mid \texttt{\#f} \mid (\texttt{and } \textit{exp exp}) \mid (\texttt{or } \textit{exp exp}) \mid (\texttt{not } \textit{exp}) \\
& \mid & (\textit{cmp exp exp}) \mid (\texttt{if } \textit{exp exp exp}) \\
& \mid & (\texttt{vector } \textit{exp} \dots) \mid (\texttt{vector-length } \textit{exp}) \\
& \mid & (\texttt{vector-ref } \textit{exp int}) \mid (\texttt{vector-set! } \textit{exp int exp}) \\
& \mid & (\texttt{void}) \mid (\texttt{has-type } \textit{exp type}) \\
R_{\texttt{Vec}} & ::= & \textit{exp}
\end{array}
$$

Figure 5.1: The concrete syntax of $R_{\texttt{Vec}}$, extending $R_{\texttt{If}}$ (Figure 4.1).

```
(let ([t (vector 40 #t (vector 2))])
  (if (vector-ref t 1)
      (+ (vector-ref t 0)
         (vector-ref (vector-ref t 2) 0))
      44))
```

Figure 5.2: Example program that creates tuples and reads from them.

5.1   The $R_{\texttt{Vec}}$ Language

Figure 5.1 defines the concrete syntax for $R_{\texttt{Vec}}$ and Figure 5.3 defines the abstract syntax. The $R_{\texttt{Vec}}$ language includes three new forms: `vector` for creating a tuple, `vector-ref` for reading an element of a tuple, and `vector-set!` for writing to an element of a tuple. The program in Figure 5.2 shows the usage of tuples in Racket. We create a 3-tuple `t` and a 1-tuple that is stored at index 2 of the 3-tuple, demonstrating that tuples are first-class values. The element at index 1 of `t` is `#t`, so the "then" branch of the `if` is taken. The element at index 0 of `t` is `40`, to which we add `2`, the element at index 0 of the 1-tuple. So the result of the program is `42`.

Tuples are our first encounter with heap-allocated data, which raises several interesting issues. First, variable binding performs a shallow-copy when dealing with tuples, which means that different variables can refer to the same tuple, that is, different variables can be *aliases* for the same entity. Consider the following example in which both `t1` and `t2` refer to the same tuple. Thus, the mutation through `t2` is visible when referencing the tuple from `t1`, so the result of this program is `42`.

```
op    ::=  ... | vector | vector-length
exp   ::=  (Int int) | (Var var) | (Let var exp exp)
      |    (Prim op (exp...)) | (Bool bool) | (If exp exp exp)
      |    (Prim vector-ref ( exp (Int int)))
      |    (Prim vector-set! ( exp (Int int) exp))
      |    (Void) | (HasType exp type)
R_Vec ::=  (Program '() exp)
```

Figure 5.3: The abstract syntax of R_{Vec}.

```
(let ([t1 (vector 3 7)])
  (let ([t2 t1])
    (let ([_ (vector-set! t2 0 42)])
      (vector-ref t1 0))))
```

The next issue concerns the lifetime of tuples. Of course, they are created by the `vector` form, but when does their lifetime end? Notice that R_{Vec} does not include an operation for deleting tuples. Furthermore, the lifetime of a tuple is not tied to any notion of static scoping. For example, the following program returns `42` even though the variable `w` goes out of scope prior to the `vector-ref` that reads from the vector it was bound to.

```
(let ([v (vector (vector 44))])
  (let ([x (let ([w (vector 42)])
             (let ([_ (vector-set! v 0 w)])
               0))])
    (+ x (vector-ref (vector-ref v 0) 0))))
```

From the perspective of programmer-observable behavior, tuples live forever. Of course, if they really lived forever, then many programs would run out of memory.[2] A Racket implementation must therefore perform automatic garbage collection.

Figure 5.4 shows the definitional interpreter for the R_{Vec} language. We define the `vector`, `vector-length`, `vector-ref`, and `vector-set!` operations for R_{Vec} in terms of the corresponding operations in Racket. One subtle point is that the `vector-set!` operation returns the `#<void>` value.

---

[2]The R_{Vec} language does not have looping or recursive functions, so it is nigh impossible to write a program in R_{Vec} that will run out of memory. However, we add recursive functions in the next Chapter!

The `#<void>` value can be passed around just like other values inside an $R_{\tt Vec}$ program and a `#<void>` value can be compared for equality with another `#<void>` value. However, there are no other operations specific to the the `#<void>` value in $R_{\tt Vec}$. In contrast, Racket defines the `void?` predicate that returns `#t` when applied to `#<void>` and `#f` otherwise.

Figure 5.5 shows the type checker for $R_{\tt Vec}$, which deserves some explanation. When allocating a vector, we need to know which elements of the vector are pointers (i.e. are also vectors). We can obtain this information during type checking. The type checker in Figure 5.5 not only computes the type of an expression, it also wraps every `vector` creation with the form (`HasType` e T), where T is the vector's type. To create the s-expression for the `Vector` type in Figure 5.5, we use the unquote-splicing operator `,@` to insert the list `t*` without its usual start and end parentheses.

5.2   Garbage Collection

Here we study a relatively simple algorithm for garbage collection that is the basis of state-of-the-art garbage collectors [68, 98, 56, 27, 31, 96]. In particular, we describe a two-space copying collector [101] that uses Cheney's algorithm to perform the copy [21].   Figure 5.6 gives a coarse-grained depiction of what happens in a two-space collector, showing two time steps, prior to garbage collection (on the top) and after garbage collection (on the bottom). In a two-space collector, the heap is divided into two parts named the FromSpace and the ToSpace. Initially, all allocations go to the FromSpace until there is not enough room for the next allocation request. At that point, the garbage collector goes to work to make more room.

The garbage collector must be careful not to reclaim tuples that will be used by the program in the future. Of course, it is impossible in general to predict what a program will do, but we can over approximate the will-be-used tuples by preserving all tuples that could be accessed by *any* program given the current computer state. A program could access any tuple whose address is in a register or on the procedure call stack. These addresses are called the *root set*. In addition, a program could access any tuple that is transitively reachable from the root set. Thus, it is safe for the garbage collector to reclaim the tuples that are not reachable in this way.

So the goal of the garbage collector is twofold:

1. preserve all tuple that are reachable from the root set via a path of pointers, that is, the *live* tuples, and

```
(define interp-Rvec-class
  (class interp-Rif-class
    (super-new)

    (define/override (interp-op op)
      (match op
        ['eq? (lambda (v1 v2)
                (cond [(or (and (fixnum? v1) (fixnum? v2))
                           (and (boolean? v1) (boolean? v2))
                           (and (vector? v1) (vector? v2))
                           (and (void? v1) (void? v2)))
                       (eq? v1 v2)]))]
        ['vector vector]
        ['vector-length vector-length]
        ['vector-ref vector-ref]
        ['vector-set! vector-set!]
        [else (super interp-op op)]
        ))

    (define/override ((interp-exp env) e)
      (define recur (interp-exp env))
      (match e
        [(HasType e t) (recur e)]
        [(Void) (void)]
        [else ((super interp-exp env) e)]
        ))
    ))

(define (interp-Rvec p)
  (send (new interp-Rvec-class) interp-program p))
```

Figure 5.4: Interpreter for the R_{Vec} language.

```
(define type-check-Rvec-class
  (class type-check-Rif-class
    (super-new)
    (inherit check-type-equal?)

    (define/override (type-check-exp env)
      (lambda (e)
        (define recur (type-check-exp env))
        (match e
          [(Void) (values (Void) 'Void)]
          [(Prim 'vector es)
           (define-values (e* t*) (for/lists (e* t*) ([e es]) (recur e)))
           (define t `(Vector ,@t*))
           (values (HasType (Prim 'vector e*) t) t)]
          [(Prim 'vector-ref (list e1 (Int i)))
           (define-values (e1^ t) (recur e1))
           (match t
             [`(Vector ,ts ...)
              (unless (and (0 . <= . i) (i . < . (length ts)))
                (error 'type-check "index ~a out of bounds\nin ~v" i e))
              (values (Prim 'vector-ref (list e1^ (Int i))) (list-ref ts i))]
             [else (error 'type-check "expect Vector, not ~a\nin ~v" t e)])]
          [(Prim 'vector-set! (list e1 (Int i) arg) )
           (define-values (e-vec t-vec) (recur e1))
           (define-values (e-arg^ t-arg) (recur arg))
           (match t-vec
             [`(Vector ,ts ...)
              (unless (and (0 . <= . i) (i . < . (length ts)))
                (error 'type-check "index ~a out of bounds\nin ~v" i e))
              (check-type-equal? (list-ref ts i) t-arg e)
              (values (Prim 'vector-set! (list e-vec (Int i) e-arg^)) 'Void)]
             [else (error 'type-check "expect Vector, not ~a\nin ~v" t-vec e)])]
          [(Prim 'vector-length (list e))
           (define-values (e^ t) (recur e))
           (match t
             [`(Vector ,ts ...)
              (values (Prim 'vector-length (list e^)) 'Integer)]
             [else (error 'type-check "expect Vector, not ~a\nin ~v" t e)])]
          [(Prim 'eq? (list arg1 arg2))
           (define-values (e1 t1) (recur arg1))
           (define-values (e2 t2) (recur arg2))
           (match* (t1 t2)
             [(`(Vector ,ts1 ...) `(Vector ,ts2 ...)) (void)]
             [(other wise) (check-type-equal? t1 t2 e)])
           (values (Prim 'eq? (list e1 e2)) 'Boolean)]
          [(HasType (Prim 'vector es) t)
           ((type-check-exp env) (Prim 'vector es))]
          [(HasType e1 t)
           (define-values (e1^ t^) (recur e1))
           (check-type-equal? t t^ e)
           (values (HasType e1^ t) t)]
          [else ((super type-check-exp env) e)]
          )))
    ))

(define (type-check-Rvec p)
  (send (new type-check-Rvec-class) type-check-program p))
```

Figure 5.5: Type checker for the R_{Vec} language.

2. reclaim the memory of everything else, that is, the *garbage.*

A copying collector accomplishes this by copying all of the live objects from the FromSpace into the ToSpace and then performs a sleight of hand, treating the ToSpace as the new FromSpace and the old FromSpace as the new ToSpace. In the example of Figure 5.6, there are three pointers in the root set, one in a register and two on the stack. All of the live objects have been copied to the ToSpace (the right-hand side of Figure 5.6) in a way that preserves the pointer relationships. For example, the pointer in the register still points to a 2-tuple whose first element is a 3-tuple and whose second element is a 2-tuple. There are four tuples that are not reachable from the root set and therefore do not get copied into the ToSpace.

The exact situation in Figure 5.6 cannot be created by a well-typed program in R_{Vec} because it contains a cycle. However, creating cycles will be possible once we get to R_{Any}. We design the garbage collector to deal with cycles to begin with so we will not need to revisit this issue.

There are many alternatives to copying collectors (and their bigger siblings, the generational collectors) when its comes to garbage collection, such as mark-and-sweep [73] and reference counting [22]. The strengths of copying collectors are that allocation is fast (just a comparison and pointer increment), there is no fragmentation, cyclic garbage is collected, and the time complexity of collection only depends on the amount of live data, and not on the amount of garbage [101]. The main disadvantages of a two-space copying collector is that it uses a lot of space and takes a long time to perform the copy, though these problems are ameliorated in generational collectors. Racket and Scheme programs tend to allocate many small objects and generate a lot of garbage, so copying and generational collectors are a good fit. Garbage collection is an active research topic, especially concurrent garbage collection [96]. Researchers are continuously developing new techniques and revisiting old trade-offs [12, 57, 85, 25, 86, 80, 54, 42]. Researchers meet every year at the International Symposium on Memory Management to present these findings.

5.2.1 Graph Copying via Cheney's Algorithm

Let us take a closer look at the copying of the live objects. The allocated objects and pointers can be viewed as a graph and we need to copy the part of the graph that is reachable from the root set. To make sure we copy all of the reachable vertices in the graph, we need an exhaustive graph traversal algorithm, such as depth-first search or breadth-first search [78, 24]. Recall
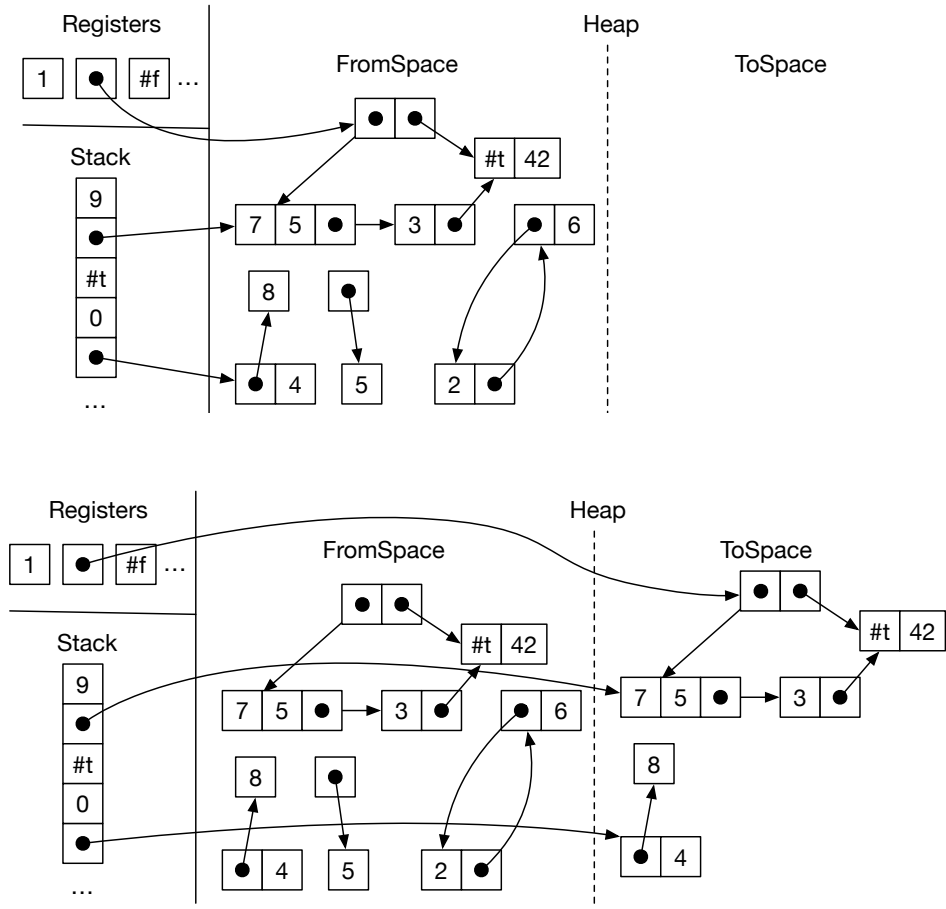
Figure 5.6: A copying collector in action.

that such algorithms take into account the possibility of cycles by marking which vertices have already been visited, so as to ensure termination of the algorithm. These search algorithms also use a data structure such as a stack or queue as a to-do list to keep track of the vertices that need to be visited. We use breadth-first search and a trick due to Cheney [21] for simultaneously representing the queue and copying tuples into the ToSpace.

Figure 5.7 shows several snapshots of the ToSpace as the copy progresses. The queue is represented by a chunk of contiguous memory at the beginning of the ToSpace, using two pointers to track the front and the back of the queue. The algorithm starts by copying all tuples that are immediately reachable from the root set into the ToSpace to form the initial queue. When we copy a tuple, we mark the old tuple to indicate that it has been visited. We discuss how this marking is accomplish in Section 5.2.2. Note that any pointers inside the copied tuples in the queue still point back to the FromSpace. Once the initial queue has been created, the algorithm enters a loop in which it repeatedly processes the tuple at the front of the queue and pops it off the queue. To process a tuple, the algorithm copies all the tuple that are directly reachable from it to the ToSpace, placing them at the back of the queue. The algorithm then updates the pointers in the popped tuple so they point to the newly copied tuples.

Getting back to Figure 5.7, in the first step we copy the tuple whose second element is 42 to the back of the queue. The other pointer goes to a tuple that has already been copied, so we do not need to copy it again, but we do need to update the pointer to the new location. This can be accomplished by storing a *forwarding pointer* to the new location in the old tuple, back when we initially copied the tuple into the ToSpace. This completes one step of the algorithm. The algorithm continues in this way until the front of the queue is empty, that is, until the front catches up with the back.

5.2.2 Data Representation

The garbage collector places some requirements on the data representations used by our compiler. First, the garbage collector needs to distinguish between pointers and other kinds of data. There are several ways to accomplish this.

1. Attached a tag to each object that identifies what type of object it is [73].

2. Store different types of objects in different regions [93].
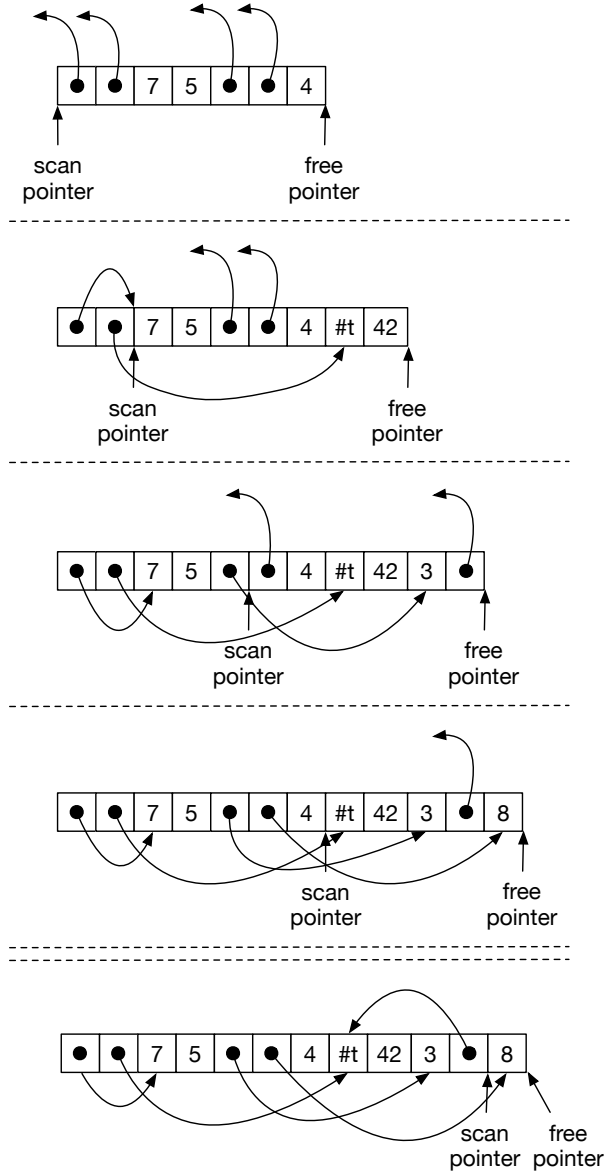
Figure 5.7: Depiction of the Cheney algorithm copying the live tuples.
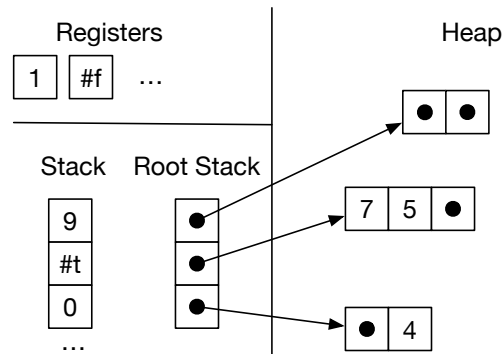
Figure 5.8: Maintaining a root stack to facilitate garbage collection.

3. Use type information from the program to either generate type-specific code for collecting or to generate tables that can guide the collector [6, 46, 29].

Dynamically typed languages, such as Lisp, need to tag objects anyways, so option 1 is a natural choice for those languages. However, $R_{\texttt{Vec}}$ is a statically typed language, so it would be unfortunate to require tags on every object, especially small and pervasive objects like integers and Booleans. Option 3 is the best-performing choice for statically typed languages, but comes with a relatively high implementation complexity. To keep this chapter within a 2-week time budget, we recommend a combination of options 1 and 2, using separate strategies for the stack and the heap.

Regarding the stack, we recommend using a separate stack for pointers, which we call a *root stack* (a.k.a. "shadow stack") [88, 49, 10]. That is, when a local variable needs to be spilled and is of type (Vector $type_1 \ldots type_n$), then we put it on the root stack instead of the normal procedure call stack. Furthermore, we always spill vector-typed variables if they are live during a call to the collector, thereby ensuring that no pointers are in registers during a collection. Figure 5.8 reproduces the example from Figure 5.6 and contrasts it with the data layout using a root stack. The root stack contains the two pointers from the regular stack and also the pointer in the second register.

The problem of distinguishing between pointers and other kinds of data also arises inside of each tuple on the heap. We solve this problem by attaching a tag, an extra 64-bits, to each tuple. Figure 5.9 zooms in on the tags for two of the tuples in the example from Figure 5.6. Note that we have
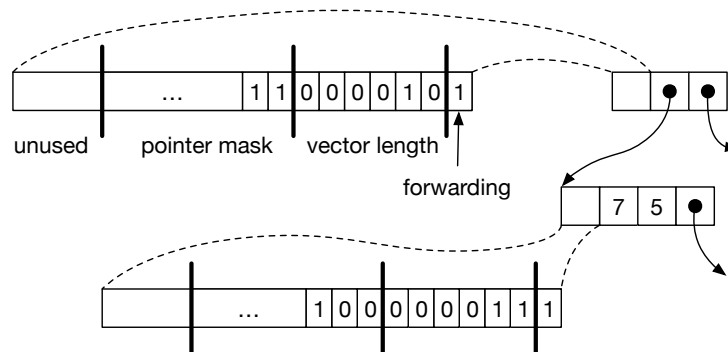
Figure 5.9: Representation of tuples in the heap.

drawn the bits in a big-endian way, from right-to-left, with bit location 0 (the least significant bit) on the far right, which corresponds to the direction of the x86 shifting instructions `salq` (shift left) and `sarq` (shift right). Part of each tag is dedicated to specifying which elements of the tuple are pointers, the part labeled "pointer mask". Within the pointer mask, a 1 bit indicates there is a pointer and a 0 bit indicates some other kind of data. The pointer mask starts at bit location 7. We have limited tuples to a maximum size of 50 elements, so we just need 50 bits for the pointer mask. The tag also contains two other pieces of information. The length of the tuple (number of elements) is stored in bits location 1 through 6. Finally, the bit at location 0 indicates whether the tuple has yet to be copied to the ToSpace. If the bit has value 1, then this tuple has not yet been copied. If the bit has value 0 then the entire tag is a forwarding pointer. (The lower 3 bits of a pointer are always zero anyways because our tuples are 8-byte aligned.)

5.2.3   Implementation of the Garbage Collector

An implementation of the copying collector is provided in the `runtime.c` file. Figure 5.10 defines the interface to the garbage collector that is used by the compiler. The `initialize` function creates the FromSpace, ToSpace, and root stack and should be called in the prelude of the `main` function. The arguments of `initialize` are the root stack size and the heap size. Both need to be multiples of 64 and 16384 is a good choice for both. The `initialize` function puts the address of the beginning of the FromSpace into the global variable `free_ptr`. The global variable `fromspace_end` points to the address that is 1-past the last element of the FromSpace. (We use half-open intervals to represent chunks of memory [28].) The `rootstack_begin` vari-

```
void initialize(uint64_t rootstack_size, uint64_t heap_size);
void collect(int64_t** rootstack_ptr, uint64_t bytes_requested);
int64_t* free_ptr;
int64_t* fromspace_begin;
int64_t* fromspace_end;
int64_t** rootstack_begin;
```

Figure 5.10: The compiler's interface to the garbage collector.

able points to the first element of the root stack.

As long as there is room left in the FromSpace, your generated code can allocate tuples simply by moving the `free_ptr` forward. The amount of room left in FromSpace is the difference between the `fromspace_end` and the `free_ptr`. The `collect` function should be called when there is not enough room left in the FromSpace for the next allocation. The `collect` function takes a pointer to the current top of the root stack (one past the last item that was pushed) and the number of bytes that need to be allocated. The `collect` function performs the copying collection and leaves the heap in a state such that the next allocation will succeed.

The introduction of garbage collection has a non-trivial impact on our compiler passes. We introduce a new compiler pass named `expose-allocation`. We make significant changes to `select-instructions`, `build-interference`, `allocate-registers`, and `print-x86` and make minor changes in several more passes. The following program will serve as our running example. It creates two tuples, one nested inside the other. Both tuples have length one. The program accesses the element in the inner tuple tuple via two vector references.

```
(vector-ref (vector-ref (vector (vector 42)) 0) 0)
```

5.3   Shrink

Recall that the `shrink` pass translates the primitives operators into a smaller set of primitives. Because this pass comes after type checking, but before the passes that require the type information in the `HasType` AST nodes, the `shrink` pass must be modified to wrap `HasType` around each AST node that it generates.

5.4   Expose Allocation

The pass `expose-allocation` lowers the `vector` creation form into a conditional call to the collector followed by the allocation. We choose to place the `expose-allocation` pass before `remove-complex-opera*` because the code generated by `expose-allocation` contains complex operands. We also place `expose-allocation` before `explicate-control` because `expose-allocation` introduces new variables using `let`, but `let` is gone after `explicate-control`.

The output of `expose-allocation` is a language $R_{\texttt{Alloc}}$ that extends $R_{\texttt{Vec}}$ with the three new forms that we use in the translation of the `vector` form.

$$exp \ ::= \ \cdots \mid (\texttt{collect} \ int) \mid (\texttt{allocate} \ int \ type) \mid (\texttt{global-value} \ name)$$

The $(\texttt{collect} \ n)$ form runs the garbage collector, requesting n bytes. It will become a call to the `collect` function in `runtime.c` in `select-instructions`. The $(\texttt{allocate} \ n \ T)$ form creates an tuple of n elements. The T parameter is the type of the tuple: $(\texttt{Vector} \ type_1 \ldots type_n)$ where $type_i$ is the type of the ith element in the tuple. The $(\texttt{global-value} \ name)$ form reads the value of a global variable, such as `free_ptr`.

In the following, we show the transformation for the `vector` form into 1) a sequence of let-bindings for the initializing expressions, 2) a conditional call to `collect`, 3) a call to `allocate`, and 4) the initialization of the vector. In the following, *len* refers to the length of the vector and *bytes* is how many total bytes need to be allocated for the vector, which is 8 for the tag plus *len* times 8.

```
(has-type (vector e_0 ... e_{n-1}) type)
⟹
  (let ([x_0 e_0]) ... (let ([x_{n-1} e_{n-1}])
  (let ([_ (if (< (+ (global-value free_ptr) bytes)
                  (global-value fromspace_end))
             (void)
             (collect bytes))])
  (let ([v (allocate len type)])
  (let ([_ (vector-set! v 0 x_0)]) ...
  (let ([_ (vector-set! v n-1 x_{n-1})])
    v) ... )))) ...)
```

In the above, we suppressed all of the `has-type` forms in the output for the sake of readability. The placement of the initializing expressions $e_0, \ldots, e_{n-1}$ prior to the `allocate` and the sequence of `vector-set!` is important, as those expressions may trigger garbage collection and we cannot have an allocated but uninitialized tuple on the heap during a collection.

Figure 5.11 shows the output of the `expose-allocation` pass on our running example.

5.5   Remove Complex Operands

The new forms `collect`, `allocate`, and `global-value` should all be treated as complex operands. Figure 5.12 shows the grammar for the output language $R_{\mathtt{Vec}}^{\mathsf{ANF}}$ of this pass, which is $R_{\mathtt{Vec}}$ in administrative normal form.

5.6   Explicate Control and the $C_{\mathtt{Vec}}$ language

The output of `explicate-control` is a program in the intermediate language $C_{\mathtt{Vec}}$, whose abstract syntax is defined in Figure 5.13. (The concrete syntax is defined in Figure 12.4 of the Appendix.) The new forms of $C_{\mathtt{Vec}}$ include the `allocate`, `vector-ref`, and `vector-set!`, and `global-value` expressions and the `collect` statement. The `explicate-control` pass can treat these new forms much like the other expression forms that we've already encoutered.

5.7   Select Instructions and the $\mathrm{x86}_{\mathtt{Global}}$ Language

In this pass we generate x86 code for most of the new operations that were needed to compile tuples, including `Allocate`, `Collect`, `vector-ref`, `vector-set!`, and `void`. We compile `GlobalValue` to `Global` because the later has a different concrete syntax (see Figures 5.14 and 5.15).

The `vector-ref` and `vector-set!` forms translate into `movq` instructions. (The plus one in the offset is to get past the tag at the beginning of the tuple representation.)

lhs = (`vector-ref` *vec n*)`;`
$\Longrightarrow$
`movq` *vec′*`, %r11`
`movq` $8(n+1)$`(%r11),` *lhs′*

lhs = (`vector-set!` *vec n arg*)`;`
$\Longrightarrow$
`movq` *vec′*`, %r11`
`movq` *arg′*`,` $8(n+1)$`(%r11)`
`movq $0,` *lhs′*

```
(vector-ref
 (vector-ref
  (let ([vecinit7976
         (let ([vecinit7972 42])
           (let ([collectret7974
                  (if (< (+ (global-value free_ptr) 16)
                         (global-value fromspace_end))
                      (void)
                      (collect 16)
                      )])
             (let ([alloc7971 (allocate 1 (Vector Integer))])
               (let ([initret7973 (vector-set! alloc7971 0 vecinit7972)])
                 alloc7971)
               )
             )
           )
         ])
    (let ([collectret7978
           (if (< (+ (global-value free_ptr) 16)
                  (global-value fromspace_end))
               (void)
               (collect 16)
               )])
      (let ([alloc7975 (allocate 1 (Vector (Vector Integer)))])
        (let ([initret7977 (vector-set! alloc7975 0 vecinit7976)])
          alloc7975)
        )
      )
    )
  0)
 0)
```

Figure 5.11: Output of the `expose-allocation` pass, minus all of the `has-type` forms.

$$
\begin{array}{rcl}
atm & ::= & (\texttt{Int}\ int)\mid(\texttt{Var}\ var)\mid(\texttt{Bool}\ bool)\mid\textbf{(Void)}\\
exp & ::= & atm\mid(\texttt{Prim}\ \texttt{read}\ ())\\
    & \mid & (\texttt{Prim}\ \texttt{-}\ (atm))\mid(\texttt{Prim}\ \texttt{+}\ (atm\ atm))\\
    & \mid & (\texttt{Let}\ var\ exp\ exp)\\
    & \mid & (\texttt{Prim}\ \texttt{'not}\ (atm))\\
    & \mid & (\texttt{Prim}\ cmp\ (atm\ atm))\mid(\texttt{If}\ exp\ exp\ exp)\\
    & \mid & \textbf{(Collect}\ int)\mid\textbf{(Allocate}\ int\ type)\mid\textbf{(GlobalValue}\ var)\\
R_3^{\dagger} & ::= & (\texttt{Program}\ \texttt{'()}\ exp)
\end{array}
$$

Figure 5.12: $R_{\texttt{Vec}}^{\textsf{ANF}}$ is $R_{\texttt{Vec}}$ in administrative normal form (ANF).

$$
\begin{array}{rcl}
atm & ::= & (\texttt{Int}\ int)\mid(\texttt{Var}\ var)\mid(\texttt{Bool}\ bool)\\
cmp & ::= & \texttt{eq?}\mid\texttt{<}\\
exp & ::= & atm\mid(\texttt{Prim}\ \texttt{read}\ ())\\
    & \mid & (\texttt{Prim}\ \texttt{-}\ (atm))\mid(\texttt{Prim}\ \texttt{+}\ (atm\ atm))\\
    & \mid & (\texttt{Prim}\ \texttt{not}\ (atm))\mid(\texttt{Prim}\ cmp\ (atm\ atm))\\
    & \mid & \textbf{(Allocate}\ int\ type)\\
    & \mid & (\texttt{Prim}\ \texttt{'vector-ref}\ (atm\ (\texttt{Int}\ int)))\\
    & \mid & (\texttt{Prim}\ \texttt{'vector-set!}\ (atm\ (\texttt{Int}\ int)\ atm))\\
    & \mid & \textbf{(GlobalValue}\ var)\mid\textbf{(Void)}\\
stmt & ::= & (\texttt{Assign}\ (\texttt{Var}\ var)\ exp)\mid\textbf{(Collect}\ int)\\
tail & ::= & (\texttt{Return}\ exp)\mid(\texttt{Seq}\ stmt\ tail)\mid(\texttt{Goto}\ label)\\
     & \mid & (\texttt{IfStmt}\ (\texttt{Prim}\ cmp\ (atm\ atm))\ (\texttt{Goto}\ label)\ (\texttt{Goto}\ label))\\
C_{\texttt{Vec}} & ::= & (\texttt{CProgram}\ info\ ((label\ .\ tail)\dots))
\end{array}
$$

Figure 5.13: The abstract syntax of $C_{\texttt{Vec}}$, extending $C_{\texttt{If}}$ (Figure 4.7).

The *lhs′*, *vec′*, and *arg′* are obtained by translating *vec* and *arg* to x86. The move of *vec′* to register `r11` ensures that offset expression $-8(n+1)$(`%r11`) contains a register operand. This requires removing `r11` from consideration by the register allocating.

Why not use `rax` instead of `r11`? Suppose we instead used `rax`. Then the generated code for `vector-set!` would be

```
movq vec′, %rax
movq arg′, 8(n + 1)(%rax)
movq $0, lhs′
```

Next, suppose that *arg′* ends up as a stack location, so `patch-instructions` would insert a move through `rax` as follows.

```
movq vec′, %rax
movq arg′, %rax
movq %rax, 8(n + 1)(%rax)
movq $0, lhs′
```

But the above sequence of instructions does not work because we're trying to use `rax` for two different values (*vec′* and *arg′*) at the same time!

We compile the `allocate` form to operations on the `free_ptr`, as shown below. The address in the `free_ptr` is the next free address in the FromSpace, so we copy it into `r11` and then move it forward by enough space for the tuple being allocated, which is $8(len+1)$ bytes because each element is 8 bytes (64 bits) and we use 8 bytes for the tag. We then initialize the *tag* and finally copy the address in `r11` to the left-hand-side. Refer to Figure 5.9 to see how the tag is organized. We recommend using the Racket operations `bitwise-ior` and `arithmetic-shift` to compute the tag during compilation. The type annotation in the `vector` form is used to determine the pointer mask region of the tag.

```
lhs = (allocate len (Vector type...));
⟹
movq free_ptr(%rip), %r11
addq 8(len + 1), free_ptr(%rip)
movq $tag, 0(%r11)
movq %r11, lhs′
```

The `collect` form is compiled to a call to the `collect` function in the runtime. The arguments to `collect` are 1) the top of the root stack and 2) the number of bytes that need to be allocated. We use another dedicated register, `r15`, to store the pointer to the top of the root stack. So `r15` is not available for use by the register allocator.

arg	::=	$\$int \mid \%reg \mid int(\%reg) \mid \%bytereg \mid var(\texttt{\%rip})$
x86_{Global}	::=	`.globl main`
		`main:` *instr* ...

Figure 5.14: The concrete syntax of x86_{Global} (extends x86_{If} of Figure 4.8).

arg	::=	$(\texttt{Int } int) \mid (\texttt{Reg } reg) \mid (\texttt{Deref } reg \ int) \mid (\texttt{ByteReg } reg)$
	|	$(\texttt{Global } var)$
x86_{Global}	::=	$(\texttt{X86Program } info \ ((label \ . \ block) \dots))$

Figure 5.15: The abstract syntax of x86_{Global} (extends x86_{If} of Figure 4.9).

```
(collect bytes)
⟹
movq %r15, %rdi
movq $bytes, %rsi
callq collect
```

The concrete and abstract syntax of the x86_{Global} language is defined in Figures 5.14 and 5.15. It differs from x86_{If} just in the addition of the form for global variables. Figure 5.16 shows the output of the `select-instructions` pass on the running example.

```
block35:                                      start:
    movq free_ptr(%rip), alloc9024                movq $42, vecinit9021
    addq $16, free_ptr(%rip)                      movq free_ptr(%rip), tmp9028
    movq alloc9024, %r11                          movq tmp9028, tmp9029
    movq $131, 0(%r11)                            addq $16, tmp9029
    movq alloc9024, %r11                          movq fromspace_end(%rip), tmp9030
    movq vecinit9025, 8(%r11)                     cmpq tmp9030, tmp9029
    movq $0, initret9026                          jl block39
    movq alloc9024, %r11                          jmp block40
    movq 8(%r11), tmp9034                     block40:
    movq tmp9034, %r11                            movq %r15, %rdi
    movq 8(%r11), %rax                            movq $16, %rsi
    jmp conclusion                               callq 'collect
block36:                                          jmp block38
    movq $0, collectret9027
    jmp block35
block38:
    movq free_ptr(%rip), alloc9020
    addq $16, free_ptr(%rip)
    movq alloc9020, %r11
    movq $3, 0(%r11)
    movq alloc9020, %r11
    movq vecinit9021, 8(%r11)
    movq $0, initret9022
    movq alloc9020, vecinit9025
    movq free_ptr(%rip), tmp9031
    movq tmp9031, tmp9032
    addq $16, tmp9032
    movq fromspace_end(%rip), tmp9033
    cmpq tmp9033, tmp9032
    jl block36
    jmp block37
block37:
    movq %r15, %rdi
    movq $16, %rsi
    callq 'collect
    jmp block35
block39:
    movq $0, collectret9023
    jmp block38
```

Figure 5.16: Output of the `select-instructions` pass.

5.8   Register Allocation

As discussed earlier in this chapter, the garbage collector needs to access all the pointers in the root set, that is, all variables that are vectors. It will be the responsibility of the register allocator to make sure that:

1. the root stack is used for spilling vector-typed variables, and

2. if a vector-typed variable is live during a call to the collector, it must be spilled to ensure it is visible to the collector.

The later responsibility can be handled during construction of the interference graph, by adding interference edges between the call-live vector-typed variables and all the callee-saved registers. (They already interfere with the caller-saved registers.) The type information for variables is in the `Program` form, so we recommend adding another parameter to the `build-interference` function to communicate this alist.

The spilling of vector-typed variables to the root stack can be handled after graph coloring, when choosing how to assign the colors (integers) to registers and stack locations. The `Program` output of this pass changes to also record the number of spills to the root stack.

5.9   Print x86

Figure 5.17 shows the output of the `print-x86` pass on the running example. In the prelude and conclusion of the `main` function, we treat the root stack very much like the regular stack in that we move the root stack pointer (`r15`) to make room for the spills to the root stack, except that the root stack grows up instead of down. For the running example, there was just one spill so we increment `r15` by 8 bytes. In the conclusion we decrement `r15` by 8 bytes.

One issue that deserves special care is that there may be a call to `collect` prior to the initializing assignments for all the variables in the root stack. We do not want the garbage collector to accidentally think that some uninitialized variable is a pointer that needs to be followed. Thus, we zero-out all locations on the root stack in the prelude of `main`. In Figure 5.17, the instruction `movq $0, (%r15)` accomplishes this task. The garbage collector tests each root to see if it is null prior to dereferencing it.

Figure 5.18 gives an overview of all the passes needed for the compilation of R_{Vec}.

```
block35:                                   start:
    movq    free_ptr(%rip), %rcx              movq    $42, %rbx
    addq    $16, free_ptr(%rip)               movq    free_ptr(%rip), %rdx
    movq    %rcx, %r11                        addq    $16, %rdx
    movq    $131, 0(%r11)                     movq    fromspace_end(%rip), %rcx
    movq    %rcx, %r11                        cmpq    %rcx, %rdx
    movq    -8(%r15), %rax                    jl block39
    movq    %rax, 8(%r11)                     movq    %r15, %rdi
    movq    $0, %rdx                          movq    $16, %rsi
    movq    %rcx, %r11                        callq   collect
    movq    8(%r11), %rcx                     jmp block38
    movq    %rcx, %r11
    movq    8(%r11), %rax                     .globl main
    jmp conclusion                    main:
block36:                                       pushq   %rbp
    movq    $0, %rcx                           movq    %rsp, %rbp
    jmp block35                                pushq   %r13
block38:                                       pushq   %r12
    movq    free_ptr(%rip), %rcx               pushq   %rbx
    addq    $16, free_ptr(%rip)               pushq   %r14
    movq    %rcx, %r11                         subq    $0, %rsp
    movq    $3, 0(%r11)                        movq $16384, %rdi
    movq    %rcx, %r11                         movq $16384, %rsi
    movq    %rbx, 8(%r11)                      callq initialize
    movq    $0, %rdx                           movq rootstack_begin(%rip), %r15
    movq    %rcx, -8(%r15)                     movq $0, (%r15)
    movq    free_ptr(%rip), %rcx               addq $8, %r15
    addq    $16, %rcx                          jmp start
    movq    fromspace_end(%rip), %rdx   conclusion:
    cmpq    %rdx, %rcx                         subq $8, %r15
    jl block36                                 addq    $0, %rsp
    movq    %r15, %rdi                         popq    %r14
    movq    $16, %rsi                          popq    %rbx
    callq   collect                            popq    %r12
    jmp block35                                popq    %r13
block39:                                       popq    %rbp
    movq    $0, %rcx                           retq
    jmp block38
```

Figure 5.17: Output of the `print-x86` pass.

5.10 Challenge: Simple Structures

Figure 5.19 defines the concrete syntax for R_3^s, which extends R^3 with support for simple structures. Recall that a `struct` in Typed Racket is a user-defined data type that contains named fields and that is heap allocated, similar to a vector. The following is an example of a structure definition, in this case the definition of a `point` type.

```
(struct point ([x : Integer] [y : Integer]) #:mutable)
```

An instance of a structure is created using function call syntax, with the name of the structure in the function position:

```
(point 7 12)
```

Function-call syntax is also used to read the value in a field of a structure. The function name is formed by the structure name, a dash, and the field name. The following example uses `point-x` and `point-y` to access the `x` and `y` fields of two point instances.

```
(let ([pt1 (point 7 12)])
  (let ([pt2 (point 4 3)])
    (+ (- (point-x pt1) (point-x pt2))
       (- (point-y pt1) (point-y pt2)))))
```

Similarly, to write to a field of a structure, use its set function, whose name starts with `set-`, followed by the structure name, then a dash, then the field name, and concluded with an exclamation mark. The following example uses `set-point-x!` to change the `x` field from 7 to 42.

```
(let ([pt (point 7 12)])
  (let ([_ (set-point-x! pt 42)])
    (point-x pt)))
```

Exercise 28. Extend your compiler with support for simple structures, compiling R_3^s to x86 assembly code. Create five new test cases that use structures and test your compiler.

5.11 Challenge: Generational Collection

The copying collector described in Section 5.2 can incur significant runtime overhead because the call to `collect` takes time proportional to all of the live data. One way to reduce this overhead is to reduce how much data is
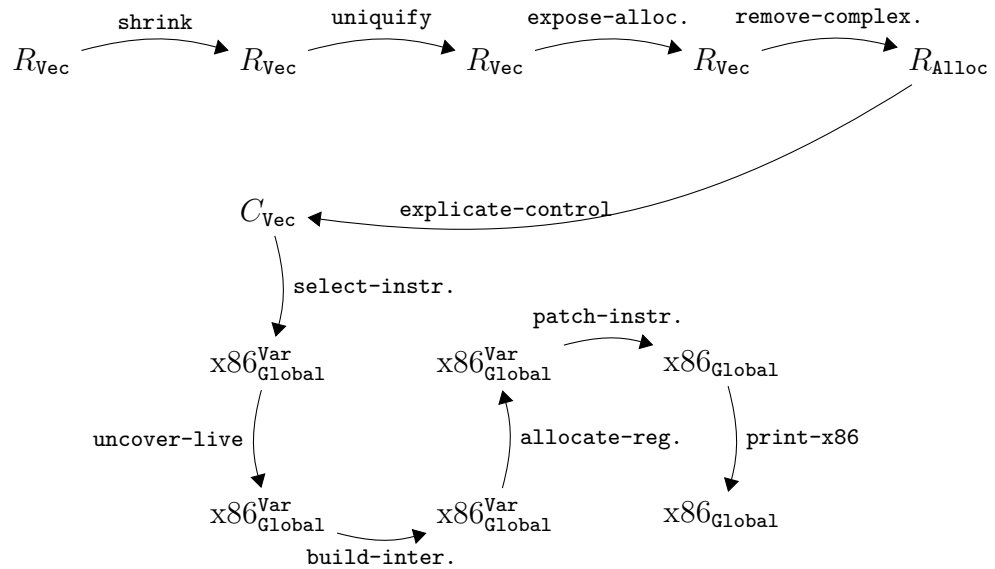
Figure 5.18: Diagram of the passes for $R_{\mathtt{Vec}}$, a language with tuples.

$$
\begin{array}{rcl}
type & ::= & \mathtt{Integer} \mid \mathtt{Boolean} \mid (\mathtt{Vector}\ type\ldots) \mid \mathtt{Void} \mid var \\
cmp & ::= & \mathtt{eq?} \mid \mathtt{<} \mid \mathtt{<=} \mid \mathtt{>} \mid \mathtt{>=} \\
exp & ::= & int \mid (\mathtt{read}) \mid (\mathtt{-}\ exp) \mid (\mathtt{+}\ exp\ exp) \mid (\mathtt{-}\ exp\ exp) \\
 & \mid & var \mid (\mathtt{let}\ ([var\ exp])\ exp) \\
 & \mid & \mathtt{\#t} \mid \mathtt{\#f} \mid (\mathtt{and}\ exp\ exp) \mid (\mathtt{or}\ exp\ exp) \mid (\mathtt{not}\ exp) \\
 & \mid & (cmp\ exp\ exp) \mid (\mathtt{if}\ exp\ exp\ exp) \\
 & \mid & (\mathtt{vector}\ exp\ldots) \mid (\mathtt{vector\text{-}ref}\ exp\ int) \\
 & \mid & (\mathtt{vector\text{-}set!}\ exp\ int\ exp) \\
 & \mid & (\mathtt{void}) \mid (var\ exp\ldots) \\
def & ::= & (\mathtt{struct}\ var\ ([var : type]\ldots)\ \mathtt{\#:mutable}) \\
R_3^s & ::= & def\ldots\ exp
\end{array}
$$

Figure 5.19: The concrete syntax of R_3^s, extending $R_{\mathtt{Vec}}$ (Figure 5.1).

inspected in each call to `collect`. In particular, researchers have observed that recently allocated data is more likely to become garbage then data that has survived one or more previous calls to `collect`. This insight motivated the creation of *generational garbage collectors* that 1) segregates data according to its age into two or more generations, 2) allocates less space for younger generations, so collecting them is faster, and more space for the older generations, and 3) performs collection on the younger generations more frequently then for older generations [101].

For this challenge assignment, the goal is to adapt the copying collector implemented in `runtime.c` to use two generations, one for young data and one for old data. Each generation consists of a FromSpace and a ToSpace. The following is a sketch of how to adapt the `collect` function to use the two generations.

1. Copy the young generation's FromSpace to its ToSpace then switch the role of the ToSpace and FromSpace

2. If there is enough space for the requested number of bytes in the young FromSpace, then return from `collect`.

3. If there is not enough space in the young FromSpace for the requested bytes, then move the data from the young generation to the old one with the following steps:

   (a) If there is enough room in the old FromSpace, copy the young FromSpace to the old FromSpace and then return.

   (b) If there is not enough room in the old FromSpace, then collect the old generation by copying the old FromSpace to the old ToSpace and swap the roles of the old FromSpace and ToSpace.

   (c) If there is enough room now, copy the young FromSpace to the old FromSpace and return. Otherwise, allocate a larger FromSpace and ToSpace for the old generation. Copy the young FromSpace and the old FromSpace into the larger FromSpace for the old generation and then return.

We recommend that you generalize the `cheney` function so that it can be used for all the copies mentioned above: between the young FromSpace and ToSpace, between the old FromSpace and ToSpace, and between the young FromSpace and old FromSpace. This can be accomplished by adding parameters to `cheney` that replace its use of the global variables `fromspace_begin`, `fromspace_end`, `tospace_begin`, and `tospace_end`.

Note that the collection of the young generation does not traverse the old generation. This introduces a potential problem: there may be young data that is only reachable through pointers in the old generation. If these pointers are not taken into account, the collector could throw away young data that is live! One solution, called *pointer recording*, is to maintain a set of all the pointers from the old generation into the new generation and consider this set as part of the root set. To maintain this set, the compiler must insert extra instructions around every `vector-set!`. If the vector being modified is in the old generation, and if the value being written is a pointer into the new generation, than that pointer must be added to the set. Also, if the value being overwritten was a pointer into the new generation, then that pointer should be removed from the set.

Exercise 29. Adapt the `collect` function in `runtime.c` to implement generational garbage collection, as outlined in this section. Update the code generation for `vector-set!` to implement pointer recording. Make sure that your new compiler and runtime passes your test suite.

6

Functions

This chapter studies the compilation of functions similar to those found in the C language. This corresponds to a subset of Typed Racket in which only top-level function definitions are allowed. This kind of function is an important stepping stone to implementing lexically-scoped functions, that is, `lambda` abstractions, which is the topic of Chapter 7.

6.1   The $R_{\mathtt{Fun}}$ Language

The concrete and abstract syntax for function definitions and function application is shown in Figures 6.1 and 6.2, where we define the $R_{\mathtt{Fun}}$ language. Programs in $R_{\mathtt{Fun}}$ begin with zero or more function definitions. The function names from these definitions are in-scope for the entire program, including all other function definitions (so the ordering of function definitions does not matter). The concrete syntax for function application is ($exp\ exp\ldots$) where the first expression must evaluate to a function and the rest are the arguments. The abstract syntax for function application is ($\mathtt{Apply}\ exp\ exp\ldots$).

   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, we introduce a function type, written

   ($type_1\ \cdots\ type_n$ -> $type_r$)

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 Racket functions) is that they are not lexically scoped. That is, the only external entities that can be referenced from inside a function body are other globally-defined functions. The syntax of $R_{\mathtt{Fun}}$ prevents functions from being nested inside each other.

$$
\begin{array}{lll}
\textit{type} & ::= & \texttt{Integer} \mid \texttt{Boolean} \mid (\texttt{Vector } \textit{type}\ldots) \mid \texttt{Void} \mid (\textit{type}\ldots \texttt{ -> } \textit{type}) \\
\textit{cmp} & ::= & \texttt{eq?} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\textit{exp} & ::= & \textit{int} \mid (\texttt{read}) \mid (\texttt{- } \textit{exp}) \mid (\texttt{+ } \textit{exp} \textit{ exp}) \mid (\texttt{- } \textit{exp} \textit{ exp}) \\
& \mid & \textit{var} \mid (\texttt{let } ([\textit{var} \textit{ exp}]) \textit{ exp}) \\
& \mid & \texttt{\#t} \mid \texttt{\#f} \mid (\texttt{and } \textit{exp} \textit{ exp}) \mid (\texttt{or } \textit{exp} \textit{ exp}) \mid (\texttt{not } \textit{exp}) \\
& \mid & (\textit{cmp } \textit{exp} \textit{ exp}) \mid (\texttt{if } \textit{exp} \textit{ exp} \textit{ exp}) \\
& \mid & (\texttt{vector } \textit{exp}\ldots) \mid (\texttt{vector-ref } \textit{exp} \textit{ int}) \\
& \mid & (\texttt{vector-set! } \textit{exp} \textit{ int} \textit{ exp}) \mid (\texttt{void}) \mid (\texttt{has-type } \textit{exp} \textit{ type}) \\
& \mid & (\textit{exp} \textit{ exp}\ldots) \\
\textit{def} & ::= & (\texttt{define } (\textit{var } [\textit{var}\!:\!\textit{type}]\ldots)\!:\!\textit{type} \textit{ exp}) \\
R_{\texttt{Fun}} & ::= & \textit{def}\ldots \textit{ exp}
\end{array}
$$

Figure 6.1: The concrete syntax of $R_{\texttt{Fun}}$, extending $R_{\texttt{Vec}}$ (Figure 5.1).

$$
\begin{array}{lll}
\textit{exp} & ::= & (\texttt{Int } \textit{int})(\texttt{Var } \textit{var}) \mid (\texttt{Let } \textit{var} \textit{ exp} \textit{ exp}) \\
& \mid & (\texttt{Prim } \textit{op} \textit{ } (\textit{exp}\ldots)) \\
& \mid & (\texttt{Bool } \textit{bool}) \mid (\texttt{If } \textit{exp} \textit{ exp} \textit{ exp}) \\
& \mid & (\texttt{Void}) \mid (\texttt{HasType } \textit{exp} \textit{ type}) \mid (\texttt{Apply } \textit{exp} \textit{ exp}\ldots) \\
\textit{def} & ::= & (\texttt{Def } \textit{var} \textit{ } ([\textit{var}\!:\!\textit{type}]\ldots) \textit{ type } \texttt{'() } \textit{exp}) \\
R_{\texttt{Fun}} & ::= & (\texttt{ProgramDefsExp '() } (\textit{def}\ldots)) \textit{ exp})
\end{array}
$$

Figure 6.2: The abstract syntax of $R_{\texttt{Fun}}$, extending $R_{\texttt{Vec}}$ (Figure 5.3).

```
(define (map-vec [f : (Integer -> Integer)]
                 [v : (Vector Integer Integer)])
       : (Vector Integer Integer)
  (vector (f (vector-ref v 0)) (f (vector-ref v 1))))

(define (add1 [x : Integer]) : Integer
  (+ x 1))

(vector-ref (map-vec add1 (vector 0 41)) 1)
```

Figure 6.3: Example of using functions in $R_{\tt Fun}$.

The program in Figure 6.3 is a representative example of defining and using functions in $R_{\tt Fun}$. We define a function `map-vec` that applies some other function `f` to both elements of a vector and returns a new vector containing the results. We also define a function `add1`. The program applies `map-vec` to `add1` and `(vector 0 41)`. The result is `(vector 1 42)`, from which we return the `42`.

The definitional interpreter for $R_{\tt Fun}$ is in Figure 6.4. The case for the `ProgramDefsExp` form is responsible for setting up the mutual recursion between the top-level function definitions. We use the classic back-patching approach that uses mutable variables and makes two passes over the function definitions [60]. In the first pass we set up the top-level environment using a mutable cons cell for each function definition. Note that the `lambda` value for each function is incomplete; it does not yet include the environment. Once the top-level environment is constructed, we then iterate over it and update the `lambda` values to use the top-level environment.

The type checker for $R_{\tt Fun}$ is is in Figure 6.5.

6.2 Functions in x86

The x86 architecture provides a few features to support the implementation of functions. We have already seen that x86 provides labels 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 and PC-relative addressing. For example, the following puts the address of the `add1` label into the `rbx` register.

```
leaq add1(%rip), %rbx
```

```
(define interp-Rfun-class
  (class interp-Rvec-class
    (super-new)

    (define/override ((interp-exp env) e)
      (define recur (interp-exp env))
      (match e
        [(Var x) (unbox (dict-ref env x))]
        [(Let x e body)
         (define new-env (dict-set env x (box (recur e))))
         ((interp-exp new-env) body)]
        [(Apply fun args)
         (define fun-val (recur fun))
         (define arg-vals (for/list ([e args]) (recur e)))
         (match fun-val
           [`(function (,xs ...) ,body ,fun-env)
            (define params-args (for/list ([x xs] [arg arg-vals])
                                  (cons x (box arg))))
            (define new-env (append params-args fun-env))
            ((interp-exp new-env) body)]
           [else (error 'interp-exp "expected function, not ~a" fun-val)])]
        [else ((super interp-exp env) e)]
        ))

    (define/public (interp-def d)
      (match d
        [(Def f (list `[,xs : ,ps] ...) rt _ body)
         (cons f (box `(function ,xs ,body ())))]))

    (define/override (interp-program p)
      (match p
        [(ProgramDefsExp info ds body)
         (let ([top-level (for/list ([d ds]) (interp-def d))])
           (for/list ([f (in-dict-values top-level)])
             (set-box! f (match (unbox f)
                           [`(function ,xs ,body ())
                            `(function ,xs ,body ,top-level)])))
           ((interp-exp top-level) body))]))
    ))

(define (interp-Rfun p)
  (send (new interp-Rfun-class) interp-program p))
```

Figure 6.4: Interpreter for the $R_{\texttt{Fun}}$ language.

```
(define type-check-Rfun-class
  (class type-check-Rvec-class
    (super-new)
    (inherit check-type-equal?)

    (define/public (type-check-apply env e es)
      (define-values (e^ ty) ((type-check-exp env) e))
      (define-values (e* ty*) (for/lists (e* ty*) ([e (in-list es)])
                                ((type-check-exp env) e)))
      (match ty
        [`(,ty^* ... -> ,rt)
         (for ([arg-ty ty*] [param-ty ty^*])
           (check-type-equal? arg-ty param-ty (Apply e es)))
         (values e^ e* rt)]))

    (define/override (type-check-exp env)
      (lambda (e)
        (match e
          [(FunRef f)
           (values (FunRef f) (dict-ref env f))]
          [(Apply e es)
           (define-values (e^ es^ rt) (type-check-apply env e es))
           (values (Apply e^ es^) rt)]
          [(Call e es)
           (define-values (e^ es^ rt) (type-check-apply env e es))
           (values (Call e^ es^) rt)]
          [else ((super type-check-exp env) e)])))

    (define/public (type-check-def env)
      (lambda (e)
        (match e
          [(Def f (and p:t* (list `[,xs : ,ps] ...)) rt info body)
           (define new-env (append (map cons xs ps) env))
           (define-values (body^ ty^) ((type-check-exp new-env) body))
           (check-type-equal? ty^ rt body)
           (Def f p:t* rt info body^)])))

    (define/public (fun-def-type d)
      (match d
        [(Def f (list `[,xs : ,ps] ...) rt info body) `(,@ps -> ,rt)]))

    (define/override (type-check-program e)
      (match e
        [(ProgramDefsExp info ds body)
         (define new-env (for/list ([d ds])
                           (cons (Def-name d) (fun-def-type d))))
         (define ds^ (for/list ([d ds]) ((type-check-def new-env) d)))
         (define-values (body^ ty) ((type-check-exp new-env) body))
         (check-type-equal? ty 'Integer body)
         (ProgramDefsExp info ds^ body^)]))))

(define (type-check-Rfun p)
  (send (new type-check-Rfun-class) type-check-program p))
```

Figure 6.5: Type checker for the $R_{\texttt{Fun}}$ language.

The instruction pointer register `rip` (aka. the program counter) always points to the next instruction to be executed. When combined with an label, as in `add1(%rip)`, the linker computes the distance d between the address of `add1` and where the `rip` would be at that moment and then changes `add1(%rip)` to d`(%rip)`, which at runtime will compute the address of `add1`.

In Section 2.2 we used of the `callq` instruction to jump to a function whose location is given by a label. To support function calls in this chapter we instead will be jumping to a function whose location is given by an address in a register, that is, we need to make an *indirect function call.* The x86 syntax for this is a `callq` instruction but with an asterisk before the register name.

```
callq *%rbx
```

6.2.1   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 (1) parameter passing, recall that the following six registers are used to pass arguments to a function, in this order.

```
rdi rsi rdx rcx r8 r9
```

If there are more than six arguments, then the convention is to use space on the frame of the caller for the rest of the arguments. However, to ease the implementation of efficient tail calls (Section 6.2.2), we arrange to never need more than six arguments. Also recall that the register `rax` is for the return value of the function.

Regarding (2) frames and the procedure call stack, recall from Section 2.2 that the stack grows down, with each function call using a chunk of space 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 are storing variables of vector type on the root stack. So the prelude needs to move the root stack pointer `r15` up 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 pointer to a vector. Otherwise the garbage collector will interpret the garbage bits in those slots as memory addresses and try to traverse them, causing serious mayhem!

Regarding (3) the sharing of registers between different functions, recall from Section 3.1 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 get overwritten with arbitrary values by the callee. That is why we recommend in Section 3.1 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. That is why we recommended in Section 3.1 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.

Also 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 offset from the base pointer (Section 2.2). Figure 6.6 shows the general layout of the caller and callee frames.

6.2.2 Efficient Tail Calls

In general, the amount of stack space used by a program is determined by the longest chain of nested function calls. That is, if function f_1 calls f_2, f_2 calls f_3, ..., and f_{n-1} calls f_n, then the amount of stack space is bounded by $O(n)$. The depth n can grow quite large in the case of recursive or mutually recursive functions. However, in some cases we can arrange to use only constant space, i.e. $O(1)$, instead of $O(n)$.

If a function call is the last action in a function body, then that call is said to be a *tail call*. For example, in the following program, the recursive call to `tail-sum` is a tail call.

```
(define (tail-sum [n : Integer] [r : Integer]) : Integer
  (if (eq? n 0)
      r
```

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	
	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	
	. . .	. . .	
	$-8(n+m)$(%rsp)	local variable m	

Figure 6.6: Memory layout of caller and callee frames.

```
    (tail-sum (- n 1) (+ n r))))
```

```
(+ (tail-sum 5 0) 27)
```

At a tail call, the frame of the caller is no longer needed, so we can pop the caller's frame before making the tail call. With this approach, a recursive function that only makes tail calls will only use $O(1)$ stack space. Functional languages like Racket typically rely heavily on recursive functions, so they typically guarantee that all tail calls will be optimized in this way.

However, some care is needed with regards to argument passing in tail calls. As mentioned above, for arguments beyond the sixth, the convention is to use space in the caller's frame for passing arguments. But for a tail call we pop the caller's frame and can no longer use it. Another alternative is to use space in the callee's frame for passing arguments. However, this option is also problematic because the caller and callee's frame overlap in memory. As we begin to copy the arguments from their sources in the caller's frame, the target locations in the callee's frame might overlap with the sources for later arguments! We solve this problem by not using the stack for passing more than six arguments but instead using the heap, as we describe in the Section 6.5.

As mentioned above, for a tail call we pop the caller's frame prior to making the tail call. The instructions for popping a frame are the instructions that we usually place in the conclusion of a function. Thus, we also need to place such code immediately before each tail call. These instructions include restoring the callee-saved registers, so it is good that the argument passing registers are all caller-saved registers.

One last note regarding which instruction to use to make the tail call. When the callee is finished, it should not return to the current function, but it should return to the function that called the current one. Thus, the return address that is already on the stack is the right one, and we should not use `callq` to make the tail call, as that would unnecessarily overwrite the return address. Instead we can simply use the `jmp` instruction. Like the indirect function call, we write an *indirect jump* with a register prefixed with an asterisk. We recommend using `rax` to hold the jump target because the preceding conclusion overwrites just about everything else.

```
jmp *%rax
```

6.3   Shrink $R_{\texttt{Fun}}$

The `shrink` pass performs a minor modification to ease the later passes. This pass introduces an explicit `main` function and changes the top `ProgramDefsExp` form to `ProgramDefs` as follows.

(`ProgramDefsExp` *info* (*def* ...) *exp*)
$\Rightarrow$ (`ProgramDefs` *info* (*def* ... *mainDef*))

where *mainDef* is

(`Def 'main '() 'Integer '()` *exp'*)

6.4   Reveal Functions and the $R_{\texttt{FunRef}}$ language

The syntax of $R_{\texttt{Fun}}$ is inconvenient for purposes of compilation in one respect: 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 than the use of a local variable; we need to use `leaq` to convert the function name (a label in x86) to an address in a register. Thus, it is a good idea to create a new pass that changes function references from just a symbol f to (`FunRef` f). This pass is named `reveal-functions` and the output language, $R_{\texttt{FunRef}}$, is defined in Figure 6.7. The concrete syntax for a function reference is (`fun-ref` f).

$$
\begin{array}{lll}
exp & ::= & \dots \mid (\texttt{FunRef } var) \\
def & ::= & (\texttt{Def } var\ ([var\!:\!type]\dots)\ type\ \text{'}()\ exp) \\
R_{\texttt{FunRef}} & ::= & (\texttt{ProgramDefs } \text{'}()\ (def\dots))
\end{array}
$$

Figure 6.7: The abstract syntax $R_{\texttt{FunRef}}$, an extension of $R_{\texttt{Fun}}$ (Figure 6.2).

Placing this pass after `uniquify` will make sure that there are no local variables and functions that share the same name. On the other hand, `reveal-functions` needs to come before the `explicate-control` pass because that pass helps us compile `FunRef` forms into assignment statements.

6.5   Limit Functions

Recall that we wish to limit the number of function parameters to six so that we do not need to use the stack for argument passing, which makes it easier to implement efficient tail calls. However, because the input language $R_{\texttt{Fun}}$ supports arbitrary numbers of function arguments, we have some work to do!

This pass transforms functions and function calls that involve more than six arguments to pass the first five arguments as usual, but it packs the rest of the arguments into a vector and passes it as the sixth argument.

Each function definition with too many parameters is transformed as follows.

$$(\texttt{Def } f\ ([x_1\!:\!T_1]\ \dots\ [x_n\!:\!T_n])\ T_r\ info\ body)$$
$$\Rightarrow$$
$$(\texttt{Def } f\ ([x_1\!:\!T_1]\ \dots\ [x_5\!:\!T_5]\ [\texttt{vec } :\ (\texttt{Vector } T_6\dots T_n)])\ T_r\ info\ body')$$

where the *body* is transformed into *body'* by replacing the occurrences of the later parameters with vector references.

$$(\texttt{Var } x_i)\ \Rightarrow\ (\texttt{Prim 'vector-ref } (\texttt{list vec } (\texttt{Int } (i-6))))$$

For function calls with too many arguments, the `limit-functions` pass transforms them in the following way.

$$(e_0\ e_1\ \dots\ e_n)\ \Rightarrow\ (e_0\ e_1\dots e_5\ (\texttt{vector } e_6\dots e_n))$$

6.6   Remove Complex Operands

The primary decisions to make for this pass is whether to classify `FunRef` and `Apply` as either atomic or complex expressions. Recall that a simple

$$
\begin{array}{rcl}
atm & ::= & (\texttt{Int}\ int)\ |\ (\texttt{Var}\ var)\ |\ (\texttt{Bool}\ bool)\ |\ (\texttt{Void}) \\
exp & ::= & atm\ |\ (\texttt{Prim}\ \texttt{read}\ ()) \\
    & | & (\texttt{Prim}\ \texttt{-}\ (atm))\ |\ (\texttt{Prim}\ \texttt{+}\ (atm\ atm)) \\
    & | & (\texttt{Let}\ var\ exp\ exp) \\
    & | & (\texttt{Prim}\ \texttt{'not}\ (atm)) \\
    & | & (\texttt{Prim}\ cmp\ (atm\ atm))\ |\ (\texttt{If}\ exp\ exp\ exp) \\
    & | & (\texttt{Collect}\ int)\ |\ (\texttt{Allocate}\ int\ type)\ |\ (\texttt{GlobalValue}\ var) \\
    & | & \mathbf{(\texttt{FunRef}\ \mathit{var})\ |\ (\texttt{Apply}\ \mathit{atm}\ \mathit{atm}\dots)} \\
def & ::= & (\texttt{Def}\ var\ ([var\!:\!type]\dots)\ type\ \texttt{'()}\ exp) \\
R_4^\dagger & ::= & (\texttt{ProgramDefs}\ \texttt{'()}\ def)
\end{array}
$$

Figure 6.8: $R_{\mathrm{Fun}}^{\mathrm{ANF}}$ is R_{Fun} in administrative normal form (ANF).

expression will eventually end up as just an immediate argument of an x86 instruction. Function application will be translated to a sequence of instructions, so `Apply` must be classified as complex expression. On the other hand, the arguments of `Apply` should be atomic expressions. Regarding `FunRef`, as discussed above, the function 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`. Figure 6.8 defines the output language $R_{\mathrm{Fun}}^{\mathrm{ANF}}$ of this pass.

6.7   Explicate Control and the C_{Fun} language

Figure 6.9 defines the abstract syntax for C_{Fun}, the output of `explicate-control`. (The concrete syntax is given in Figure 12.5 of the Appendix.) The auxiliary functions for assignment and tail contexts should be updated with cases for `Apply` and `FunRef` and the function for predicate context should be updated for `Apply` but not `FunRef`. (A `FunRef` can't be a Boolean.) In assignment and predicate contexts, `Apply` becomes `Call`, whereas in tail position `Apply` becomes `TailCall`. We recommend defining a new auxiliary function for processing function definitions. This code is similar to the case for `Program` in R_{Vec}. The top-level `explicate-control` function that handles the `ProgramDefs` form of R_{Fun} can then apply this new function to all the function definitions.

$$
\begin{array}{lll}
\textit{atm} & ::= & (\texttt{Int}\ \textit{int})\ |\ (\texttt{Var}\ \textit{var})\ |\ (\texttt{Bool}\ \textit{bool}) \\
\textit{cmp} & ::= & \texttt{eq?}\ |\ \texttt{<} \\
\textit{exp} & ::= & \textit{atm}\ |\ (\texttt{Prim}\ \texttt{read}\ ()) \\
 & & |\quad (\texttt{Prim}\ \texttt{-}\ (\textit{atm}))\ |\ (\texttt{Prim}\ \texttt{+}\ (\textit{atm}\ \textit{atm})) \\
 & & |\quad (\texttt{Prim}\ \texttt{not}\ (\textit{atm}))\ |\ (\texttt{Prim}\ \textit{cmp}\ (\textit{atm}\ \textit{atm})) \\
 & & |\quad (\texttt{Allocate}\ \textit{int}\ \textit{type}) \\
 & & |\quad (\texttt{Prim}\ \texttt{'vector-ref}\ (\textit{atm}\ (\texttt{Int}\ \textit{int}))) \\
 & & |\quad (\texttt{Prim}\ \texttt{'vector-set!}\ (\texttt{list}\ \textit{atm}\ (\texttt{Int}\ \textit{int})\ \textit{atm})) \\
 & & |\quad (\texttt{GlobalValue}\ \textit{var})\ |\ (\texttt{Void}) \\
 & & |\quad \mathbf{(FunRef}\ \textit{label})\ |\ (\texttt{Call}\ \textit{atm}\ (\textit{atm}\ldots)) \\
\textit{stmt} & ::= & (\texttt{Assign}\ (\texttt{Var}\ \textit{var})\ \textit{exp})\ |\ (\texttt{Collect}\ \textit{int}) \\
\textit{tail} & ::= & (\texttt{Return}\ \textit{exp})\ |\ (\texttt{Seq}\ \textit{stmt}\ \textit{tail})\ |\ (\texttt{Goto}\ \textit{label}) \\
 & & |\quad (\texttt{IfStmt}\ (\texttt{Prim}\ \textit{cmp}\ (\textit{atm}\ \textit{atm}))\ (\texttt{Goto}\ \textit{label})\ (\texttt{Goto}\ \textit{label})) \\
 & & |\quad \mathbf{(TailCall}\ \textit{atm}\ \textit{atm}\ldots) \\
\textit{def} & ::= & \mathbf{(Def}\ \textit{label}\ ([\textit{var}:\textit{type}]\ldots)\ \textit{type}\ \textit{info}\ ((\textit{label}\ .\ \textit{tail})\ldots)) \\
C_{\texttt{Fun}} & ::= & (\texttt{ProgramDefs}\ \textit{info}\ (\textit{def}\ldots))
\end{array}
$$

Figure 6.9: The abstract syntax of $C_{\texttt{Fun}}$, extending $C_{\texttt{Vec}}$ (Figure 5.13).

$$
\begin{array}{lll}
\textit{arg} & ::= & \$\textit{int}\ |\ \%\textit{reg}\ |\ \textit{int}(\%\textit{reg})\ |\ \%\textit{bytereg}\ |\ \textit{var}(\%\texttt{rip})\ |\ (\texttt{fun-ref}\ \textit{label}) \\
\textit{cc} & ::= & \texttt{e}\ |\ \texttt{l}\ |\ \texttt{le}\ |\ \texttt{g}\ |\ \texttt{ge} \\
\textit{instr} & ::= & \ldots\ |\ \texttt{callq}\ *\textit{arg}\ |\ \texttt{tailjmp}\ \textit{arg}\ |\ \texttt{leaq}\ \textit{arg},\ \%\textit{reg} \\
\textit{block} & ::= & \textit{instr}\ldots \\
\textit{def} & ::= & (\texttt{define}\ (\textit{label})\ ((\textit{label}\ .\ \textit{block})\ldots)) \\
\text{x86}_{\texttt{callq*}} & ::= & \textit{def}\ldots
\end{array}
$$

Figure 6.10: The concrete syntax of x86$_{\texttt{callq*}}$ (extends x86$_{\texttt{Global}}$ of Figure 5.14).

6.8   Select Instructions and the x86$_{\texttt{callq*}}$ Language

The output of select instructions is a program in the x86$_{\texttt{callq*}}$ language, whose syntax is defined in Figure 6.11.

An assignment of a function reference to a variable becomes a load-effective-address instruction as follows:

$\textit{lhs}$ = (fun-ref f); $\Rightarrow$ leaq (fun-ref f), $\textit{lhs}'$

Regarding function definitions, we need to remove the parameters and instead perform parameter passing using the conventions discussed in Section 6.2. That is, the arguments are passed in registers. We recommend

---

arg	::=	(Int *int*) | (Reg *reg*) | (Deref *reg int*) | (ByteReg *reg*)
	|	(Global *var*) | (**FunRef** *label*)
instr	::=	... | (**IndirectCallq** *arg int*) | (**TailJmp** *arg int*)
	|	(**Instr** 'leaq (*arg* (Reg *reg*)))
block	::=	(Block *info* (*instr*...))
def	::=	(Def *label* '() *type info* ((*label* . *block*)...))
x86$_{\texttt{callq}*}$	::=	(ProgramDefs *info* (*def*...))

Figure 6.11: The abstract syntax of x86$_{\texttt{callq}*}$ (extends x86$_{\texttt{Global}}$ of Figure 5.15).

turning the parameters into local variables and generating instructions at the beginning of the function to move from the argument passing registers to these local variables.

```
(Def f '([x₁ : T₁] [x₂ : T₂] ... ) Tᵣ info G)
⇒
(Def f '() 'Integer info′ G′)
```

The G' control-flow graph is the same as G 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 G shown on the left is changed to the code on the right.

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 implemented the move-biasing challenge (Section 3.7), 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 in Figure 6.13 of Section 6.12, 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 assignment of x_1 to **rsi**, which is good, because that would overwrite the argument that needs to move into x_2.

Next, consider the compilation of function calls. In the mirror image of handling the parameters of function definitions, the arguments need to be moved to the argument passing registers. The function call itself is performed with an indirect function call. The return value from the function is stored in **rax**, so it needs to be moved into the *lhs*.

```
lhs = (call fun arg₁ arg₂...);
⇒
movq arg₁, %rdi
movq arg₂, %rsi
.
.
.
callq *fun
movq %rax, lhs
```

The **IndirectCallq** AST node includes an integer for the arity of the function, i.e., 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.

For tail calls, the parameter passing is the same as non-tail calls: generate instructions to move the arguments into to the argument passing registers. After that we need to pop the frame from the procedure call stack. However, we do not yet know how big the frame is; that gets determined during register allocation. So instead of generating those instructions here, we invent a new instruction that means "pop the frame and then do an indirect jump", which we name **TailJmp**. The abstract syntax for this instruction includes an argument that specifies where to jump and an integer that represents the arity of the function being called.

Recall that in Section 2.6 we recommended using the label **start** for the initial block of a program, and in Section 2.7 we recommended 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, we will have a starting 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. (Alternatively, one could **gensym** labels for the start and conclusion and store them in the *info* field of the function definition.)

6.9 Register Allocation

6.9.1 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 call-live variables 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 `TailJmp` and `IndirectCallq` determines how many of the argument-passing registers should be considered as read by those instructions.

6.9.2 Build Interference Graph

With the addition of function definitions, we compute an interference graph for each function (not just one for the whole program).

Recall that in Section 5.8 we discussed the need to spill vector-typed variables that are live during a call to the `collect`. With the addition of functions to our language, we need to revisit this issue. Many functions perform allocation and therefore have calls to the collector inside of them. Thus, we should not only spill a vector-typed variable when it is live during a call to `collect`, but we should spill the variable if it is live during any function call. Thus, in the `build-interference` pass, we recommend adding interference edges between call-live vector-typed variables and the callee-saved registers (in addition to the usual addition of edges between call-live variables and the caller-saved registers).

6.9.3 Allocate Registers

The primary change to the `allocate-registers` pass is adding an auxiliary function for handling definitions (the *def* non-terminal in Figure 6.11) with one case for function definitions. The logic is the same as described in Chapter 3, except now register allocation is performed many times, once for each function definition, instead of just once for the whole program.

6.10 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 `TailJmp` is *rax*, our reserved register—this is to

make code generation more convenient, because we trample many registers before the tail call (as explained in the next section).

6.11   Print x86

For the `print-x86` pass, the cases for `FunRef` and `IndirectCallq` are straight-forward: output their concrete syntax.

   (FunRef *label*) ⇒ *label*(%rip)
   (IndirectCallq *arg int*) ⇒ callq **arg*'

The `TailJmp` node requires a bit work. A straightforward translation of `TailJmp` would be `jmp *`*arg*, but before the jump we need to pop the current frame. This sequence of instructions is the same as the code for the conclusion of a function, except the `retq` is replaced with `jmp *`*arg*.

Regarding function definitions, you will need to generate a prelude and conclusion for each one. This code is similar to the prelude and conclusion that you generated for the `main` function in Chapter 5. To review, the prelude of every function should carry out the following steps.

1. Start with `.global` and `.align` directives followed by the label for the function. (See Figure 6.13 for an example.)

2. Push `rbp` to the stack and set `rbp` to current stack pointer.

3. Push to the stack all of the callee-saved registers that were used for register allocation.

4. Move the stack pointer `rsp` down by the size of the stack frame for this function, which depends on the number of regular spills. (Aligned to 16 bytes.)

5. 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 vectors.

6. Initialize to zero all of the entries in the root-stack frame.

7. Jump to the start block.

The prelude of the `main` function has one additional task: call the `initialize` function to set up the garbage collector and move the value of the global `rootstack_begin` in `r15`. This should happen before step 5 above, which depends on `r15`.

The conclusion of every function should do the following.

1. Move the stack pointer back up by the size of the stack frame for this function.

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.

Exercise 30. Expand your compiler to handle $R_{\texttt{Fun}}$ as outlined in this chapter. Create 5 new programs that use functions, including examples that pass functions and return functions from other functions, recursive functions, functions that create vectors, and functions that make tail calls. Test your compiler on these new programs and all of your previously created test programs.

Figure 6.12 gives an overview of the passes for compiling $R_{\texttt{Fun}}$ to x86.

6.12 An Example Translation

Figure 6.13 shows an example translation of a simple function in $R_{\texttt{Fun}}$ to x86. The figure also includes the results of the `explicate-control` and `select-instructions` passes.

$R_{\texttt{Fun}}$ $\xrightarrow{\text{shrink}}$ $R_{\texttt{Fun}}$ $\xrightarrow{\text{uniquify}}$ $R_{\texttt{Fun}}$ $\xrightarrow{\text{reveal-functions}}$ $R_{\texttt{FunRef}}$

$R_{\texttt{FunRef}}^{\texttt{Alloc}}$ $\xleftarrow{\text{remove-complex.}}$ $R_{\texttt{FunRef}}^{\texttt{Alloc}}$ $\xleftarrow{\text{expose-alloc.}}$ $R_{\texttt{FunRef}}$ $\xleftarrow{\text{limit-functions}}$ $R_{\texttt{FunRef}}$

$R_{\texttt{FunRef}}^{\texttt{Alloc}}$ $\xrightarrow{\text{explicate-control}}$ $C_{\texttt{Fun}}$

$C_{\texttt{Fun}}$ $\xrightarrow{\text{select-instr.}}$ $\text{x86}_{\texttt{callq*}}^{\texttt{Var}}$

$\text{x86}_{\texttt{callq*}}^{\texttt{Var}}$ $\xrightarrow{\text{patch-instr.}}$ $\text{x86}_{\texttt{callq*}}$

$\text{x86}_{\texttt{callq*}}^{\texttt{Var}}$ $\xrightarrow{\text{uncover-live}}$ $\text{x86}_{\texttt{callq*}}^{\texttt{Var}}$ $\xrightarrow{\text{allocate-reg.}}$ $\text{x86}_{\texttt{callq*}}^{\texttt{Var}}$ $\xrightarrow{\text{print-x86}}$ $\text{x86}_{\texttt{callq*}}$

$\text{x86}_{\texttt{callq*}}^{\texttt{Var}}$ $\xrightarrow{\text{build-inter.}}$ $\text{x86}_{\texttt{callq*}}^{\texttt{Var}}$

Figure 6.12: Diagram of the passes for $R_{\texttt{Fun}}$, a language with functions.

```
(define (add [x : Integer] [y : Integer])       (define (add86) : Integer
   : Integer                                        add86start:
   (+ x y))                                             movq %rdi, x87
(add 40 2)                                             movq %rsi, y88
                                                      movq x87, %rax
⇓                                                    addq y88, %rax
                                                      jmp add11389conclusion
(define (add86 [x87 : Integer]                     )
              [y88 : Integer]) : Integer   ⇒   (define (main) : Integer
   add86start:                                       mainstart:
      return (+ x87 y88);                               leaq (fun-ref add86), tmp89
   )                                                    movq $40, %rdi
(define (main) : Integer ()                            movq $2, %rsi
   mainstart:                                           tail-jmp tmp89
      tmp89 = (fun-ref add86);                       )
      (tail-call tmp89 40 2)
   )                                                ⇓
```

```
                                                  .globl main
                                                  .align 16
                                          main:
                                                  pushq  %rbp
                .globl add86                      movq   %rsp, %rbp
                .align 16                          movq   $16384, %rdi
add86:                                             movq   $16384, %rsi
        pushq  %rbp                                callq  initialize
        movq   %rsp, %rbp                          movq   rootstack_begin(%rip), %r15
        jmp    add86start                          jmp    mainstart
add86start:                               mainstart:
        movq   %rdi, %rax                          leaq   add86(%rip), %rcx
        addq   %rsi, %rax                          movq   $40, %rdi
        jmp add86conclusion                        movq   $2, %rsi
add86conclusion:                                   movq   %rcx, %rax
        popq   %rbp                                popq   %rbp
        retq                                       jmp    *%rax
                                          mainconclusion:
                                                  popq   %rbp
                                                  retq
```

Figure 6.13: Example compilation of a simple function to x86.

7

Lexically Scoped Functions

This chapter studies lexically scoped functions as they appear in functional languages such as Racket. By lexical scoping we mean that a function's body may refer to variables whose binding site is outside of the function, in an enclosing scope. Consider the example in Figure 7.1 written in R_λ, which extends $R_{\texttt{Fun}}$ with anonymous functions using the `lambda` form. The body of the `lambda`, refers to three variables: x, y, and z. The binding sites for x and y are outside of the `lambda`. Variable y is bound by the enclosing `let` and x is a parameter of function f. The `lambda` is returned from the function f. The main expression of the program includes two calls to f with different arguments for x, first 5 then 3. The functions returned from f are bound to variables g and h. Even though these two functions were created by the same `lambda`, they are really different functions because they use different values for x. Applying g to 11 produces 20 whereas applying h to 15 produces 22. The result of this program is 42.

   The approach that we take for implementing lexically scoped functions

```
(define (f [x : Integer]) : (Integer -> Integer)
   (let ([y 4])
      (lambda: ([z : Integer]) : Integer
         (+ x (+ y z)))))

(let ([g (f 5)])
  (let ([h (f 3)])
    (+ (g 11) (h 15))))
```

Figure 7.1: Example of a lexically scoped function.

137

is to compile them into top-level function definitions, translating from R_λ into $R_{\texttt{Fun}}$. However, the compiler will need to provide special treatment for variable occurrences such as x and y in the body of the lambda of Figure 7.1. After all, an $R_{\texttt{Fun}}$ function may not refer to variables defined outside of it. To identify such variable occurrences, we review the standard notion of free variable.

Definition 31. *A variable is* free in expression *e if the variable occurs inside e but does not have an enclosing binding in e.*

For example, in the expression (+ x (+ y z)) the variables x, y, and z are all free. On the other hand, only x and y are free in the following expression because z is bound by the lambda.

```
(lambda: ([z : Integer]) : Integer
  (+ x (+ y z)))
```

So the free variables of a lambda are the ones that will need special treatment. We need to arrange for some way to transport, at runtime, the values of those variables from the point where the lambda was created to the point where the lambda is applied. An efficient solution to the problem, due to Cardelli [18], is to bundle into a vector the values of the free variables together with the function pointer for the lambda's code, an arrangement called a *flat closure* (which we shorten to just "closure"). Fortunately, we have all the ingredients to make closures, Chapter 5 gave us vectors and Chapter 6 gave us function pointers. The function pointer resides at index 0 and the values for the free variables will fill in the rest of the vector.

Let us revisit the example in Figure 7.1 to see how closures work. It's a three-step dance. The program first calls function f, which creates a closure for the lambda. The closure is a vector whose first element is a pointer to the top-level function that we will generate for the lambda, the second element is the value of x, which is 5, and the third element is 4, the value of y. The closure does not contain an element for z because z is not a free variable of the lambda. Creating the closure is step 1 of the dance. The closure is returned from f and bound to g, as shown in Figure 7.2. The second call to f creates another closure, this time with 3 in the second slot (for x). This closure is also returned from f but bound to h, which is also shown in Figure 7.2.

Continuing with the example, consider the application of g to 11 in Figure 7.1. To apply a closure, we obtain the function pointer in the first element of the closure and call it, passing in the closure itself and then the regular arguments, in this case 11. This technique for applying a closure
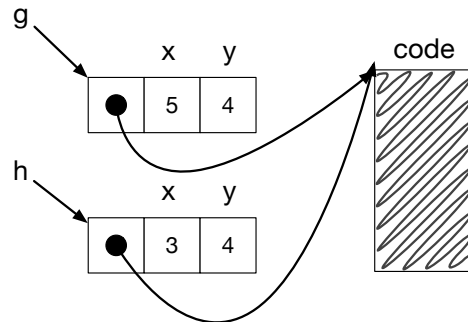
Figure 7.2: Example closure representation for the `lambda`'s in Figure 7.1.

is step 2 of the dance. But doesn't this `lambda` only take 1 argument, for parameter `z`? The third and final step of the dance is generating a top-level function for a `lambda`. We add an additional parameter for the closure and we insert a `let` at the beginning of the function for each free variable, to bind those variables to the appropriate elements from the closure parameter. This three-step dance is known as *closure conversion.* We discuss the details of closure conversion in Section 7.3 and the code generated from the example in Section 7.4. But first we define the syntax and semantics of R_λ in Section 7.1.

7.1 The R_λ Language

The concrete and abstract syntax for R_λ, a language with anonymous functions and lexical scoping, is defined in Figures 7.3 and 7.4. It adds the `lambda` form to the grammar for $R_{\mathtt{Fun}}$, which already has syntax for function application.

Figure 7.5 shows the definitional interpreter for R_λ. The case for `lambda` saves the current environment inside the returned `lambda`. Then the case for `Apply` uses the environment from the `lambda`, the `lam-env`, when interpreting the body of the `lambda`. The `lam-env` environment is extended with the mapping of parameters to argument values.

Figure 7.6 shows how to type check the new `lambda` form. The body of the `lambda` is checked in an environment that includes the current environment (because it is lexically scoped) and also includes the `lambda`'s parameters. We require the body's type to match the declared return type.

```
type   ::=   Integer | Boolean | (Vector type ...) | Void | (type ... -> type)
exp    ::=   int | (read) | (- exp) | (+ exp exp) | (- exp exp)
        |    var | (let ([var exp]) exp)
        |    #t | #f | (and exp exp) | (or exp exp) | (not exp)
        |    (eq? exp exp) | (if exp exp exp)
        |    (vector exp ...) | (vector-ref exp int)
        |    (vector-set! exp int exp) | (void) | (exp exp ...)
        |    (procedure-arity exp)
        |    (lambda: ([var : type] ...) : type exp)
def    ::=   (define (var [var : type] ...) : type exp)
R_λ    ::=   def ... exp
```

Figure 7.3: The concrete syntax of R_λ, extending R_{Fun} (Figure 6.1) with `lambda`.

```
op    ::=   ... | procedure-arity
exp   ::=   (Int int) (Var var) | (Let var exp exp)
       |    (Prim op (exp ...))
       |    (Bool bool) | (If exp exp exp)
       |    (Void) | (HasType exp type) | (Apply exp exp ...)
       |    (Lambda ([var : type] ...) type exp)
def   ::=   (Def var ([var : type] ...) type '() exp)
R_λ   ::=   (ProgramDefsExp '() (def ...) exp)
```

Figure 7.4: The abstract syntax of R_λ, extending R_{Fun} (Figure 6.2).

```
(define interp-Rlambda-class
  (class interp-Rfun-class
    (super-new)

    (define/override (interp-op op)
      (match op
        ['procedure-arity
         (lambda (v)
           (match v
             [`(function (,xs ...) ,body ,lam-env) (length xs)]
             [else (error 'interp-op "expected a function, not ~a" v)]))]
        [else (super interp-op op)]))

    (define/override ((interp-exp env) e)
      (define recur (interp-exp env))
      (match e
        [(Lambda (list `[,xs : ,Ts] ...) rT body)
         `(function ,xs ,body ,env)]
        [else ((super interp-exp env) e)]))
    ))

(define (interp-Rlambda p)
  (send (new interp-Rlambda-class) interp-program p))
```

Figure 7.5: Interpreter for R_λ.

```
(define (type-check-Rlambda env)
  (lambda (e)
    (match e
      [(Lambda (and params `([,xs : ,Ts] ...)) rT body)
       (define-values (new-body bodyT)
         ((type-check-exp (append (map cons xs Ts) env)) body))
       (define ty `(,@Ts -> ,rT))
       (cond
         [(equal? rT bodyT)
          (values (HasType (Lambda params rT new-body) ty) ty)]
         [else
          (error "mismatch in return type" bodyT rT)])]
      ...
      )))
```

Figure 7.6: Type checking the `lambda`'s in R_λ.

$$
\begin{array}{lll}
exp & ::= & \ldots \mid (\texttt{FunRefArity}\ var\ int) \\
def & ::= & (\texttt{Def}\ var\ ([var\!:\!type]\ldots)\ type\ \texttt{'()}\ exp) \\
F_2 & ::= & (\texttt{ProgramDefs}\ \texttt{'()}\ (def\ldots))
\end{array}
$$

Figure 7.7: The abstract syntax F_2, an extension of R_λ (Figure 7.4).

7.2 Reveal Functions and the F_2 language

To support the `procedure-arity` operator we need to communicate the arity of a function to the point of closure creation. We can accomplish this by replacing the (`FunRef` *var*) struct with one that has a second field for the arity: (`FunRefArity` *var int*). The output of this pass is the language F_2, whose syntax is defined in Figure 7.7.

7.3 Closure Conversion

The compiling of lexically-scoped functions into top-level function definitions is accomplished in the pass `convert-to-closures` that comes after `reveal-functions` and before `limit-functions`.

As usual, we implement the pass as a recursive function over the AST. All of the action is in the cases for `Lambda` and `Apply`. We transform a `Lambda` expression into an expression that creates a closure, that is, a vector whose first element is a function pointer and the rest of the elements are the free variables of the `Lambda`. We use the struct `Closure` here instead of using `vector` so that we can distinguish closures from vectors in Section 7.8 and to record the arity. In the generated code below, the *name* is a unique symbol generated to identify the function and the *arity* is the number of parameters (the length of *ps*).

```
(Lambda ps rt body)
⇒
(Closure arity (cons (FunRef name) fvs))
```

In addition to transforming each `Lambda` into a `Closure`, we create a top-level function definition for each `Lambda`, as shown below.

```
(Def name ([clos : (Vector _ fvts ...)] ps' ...) rt'
   (Let fvs₁ (Prim 'vector-ref (list (Var clos) (Int 1)))
     ...
     (Let fvsₙ (Prim 'vector-ref (list (Var clos) (Int n)))
       body')...))
```

The `clos` parameter refers to the closure. Translate the type annotations in *ps* and the return type *rt*, as discussed in the next paragraph, to obtain *ps'* and *rt'*. The types *fvts* are the types of the free variables in the lambda and the underscore `_` is a dummy type that we use because it is rather difficult to give a type to the function in the closure's type.[1] The dummy type is considered to be equal to any other type during type checking. The sequence of `Let` forms bind the free variables to their values obtained from the closure.

Closure conversion turns functions into vectors, so the type annotations in the program must also be translated. We recommend defining a auxiliary recursive function for this purpose. Function types should be translated as follows.

$(T_1, \ldots, T_n$ `->` $T_r)$
$\Rightarrow$
(`Vector` ((`Vector` `_`) $T'_1, \ldots, T'_n$ `->` T'_r))

The above type says that the first thing in the vector is a function pointer. The first parameter of the function pointer is a vector (a closure) and the rest of the parameters are the ones from the original function, with types $T'_1, \ldots, T'_n$. The `Vector` type for the closure omits the types of the free variables because 1) those types are not available in this context and 2) we do not need them in the code that is generated for function application.

We transform function application into code that retrieves the function pointer from the closure and then calls the function, passing in the closure as the first argument. We bind e' to a temporary variable to avoid code duplication.

(`Apply` e es)
$\Rightarrow$
(`Let` *tmp* e'
  (`Apply` (`Prim` 'vector-ref (`list` (`Var` *tmp*) (`Int` 0))) (`cons` *tmp* es')))

There is also the question of what to do with references top-level function definitions. To maintain a uniform translation of function application, we turn function references into closures.

(`FunRefArity` f n) $\Rightarrow$ (`Closure` n (`FunRef` f) '())

The top-level function definitions need to be updated as well to take an extra closure parameter.

---

[1] To give an accurate type to a closure, we would need to add existential types to the type checker [77].

```
(define (f6 [x7 : Integer]) : (Integer -> Integer)
  (let ([y8 4])
    (lambda: ([z9 : Integer]) : Integer
      (+ x7 (+ y8 z9)))))

(define (main) : Integer
  (let ([g0 ((fun-ref-arity f6 1) 5)])
    (let ([h1 ((fun-ref-arity f6 1) 3)])
      (+ (g0 11) (h1 15)))))
```

$\Rightarrow$

```
(define (f6 [fvs4 : _] [x7 : Integer]) : (Vector ((Vector _) Integer -> Integer))
  (let ([y8 4])
    (closure 1 (list (fun-ref lambda2) x7 y8))))

(define (lambda2 [fvs3 : (Vector _ Integer Integer)] [z9 : Integer]) : Integer
  (let ([x7 (vector-ref fvs3 1)])
    (let ([y8 (vector-ref fvs3 2)])
      (+ x7 (+ y8 z9)))))

(define (main) : Integer
  (let ([g0 (let ([clos5 (closure 1 (list (fun-ref f6)))])
              ((vector-ref clos5 0) clos5 5))])
    (let ([h1 (let ([clos6 (closure 1 (list (fun-ref f6)))])
                ((vector-ref clos6 0) clos6 3))])
      (+ ((vector-ref g0 0) g0 11) ((vector-ref h1 0) h1 15)))))
```

Figure 7.8: Example of closure conversion.

7.4   An Example Translation

Figure 7.8 shows the result of `reveal-functions` and `convert-to-closures` for the example program demonstrating lexical scoping that we discussed at the beginning of this chapter.

Exercise 32. Expand your compiler to handle R_λ as outlined in this chapter. Create 5 new programs that use `lambda` functions and make use of lexical scoping. Test your compiler on these new programs and all of your previously created test programs.

7.5   Expose Allocation

Compile the (Closure *arity* (*exp*...)) form into code that allocates and initializes a vector, similar to the translation of the `vector` operator in

$$
\begin{array}{lll}
exp & ::= & \ldots \mid (\texttt{AllocateClosure}\ int\ type\ int) \\
stmt & ::= & (\texttt{Assign}\ (\texttt{Var}\ var)\ exp) \mid (\texttt{Collect}\ int) \\
tail & ::= & (\texttt{Return}\ exp) \mid (\texttt{Seq}\ stmt\ tail) \mid (\texttt{Goto}\ label) \\
& \mid & (\texttt{IfStmt}\ (\texttt{Prim}\ cmp\ (atm\ atm))\ (\texttt{Goto}\ label)\ (\texttt{Goto}\ label)) \\
& \mid & (\texttt{TailCall}\ atm\ atm\ldots) \\
def & ::= & (\texttt{Def}\ label\ ([var\!:\!type]\ldots)\ type\ info\ ((label\ .\ tail)\ldots)) \\
C_{\texttt{Clos}} & ::= & (\texttt{ProgramDefs}\ info\ (def\ldots))
\end{array}
$$

Figure 7.9: The abstract syntax of $C_{\texttt{Clos}}$, extending $C_{\texttt{Fun}}$ (Figure 6.9).

Section 5.4. The only difference is replacing the use of (`Allocate` *len type*) with (`AllocateClosure` *len type arity*).

7.6 Explicate Control and C_{Clos}

The output language of `explicate-control` is $C_{\texttt{Clos}}$ whose abstract syntax is defined in Figure 7.9. The only difference with respect to $C_{\texttt{Fun}}$ is the addition of the `AllocateClosure` form to the grammar for *exp*. The handling of `AllocateClosure` in the `explicate-control` pass is similar to the handling of other expressions such as primitive operators.

7.7 Select Instructions

Compile (`AllocateClosure` *len type arity*) in almost the same way as the (`Allocate` *len type*) form (Section 5.7). The only difference is that you should place the *arity* in the tag that is stored at position 0 of the vector. Recall that in Section 5.7 we used the first 56 bits of the 64-bit tag, but that the rest were unused. So the arity goes into the tag in bit positions 57 through 63.

Compile the `procedure-arity` operator into a sequence of instructions that access the tag from position 0 of the vector and shift it by 57 bits to the right.

Figure 7.10 provides an overview of all the passes needed for the compilation of R_{λ}.

$$R_{\texttt{Fun}} \xrightarrow{\texttt{shrink}} R_{\texttt{Fun}} \xrightarrow{\texttt{uniquify}} R_{\texttt{Fun}} \xrightarrow{\texttt{reveal-functions}} R_{\texttt{FunRef}}$$

$$F_1 \xleftarrow{\texttt{remove-complex.}} F_1 \xleftarrow{\texttt{expose-alloc.}} F_1 \xleftarrow{\texttt{limit-fun.}} R_{\texttt{FunRef}} \xleftarrow{\texttt{convert-to-clos.}} R_{\texttt{FunRef}}$$

$$F_1 \xrightarrow{\texttt{explicate-control}} C_{\texttt{Fun}}$$

$$C_{\texttt{Fun}} \xrightarrow{\texttt{select-instr.}} \text{x86}^{\texttt{Var}}_{\texttt{callq*}}$$

$$\text{x86}^{\texttt{Var}}_{\texttt{callq*}} \xrightarrow{\texttt{patch-instr.}} \text{x86}_{\texttt{callq*}}$$

$$\text{x86}^{\texttt{Var}}_{\texttt{callq*}} \xleftarrow{\texttt{uncover-live}}$$

$$\text{allocate-reg.} \qquad \texttt{print-x86}$$

$$\text{x86}^{\texttt{Var}}_{\texttt{callq*}} \xrightarrow{\texttt{build-inter.}} \text{x86}^{\texttt{Var}}_{\texttt{callq*}} \qquad \text{x86}_{\texttt{callq*}}$$
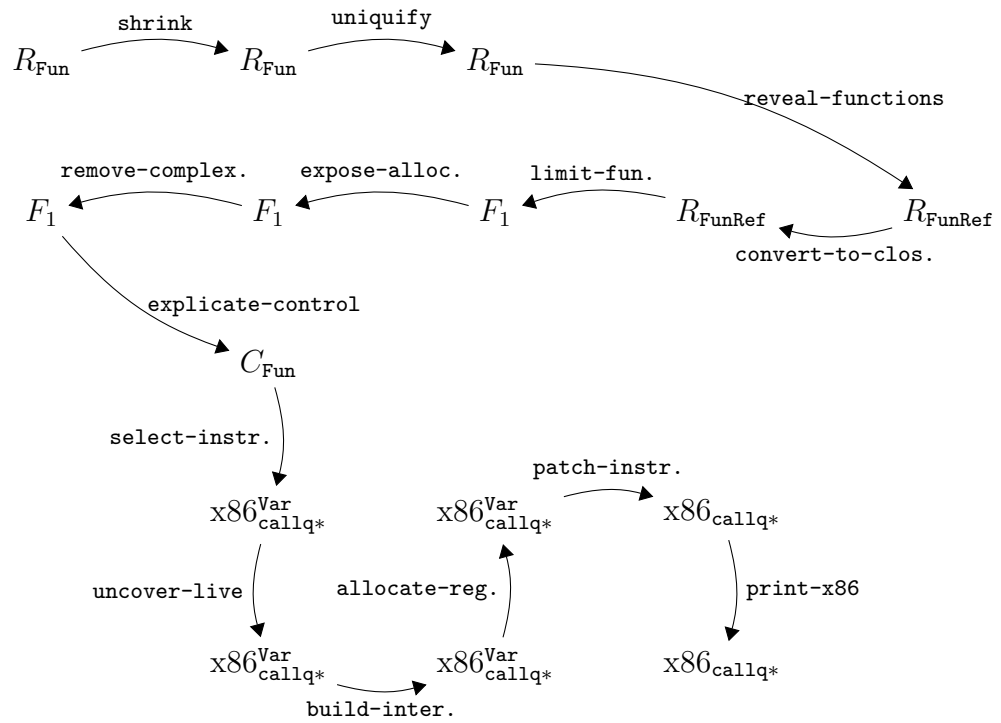
Figure 7.10: Diagram of the passes for R_λ, a language with lexically-scoped functions.

7.8 Challenge: Optimize Closures

In this chapter we compiled lexically-scoped functions into a relatively efficient representation: flat closures. However, even this representation comes with some overhead. For example, consider the following program with a function `tail-sum` that does not have any free variables and where all the uses of `tail-sum` are in applications where we know that only `tail-sum` is being applied (and not any other functions).

```
(define (tail-sum [n : Integer] [r : Integer]) : Integer
  (if (eq? n 0)
      r
      (tail-sum (- n 1) (+ n r))))

(+ (tail-sum 5 0) 27)
```

As described in this chapter, we uniformly apply closure conversion to all functions, obtaining the following output for this program.

```
(define (tail_sum1 [fvs5 : _] [n2 : Integer] [r3 : Integer]) : Integer
  (if (eq? n2 0)
      r3
      (let ([clos4 (closure (list (fun-ref tail_sum1)))])
        ((vector-ref clos4 0) clos4 (+ n2 -1) (+ n2 r3)))))

(define (main) : Integer
  (+ (let ([clos6 (closure (list (fun-ref tail_sum1)))])
       ((vector-ref clos6 0) clos6 5 0)) 27))
```

In the previous Chapter, there would be no allocation in the program and the calls to `tail-sum` would be direct calls. In contrast, the above program allocates memory for each `closure` and the calls to `tail-sum` are indirect. These two differences incur considerable overhead in a program such as this one, where the allocations and indirect calls occur inside a tight loop.

One might think that this problem is trivial to solve: can't we just recognize calls of the form $((\text{fun-ref } f) \; e_1 \ldots e_n)$ and compile them to direct calls $((\text{fun-ref } f) \; e'_1 \ldots e'_n)$ instead of treating it like a call to a closure? We would also drop the `fvs5` parameter of `tail_sum1`. However, this problem is not so trivial because a global function may "escape" and become involved in applications that also involve closures. Consider the following example in which the application (`f 41`) needs to be compiled

into a closure application, because the `lambda` may get bound to `f`, but the `add1` function might also get bound to `f`.

```
(define (add1 [x : Integer]) : Integer
  (+ x 1))

(let ([y (read)])
  (let ([f (if (eq? (read) 0)
               add1
               (lambda: ([x : Integer]) : Integer (- x y)))])
    (f 41)))
```

If a global function name is used in any way other than as the operator in a direct call, then we say that the function *escapes*. If a global function does not escape, then we do not need to perform closure conversion on the function.

Exercise 33. Implement an auxiliary function for detecting which global functions escape. Using that function, implement an improved version of closure conversion that does not apply closure conversion to global functions that do not escape but instead compiles them as regular functions. Create several new test cases that check whether you properly detect whether global functions escape or not.

So far we have reduced the overhead of calling global functions, but it would also be nice to reduce the overhead of calling a `lambda` when we can determine at compile time which `lambda` will be called. We refer to such calls as *known calls*. Consider the following example in which a `lambda` is bound to `f` and then applied.

```
(let ([y (read)])
  (let ([f (lambda: ([x : Integer]) : Integer
             (+ x y))])
    (f 21)))
```

Closure conversion compiles (`f 21`) into an indirect call:

```
(define (lambda5 [fvs6 : (Vector _ Integer)] [x3 : Integer]) : Integer
  (let ([y2 (vector-ref fvs6 1)])
    (+ x3 y2)))

(define (main) : Integer
  (let ([y2 (read)])
    (let ([f4 (Closure 1 (list (fun-ref lambda5) y2))])
      ((vector-ref f4 0) f4 21))))
```

but we can instead compile the application (`f 21`) into a direct call to
`lambda5`:

```
(define (main) : Integer
  (let ([y2 (read)])
    (let ([f4 (Closure 1 (list (fun-ref lambda5) y2))])
      ((fun-ref lambda5) f4 21))))
```

The problem of determining which lambda will be called from a partic-
ular application is quite challenging in general and the topic of considerable
research [87, 45]. For the following exercise we recommend that you compile
an application to a direct call when the operator is a variable and the vari-
able is `let`-bound to a closure. This can be accomplished by maintaining an
environment mapping `let`-bound variables to function names. Extend the
environment whenever you encounter a closure on the right-hand side of a
`let`, mapping the `let`-bound variable to the name of the global function for
the closure. This pass should come after closure conversion.

Exercise 34. Implement a compiler pass, named `optimize-known-calls`,
that compiles known calls into direct calls. Verify that your compiler is
successful in this regard on several example programs.

These exercises only scratches the surface of optimizing of closures. A
good next step for the interested reader is to look at the work of Keep et al.
[59].

8

Dynamic Typing

In this chapter we discuss the compilation of R_{Dyn}, a dynamically typed language that is a subset of Racket. This is in contrast to the previous chapters, which have studied the compilation of Typed Racket. In dynamically typed languages such as R_{Dyn}, a given expression may produce a value of a different type each time it is executed. Consider the following example with a conditional `if` expression that may return a Boolean or an integer depending on the input to the program.

```
(not (if (eq? (read) 1) #f 0))
```

Languages that allow expressions to produce different kinds of values are called *polymorphic*, a word composed of the Greek roots "poly", meaning "many", and "morph", meaning "shape". There are several kinds of polymorphism in programming languages, such as subtype polymorphism and parametric polymorphism [20]. The kind of polymorphism we study in this chapter does not have a special name but it is the kind that arises in dynamically typed languages.

Another characteristic of dynamically typed languages is that primitive operations, such as `not`, are often defined to operate on many different types of values. In fact, in Racket, the `not` operator produces a result for any kind of value: given `#f` it returns `#t` and given anything else it returns `#f`. Furthermore, even when primitive operations restrict their inputs to values of a certain type, this restriction is enforced at runtime instead of during compilation. For example, the following vector reference results in a run-time contract violation because the index must be in integer, not a Boolean such as `#t`.

```
(vector-ref (vector 42) #t)
```

$$
\begin{array}{rcl}
cmp & ::= & \texttt{eq?} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
exp & ::= & int \mid (\texttt{read}) \mid (\texttt{-}\ exp) \mid (\texttt{+}\ exp\ exp) \mid (\texttt{-}\ exp\ exp) \\
    & \mid & var \mid (\texttt{let}\ ([var\ exp])\ exp) \\
    & \mid & \texttt{\#t} \mid \texttt{\#f} \mid (\texttt{and}\ exp\ exp) \mid (\texttt{or}\ exp\ exp) \mid (\texttt{not}\ exp) \\
    & \mid & (cmp\ exp\ exp) \mid (\texttt{if}\ exp\ exp\ exp) \\
    & \mid & (\texttt{vector}\ exp\ldots) \mid (\texttt{vector-ref}\ exp\ exp) \\
    & \mid & (\texttt{vector-set!}\ exp\ exp\ exp) \mid (\texttt{void}) \\
    & \mid & (exp\ exp\ldots) \mid (\texttt{lambda}\ (var\ldots)\ exp) \\
    & \mid & (\texttt{boolean?}\ exp) \mid (\texttt{integer?}\ exp) \\
    & \mid & (\texttt{vector?}\ exp) \mid (\texttt{procedure?}\ exp) \mid (\texttt{void?}\ exp) \\
def & ::= & (\texttt{define}\ (var\ var\ldots)\ exp) \\
R_{\texttt{Dyn}} & ::= & def\ldots\ exp
\end{array}
$$

Figure 8.1: Syntax of $R_{\texttt{Dyn}}$, an untyped language (a subset of Racket).

$$
\begin{array}{rcl}
exp & ::= & (\texttt{Int}\ int) \mid (\texttt{Var}\ var) \mid (\texttt{Let}\ var\ exp\ exp) \\
    & \mid & (\texttt{Prim}\ op\ (exp\ldots)) \\
    & \mid & (\texttt{Bool}\ bool) \mid (\texttt{If}\ exp\ exp\ exp) \\
    & \mid & (\texttt{Void}) \mid (\texttt{Apply}\ exp\ exp\ldots) \\
    & \mid & (\texttt{Lambda}\ (var\ldots)\ \texttt{'Any}\ exp) \\
def & ::= & (\texttt{Def}\ var\ (var\ldots)\ \texttt{'Any}\ \texttt{'()}\ exp) \\
R_{\texttt{Dyn}} & ::= & (\texttt{ProgramDefsExp}\ \texttt{'()}\ (def\ldots)\ exp)
\end{array}
$$

Figure 8.2: The abstract syntax of $R_{\texttt{Dyn}}$.

The concrete and abstract syntax of R_{Dyn}, our subset of Racket, is defined in Figures 8.1 and 8.2. There is no type checker for R_{Dyn} because it is not a statically typed language (it's dynamically typed!).

The definitional interpreter for R_{Dyn} is presented in Figure 8.3 and its auxiliary functions are defined in Figure 8.4. Consider the match case for `(Int n)`. Instead of simply returning the integer `n` (as in the interpreter for R_{Var} in Figure 2.3), the interpreter for R_{Dyn} creates a *tagged value* that combines an underlying value with a tag that identifies what kind of value it is. We define the following struct to represented tagged values.

```
(struct Tagged (value tag) #:transparent)
```

The tags are `Integer`, `Boolean`, `Void`, `Vector`, and `Procedure`. Tags are closely related to types but don't always capture all the information that a type does. For example, a vector of type `(Vector Any Any)` is tagged with `Vector` and a procedure of type `(Any Any -> Any)` is tagged with `Procedure`.

Next consider the match case for `vector-ref`. The `check-tag` auxiliary function (Figure 8.4) is used to ensure that the first argument is a vector and the second is an integer. If they are not, a `trapped-error` is raised. Recall from Section 1.5 that when a definition interpreter raises a `trapped-error` error, the compiled code must also signal an error by exiting with return code `255`. A `trapped-error` is also raised if the index is not less than length of the vector.

```
(define ((interp-Rdyn-exp env) ast)
  (define recur (interp-Rdyn-exp env))
  (match ast
    [(Var x) (lookup x env)]
    [(Int n) (Tagged n 'Integer)]
    [(Bool b) (Tagged b 'Boolean)]
    [(Lambda xs rt body)
     (Tagged `(function ,xs ,body ,env) 'Procedure)]
    [(Prim 'vector es)
     (Tagged (apply vector (for/list ([e es]) (recur e))) 'Vector)]
    [(Prim 'vector-ref (list e1 e2))
     (define vec (recur e1)) (define i (recur e2))
     (check-tag vec 'Vector ast) (check-tag i 'Integer ast)
     (unless (< (Tagged-value i) (vector-length (Tagged-value vec)))
       (error 'trapped-error "index ~a too big\nin ~v" (Tagged-value i) ast))
     (vector-ref (Tagged-value vec) (Tagged-value i))]
    [(Prim 'vector-set! (list e1 e2 e3))
     (define vec (recur e1)) (define i (recur e2)) (define arg (recur e3))
     (check-tag vec 'Vector ast) (check-tag i 'Integer ast)
     (unless (< (Tagged-value i) (vector-length (Tagged-value vec)))
       (error 'trapped-error "index ~a too big\nin ~v" (Tagged-value i) ast))
     (vector-set! (Tagged-value vec) (Tagged-value i) arg)
     (Tagged (void) 'Void)]
    [(Let x e body) ((interp-Rdyn-exp (cons (cons x (recur e)) env)) body)]
    [(Prim 'and (list e1 e2)) (recur (If e1 e2 (Bool #f)))]
    [(Prim 'or (list e1 e2))
     (define v1 (recur e1))
     (match (Tagged-value v1) [#f (recur e2)] [else v1])]
    [(Prim 'eq? (list l r)) (Tagged (equal? (recur l) (recur r)) 'Boolean)]
    [(Prim op (list e1))
     #:when (set-member? type-predicates op)
     (tag-value ((interp-op op) (Tagged-value (recur e1))))]
    [(Prim op es)
     (define args (map recur es))
     (define tags (for/list ([arg args]) (Tagged-tag arg)))
     (unless (for/or ([expected-tags (op-tags op)])
               (equal? expected-tags tags))
       (error 'trapped-error "illegal argument tags ~a\nin ~v" tags ast))
     (tag-value
      (apply (interp-op op) (for/list ([a args]) (Tagged-value a))))]
    [(If q t f)
     (match (Tagged-value (recur q)) [#f (recur f)] [else (recur t)])]
    [(Apply f es)
     (define new-f (recur f)) (define args (map recur es))
     (check-tag new-f 'Procedure ast) (define f-val (Tagged-value new-f))
     (match f-val
       [`(function ,xs ,body ,lam-env)
        (unless (eq? (length xs) (length args))
          (error 'trapped-error "~a != ~a\nin ~v" (length args) (length xs) ast))
        (define new-env (append (map cons xs args) lam-env))
        ((interp-Rdyn-exp new-env) body)]
       [else (error "interp-Rdyn-exp, expected function, not" f-val)])]))
```

Figure 8.3: Interpreter for the R_{Dyn} language.

```
(define (interp-op op)
  (match op
    ['+ fx+]
    ['- fx-]
    ['read read-fixnum]
    ['not (lambda (v) (match v [#t #f] [#f #t]))]
    ['< (lambda (v1 v2)
          (cond [(and (fixnum? v1) (fixnum? v2)) (< v1 v2)]))]
    ['<= (lambda (v1 v2)
           (cond [(and (fixnum? v1) (fixnum? v2)) (<= v1 v2)]))]
    ['> (lambda (v1 v2)
          (cond [(and (fixnum? v1) (fixnum? v2)) (> v1 v2)]))]
    ['>= (lambda (v1 v2)
           (cond [(and (fixnum? v1) (fixnum? v2)) (>= v1 v2)]))]
    ['boolean? boolean?]
    ['integer? fixnum?]
    ['void? void?]
    ['vector? vector?]
    ['vector-length vector-length]
    ['procedure? (match-lambda
                   [`(functions ,xs ,body ,env) #t] [else #f])]
    [else (error 'interp-op "unknown operator" op)]))

(define (op-tags op)
  (match op
    ['+ '((Integer Integer))]
    ['- '((Integer Integer) (Integer))]
    ['read '(())]
    ['not '((Boolean))]
    ['< '((Integer Integer))]
    ['<= '((Integer Integer))]
    ['> '((Integer Integer))]
    ['>= '((Integer Integer))]
    ['vector-length '((Vector))]))

(define type-predicates
  (set 'boolean? 'integer? 'vector? 'procedure? 'void?))

(define (tag-value v)
  (cond [(boolean? v) (Tagged v 'Boolean)]
        [(fixnum? v) (Tagged v 'Integer)]
        [(procedure? v) (Tagged v 'Procedure)]
        [(vector? v) (Tagged v 'Vector)]
        [(void? v) (Tagged v 'Void)]
        [else (error 'tag-value "unidentified value ~a" v)]))

(define (check-tag val expected ast)
  (define tag (Tagged-tag val))
  (unless (eq? tag expected)
    (error 'trapped-error "expected ~a, not ~a\nin ~v" expected tag ast)))
```

Figure 8.4: Auxiliary functions for the R_{Dyn} interpreter.

8.1   Representation of Tagged Values

The interpreter for R_{Dyn} introduced a new kind of value, a tagged value. To compile R_{Dyn} to x86 we must decide how to represent tagged values at the bit level. Because almost every operation in R_{Dyn} involves manipulating tagged values, the representation must be efficient. Recall that all of our values are 64 bits. We shall steal the 3 right-most bits to encode the tag. We use 001 to identify integers, 100 for Booleans, 010 for vectors, 011 for procedures, and 101 for the void value. We define the following auxiliary function for mapping types to tag codes.

$$tagof(\texttt{Integer}) = 001$$
$$tagof(\texttt{Boolean}) = 100$$
$$tagof((\texttt{Vector}\ldots)) = 010$$
$$tagof((\ldots\texttt{->}\ldots)) = 011$$
$$tagof(\texttt{Void}) = 101$$

This stealing of 3 bits comes at some price: our integers are reduced to ranging from -2^{60} to 2^{60}. The stealing does not adversely affect vectors and procedures because those values are addresses, and our addresses are 8-byte aligned so the rightmost 3 bits are unused, they are always 000. Thus, we do not lose information by overwriting the rightmost 3 bits with the tag and we can simply zero-out the tag to recover the original address.

   To make tagged values into first-class entities, we can give them a type, called `Any`, and define operations such as `Inject` and `Project` for creating and using them, yielding the $R_{\texttt{Any}}$ intermediate language. We describe how to compile R_{Dyn} to $R_{\texttt{Any}}$ in Section 8.3 but first we describe the $R_{\texttt{Any}}$ language in greater detail.

8.2   The $R_{\texttt{Any}}$ Language

The abstract syntax of $R_{\texttt{Any}}$ is defined in Figure 8.5. (The concrete syntax of $R_{\texttt{Any}}$ is in the Appendix, Figure 12.1.) The (`Inject` $e\ T$) form converts the value produced by expression e of type T into a tagged value. The (`Project` $e\ T$) form converts the tagged value produced by expression e into a value of type T or else halts the program if the type tag is not equivalent to T. Note that in both `Inject` and `Project`, the type T is restricted to a flat type *ftype*, which simplifies the implementation and corresponds with what is needed for compiling R_{Dyn}.

$$
\begin{array}{lll}
\textit{type} & ::= & \ldots \mid \texttt{Any} \\
\textit{op} & ::= & \ldots \mid \texttt{any-vector-length} \mid \texttt{any-vector-ref} \mid \texttt{any-vector-set!} \\
& \mid & \texttt{boolean?} \mid \texttt{integer?} \mid \texttt{vector?} \mid \texttt{procedure?} \mid \texttt{void?} \\
\textit{exp} & ::= & \ldots \mid (\texttt{Prim } \textit{op } (\textit{exp} \ldots)) \\
& \mid & (\texttt{Inject } \textit{exp ftype}) \mid (\texttt{Project } \textit{exp ftype}) \\
\textit{def} & ::= & (\texttt{Def } \textit{var } ([\textit{var}\!:\!\textit{type}] \ldots) \textit{ type } \texttt{'() } \textit{exp}) \\
R_{\texttt{Any}} & ::= & (\texttt{ProgramDefsExp '() } (\textit{def} \ldots) \textit{ exp})
\end{array}
$$

Figure 8.5: The abstract syntax of $R_{\texttt{Any}}$, extending R_λ (Figure 7.4).

The `any-vector` operators adapt the vector operations so that they can be applied to a value of type `Any`. They also generalize the vector operations in that the index is not restricted to be a literal integer in the grammar but is allowed to be any expression.

The type predicates such as `boolean?` expect their argument to produce a tagged value; they return `#t` if the tag corresponds to the predicate and they return `#f` otherwise.

The type checker for $R_{\texttt{Any}}$ is shown in Figures 8.6 and 8.7 and uses the auxiliary functions in Figure 8.8. The interpreter for $R_{\texttt{Any}}$ is in Figure 8.9 and the auxiliary functions `apply-inject` and `apply-project` are in Figure 8.10.

```
(define type-check-Rany-class
  (class type-check-Rlambda-class
    (super-new)
    (inherit check-type-equal?)

    (define/override (type-check-exp env)
      (lambda (e)
        (define recur (type-check-exp env))
        (match e
          [(Inject e1 ty)
           (unless (flat-ty? ty)
             (error 'type-check "may only inject from flat type, not ~a" ty))
           (define-values (new-e1 e-ty) (recur e1))
           (check-type-equal? e-ty ty e)
           (values (Inject new-e1 ty) 'Any)]
          [(Project e1 ty)
           (unless (flat-ty? ty)
             (error 'type-check "may only project to flat type, not ~a" ty))
           (define-values (new-e1 e-ty) (recur e1))
           (check-type-equal? e-ty 'Any e)
           (values (Project new-e1 ty) ty)]
          [(Prim 'any-vector-length (list e1))
           (define-values (e1^ t1) (recur e1))
           (check-type-equal? t1 'Any e)
           (values (Prim 'any-vector-length (list e1^)) 'Integer)]
          [(Prim 'any-vector-ref (list e1 e2))
           (define-values (e1^ t1) (recur e1))
           (define-values (e2^ t2) (recur e2))
           (check-type-equal? t1 'Any e)
           (check-type-equal? t2 'Integer e)
           (values (Prim 'any-vector-ref (list e1^ e2^)) 'Any)]
          [(Prim 'any-vector-set! (list e1 e2 e3))
           (define-values (e1^ t1) (recur e1))
           (define-values (e2^ t2) (recur e2))
           (define-values (e3^ t3) (recur e3))
           (check-type-equal? t1 'Any e)
           (check-type-equal? t2 'Integer e)
           (check-type-equal? t3 'Any e)
           (values (Prim 'any-vector-set! (list e1^ e2^ e3^)) 'Void)]
```

Figure 8.6: Type checker for the $R_{\tt Any}$ language, part 1.

```
      [(ValueOf e ty)
       (define-values (new-e e-ty) (recur e))
       (values (ValueOf new-e ty) ty)]
      [(Prim pred (list e1))
       #:when (set-member? (type-predicates) pred)
       (define-values (new-e1 e-ty) (recur e1))
       (check-type-equal? e-ty 'Any e)
       (values (Prim pred (list new-e1)) 'Boolean)]
      [(If cnd thn els)
       (define-values (cnd^ Tc) (recur cnd))
       (define-values (thn^ Tt) (recur thn))
       (define-values (els^ Te) (recur els))
       (check-type-equal? Tc 'Boolean cnd)
       (check-type-equal? Tt Te e)
       (values (If cnd^ thn^ els^) (combine-types Tt Te))]
      [(Exit) (values (Exit) '_)]
      [(Prim 'eq? (list arg1 arg2))
       (define-values (e1 t1) (recur arg1))
       (define-values (e2 t2) (recur arg2))
       (match* (t1 t2)
         [(`(Vector ,ts1 ...) `(Vector ,ts2 ...)) (void)]
         [(other wise) (check-type-equal? t1 t2 e)])
       (values (Prim 'eq? (list e1 e2)) 'Boolean)]
      [else ((super type-check-exp env) e)])))

))
```

Figure 8.7: Type checker for the R_{Any} language, part 2.

```
(define/override (operator-types)
  (append
   '((integer? . ((Any) . Boolean))
     (vector? . ((Any) . Boolean))
     (procedure? . ((Any) . Boolean))
     (void? . ((Any) . Boolean))
     (tag-of-any . ((Any) . Integer))
     (make-any . ((_ Integer) . Any))
     )
   (super operator-types)))

(define/public (type-predicates)
  (set 'boolean? 'integer? 'vector? 'procedure? 'void?))

(define/public (combine-types t1 t2)
  (match (list t1 t2)
    [(list '_ t2) t2]
    [(list t1 '_) t1]
    [(list `(Vector ,ts1 ...)
           `(Vector ,ts2 ...))
     `(Vector ,@(for/list ([t1 ts1] [t2 ts2])
                  (combine-types t1 t2)))]
    [(list `(,ts1 ... -> ,rt1)
           `(,ts2 ... -> ,rt2))
     `(,@(for/list ([t1 ts1] [t2 ts2])
           (combine-types t1 t2))
       -> ,(combine-types rt1 rt2))]
    [else t1]))

(define/public (flat-ty? ty)
  (match ty
    [(or `Integer `Boolean '_ `Void) #t]
    [`(Vector ,ts ...) (for/and ([t ts]) (eq? t 'Any))]
    [`(,ts ... -> ,rt)
     (and (eq? rt 'Any) (for/and ([t ts]) (eq? t 'Any)))]
    [else #f]))
```

Figure 8.8: Auxiliary methods for type checking R_{Any}.

```
(define interp-Rany-class
  (class interp-Rlambda-class
    (super-new)

    (define/override (interp-op op)
      (match op
        ['boolean? (match-lambda
                     [`(tagged ,v1 ,tg) (equal? tg (any-tag 'Boolean))]
                     [else #f])]
        ['integer? (match-lambda
                     [`(tagged ,v1 ,tg) (equal? tg (any-tag 'Integer))]
                     [else #f])]
        ['vector? (match-lambda
                     [`(tagged ,v1 ,tg) (equal? tg (any-tag `(Vector Any)))]
                     [else #f])]
        ['procedure? (match-lambda
                       [`(tagged ,v1 ,tg) (equal? tg (any-tag `(Any -> Any)))]
                       [else #f])]
        ['eq? (match-lambda*
                [`((tagged ,v1^ ,tg1) (tagged ,v2^ ,tg2))
                 (and (eq? v1^ v2^) (equal? tg1 tg2))]
                [ls (apply (super interp-op op) ls)])]
        ['any-vector-ref (lambda (v i)
                           (match v [`(tagged ,v^ ,tg) (vector-ref v^ i)]))]
        ['any-vector-set! (lambda (v i a)
                            (match v [`(tagged ,v^ ,tg) (vector-set! v^ i a)]))]
        ['any-vector-length (lambda (v)
                              (match v [`(tagged ,v^ ,tg) (vector-length v^)]))]
        [else (super interp-op op)]))

    (define/override ((interp-exp env) e)
      (define recur (interp-exp env))
      (match e
        [(Inject e ty) `(tagged ,(recur e) ,(any-tag ty))]
        [(Project e ty2) (apply-project (recur e) ty2)]
        [else ((super interp-exp env) e)]))
  ))

(define (interp-Rany p)
  (send (new interp-Rany-class) interp-program p))
```

Figure 8.9: Interpreter for R_{Any}.

```
(define/public (apply-inject v tg) (Tagged v tg))

(define/public (apply-project v ty2)
  (define tag2 (any-tag ty2))
  (match v
    [(Tagged v1 tag1)
     (cond
       [(eq? tag1 tag2)
        (match ty2
          [`(Vector ,ts ...)
           (define l1 ((interp-op 'vector-length) v1))
           (cond
             [(eq? l1 (length ts)) v1]
             [else (error 'apply-project "vector length mismatch, ~a != ~a"
                          l1 (length ts))])]
          [`(,ts ... -> ,rt)
           (match v1
             [`(function ,xs ,body ,env)
              (cond [(eq? (length xs) (length ts)) v1]
                    [else
                     (error 'apply-project "arity mismatch ~a != ~a"
                            (length xs) (length ts))])]
             [else (error 'apply-project "expected function not ~a" v1)])]
          [else v1])]
       [else (error 'apply-project "tag mismatch ~a != ~a" tag1 tag2)])]
    [else (error 'apply-project "expected tagged value, not ~a" v)]))
```

Figure 8.10: Auxiliary functions for injection and projection.

8.3 Cast Insertion: Compiling R_{Dyn} to R_{Any}

The `cast-insert` pass compiles from R_{Dyn} to R_{Any}. Figure 8.11 shows the compilation of many of the R_{Dyn} forms into R_{Any}. An important invariant of this pass is that given a subexpression e in the R_{Dyn} program, the pass will produce an expression e' in R_{Any} that has type `Any`. For example, the first row in Figure 8.11 shows the compilation of the Boolean `#t`, which must be injected to produce an expression of type `Any`. The second row of Figure 8.11, the compilation of addition, is representative of compilation for many primitive operations: the arguments have type `Any` and must be projected to `Integer` before the addition can be performed.

The compilation of `lambda` (third row of Figure 8.11) shows what happens when we need to produce type annotations: we simply use `Any`. The compilation of `if` and `eq?` demonstrate how this pass has to account for some differences in behavior between R_{Dyn} and R_{Any}. The R_{Dyn} language is more permissive than R_{Any} regarding what kind of values can be used in various places. For example, the condition of an `if` does not have to be a Boolean. For `eq?`, the arguments need not be of the same type (in that case the result is `#f`).

8.4 Reveal Casts

In the `reveal-casts` pass we recommend compiling `project` into an `if` expression that checks whether the value's tag matches the target type; if it does, the value is converted to a value of the target type by removing the tag; if it does not, the program exits. To perform these actions we need a new primitive operation, `tag-of-any`, and two new forms, `ValueOf` and `Exit`. The `tag-of-any` operation retrieves the type tag from a tagged value of type `Any`. The `ValueOf` form retrieves the underlying value from a tagged value. The `ValueOf` form includes the type for the underlying value which is used by the type checker. Finally, the `Exit` form ends the execution of the program.

If the target type of the projection is `Boolean` or `Integer`, then `Project` can be translated as follows.

`#t`	$\Rightarrow$	`(inject #t Boolean)`

`(+ `e_1` `e_2`)`	$\Rightarrow$	`(inject`
		`   (+ (project `e_1'` Integer)`
		`      (project `e_2'` Integer))`
		`   Integer)`

`(lambda (`$x_1 \ldots x_n$`) `e`)`	$\Rightarrow$	`(inject`
		`   (lambda: ([`x_1`:Any]...[`x_n`:Any]):Any `e'`)`
		`   (Any...Any -> Any))`

`(`e_0` `$e_1 \ldots e_n$`)`	$\Rightarrow$	`((project `e_0'` (Any...Any -> Any)) `$e_1' \ldots e_n'$`)`
`(vector-ref `e_1` `e_2`)`	$\Rightarrow$	`(any-vector-ref `e_1'` `e_2'`)`
`(if `e_1` `e_2` `e_3`)`	$\Rightarrow$	`(if (eq? `e_1'` (inject #f Boolean)) `e_3'` `e_2'`)`
`(eq? `e_1` `e_2`)`	$\Rightarrow$	`(inject (eq? `e_1'` `e_2'`) Boolean)`
`(not `e_1`)`	$\Rightarrow$	`(if (eq? `e_1'` (inject #f Boolean))` `    (inject #t Boolean) (inject #f Boolean))`

Figure 8.11: Cast Insertion

```
(Project e ftype)
⇒
(Let tmp e′
   (If (Prim 'eq? (list (Prim 'tag-of-any (list (Var tmp)))
                        (Int tagof(ftype))))
       (ValueOf tmp ftype)
       (Exit)))
```

If the target type of the projection is a vector or function type, then there is a bit more work to do. For vectors, check that the length of the vector type matches the length of the vector (using the `vector-length` primitive). For functions, check that the number of parameters in the function type matches the function's arity (using `procedure-arity`).

Regarding `inject`, we recommend compiling it to a slightly lower-level primitive operation named `make-any`. This operation takes a tag instead of a type.

```
(Inject e ftype)
⇒
(Prim 'make-any (list e′ (Int tagof(ftype))))
```

The type predicates (`boolean?`, etc.) can be translated into uses of `tag-of-any` and `eq?` in a similar way as in the translation of `Project`.

The `any-vector-ref` and `any-vector-set!` operations combine the projection action with the vector operation. Also, the read and write operations allow arbitrary expressions for the index so the type checker for R_{Any} (Figure 8.6) cannot guarantee that the index is within bounds. Thus, we insert code to perform bounds checking at runtime. The translation for `any-vector-ref` is as follows and the other two operations are translated in a similar way.

```
(Prim 'any-vector-ref (list e₁ e₂))
⇒
(Let v e′₁
  (Let i e′₂
    (If (Prim 'eq? (list (Prim 'tag-of-any (list (Var v))) (Int 2)))
       (If (Prim '< (list (Var i)
             (Prim 'any-vector-length (list (Var v)))))
         (Prim 'any-vector-ref (list (Var v) (Var i)))
         (Exit))))
```

$$
\begin{array}{lll}
exp & ::= & \dots \mid (\texttt{Prim 'any-vector-ref } (atm\ atm)) \\
 & \mid & (\texttt{Prim 'any-vector-set! } (\texttt{list } atm\ atm\ atm)) \\
 & \mid & (\texttt{ValueOf } exp\ ftype) \\
stmt & ::= & (\texttt{Assign } (\texttt{Var } var)\ exp) \mid (\texttt{Collect } int) \\
tail & ::= & (\texttt{Return } exp) \mid (\texttt{Seq } stmt\ tail) \mid (\texttt{Goto } label) \\
 & \mid & (\texttt{IfStmt } (\texttt{Prim } cmp\ (atm\ atm))\ (\texttt{Goto } label)\ (\texttt{Goto } label)) \\
 & \mid & (\texttt{TailCall } atm\ atm \dots) \mid (\texttt{Exit}) \\
def & ::= & (\texttt{Def } label\ ([var{:}type] \dots)\ type\ info\ ((label\ .\ tail) \dots)) \\
C_{\texttt{Clos}} & ::= & (\texttt{ProgramDefs } info\ (def \dots))
\end{array}
$$

Figure 8.12: The abstract syntax of $C_{\texttt{Any}}$, extending $C_{\texttt{Clos}}$ (Figure 7.9).

8.5   Remove Complex Operands

The `ValueOf` and `Exit` forms are both complex expressions. The subexpression of `ValueOf` must be atomic.

8.6   Explicate Control and $C_{\texttt{Any}}$

The output of `explicate-control` is the $C_{\texttt{Any}}$ language whose syntax is defined in Figure 8.12. The `ValueOf` form that we added to $R_{\texttt{Any}}$ remains an expression and the `Exit` expression becomes a *tail*. Also, note that the index argument of `vector-ref` and `vector-set!` is an *atm* instead of an integer, as in $C_{\texttt{Vec}}$ (Figure 5.13).

8.7   Select Instructions

In the `select-instructions` pass we translate the primitive operations on the `Any` type to x86 instructions that involve manipulating the 3 tag bits of the tagged value.

Make-any   We recommend compiling the `make-any` primitive as follows if the tag is for `Integer` or `Boolean`. The `salq` instruction shifts the destination to the left by the number of bits specified its source argument (in this case 3, the length of the tag) and it preserves the sign of the integer. We use the `orq` instruction to combine the tag and the value to form the tagged value.

```
(Assign lhs (Prim 'make-any (list e (Int tag))))
⇒
movq e′, lhs′
salq $3, lhs′
orq $tag, lhs′
```

The instruction selection for vectors and procedures is different because their is no need to shift them to the left. The rightmost 3 bits are already zeros as described at the beginning of this chapter. So we just combine the value and the tag using `orq`.

```
(Assign lhs (Prim 'make-any (list e (Int tag))))
⇒
movq e′, lhs′
orq $tag, lhs′
```

Tag-of-any   Recall that the `tag-of-any` operation extracts the type tag from a value of type `Any`. The type tag is the bottom three bits, so we obtain the tag by taking the bitwise-and of the value with 111 (7 in decimal).

```
(Assign lhs (Prim 'tag-of-any (list e)))
⇒
movq e′, lhs′
andq $7, lhs′
```

ValueOf   Like `make-any`, the instructions for `ValueOf` are different depending on whether the type T is a pointer (vector or procedure) or not (Integer or Boolean). The following shows the instruction selection for Integer and Boolean. We produce an untagged value by shifting it to the right by 3 bits.

```
(Assign lhs (ValueOf e T))
⇒
movq e′, lhs′
sarq $3, lhs′
```

In the case for vectors and procedures, there is no need to shift. Instead we just need to zero-out the rightmost 3 bits. We accomplish this by creating the bit pattern . . . 0111 (7 in decimal) and apply `bitwise-not` to obtain . . . 11111000 (-8 in decimal) which we `movq` into the destination *lhs*. We then apply `andq` with the tagged value to get the desired result.

```
(Assign lhs (ValueOf e T))
⇒
movq $−8, lhs′
andq e′, lhs′
```

```
(Assign lhs (Prim 'any-vector-length (list a₁)))
⟹
movq ¬111, %r11
andq a₁′, %r11
movq 0(%r11), %r11
andq $126, %r11
sarq $1, %r11
movq %r11, lhs′
```

Any-vector-ref   The index may be an arbitrary atom so instead of computing the offset at compile time, instructions need to be generated to compute the offset at runtime as follows. Note the use of the new instruction `imulq`.

```
(Assign lhs (Prim 'any-vector-ref (list a₁ a₂)))
⟹
movq ¬111, %r11
andq a₁′, %r11
movq a₂′, %rax
addq $1, %rax
imulq $8, %rax
addq %rax, %r11
movq 0(%r11) lhs′
```

Any-vector-set!   The code generation for `any-vector-set!` is similar to the other `any-vector` operations.

8.8   Register Allocation for $R_{\texttt{Any}}$

There is an interesting interaction between tagged values and garbage collection that has an impact on register allocation. A variable of type `Any` might refer to a vector and therefore it might be a root that needs to be inspected and copied during garbage collection. Thus, we need to treat variables of

type `Any` in a similar way to variables of type `Vector` for purposes of register allocation. In particular,
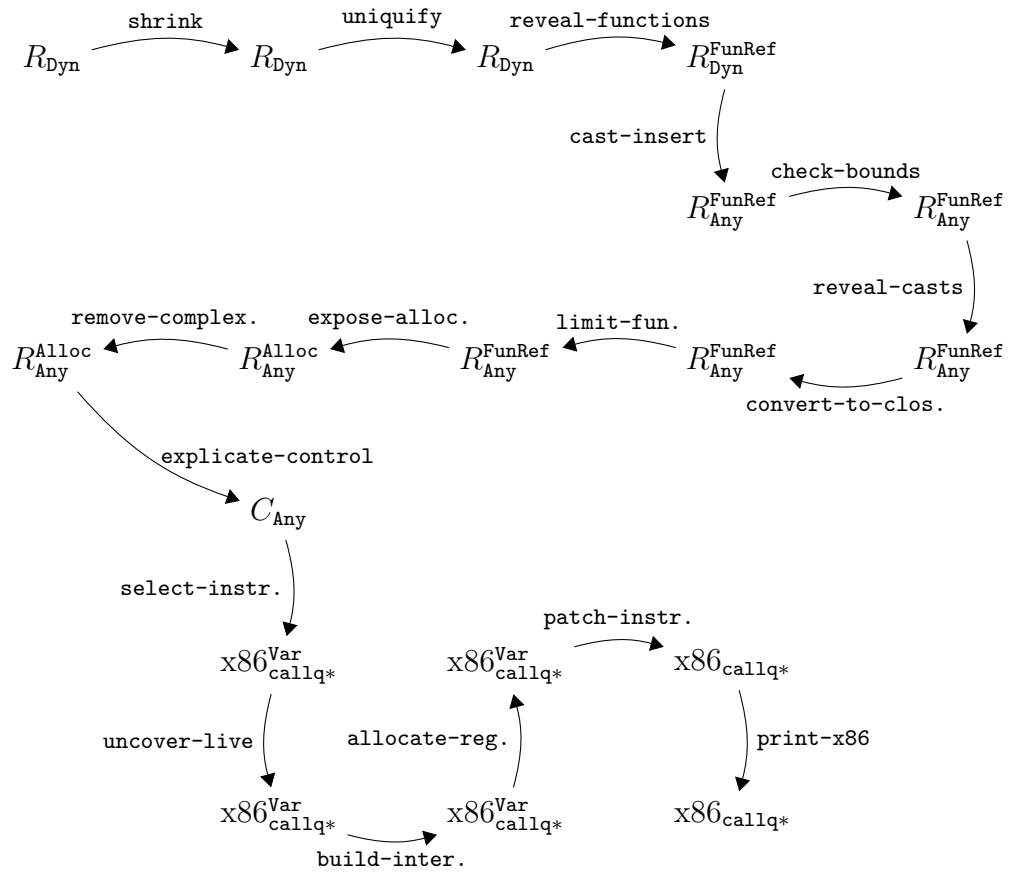
- If a variable of type `Any` is live during a function call, then it must be spilled. This can be accomplished by changing `build-interference` to mark all variables of type `Any` that are live after a `callq` as interfering with all the registers.

- If a variable of type `Any` is spilled, it must be spilled to the root stack instead of the normal procedure call stack.

Another concern regarding the root stack is that the garbage collector needs to differentiate between (1) plain old pointers to tuples, (2) a tagged value that points to a tuple, and (3) a tagged value that is not a tuple. We enable this differentiation by choosing not to use the tag 000 in the *tagof* function. Instead, that bit pattern is reserved for identifying plain old pointers to tuples. That way, if one of the first three bits is set, then we have a tagged value and inspecting the tag can differentiation between vectors (010) and the other kinds of values.

Exercise 35. Expand your compiler to handle $R_{\texttt{Any}}$ as discussed in the last few sections. Create 5 new programs that use the `Any` type and the new operations (`inject`, `project`, `boolean?`, etc.). Test your compiler on these new programs and all of your previously created test programs.

Exercise 36. Expand your compiler to handle $R_{\texttt{Dyn}}$ as outlined in this chapter. Create tests for $R_{\texttt{Dyn}}$ by adapting ten of your previous test programs by removing type annotations. Add 5 more tests programs that specifically rely on the language being dynamically typed. That is, they should not be legal programs in a statically typed language, but nevertheless, they should be valid $R_{\texttt{Dyn}}$ programs that run to completion without error.

Figure 8.13 provides an overview of all the passes needed for the compilation of $R_{\texttt{Dyn}}$.

$$R_{\texttt{Dyn}} \xrightarrow{\text{shrink}} R_{\texttt{Dyn}} \xrightarrow{\text{uniquify}} R_{\texttt{Dyn}} \xrightarrow{\text{reveal-functions}} R_{\texttt{Dyn}}^{\texttt{FunRef}}$$

cast-insert

$$R_{\texttt{Any}}^{\texttt{FunRef}} \xrightarrow{\text{check-bounds}} R_{\texttt{Any}}^{\texttt{FunRef}}$$

reveal-casts

remove-complex. $\quad$ expose-alloc. $\quad$ limit-fun.

$$R_{\texttt{Any}}^{\texttt{Alloc}} \leftarrow R_{\texttt{Any}}^{\texttt{Alloc}} \leftarrow R_{\texttt{Any}}^{\texttt{FunRef}} \leftarrow R_{\texttt{Any}}^{\texttt{FunRef}} \leftarrow R_{\texttt{Any}}^{\texttt{FunRef}}$$

convert-to-clos.

explicate-control

$$C_{\texttt{Any}}$$

select-instr.

$$\text{x86}_{\texttt{callq*}}^{\texttt{Var}} \qquad \text{x86}_{\texttt{callq*}}^{\texttt{Var}} \xrightarrow{\text{patch-instr.}} \text{x86}_{\texttt{callq*}}$$

uncover-live $\qquad$ allocate-reg. $\qquad$ print-x86

$$\text{x86}_{\texttt{callq*}}^{\texttt{Var}} \xrightarrow{\text{build-inter.}} \text{x86}_{\texttt{callq*}}^{\texttt{Var}} \qquad \text{x86}_{\texttt{callq*}}$$

Figure 8.13: Diagram of the passes for $R_{\texttt{Dyn}}$, a dynamically typed language.

9

Loops and Assignment

In this chapter we study two features that are the hallmarks of imperative programming languages: loops and assignments to local variables. The following example demonstrates these new features by computing the sum of the first five positive integers.

```
(let ([sum 0])
  (let ([i 5])
    (begin
      (while (> i 0)
        (begin
          (set! sum (+ sum i))
          (set! i (- i 1))))
      sum)))
```

The `while` loop consists of a condition and a body. The `set!` consists of a variable and a right-hand-side expression. The primary purpose of both the `while` loop and `set!` is to cause side effects, so it is convenient to also include in a language feature for sequencing side effects: the `begin` expression. It consists of one or more subexpressions that are evaluated left-to-right.

9.1   The $R_{\texttt{While}}$ Language

The concrete syntax of $R_{\texttt{While}}$ is defined in Figure 9.1 and its abstract syntax is defined in Figure 9.2. The definitional interpreter for $R_{\texttt{While}}$ is shown in Figure 9.3. We add three new cases for `SetBang`, `WhileLoop`, and `Begin` and we make changes to the cases for `Var`, `Let`, and `Apply` regarding variables. To support assignment to variables and to make their lifetimes indefinite (see the second example in Section 9.2), we box the value that is bound to

$$
\begin{aligned}
exp \quad &::= \quad int \mid (\texttt{read}) \mid (\texttt{-}\ exp) \mid (\texttt{+}\ exp\ exp) \mid (\texttt{-}\ exp\ exp) \\
&\mid \quad var \mid (\texttt{let}\ ([var\ exp])\ exp) \\
&\mid \quad \texttt{\#t} \mid \texttt{\#f} \mid (\texttt{and}\ exp\ exp) \mid (\texttt{or}\ exp\ exp) \mid (\texttt{not}\ exp) \\
&\mid \quad (\texttt{eq?}\ exp\ exp) \mid (\texttt{if}\ exp\ exp\ exp) \\
&\mid \quad (\texttt{vector}\ exp\ldots) \mid (\texttt{vector-ref}\ exp\ int) \\
&\mid \quad (\texttt{vector-set!}\ exp\ int\ exp) \mid (\texttt{void}) \mid (exp\ exp\ldots) \\
&\mid \quad (\texttt{procedure-arity}\ exp) \mid (\texttt{lambda:}\ ([var\!:\!type]\ldots)\!:\!type\ exp) \\
&\mid \quad \textbf{(set!}\ var\ exp\textbf{)} \mid \textbf{(begin}\ exp\ldots\ exp\textbf{)} \mid \textbf{(while}\ exp\ exp\textbf{)} \\
def \quad &::= \quad (\texttt{define}\ (var\ [var\!:\!type]\ldots)\!:\!type\ exp) \\
R_{\texttt{While}} \quad &::= \quad def\ldots\ exp
\end{aligned}
$$

Figure 9.1: The concrete syntax of $R_{\texttt{While}}$, extending $R_{\texttt{Any}}$ (Figure 12.1).

$$
\begin{aligned}
exp \quad &::= \quad (\texttt{Int}\ int)(\texttt{Var}\ var) \mid (\texttt{Let}\ var\ exp\ exp) \\
&\mid \quad (\texttt{Prim}\ op\ (exp\ldots)) \\
&\mid \quad (\texttt{Bool}\ bool) \mid (\texttt{If}\ exp\ exp\ exp) \\
&\mid \quad (\texttt{Void}) \mid (\texttt{HasType}\ exp\ type) \mid (\texttt{Apply}\ exp\ exp\ldots) \\
&\mid \quad (\texttt{Lambda}\ ([var\!:\!type]\ldots)\ type\ exp) \\
&\mid \quad \textbf{(SetBang}\ var\ exp\textbf{)} \mid \textbf{(Begin}\ (exp\ldots)\ exp\textbf{)} \mid \textbf{(WhileLoop}\ exp\ exp\textbf{)} \\
def \quad &::= \quad (\texttt{Def}\ var\ ([var\!:\!type]\ldots)\ type\ \texttt{'()}\ exp) \\
R_{\texttt{While}} \quad &::= \quad (\texttt{ProgramDefsExp}\ \texttt{'()}\ (def\ldots)\ exp)
\end{aligned}
$$

Figure 9.2: The abstract syntax of $R_{\texttt{While}}$, extending $R_{\texttt{Any}}$ (Figure 8.5).

```
(define interp-Rwhile-class
  (class interp-Rany-class
    (super-new)

    (define/override ((interp-exp env) e)
      (define recur (interp-exp env))
      (match e
        [(SetBang x rhs)
         (set-box! (lookup x env) (recur rhs))]
        [(WhileLoop cnd body)
         (define (loop)
           (cond [(recur cnd) (recur body) (loop)]
                 [else        (void)]))
         (loop)]
        [(Begin es body)
         (for ([e es]) (recur e))
         (recur body)]
        [else ((super interp-exp env) e)]))
    ))

(define (interp-Rwhile p)
  (send (new interp-Rwhile-class) interp-program p))
```

Figure 9.3: Interpreter for $R_{\texttt{While}}$.

each variable (in `Let`) and function parameter (in `Apply`). The case for `Var` unboxes the value. Now to discuss the new cases. For `SetBang`, we lookup the variable in the environment to obtain a boxed value and then we change it using `set-box!` to the result of evaluating the right-hand side. The result value of a `SetBang` is void. For the `WhileLoop`, we repeatedly 1) evaluate the condition, and if the result is true, 2) evaluate the body. The result value of a `while` loop is also void. Finally, the (`Begin` *es body*) expression evaluates the subexpressions *es* for their effects and then evaluates and returns the result from *body*.

The type checker for $R_{\texttt{While}}$ is define in Figure 9.4. For `SetBang`, the type of the variable and the right-hand-side must agree. The result type is `Void`. For the `WhileLoop`, the condition must be a `Boolean`. The result type is also `Void`. For `Begin`, the result type is the type of its last subexpression.

At first glance, the translation of these language features to x86 seems straightforward because the $C_{\texttt{Fun}}$ intermediate language already supports all of the ingredients that we need: assignment, `goto`, conditional branching, and sequencing. However, there are two complications that arise which we discuss in the next two sections. After that we introduce one new compiler

```
(define type-check-Rwhile-class
  (class type-check-Rany-class
    (super-new)
    (inherit check-type-equal?)

    (define/override (type-check-exp env)
      (lambda (e)
        (define recur (type-check-exp env))
        (match e
          [(SetBang x rhs)
           (define-values (rhs^ rhsT) (recur rhs))
           (define varT (dict-ref env x))
           (check-type-equal? rhsT varT e)
           (values (SetBang x rhs^) 'Void)]
          [(WhileLoop cnd body)
           (define-values (cnd^ Tc) (recur cnd))
           (check-type-equal? Tc 'Boolean e)
           (define-values (body^ Tbody) ((type-check-exp env) body))
           (values (WhileLoop cnd^ body^) 'Void)]
          [(Begin es body)
           (define-values (es^ ts)
             (for/lists (l1 l2) ([e es]) (recur e)))
           (define-values (body^ Tbody) (recur body))
           (values (Begin es^ body^) Tbody)]
          [else ((super type-check-exp env) e)]))))
    ))

(define (type-check-Rwhile p)
  (send (new type-check-Rwhile-class) type-check-program p))
```

Figure 9.4: Type checking `SetBang`, `WhileLoop`, and `Begin` in $R_{\texttt{While}}$.

pass and the changes necessary to the existing passes.

9.2   Assignment and Lexically Scoped Functions

The addition of assignment raises a problem with our approach to implementing lexically-scoped functions. Consider the following example in which function `f` has a free variable `x` that is changed after `f` is created but before the call to `f`.

```
(let ([x 0])
  (let ([y 0])
    (let ([z 20])
      (let ([f (lambda: ([a : Integer]) : Integer (+ a (+ x z)))])
        (begin
          (set! x 10)
          (set! y 12)
          (f y))))))
```

The correct output for this example is `42` because the call to `f` is required to use the current value of `x` (which is `10`). Unfortunately, the closure conversion pass (Section 7.3) generates code for the `lambda` that copies the old value of `x` into a closure. Thus, if we naively add support for assignment to our current compiler, the output of this program would be `32`.

A first attempt at solving this problem would be to save a pointer to `x` in the closure and change the occurrences of `x` inside the lambda to dereference the pointer. Of course, this would require assigning `x` to the stack and not to a register. However, the problem goes a bit deeper. Consider the following example in which we create a counter abstraction by creating a pair of functions that share the free variable `x`.

```
(define (f [x : Integer]) : (Vector ( -> Integer) ( -> Void))
  (vector
   (lambda: () : Integer x)
   (lambda: () : Void (set! x (+ 1 x)))))

(let ([counter (f 0)])
  (let ([get (vector-ref counter 0)])
    (let ([inc (vector-ref counter 1)])
      (begin
        (inc)
        (get)))))
```

In this example, the lifetime of `x` extends beyond the lifetime of the call to `f`. Thus, if we were to store `x` on the stack frame for the call to `f`, it would

be gone by the time we call `inc` and `get`, leaving us with dangling pointers
for `x`. This example demonstrates that when a variable occurs free inside a
`lambda`, its lifetime becomes indefinite. Thus, the value of the variable needs
to live on the heap. The verb "box" is often used for allocating a single value
on the heap, producing a pointer, and "unbox" for dereferencing the pointer.

We recommend solving these problems by "boxing" the local variables
that are in the intersection of 1) variables that appear on the left-hand-side of
a `set!` and 2) variables that occur free inside a `lambda`. We shall introduce
a new pass named `convert-assignments` in Section 9.4 to perform this
translation. But before diving into the compiler passes, we one more problem
to discuss.

9.3   Cyclic Control Flow and Dataflow Analysis

Up until this point the control-flow graphs generated in `explicate-control`
were guaranteed to be acyclic. However, each `while` loop introduces a cycle
in the control-flow graph. But does that matter? Indeed it does. Recall that
for register allocation, the compiler performs liveness analysis to determine
which variables can share the same register. In Section 4.10.1 we analyze
the control-flow graph in reverse topological order, but topological order is
only well-defined for acyclic graphs.

Let us return to the example of computing the sum of the first five
positive integers. Here is the program after instruction selection but before
register allocation.

```
(define (main) : Integer
   mainstart:
      movq $0, sum1
      movq $5, i2
      jmp block5
   block5:
      movq i2, tmp3
      cmpq tmp3, $0
      jl block7
      jmp block8
```
```
block7:
      addq i2, sum1
      movq $1, tmp4
      negq tmp4
      addq tmp4, i2
      jmp block5
   block8:
      movq $27, %rax
      addq sum1, %rax
      jmp mainconclusion
)
```

Recall that liveness analysis works backwards, starting at the end of each
function. For this example we could start with `block8` because we know

what is live at the beginning of the conclusion, just `rax` and `rsp`. So the live-before set for `block8` is {`rsp`, `sum1`}. Next we might try to analyze `block5` or `block7`, but `block5` jumps to `block7` and vice versa, so it seems that we are stuck.

The way out of this impasse comes from the realization that one can perform liveness analysis starting with an empty live-after set to compute an under-approximation of the live-before set. By *under-approximation*, we mean that the set only contains variables that are really live, but it may be missing some. Next, the under-approximations for each block can be improved by 1) updating the live-after set for each block using the approximate live-before sets from the other blocks and 2) perform liveness analysis again on each block. In fact, by iterating this process, the under-approximations eventually become the correct solutions! This approach of iteratively analyzing a control-flow graph is applicable to many static analysis problems and goes by the name *dataflow analysis*. It was invented by Kildall [62] in his Ph.D. thesis at the University of Washington.

Let us apply this approach to the above example. We use the empty set for the initial live-before set for each block. Let m_0 be the following mapping from label names to sets of locations (variables and registers).

```
mainstart: {}
block5: {}
block7: {}
block8: {}
```

Using the above live-before approximations, we determine the live-after for each block and then apply liveness analysis to each block. This produces our next approximation m_1 of the live-before sets.

```
mainstart: {}
block5: {i2}
block7: {i2, sum1}
block8: {rsp, sum1}
```

For the second round, the live-after for `mainstart` is the current live-before for `block5`, which is {`i2`}. So the liveness analysis for `mainstart` computes the empty set. The live-after for `block5` is the union of the live-before sets for `block7` and `block8`, which is {`i2`, `rsp`, `sum1`}. So the liveness analysis for `block5` computes {`i2`, `rsp`, `sum1`}. The live-after for `block7` is the live-before for `block5` (from the previous iteration), which is {`i2`}. So the liveness analysis for `block7` remains {`i2`, `sum1`}. Together these yield the following approximation m_2 of the live-before sets.

```
mainstart: {}
block5: {i2, rsp, sum1}
block7: {i2, sum1}
block8: {rsp, sum1}
```

In the preceding iteration, only `block5` changed, so we can limit our attention to `mainstart` and `block7`, the two blocks that jump to `block5`. As a result, the live-before sets for `mainstart` and `block7` are updated to include `rsp`, yielding the following approximation m_3.

```
mainstart: {rsp}
block5: {i2, rsp, sum1}
block7: {i2, rsp, sum1}
block8: {rsp, sum1}
```

Because `block7` changed, we analyze `block5` once more, but its live-before set remains `{ i2, rsp, sum1 }`. At this point our approximations have converged, so m_3 is the solution.

This iteration process is guaranteed to converge to a solution by the Kleene Fixed-Point Theorem, a general theorem about functions on lattices [63]. Roughly speaking, a *lattice* is any collection that comes with a partial ordering $\sqsubseteq$ on its elements, a least element $\bot$ (pronounced bottom), and a join operator $\sqcup$.[1] When two elements are ordered $m_i \sqsubseteq m_j$, it means that m_j contains at least as much information as m_i, so we can think of m_j as a better-or-equal approximation than m_i. The bottom element $\bot$ represents the complete lack of information, i.e., the worst approximation. The join operator takes two lattice elements and combines their information, i.e., it produces the least upper bound of the two.

A dataflow analysis typically involves two lattices: one lattice to represent abstract states and another lattice that aggregates the abstract states of all the blocks in the control-flow graph. For liveness analysis, an abstract state is a set of locations. We form the lattice L by taking its elements to be sets of locations, the ordering to be set inclusion ($\subseteq$), the bottom to be the empty set, and the join operator to be set union. We form a second lattice M by taking its elements to be mappings from the block labels to sets of locations (elements of L). We order the mappings point-wise, using the ordering of L. So given any two mappings m_i and m_j, $m_i \sqsubseteq_M m_j$ when $m_i(\ell) \subseteq m_j(\ell)$ for every block label ℓ in the program. The bottom

---

[1]Technically speaking, we will be working with join semi-lattices.

element of M is the mapping $\perp_M$ that sends every label to the empty set, i.e., $\perp_M(\ell) = \emptyset$.

We can think of one iteration of liveness analysis as being a function f on the lattice M. It takes a mapping as input and computes a new mapping.

$$f(m_i) = m_{i+1}$$

Next let us think for a moment about what a final solution m_s should look like. If we perform liveness analysis using the solution m_s as input, we should get m_s again as the output. That is, the solution should be a *fixed point* of the function f.

$$f(m_s) = m_s$$

Furthermore, the solution should only include locations that are forced to be there by performing liveness analysis on the program, so the solution should be the *least* fixed point.

The Kleene Fixed-Point Theorem states that if a function f is monotone (better inputs produce better outputs), then the least fixed point of f is the least upper bound of the *ascending Kleene chain* obtained by starting at $\perp$ and iterating f as follows.

$$\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq \cdots \sqsubseteq f^n(\perp) \sqsubseteq \cdots$$

When a lattice contains only finitely-long ascending chains, then every Kleene chain tops out at some fixed point after a number of iterations of f. So that fixed point is also a least upper bound of the chain.

$$\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq \cdots \sqsubseteq f^k(\perp) = f^{k+1}(\perp) = m_s$$

The liveness analysis is indeed a monotone function and the lattice M only has finitely-long ascending chains because there are only a finite number of variables and blocks in the program. Thus we are guaranteed that iteratively applying liveness analysis to all blocks in the program will eventually produce the least fixed point solution.

Next let us consider dataflow analysis in general and discuss the generic work list algorithm (Figure 9.5). The algorithm has four parameters: the control-flow graph `G`, a function `transfer` that applies the analysis to one block, the `bottom` and `join` operator for the lattice of abstract states. The algorithm begins by creating the bottom mapping, represented by a hash table. It then pushes all of the nodes in the control-flow graph onto the work list (a queue). The algorithm repeats the `while` loop as long as there are items in the work list. In each iteration, a node is popped from the work list

```
(define (analyze-dataflow G transfer bottom join)
  (define mapping (make-hash))
  (for ([v (in-vertices G)])
    (dict-set! mapping v bottom))
  (define worklist (make-queue))
  (for ([v (in-vertices G)])
    (enqueue! worklist v))
  (define trans-G (transpose G))
  (while (not (queue-empty? worklist))
    (define node (dequeue! worklist))
    (define input (for/fold ([state bottom])
                            ([pred (in-neighbors trans-G node)])
                   (join state (dict-ref mapping pred))))
    (define output (transfer node input))
    (cond [(not (equal? output (dict-ref mapping node)))
           (dict-set! mapping node output)
           (for ([v (in-neighbors G node)])
             (enqueue! worklist v))]))
  mapping)
```

Figure 9.5: Generic work list algorithm for dataflow analysis

and processed. The `input` for the node is computed by taking the join of
the abstract states of all the predecessor nodes. The `transfer` function is
then applied to obtain the `output` abstract state. If the output differs from
the previous state for this block, the mapping for this block is updated and
its successor nodes are pushed onto the work list.

Having discussed the two complications that arise from adding support
for assignment and loops, we turn to discussing the one new compiler pass
and the significant changes to existing passes.

9.4   Convert Assignments

Recall that in Section 9.2 we learned that the combination of assignments
and lexically-scoped functions requires that we box those variables that are
both assigned-to and that appear free inside a `lambda`. The purpose of
the `convert-assignments` pass is to carry out that transformation. We
recommend placing this pass after `uniquify` but before `reveal-functions`.

Consider again the first example from Section 9.2:

```
(let ([x 0])
```

```
(let ([y 0])
  (let ([z 20])
    (let ([f (lambda: ([a : Integer]) : Integer (+ a (+ x z)))])
      (begin
        (set! x 10)
        (set! y 12)
        (f y))))))
```

The variables x and y are assigned-to. The variables x and z occur free inside the lambda. Thus, variable x needs to be boxed but not y and z. The boxing of x consists of three transformations: initialize x with a vector, replace reads from x with vector-ref's, and replace each set! on x with a vector-set!. The output of convert-assignments for this example is as follows.

```
(define (main) : Integer
  (let ([x0 (vector 0)])
    (let ([y1 0])
      (let ([z2 20])
        (let ([f4 (lambda: ([a3 : Integer]) : Integer
                    (+ a3 (+ (vector-ref x0 0) z2)))])
          (begin
            (vector-set! x0 0 10)
            (set! y1 12)
            (f4 y1)))))))
```

Assigned & Free   We recommend defining an auxiliary function named assigned&free that takes an expression and simultaneously computes 1) a set of assigned variables A, 2) a set F of variables that occur free within lambda's, and 3) a new version of the expression that records which bound variables occurred in the intersection of A and F. You can use the struct AssignedFree to do this. Consider the case for (Let *x rhs body*). Suppose the the recursive call on *rhs* produces rhs', A_r, and F_r and the recursive call on the *body* produces $body'$, A_b, and F_b. If x is in $A_b \cap F_b$, then transforms the Let as follows.

(Let x *rhs body*)
$\Rightarrow$
(Let (AssignedFree x) rhs' $body'$)

If x is not in $A_b \cap F_b$ then omit the use of AssignedFree. The set of assigned variables for this Let is $A_r \cup (A_b - \{x\})$ and the set of variables free in lambda's is $F_r \cup (F_b - \{x\})$.

The case for (SetBang *x rhs*) is straightforward but important. Recursively process *rhs* to obtain rhs', A_r, and F_r. The result is (SetBang *x rhs′*), $\{x\} \cup A_r$, and F_r.

The case for (Lambda *params T body*) is a bit more involved. Let *body′*, A_b, and F_b be the result of recursively processing *body*. Wrap each of parameter that occurs in $A_b \cap F_b$ with AssignedFree to produce *params′*. Let P be the set of parameter names in *params*. The result is (Lambda *params′ T body′*), $A_b - P$, and $(F_b \cup \mathrm{FV}(body)) - P$, where FV computes the free variables of an expression (see Chapter 7).

Convert Assignments Next we discuss the convert-assignment pass with its auxiliary functions for expressions and definitions. The function for expressions, cnvt-assign-exp, should take an expression and a set of assigned-and-free variables (obtained from the result of assigned&free. In the case for (Var *x*), if x is assigned-and-free, then unbox it by translating (Var *x*) to a vector-ref.

```
(Var x)
⇒
(Prim 'vector-ref (list (Var x) (Int 0)))
```

In the case for (Let (AssignedFree *x*) *rhs body*), recursively process *rhs* to obtain *rhs′*. Next, recursively process *body* to obtain *body′* but with x added to the set of assigned-and-free variables. Translate the let-expression as follows to bind x to a boxed value.

```
(Let (AssignedFree x) rhs body)
⇒
(Let x (Prim 'vector (list rhs′)) body′)
```

In the case for (SetBang *x rhs*), recursively process *rhs* to obtain *rhs′*. If x is in the assigned-and-free variables, translate the set! into a vector-set! as follows.

```
(SetBang x rhs)
⇒
(Prim 'vector-set! (list (Var x) (Int 0) rhs′))
```

The case for Lambda is non-trivial, but it is similar to the case for function definitions, which we discuss next.

The auxiliary function for definitions, cnvt-assign-def, applies assignment conversion to function definitions. We translate a function definition as follows.

$$
\begin{array}{rcl}
atm & ::= & (\text{Int } int) \mid (\text{Var } var) \mid (\text{Bool } bool) \mid (\text{Void}) \\
exp & ::= & \ldots \mid (\text{Let } var\ exp\ exp) \\
    & \mid & (\text{WhileLoop } exp\ exp) \mid (\text{SetBang } var\ exp) \mid (\text{Begin } (exp\ldots)\ exp) \\
def & ::= & (\text{Def } var\ ([var\!:\!type]\ldots)\ type\ \text{'()}\ exp) \\
R_8^\dagger & ::= & (\text{ProgramDefs '()}\ def)
\end{array}
$$

Figure 9.6: $R_{\texttt{While}}^{\textsf{ANF}}$ is $R_{\texttt{While}}$ in administrative normal form (ANF).

```
(Def f params T info body₁)
⇒
(Def f params' T info body₄)
```

So it remains to explain *params'* and *body₄*. Let *body₂*, A_b, and F_b be the result of `assigned&free` on *body₁*. Let P be the parameter names in *params*. We then apply `cnvt-assign-exp` to *body₂* to obtain *body₃*, passing $A_b \cap F_b \cap P$ as the set of assigned-and-free variables. Finally, we obtain *body₄* by wrapping *body₃* in a sequence of let-expressions that box the parameters that are in $A_b \cap F_b$. Regarding *params'*, change the names of the parameters that are in $A_b \cap F_b$ to maintain uniqueness (and so the let-bound variables can retain the original names). Recall the second example in Section 9.2 involving a counter abstraction. The following is the output of assignment version for function `f`.

```
(define (f0 [x1 : Integer]) : (Vector ( -> Integer) ( -> Void))
  (vector
   (lambda: () : Integer x1)
   (lambda: () : Void (set! x1 (+ 1 x1)))))
⇒
(define (f0 [param_x1 : Integer]) : (Vector (-> Integer) (-> Void))
  (let ([x1 (vector param_x1)])
    (vector (lambda: () : Integer (vector-ref x1 0))
            (lambda: () : Void
               (vector-set! x1 0 (+ 1 (vector-ref x1 0)))))))
```

9.5   Remove Complex Operands

The three new language forms, `while`, `set!`, and `begin` are all complex expressions and their subexpressions are allowed to be complex. Figure 6.8 defines the output language $R_{\texttt{Fun}}^{\textsf{ANF}}$ of this pass.

As usual, when a complex expression appears in a grammar position that needs to be atomic, such as the argument of a primitive operator, we

must introduce a temporary variable and bind it to the complex expression. This approach applies, unchanged, to handle the new language forms. For example, in the following code there are two `begin` expressions appearing as arguments to `+`. The output of `rco-exp` is shown below, in which the `begin` expressions have been bound to temporary variables. Recall that `let` expressions in $R_{\mathtt{While}}^{\mathsf{ANF}}$ are allowed to have arbitrary expressions in their right-hand-side expression, so it is fine to place `begin` there.

```
(let ([x0 10])
  (let ([y1 0])
    (+ (+ (begin (set! y1 (read)) x0)
          (begin (set! x0 (read)) y1))
       x0)))
⇒
(let ([x0 10])
  (let ([y1 0])
    (let ([tmp2 (begin (set! y1 (read)) x0)])
      (let ([tmp3 (begin (set! x0 (read)) y1)])
        (let ([tmp4 (+ tmp2 tmp3)])
          (+ tmp4 x0))))))
```

9.6   Explicate Control and $C_{\circlearrowleft}$

Recall that in the `explicate-control` pass we define one helper function for each kind of position in the program. For the $R_{\mathtt{Var}}$ language of integers and variables we needed kinds of positions: assignment and tail. The `if` expressions of $R_{\mathtt{If}}$ introduced predicate positions. For $R_{\mathtt{While}}$, the `begin` expression introduces yet another kind of position: effect position. Except for the last subexpression, the subexpressions inside a `begin` are evaluated only for their effect. Their result values are discarded. We can generate better code by taking this fact into account.

The output language of `explicate-control` is $C_{\circlearrowleft}$ (Figure 9.7), which is nearly identical to $C_{\mathtt{Clos}}$. The only syntactic difference is that `Call`, `vector-set!`, and `read` may also appear as statements. The most significant difference between $C_{\mathtt{Clos}}$ and $C_{\circlearrowleft}$ is that the control-flow graphs of the later may contain cycles.

The new auxiliary function `explicate-effect` takes an expression (in an effect position) and a promise of a continuation block. The function returns a promise for a *tail* that includes the generated code for the input expression followed by the continuation block. If the expression is obviously pure, that is, never causes side effects, then the expression can be removed, so the

$$
\begin{array}{rcl}
stmt & ::= & (\texttt{Assign (Var } var) \; exp) \mid (\texttt{Collect } int) \\
 & \mid & (\texttt{Call } atm \; (atm \dots)) \mid (\texttt{Prim read ()}) \\
 & \mid & (\texttt{Prim 'vector-set! (list } atm \; (\texttt{Int } int) \; atm)) \\
def & ::= & (\texttt{Def } label \; ([var\!:\!type] \dots) \; type \; info \; ((label \,.\, tail) \dots)) \\
C_{\circlearrowleft} & ::= & (\texttt{ProgramDefs } info \; (def \dots))
\end{array}
$$

Figure 9.7: The abstract syntax of $C_{\circlearrowleft}$, extending $C_{\texttt{Clos}}$ (Figure 7.9).

result is just the continuation block. The (`WhileLoop` *cnd body*) expression is the most interesting case. First, you will need a fresh label *loop* for the top of the loop. Recursively process the *body* (in effect position) with the a `goto` to *loop* as the continuation, producing *body′*. Next, process the *cnd* (in predicate position) with *body′* as the then-branch and the continuation block as the else-branch. The result should be added to the control-flow graph with the label *loop*. The result for the whole `while` loop is a `goto` to the *loop* label. Note that the loop should only be added to the control-flow graph if the loop is indeed used, which can be accomplished using `delay`.

The auxiliary functions for tail, assignment, and predicate positions need to be updated. The three new language forms, `while`, `set!`, and `begin`, can appear in assignment and tail positions. Only `begin` may appear in predicate positions; the other two have result type `Void`.

9.7 Select Instructions

Only three small additions are needed in the `select-instructions` pass to handle the changes to $C_{\circlearrowleft}$. That is, `Call`, `read`, and `vector-set!` may now appear as stand-alone statements instead of only appearing on the right-hand side of an assignment statement. The code generation is nearly identical; just leave off the instruction for moving the result into the left-hand side.

9.8 Register Allocation

As discussed in Section 9.3, the presence of loops in $R_{\texttt{While}}$ means that the control-flow graphs may contain cycles, which complicates the liveness analysis needed for register allocation.
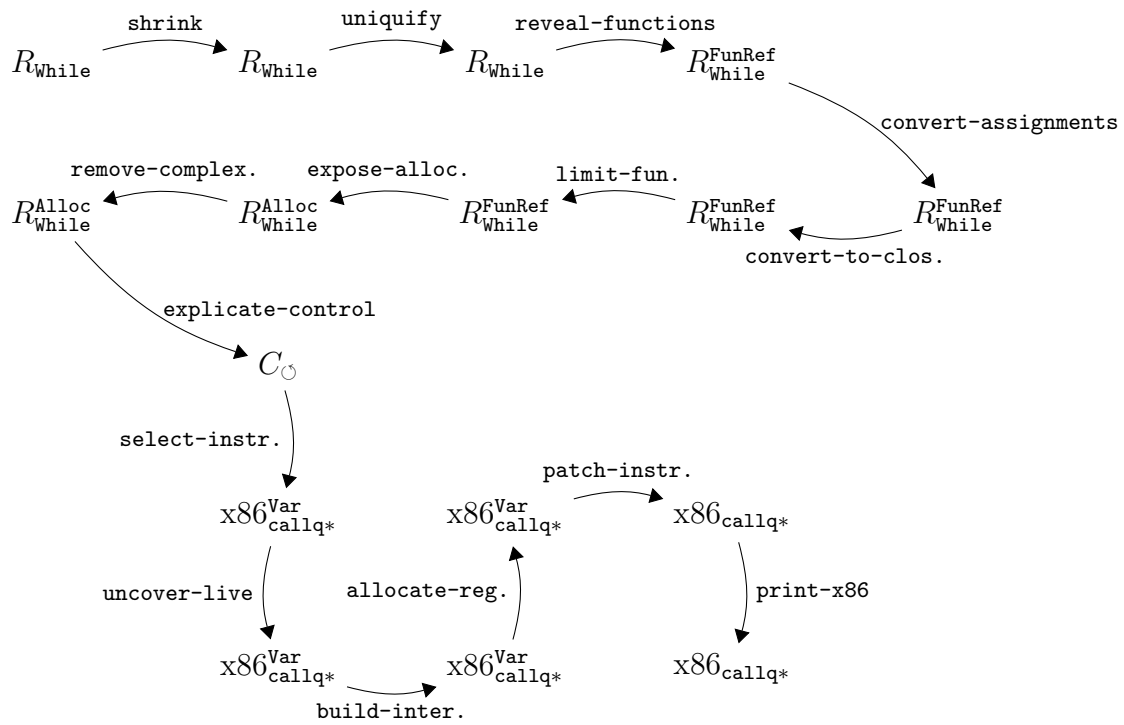
9.8.1   Liveness Analysis

We recommend using the generic `analyze-dataflow` function that was presented at the end of Section 9.3 to perform liveness analysis, replacing the code in `uncover-live-CFG` that processed the basic blocks in topological order (Section 4.10.1).

The `analyze-dataflow` function has four parameters.

1. The first parameter `G` should be a directed graph from the `racket/graph` package (see the sidebar in Section 3.3) that represents the control-flow graph.

2. The second parameter `transfer` is a function that applies liveness analysis to a basic block. It takes two parameters: the label for the block to analyze and the live-after set for that block. The transfer function should return the live-before set for the block. Also, as a side-effect, it should update the block's *info* with the liveness information for each instruction. To implement the `transfer` function, you should be able to reuse the code you already have for analyzing basic blocks.

3. The third and fourth parameters of `analyze-dataflow` are `bottom` and `join` for the lattice of abstract states, i.e. sets of locations. The bottom of the lattice is the empty set `(set)` and the join operator is `set-union`.

Figure 9.8 provides an overview of all the passes needed for the compilation of $R_{\texttt{While}}$.

$R_{\texttt{While}} \xrightarrow{\text{shrink}} R_{\texttt{While}} \xrightarrow{\text{uniquify}} R_{\texttt{While}} \xrightarrow{\text{reveal-functions}} R_{\texttt{While}}^{\texttt{FunRef}}$

convert-assignments

$R_{\texttt{While}}^{\texttt{Alloc}} \xleftarrow{\text{remove-complex.}} R_{\texttt{While}}^{\texttt{Alloc}} \xleftarrow{\text{expose-alloc.}} R_{\texttt{While}}^{\texttt{FunRef}} \xleftarrow{\text{limit-fun.}} R_{\texttt{While}}^{\texttt{FunRef}} \xleftarrow{\text{convert-to-clos.}} R_{\texttt{While}}^{\texttt{FunRef}}$

explicate-control

$C_{\circlearrowleft}$

select-instr.

$\text{x86}_{\texttt{callq*}}^{\texttt{Var}}$

uncover-live

$\text{x86}_{\texttt{callq*}}^{\texttt{Var}} \xrightarrow{\text{patch-instr.}} \text{x86}_{\texttt{callq*}}$

allocate-reg.

print-x86

$\text{x86}_{\texttt{callq*}}^{\texttt{Var}} \xrightarrow{\text{build-inter.}} \text{x86}_{\texttt{callq*}}^{\texttt{Var}}$

$\text{x86}_{\texttt{callq*}}$

Figure 9.8: Diagram of the passes for $R_{\texttt{While}}$ (loops and assignment).

10

Gradual Typing

This chapter studies a language, $R_?$, in which the programmer can choose between static and dynamic type checking in different parts of a program, thereby mixing the statically typed $R_{\texttt{While}}$ language with the dynamically typed $R_{\texttt{Dyn}}$. There are several approaches to mixing static and dynamic typing, including multi-language integration [97, 71] and hybrid type checking [36, 47]. In this chapter we focus on *gradual typing*, in which the programmer controls the amount of static versus dynamic checking by adding or removing type annotations on parameters and variables [5, 90]. The concrete syntax of $R_?$ is defined in Figure 10.1 and its abstract syntax is defined in Figure 10.2. The main syntactic difference between $R_{\texttt{While}}$ and $R_?$ is the additional *param* and *ret* non-terminals that make type annotations optional. The return types are not optional in the abstract syntax; the parser fills in `Any` when the return type is not specified in the concrete syntax.

Both the type checker and the interpreter for $R_?$ require some interesting changes to enable gradual typing, which we discuss in the next two sections in the context of the `map-vec` example from Chapter 6. In Figure 10.3 we revised the `map-vec` example, omitting the type annotations from the `add1` function.

10.1 Type Checking $R_?$, Casts, and $R_{\texttt{cast}}$

The type checker for $R_?$ uses the `Any` type for missing parameter and return types. For example, the `x` parameter of `add1` in Figure 10.3 is given the type `Any` and the return type of `add1` is `Any`. Next consider the `+` operator inside `add1`. It expects both arguments to have type `Integer`, but its first argument `x` has type `Any`. In a gradually typed language, such differences

189

$$
\begin{array}{lll}
param & ::= & var \mid [var\!:\!type] \\
ret & ::= & \epsilon \mid :type \\
exp & ::= & int \mid (\text{read}) \mid (\text{-}\ exp) \mid (\text{+}\ exp\ exp) \mid (\text{-}\ exp\ exp) \\
& \mid & var \mid (\text{let } ([var\ exp])\ exp) \\
& \mid & \text{\#t} \mid \text{\#f} \mid (\text{and}\ exp\ exp) \mid (\text{or}\ exp\ exp) \mid (\text{not}\ exp) \\
& \mid & (\text{eq?}\ exp\ exp) \mid (\text{if}\ exp\ exp\ exp) \\
& \mid & (\text{vector}\ exp\ldots) \mid (\text{vector-ref}\ exp\ int) \\
& \mid & (\text{vector-set!}\ exp\ int\ exp) \mid (\text{void}) \mid (exp\ exp\ldots) \\
& \mid & (\text{procedure-arity}\ exp) \mid (\textbf{lambda:}\ (param\ldots)\ ret\ exp) \\
& \mid & (\text{set!}\ var\ exp) \mid (\text{begin}\ exp\ldots\ exp) \mid (\text{while}\ exp\ exp) \\
def & ::= & (\textbf{define}\ (var\ param\ldots)\ ret\ exp) \\
R_? & ::= & def\ldots\ exp
\end{array}
$$

Figure 10.1: The concrete syntax of $R_?$, extending $R_{\texttt{While}}$ (Figure 9.1).

$$
\begin{array}{lll}
param & ::= & var \mid [var\!:\!type] \\
exp & ::= & (\text{Int}\ int)(\text{Var}\ var) \mid (\text{Let}\ var\ exp\ exp) \\
& \mid & (\text{Prim}\ op\ (exp\ldots)) \\
& \mid & (\text{Bool}\ bool) \mid (\text{If}\ exp\ exp\ exp) \\
& \mid & (\text{Void}) \mid (\text{HasType}\ exp\ type) \mid (\text{Apply}\ exp\ exp\ldots) \\
& \mid & (\textbf{Lambda}\ (param\ldots)\ type\ exp) \\
& \mid & (\text{SetBang}\ var\ exp) \mid (\text{Begin}\ (exp\ldots)\ exp) \\
& \mid & (\text{WhileLoop}\ exp\ exp) \\
def & ::= & (\textbf{Def}\ var\ (param\ldots)\ type\ \text{'()}\ exp) \\
R_? & ::= & (\text{ProgramDefsExp '()}\ (def\ldots)\ exp)
\end{array}
$$

Figure 10.2: The abstract syntax of $R_?$, extending $R_{\texttt{While}}$ (Figure 9.2).

```
(define (map-vec [f : (Integer -> Integer)]
                 [v : (Vector Integer Integer)])
                : (Vector Integer Integer)
  (vector (f (vector-ref v 0)) (f (vector-ref v 1)))))

(define (add1 x) (+ x 1))

(vector-ref (map-vec add1 (vector 0 41)) 1)
```

Figure 10.3: A partially-typed version of the map-vec example.

```
(define/public (consistent? t1 t2)
  (match* (t1 t2)
    [('Integer 'Integer) #t]
    [('Boolean 'Boolean) #t]
    [('Void 'Void) #t]
    [('Any t2) #t]
    [(t1 'Any) #t]
    [(`(Vector ,ts1 ...) `(Vector ,ts2 ...))
     (for/and ([t1 ts1] [t2 ts2]) (consistent? t1 t2))]
    [(`(,ts1 ... -> ,rt1) `(,ts2 ... -> ,rt2))
     (and (for/and ([t1 ts1] [t2 ts2]) (consistent? t1 t2))
          (consistent? rt1 rt2))]
    [(other wise) #f]))
```

Figure 10.4: The consistency predicate on types, a method in `type-check-gradual-class`.

are allowed so long as the types are *consistent*, that is, they are equal except in places where there is an `Any` type. The type `Any` is consistent with every other type. Figure 10.4 defines the `consistent?` predicate.

Returning to the `map-vec` example of Figure 10.3, the `add1` function has type (`Any -> Any`) but parameter `f` of `map-vec` has type (`Integer -> Integer`). The type checker for $R_?$ allows this because the two types are consistent. In particular, `->` is equal to `->` and because `Any` is consistent with `Integer`.

Next consider a program with an error, such as applying the `map-vec` to a function that sometimes returns a Boolean, as shown in Figure 10.6. The type checker for $R_?$ accepts this program because the type of `maybe-add1` is consistent with the type of parameter `f` of `map-vec`, that is, (`Any -> Any`) is consistent with (`Integer -> Integer`). One might say that a gradual type checker is optimistic in that it accepts programs that might execute without a runtime type error. Unfortunately, running this program with input `1` triggers an error when the `maybe-add1` function returns `#t`. $R_?$ performs checking at runtime to ensure the integrity of the static types, such as the (`Integer -> Integer`) annotation on parameter `f` of `map-vec`. This runtime checking is carried out by a new `Cast` form that is inserted by the type checker. Thus, the output of the type checker is a program in the $R_{\texttt{cast}}$ language, which adds `Cast` to $R_{\texttt{While}}$, as shown in Figure 10.5.

Figure 10.7 shows the output of the type checker for `map-vec` and `maybe-add1`. The idea is that `Cast` is inserted every time the type checker sees two types

$$
\begin{array}{lll}
exp & ::= & \dots \mid (\texttt{Cast}\ exp\ type\ type) \\
R_{\mathsf{cast}} & ::= & (\texttt{ProgramDefsExp '() } (def \dots)\ exp)
\end{array}
$$

Figure 10.5: The abstract syntax of R_{cast}, extending $R_{\mathtt{While}}$ (Figure 9.2).

```
(define (map-vec [f : (Integer -> Integer)]
                 [v : (Vector Integer Integer)])
                 : (Vector Integer Integer)
  (vector (f (vector-ref v 0)) (f (vector-ref v 1))))
(define (add1 x) (+ x 1))
(define (true) #t)
(define (maybe-add1 x) (if (eq? 0 (read)) (add1 x) (true)))

(vector-ref (map-vec maybe-add1 (vector 0 41)) 0)
```

Figure 10.6: A variant of the `map-vec` example with an error.

that are consistent but not equal. In the `add1` function, `x` is cast to `Integer` and the result of the `+` is cast to `Any`. In the call to `map-vec`, the `add1` argument is cast from `(Any -> Any)` to `(Integer -> Integer)`.

The type checker for $R_?$ is defined in Figures 10.8, 10.9, and 10.10.

```
(define (map-vec [f : (Integer -> Integer)] [v : (Vector Integer Integer)])
                 : (Vector Integer Integer)
  (vector (f (vector-ref v 0)) (f (vector-ref v 1))))
(define (add1 [x : Any]) : Any
  (cast (+ (cast x Any Integer) 1) Integer Any))
(define (true) : Any (cast #t Boolean Any))
(define (maybe-add1 [x : Any]) : Any
  (if (eq? 0 (read)) (add1 x) (true)))

(vector-ref (map-vec (cast maybe-add1 (Any -> Any) (Integer -> Integer))
                     (vector 0 41)) 0)
```

Figure 10.7: Output of type checking `map-vec` and `maybe-add1`.

```
(define type-check-gradual-class
  (class type-check-Rwhile-class
    (super-new)
    (inherit operator-types type-predicates)

    (define/override (type-check-exp env)
      (lambda (e)
        (define recur (type-check-exp env))
        (match e
          [(Prim 'vector-length (list e1))
           (define-values (e1^ t) (recur e1))
           (match t
             [`(Vector ,ts ...)
              (values (Prim 'vector-length (list e1^)) 'Integer)]
             ['Any (values (Prim 'any-vector-length (list e1^)) 'Integer)])]
          [(Prim 'vector-ref (list e1 e2))
           (define-values (e1^ t1) (recur e1))
           (define-values (e2^ t2) (recur e2))
           (check-consistent? t2 'Integer e)
           (match t1
             [`(Vector ,ts ...)
              (match e2^
                [(Int i)
                 (unless (and (0 . <= . i) (i . < . (length ts)))
                   (error 'type-check "invalid index ~a in ~a" i e))
                 (values (Prim 'vector-ref (list e1^ (Int i))) (list-ref ts i))]
                [else (define e1^^ (make-cast e1^ t1 'Any))
                      (define e2^^ (make-cast e2^ t2 'Integer))
                      (values (Prim 'any-vector-ref (list e1^^ e2^^)) 'Any)])]
             ['Any
              (define e2^^ (make-cast e2^ t2 'Integer))
              (values (Prim 'any-vector-ref (list e1^ e2^^)) 'Any)]
             [else (error 'type-check "expected vector not ~a\nin ~v" t1 e)])]
          [(Prim 'vector-set! (list e1 e2 e3) )
           (define-values (e1^ t1) (recur e1))
           (define-values (e2^ t2) (recur e2))
           (define-values (e3^ t3) (recur e3))
           (check-consistent? t2 'Integer e)
           (match t1
             [`(Vector ,ts ...)
              (match e2^
                [(Int i)
                 (unless (and (0 . <= . i) (i . < . (length ts)))
                   (error 'type-check "invalid index ~a in ~a" i e))
                 (check-consistent? (list-ref ts i) t3 e)
                 (define e3^^ (make-cast e3^ t3 (list-ref ts i)))
                 (values (Prim 'vector-set! (list e1^ (Int i) e3^^)) 'Void)]
                [else
                 (define e1^^ (make-cast e1^ t1 'Any))
                 (define e2^^ (make-cast e2^ t2 'Integer))
                 (define e3^^ (make-cast e3^ t3 'Any))
                 (values (Prim 'any-vector-set! (list e1^^ e2^^ e3^^)) 'Void)])]
             ['Any
              (define e2^^ (make-cast e2^ t2 'Integer))
              (define e3^^ (make-cast e3^ t3 'Any))
              (values (Prim 'any-vector-set! (list e1^ e2^^ e3^^)) 'Void)]
             [else (error 'type-check "expected vector not ~a\nin ~v" t1 e)])]
```

Figure 10.8: Type checker for the $R_?$ language, part 1.

```
[(Prim 'eq? (list e1 e2))
 (define-values (e1^ t1) (recur e1))
 (define-values (e2^ t2) (recur e2))
 (check-consistent? t1 t2 e)
 (define T (meet t1 t2))
 (values (Prim 'eq? (list (make-cast e1^ t1 T) (make-cast e2^ t2 T)))
         'Boolean)]
[(Prim 'not (list e1))
 (define-values (e1^ t1) (recur e1))
 (match t1
   ['Any
    (recur (If (Prim 'eq? (list e1 (Inject (Bool #f) 'Boolean)))
               (Bool #t) (Bool #f)))]
   [else
    (define-values (t-ret new-es^)
      (type-check-op 'not (list t1) (list e1^) e))
    (values (Prim 'not new-es^) t-ret)])]
[(Prim 'and (list e1 e2))
 (recur (If e1 e2 (Bool #f)))]
[(Prim 'or (list e1 e2))
 (define tmp (gensym 'tmp))
 (recur (Let tmp e1 (If (Var tmp) (Var tmp) e2)))]
[(Prim op es)
 #:when (not (set-member? explicit-prim-ops op))
 (define-values (new-es ts)
   (for/lists (exprs types) ([e es])
     (recur e)))
 (define-values (t-ret new-es^) (type-check-op op ts new-es e))
 (values (Prim op new-es^) t-ret)]
[(If e1 e2 e3)
 (define-values (e1^ T1) (recur e1))
 (define-values (e2^ T2) (recur e2))
 (define-values (e3^ T3) (recur e3))
 (check-consistent? T2 T3 e)
 (match T1
   ['Boolean
    (define Tif (join T2 T3))
    (values (If e1^ (make-cast e2^ T2 Tif)
                (make-cast e3^ T3 Tif)) Tif)]
   ['Any
    (define Tif (meet T2 T3))
    (values (If (Prim 'eq? (list e1^ (Inject (Bool #f) 'Boolean)))
                (make-cast e3^ T3 Tif) (make-cast e2^ T2 Tif))
            Tif)]
   [else (error 'type-check "expected Boolean not ~a\nin ~v" T1 e)])]
[(HasType e1 T)
 (define-values (e1^ T1) (recur e1))
 (check-consistent? T1 T)
 (values (make-cast e1^ T1 T) T)]
[(SetBang x e1)
 (define-values (e1^ T1) (recur e1))
 (define varT (dict-ref env x))
 (check-consistent? T1 varT e)
 (values (SetBang x (make-cast e1^ T1 varT)) 'Void)]
[(WhileLoop e1 e2)
 (define-values (e1^ T1) (recur e1))
 (check-consistent? T1 'Boolean e)
 (define-values (e2^ T2) ((type-check-exp env) e2))
 (values (WhileLoop (make-cast e1^ T1 'Boolean) e2^) 'Void)]
```

Figure 10.9: Type checker for the $R_?$ language, part 2.

```
[(Apply e1 e2s)
 (define-values (e1^ T1) (recur e1))
 (define-values (e2s^ T2s) (for/lists (e* ty*) ([e2 e2s]) (recur e2)))
 (match T1
   [`(,T1ps ... -> ,T1rt)
    (for ([T2 T2s] [Tp T1ps])
      (check-consistent? T2 Tp e))
    (define e2s^^ (for/list ([e2 e2s^] [src T2s] [tgt T1ps])
                   (make-cast e2 src tgt)))
    (values (Apply e1^ e2s^^) T1rt)]
   [`Any
    (define e1^^ (make-cast e1^ 'Any
                            `(,@(for/list ([e e2s]) 'Any) -> Any)))
    (define e2s^^ (for/list ([e2 e2s^] [src T2s])
                   (make-cast e2 src 'Any)))
    (values (Apply e1^^ e2s^^) 'Any)]
   [else (error 'type-check "expected function not ~a\in ~v" T1 e)])])
[(Lambda params Tr e1)
 (define-values (xs Ts) (for/lists (l1 l2) ([p params])
                         (match p
                           [`[,x : ,T] (values x T)]
                           [(? symbol? x) (values x 'Any)])))
 (define-values (e1^ T1)
   ((type-check-exp (append (map cons xs Ts) env)) e1))
 (check-consistent? Tr T1 e)
 (values (Lambda (for/list ([x xs] [T Ts]) `[,x : ,T]) Tr
           (make-cast e1^ T1 Tr)) `(,@Ts -> ,Tr))]
[else ((super type-check-exp env) e)]
)))
```

Figure 10.10: Type checker for the $R_?$ language, part 3.

```
(define/public (join t1 t2)
  (match* (t1 t2)
    [('Integer 'Integer) 'Integer]
    [('Boolean 'Boolean) 'Boolean]
    [('Void 'Void) 'Void]
    [('Any t2) t2]
    [(t1 'Any) t1]
    [(`(Vector ,ts1 ...) `(Vector ,ts2 ...))
     `(Vector ,@(for/list ([t1 ts1] [t2 ts2]) (join t1 t2)))]
    [(`(,ts1 ... -> ,rt1) `(,ts2 ... -> ,rt2))
     `(,@(for/list ([t1 ts1] [t2 ts2]) (join t1 t2))
        -> ,(join rt1 rt2))]))

(define/public (meet t1 t2)
  (match* (t1 t2)
    [('Integer 'Integer) 'Integer]
    [('Boolean 'Boolean) 'Boolean]
    [('Void 'Void) 'Void]
    [('Any t2) 'Any]
    [(t1 'Any) 'Any]
    [(`(Vector ,ts1 ...) `(Vector ,ts2 ...))
     `(Vector ,@(for/list ([t1 ts1] [t2 ts2]) (meet t1 t2)))]
    [(`(,ts1 ... -> ,rt1) `(,ts2 ... -> ,rt2))
     `(,@(for/list ([t1 ts1] [t2 ts2]) (meet t1 t2))
        -> ,(meet rt1 rt2))]))

(define/public (make-cast e src tgt)
  (cond [(equal? src tgt) e] [else (Cast e src tgt)]))

(define/public (check-consistent? t1 t2 e)
  (unless (consistent? t1 t2)
    (error 'type-check "~a is inconsistent with ~a\nin ~v" t1 t2 e)))

(define/override (type-check-op op arg-types args e)
  (match (dict-ref (operator-types) op)
    [`(,param-types . ,return-type)
     (for ([at arg-types] [pt param-types])
       (check-consistent? at pt e))
     (values return-type
             (for/list ([e args] [s arg-types] [t param-types])
               (make-cast e s t)))]
    [else (error 'type-check-op "unrecognized ~a" op)]))

(define explicit-prim-ops
  (set-union
   (type-predicates)
   (set 'procedure-arity 'eq?
        'vector 'vector-length 'vector-ref 'vector-set!
        'any-vector-length 'any-vector-ref 'any-vector-set!)))

(define/override (fun-def-type d)
  (match d
    [(Def f params rt info body)
     (define ps
       (for/list ([p params])
         (match p
           [`[,x : ,T] T]
           [(? symbol?) 'Any]
           [else (error 'fun-def-type "unmatched parameter ~a" p)])))
     `(,@ps -> ,rt)]
    [else (error 'fun-def-type "ill-formed function definition in ~a" d)]))
```

Figure 10.11: Auxiliary functions for type checking $R_?$.

10.2   Interpreting $R_{\tt cast}$

The runtime behavior of first-order casts is straightforward, that is, casts involving simple types such as `Integer` and `Boolean`. For example, a cast from `Integer` to `Any` can be accomplished with the `Inject` operator of $R_{\tt Any}$, which puts the integer into a tagged value (Figure 8.9). Similarly, a cast from `Any` to `Integer` is accomplished with the `Project` operator, that is, by checking the value's tag and either retrieving the underlying integer or signaling an error if it the tag is not the one for integers (Figure 8.10). Things get more interesting for higher-order casts, that is, casts involving function or vector types.

Consider the cast of the function `maybe-add1` from (`Any -> Any`) to (`Integer -> Integer`). When a function flows through this cast at runtime, we can't know in general whether the function will always return an integer.[1] The $R_{\tt cast}$ interpreter therefore delays the checking of the cast until the function is applied. This is accomplished by wrapping `maybe-add1` in a new function that casts its parameter from `Integer` to `Any`, applies `maybe-add1`, and then casts the return value from `Any` to `Integer`.

Turning our attention to casts involving vector types, we consider the example in Figure 10.12 that defines a partially-typed version of `map-vec` whose parameter v has type (`Vector Any Any`) and that updates v in place instead of returning a new vector. So we name this function `map-vec!`. We apply `map-vec!` to a vector of integers, so the type checker inserts a cast from (`Vector Integer Integer`) to (`Vector Any Any`). A naive way for the $R_{\tt cast}$ interpreter to cast between vector types would be a build a new vector whose elements are the result of casting each of the original elements to the appropriate target type. However, this approach is only valid for immutable vectors; and our vectors are mutable. In the example of Figure 10.12, if the cast created a new vector, then the updates inside of `map-vec!` would happen to the new vector and not the original one.

Instead the interpreter needs to create a new kind of value, a *vector proxy*, that intercepts every vector operation. On a read, the proxy reads from the underlying vector and then applies a cast to the resulting value. On a write, the proxy casts the argument value and then performs the write to the underlying vector. For the first (`vector-ref v 0`) in `map-vec!`, the proxy casts 0 from `Integer` to `Any`. For the first `vector-set!`, the proxy casts a tagged 1 from `Any` to `Integer`.

---

[1]Predicting the return value of a function is equivalent to the halting problem, which is undecidable.

```
(define (map-vec! [f : (Any -> Any)]
                  [v : (Vector Any Any)]) : Void
  (begin
    (vector-set! v 0 (f (vector-ref v 0)))
    (vector-set! v 1 (f (vector-ref v 1)))))

(define (add1 x) (+ x 1))

(let ([v (vector 0 41)])
  (begin (map-vec! add1 v) (vector-ref v 1)))
```

Figure 10.12: An example involving casts on vectors.

```
(define (map-vec! [f : (Any -> Any)] v) : Void
  (begin
    (vector-set! v 0 (f (vector-ref v 0)))
    (vector-set! v 1 (f (vector-ref v 1)))))

(define (add1 x) (+ x 1))

(let ([v (vector 0 41)])
  (begin (map-vec! add1 v) (vector-ref v 1)))
```

Figure 10.13: Casting a vector to `Any`.

The final category of cast that we need to consider are casts between the `Any` type and either a function or a vector type. Figure 10.13 shows a variant of `map-vec!` in which parameter `v` does not have a type annotation, so it is given type `Any`. In the call to `map-vec!`, the vector has type `(Vector Integer Integer)` so the type checker inserts a cast from `(Vector Integer Integer)` to `Any`. A first thought is to use `Inject`, but that doesn't work because `(Vector Integer Integer)` is not a flat type. Instead, we must first cast to `(Vector Any Any)` (which is flat) and then inject to `Any`.

The R_{cast} interpreter uses an auxiliary function named `apply-cast` to cast a value from a source type to a target type, shown in Figure 10.14. You'll find that it handles all of the kinds of casts that we've discussed in this section.

The interpreter for R_{cast} is defined in Figure 10.15, with the case for `Cast` dispatching to `apply-cast`. To handle the addition of vector proxies, we update the vector primitives in `interp-op` using the functions in

```
(define/public (apply-cast v s t)
  (match* (s t)
    [(t1 t2) #:when (equal? t1 t2) v]
    [('Any t2)
     (match t2
       [`(,ts ... -> ,rt)
        (define any->any `(,@(for/list ([t ts]) 'Any) -> Any))
        (define v^ (apply-project v any->any))
        (apply-cast v^ any->any `(,@ts -> ,rt))]
       [`(Vector ,ts ...)
        (define vec-any `(Vector ,@(for/list ([t ts]) 'Any)))
        (define v^ (apply-project v vec-any))
        (apply-cast v^ vec-any `(Vector ,@ts))]
       [else (apply-project v t2)])]
    [(t1 'Any)
     (match t1
       [`(,ts ... -> ,rt)
        (define any->any `(,@(for/list ([t ts]) 'Any) -> Any))
        (define v^ (apply-cast v `(,@ts -> ,rt) any->any))
        (apply-inject v^ (any-tag any->any))]
       [`(Vector ,ts ...)
        (define vec-any `(Vector ,@(for/list ([t ts]) 'Any)))
        (define v^ (apply-cast v `(Vector ,@ts) vec-any))
        (apply-inject v^ (any-tag vec-any))]
       [else (apply-inject v (any-tag t1))])]
    [(`(Vector ,ts1 ...) `(Vector ,ts2 ...))
     (define x (gensym 'x))
     (define cast-reads (for/list ([t1 ts1] [t2 ts2])
                          `(function (,x) ,(Cast (Var x) t1 t2) ()))))
     (define cast-writes
       (for/list ([t1 ts1] [t2 ts2])
         `(function (,x) ,(Cast (Var x) t2 t1) ()))))
     `(vector-proxy ,(vector v (apply vector cast-reads)
                             (apply vector cast-writes)))]
    [(`(,ts1 ... -> ,rt1) `(,ts2 ... -> ,rt2))
     (define xs (for/list ([t2 ts2]) (gensym 'x)))
     `(function ,xs ,(Cast
                       (Apply (Value v)
                              (for/list ([x xs][t1 ts1][t2 ts2])
                                (Cast (Var x) t2 t1)))
                       rt1 rt2) ()))]
  ))
```

Figure 10.14: The `apply-cast` auxiliary method.

Figure 10.16.

10.3   Lower Casts

The next step in the journey towards x86 is the `lower-casts` pass that
translates the casts in $R_{\tt cast}$ to the lower-level `Inject` and `Project` oper-
ators and a new operator for creating vector proxies, extending the $R_{\tt While}$
language to create $R_{\tt proxy}$. We recommend creating an auxiliary function
named `lower-cast` that takes an expression (in $R_{\tt cast}$), a source type, and
a target type, and translates it to expression in $R_{\tt proxy}$ that has the same
behavior as casting the expression from the source to the target type in the
interpreter.

The `lower-cast` function can follow a code structure similar to the
`apply-cast` function (Figure 10.14) used in the interpreter for $R_{\tt cast}$ be-
cause it must handle the same cases as `apply-cast` and it needs to mimic
the behavior of `apply-cast`. The most interesting cases are those concern-
ing the casts between two vector types and between two function types.

As mentioned in Section 10.2, a cast from one vector type to another vec-
tor type is accomplished by creating a proxy that intercepts the operations
on the underlying vector. Here we make the creation of the proxy explicit
with the `vector-proxy` primitive operation. It takes three arguments, the
first is an expression for the vector, the second is a vector of functions for
casting an element that is being read from the vector, and the third is a
vector of functions for casting an element that is being written to the vec-
tor. You can create the functions using `Lambda`. Also, as we shall see in the
next section, we need to differentiate these vectors from the user-created
ones, so we recommend using a new primitive operator named `raw-vector`
instead of `vector` to create these vectors of functions. Figure 10.17 shows
the output of `lower-casts` on the example in Figure 10.12 that involved
casting a vector of integers to a vector of `Any`.

A cast from one function type to another function type is accomplished
by generating a `Lambda` whose parameter and return types match the target
function type. The body of the `Lambda` should cast the parameters from the
target type to the source type (yes, backwards! functions are contravariant in
the parameters), then call the underlying function, and finally cast the result
from the source return type to the target return type. Figure 10.18 shows
the output of the `lower-casts` pass on the `map-vec` example in Figure 10.3.
Note that the `add1` argument in the call to `map-vec` is wrapped in a `lambda`.

```
(define interp-Rcast-class
  (class interp-Rwhile-class
    (super-new)
    (inherit apply-fun apply-inject apply-project)

    (define/override (interp-op op)
      (match op
        ['vector-length guarded-vector-length]
        ['vector-ref guarded-vector-ref]
        ['vector-set! guarded-vector-set!]
        ['any-vector-ref (lambda (v i)
                           (match v [`(tagged ,v^ ,tg)
                                     (guarded-vector-ref v^ i)]))]
        ['any-vector-set! (lambda (v i a)
                            (match v [`(tagged ,v^ ,tg)
                                      (guarded-vector-set! v^ i a)]))]
        ['any-vector-length (lambda (v)
                               (match v [`(tagged ,v^ ,tg)
                                         (guarded-vector-length v^)]))]
        [else (super interp-op op)]
        ))

    (define/override ((interp-exp env) e)
      (define (recur e) ((interp-exp env) e))
      (match e
        [(Value v) v]
        [(Cast e src tgt) (apply-cast (recur e) src tgt)]
        [else ((super interp-exp env) e)]))
  ))

(define (interp-Rcast p)
  (send (new interp-Rcast-class) interp-program p))
```

Figure 10.15: The interpreter for R_{cast}.

```
(define (guarded-vector-ref vec i)
  (match vec
    [`(vector-proxy ,proxy)
     (define val (guarded-vector-ref (vector-ref proxy 0) i))
     (define rd (vector-ref (vector-ref proxy 1) i))
     (apply-fun rd (list val) 'guarded-vector-ref)]
    [else (vector-ref vec i)]))

(define (guarded-vector-set! vec i arg)
  (match vec
    [`(vector-proxy ,proxy)
     (define wr (vector-ref (vector-ref proxy 2) i))
     (define arg^ (apply-fun wr (list arg) 'guarded-vector-set!))
     (guarded-vector-set! (vector-ref proxy 0) i arg^)]
    [else (vector-set! vec i arg)]))

(define (guarded-vector-length vec)
  (match vec
    [`(vector-proxy ,proxy)
     (guarded-vector-length (vector-ref proxy 0))]
    [else (vector-length vec)]))
```

Figure 10.16: The guarded-vector auxiliary functions.

10.4   Differentiate Proxies

So far the job of differentiating vectors and vector proxies has been the
job of the interpreter. For example, the interpreter for R_{cast} implements
`vector-ref` using the `guarded-vector-ref` function in Figure 10.16. In the
`differentiate-proxies` pass we shift this responsibility to the generated
code.

We begin by designing the output language R_8^p. In $R_?$ we used the type
`Vector` for both real vectors and vector proxies. In R_8^p we return the `Vector`
type to its original meaning, as the type of real vectors, and we introduce a
new type, `PVector`, whose values can be either real vectors or vector proxies.
This new type comes with a suite of new primitive operations for creating
and using values of type `PVector`. We don't need to introduce a new type to
represent vector proxies. A proxy is represented by a vector containing three
things: 1) the underlying vector, 2) a vector of functions for casting elements
that are read from the vector, and 3) a vector of functions for casting values
to be written to the vector. So we define the following abbreviation for the

```
(define (map-vec! [f : (Any -> Any)] [v : (Vector Any Any)]) : Void
  (begin
    (vector-set! v 0 (f (vector-ref v 0)))
    (vector-set! v 1 (f (vector-ref v 1)))))

(define (add1 [x : Any]) : Any
  (inject (+ (project x Integer) 1) Integer))

(let ([v (vector 0 41)])
  (begin
    (map-vec! add1 (vector-proxy v
                      (raw-vector (lambda: ([x9 : Integer]) : Any
                                    (inject x9 Integer))
                                  (lambda: ([x9 : Integer]) : Any
                                    (inject x9 Integer)))
                      (raw-vector (lambda: ([x9 : Any]) : Integer
                                    (project x9 Integer))
                                  (lambda: ([x9 : Any]) : Integer
                                    (project x9 Integer)))))
    (vector-ref v 1)))
```

Figure 10.17: Output of `lower-casts` on the example in Figure 10.12.

```
(define (map-vec [f : (Integer -> Integer)]
                 [v : (Vector Integer Integer)])
                 : (Vector Integer Integer)
  (vector (f (vector-ref v 0)) (f (vector-ref v 1))))

(define (add1 [x : Any]) : Any
  (inject (+ (project x Integer) 1) Integer))

(vector-ref (map-vec (lambda: ([x9 : Integer]) : Integer
                       (project (add1 (inject x9 Integer)) Integer))
                     (vector 0 41)) 1)
```

Figure 10.18: Output of `lower-casts` on the example in Figure 10.3.

type of a vector proxy:

$$Proxy(T \ldots \Rightarrow T' \ldots) = (\texttt{Vector } (\texttt{PVector } T \ldots) \ R \ W) \rightarrow (\texttt{PVector } T' \ldots)$$

where $R = (\texttt{Vector } (T \rightarrow T') \ldots)$ and $W = (\texttt{Vector } (T' \rightarrow T) \ldots)$. Next we describe each of the new primitive operations.

inject-vector : $(\texttt{Vector } T \ldots) \rightarrow (\texttt{PVector } T \ldots)$
>   This operation brands a vector as a value of the **PVector** type.

inject-proxy : $Proxy(T \ldots \Rightarrow T' \ldots) \rightarrow (\texttt{PVector } T' \ldots)$
>   This operation brands a vector proxy as value of the **PVector** type.

proxy? : $(\texttt{PVector } T \ldots) \rightarrow \texttt{Boolean}$
>   returns true if the value is a vector proxy and false if it is a real vector.

project-vector : $(\texttt{PVector } T \ldots) \rightarrow (\texttt{Vector } T \ldots)$
>   Assuming that the input is a vector (and not a proxy), this operation returns the vector.

proxy-vector-length : $(\texttt{PVector } T \ldots) \rightarrow \texttt{Boolean}$
>   Given a vector proxy, this operation returns the length of the underlying vector.

proxy-vector-ref : $(\texttt{PVector } T \ldots) \rightarrow (i : \texttt{Integer}) \rightarrow T_i$
>   Given a vector proxy, this operation returns the ith element of the underlying vector.

proxy-vector-set! : $(\texttt{PVector } T \ldots) \rightarrow (i : \texttt{Integer}) \rightarrow T_i \rightarrow \texttt{Void}$
>   Given a vector proxy, this operation writes a value to the ith element of the underlying vector.

Now to discuss the translation that differentiates vectors from proxies. First, every type annotation in the program must be translated (recursively) to replace **Vector** with **PVector**. Next, we must insert uses of **PVector** operations in the appropriate places. For example, we wrap every vector creation with an **inject-vector**.

```
(vector  e_1 … e_n)
⇒
(inject-vector (vector  e'_1 … e'_n))
```

The **raw-vector** operator that we introduced in the previous section does not get injected.

```
(raw-vector e₁...eₙ)
```
$\Rightarrow$
```
(vector e'₁...e'ₙ)
```

The `vector-proxy` primitive translates as follows.

```
(vector-proxy e₁ e₂ e₃)
```
$\Rightarrow$
```
(inject-proxy (vector e'₁ e'₂ e'₃))
```

We translate the vector operations into conditional expressions that check whether the value is a proxy and then dispatch to either the appropriate proxy vector operation or the regular vector operation. For example, the following is the translation for `vector-ref`.

```
(vector-ref e₁ i)
```
$\Rightarrow$
```
(let ([v e₁])
  (if (proxy? v)
    (proxy-vector-ref v i)
    (vector-ref (project-vector v) i)
```

Note in the case of a real vector, we must apply `project-vector` before the `vector-ref`.

10.5 Reveal Casts

Recall that the `reveal-casts` pass (Section 8.4) is responsible for lowering `Inject` and `Project` into lower-level operations. In particular, `Project` turns into a conditional expression that inspects the tag and retrieves the underlying value. Here we need to augment the translation of `Project` to handle the situation when the target type is `PVector`. Instead of using `vector-length` we need to use `proxy-vector-length`.

```
(project e (PVector Any₁ ... Anyₙ))
```
$\Rightarrow$
```
(let tmp e'
  (if (eq? (tag-of-any tmp 2))
    (let vec (value-of tmp (PVector Any ... Any))
      (if (eq? (proxy-vector-length vec) n) vec (exit)))
    (exit)))
```

10.6   Closure Conversion

The closure conversion pass only requires one minor adjustment. The auxiliary function that translates type annotations needs to be updated to handle the `PVector` type.

10.7   Explicate Control

Update the `explicate-control` pass to handle the new primitive operations on the `PVector` type.

10.8   Select Instructions

Recall that the `select-instructions` pass is responsible for lowering the primitive operations into x86 instructions. So we need to translate the new `PVector` operations to x86. To do so, the first question we need to answer is how will we differentiate the two kinds of values (vectors and proxies) that can inhabit `PVector`. We need just one bit to accomplish this, so we use the 57th bit of the 64-bit tag at the front of every vector (see Figure 5.9). So far, this bit has been set to 0, so for `inject-vector` we leave it that way.

```
(Assign lhs (Prim 'inject-vector (list e_1)))
```
$\Rightarrow$
```
movq e'_1, lhs'
```

On the other hand, `inject-proxy` sets the 57th bit to 1.

```
(Assign lhs (Prim 'inject-proxy (list e_1)))
```
$\Rightarrow$
```
movq e'_1, %r11
movq (1 << 57), %rax
orq 0(%r11), %rax
movq %rax, 0(%r11)
movq %r11, lhs'
```

The `proxy?` operation consumes the information so carefully stashed away by `inject-vector` and `inject-proxy`. It isolates the 57th bit to tell whether the value is a real vector or a proxy.

```
(Assign lhs (Prim 'proxy? (list e)))
```
$\Rightarrow$
```
movq e'_1, %r11
movq 0(%r11), %rax
sarq $57, %rax
```

```
andq $1, %rax
movq %rax, lhs′
```

The `project-vector` operation is straightforward to translate, so we leave it up to the reader.

Regarding the `proxy-vector` operations, the runtime provides procedures that implement them (they are recursive functions!) so here we simply need to translate these vector operations into the appropriate function call. For example, here is the translation for `proxy-vector-ref`.

```
(Assign lhs (Prim 'proxy-vector-ref (list e₁ e₂)))
⇒
movq e₁′, %rdi
movq e₂′, %rsi
callq proxy_vector_ref
movq %rax, lhs′
```

We have another batch of vector operations to deal with, those for the `Any` type. Recall that the type checker for $R_?$ generates an `any-vector-ref` when there is a `vector-ref` on something of type `Any`, and similarly for `any-vector-set!` and `any-vector-length` (Figure 10.8). In Section 8.7 we selected instructions for these operations based on the idea that the underlying value was a real vector. But in the current setting, the underlying value is of type `PVector`. So `any-vector-ref` can be translates to pseudo-x86 as follows. We begin by projecting the underlying value out of the tagged value and then call the `proxy_vector_ref` procedure in the runtime.

```
(Assign lhs (Prim 'any-vector-ref (list e₁ e₂)))
movq ¬111, %rdi
andq e₁′, %rdi
movq e₂′, %rsi
callq proxy_vector_ref
movq %rax, lhs′
```

The `any-vector-set!` and `any-vector-length` operators can be translated in a similar way.

Exercise 37. Implement a compiler for the gradually-typed $R_?$ language by extending and adapting your compiler for $R_{\texttt{While}}$. Create 10 new partially-typed test programs. In addition to testing with these new programs, also test your compiler on all the tests for $R_{\texttt{While}}$ and tests for $R_{\texttt{Dyn}}$. Sometimes you may get a type checking error on the $R_{\texttt{Dyn}}$ programs but you can adapt them by inserting a cast to the `Any` type around each subexpression causing a type error. While $R_{\texttt{Dyn}}$ doesn't have explicit casts, you can induce one

by wrapping the subexpression `e` with a call to an un-annotated identity function, like this: `((lambda (x) x) e)`.

Figure 10.19 provides an overview of all the passes needed for the compilation of $R_?$.

10.9   Further Reading

This chapter just scratches the surface of gradual typing. The basic approach described here is missing two key ingredients that one would want in a implementation of gradual typing: blame tracking [97, 99] and space-efficient casts [51, 52]. The problem addressed by blame tracking is that when a cast on a higher-order value fails, it often does so at a point in the program that is far removed from the original cast. Blame tracking is a technique for propagating extra information through casts and proxies so that when a cast fails, the error message can point back to the original location of the cast in the source program.

The problem addressed by space-efficient casts also relates to higher-order casts. It turns out that in partially typed programs, a function or vector can flow through very-many casts at runtime. With the approach described in this chapter, each cast adds another `lambda` wrapper or a vector proxy. Not only does this take up considerable space, but it also makes the function calls and vector operations slow. For example, a partially-typed version of quicksort could, in the worst case, build a chain of proxies of length $O(n)$ around the vector, changing the overall time complexity of the algorithm from $O(n^2)$ to $O(n^3)$! Herman et al. [51] suggested a solution to this problem by representing casts using the coercion calculus of Henglein [50], which prevents the creation of long chains of proxies by compressing them into a concise normal form. Siek et al. [91] give and algorithm for compressing coercions and Kuhlenschmidt et al. [66] show how to implement these ideas in the Grift compiler.

<div align="center">

`https://github.com/Gradual-Typing/Grift`

</div>

There are also interesting interactions between gradual typing and other language features, such as parametetric polymorphism, information-flow types, and type inference, to name a few. We recommend the reader to the online gradual typing bibliography:
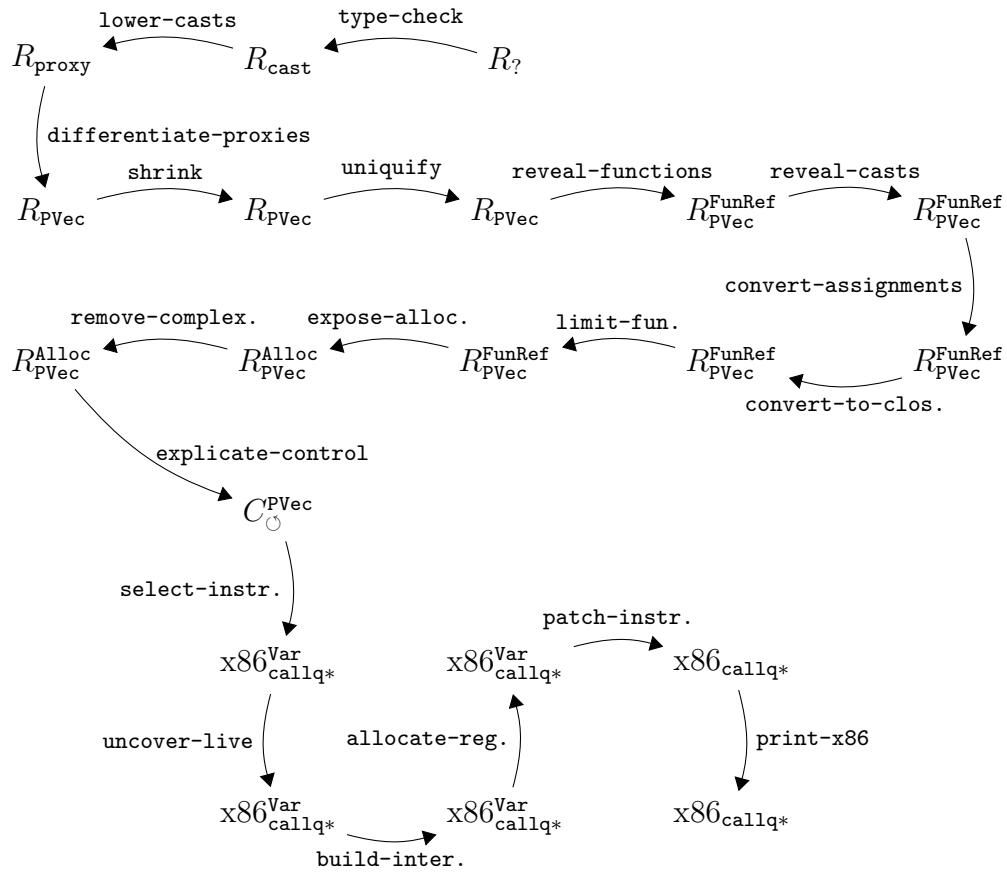
<div align="center">

`http://samth.github.io/gradual-typing-bib/`

</div>

$$R_{\text{proxy}} \xleftarrow{\text{lower-casts}} R_{\text{cast}} \xleftarrow{\text{type-check}} R_?$$

differentiate-proxies

$$R_{\text{PVec}} \xrightarrow{\text{shrink}} R_{\text{PVec}} \xrightarrow{\text{uniquify}} R_{\text{PVec}} \xrightarrow{\text{reveal-functions}} R_{\text{PVec}}^{\text{FunRef}} \xrightarrow{\text{reveal-casts}} R_{\text{PVec}}^{\text{FunRef}}$$

convert-assignments

$$R_{\text{PVec}}^{\text{Alloc}} \xleftarrow{\text{remove-complex.}} R_{\text{PVec}}^{\text{Alloc}} \xleftarrow{\text{expose-alloc.}} R_{\text{PVec}}^{\text{FunRef}} \xleftarrow{\text{limit-fun.}} R_{\text{PVec}}^{\text{FunRef}} \xleftarrow{} R_{\text{PVec}}^{\text{FunRef}}$$

convert-to-clos.

explicate-control

$$C_{\circlearrowleft}^{\text{PVec}}$$

select-instr.

$$\text{x86}_{\text{callq*}}^{\text{Var}} \qquad \text{x86}_{\text{callq*}}^{\text{Var}} \xrightarrow{\text{patch-instr.}} \text{x86}_{\text{callq*}}$$

uncover-live   allocate-reg.   print-x86

$$\text{x86}_{\text{callq*}}^{\text{Var}} \xrightarrow{\text{build-inter.}} \text{x86}_{\text{callq*}}^{\text{Var}} \qquad \text{x86}_{\text{callq*}}$$

Figure 10.19: Diagram of the passes for $R_?$ (gradual typing).

11

Parametric Polymorphism

This chapter studies the compilation of parametric polymorphism (aka. generics) in the subset $R_{\texttt{Poly}}$ of Typed Racket. Parametric polymorphism enables improved code reuse by parameterizing functions and data structures with respect to the types that they operate on. For example, Figure 11.1 revisits the `map-vec` example but this time gives it a more fitting type. This `map-vec` function is parameterized with respect to the element type of the vector. The type of `map-vec` is the following polymorphic type as specified by the `All` and the type parameter `a`.

```
(All (a) ((a -> a) (Vector a a) -> (Vector a a)))
```

The idea is that `map-vec` can be used at *all* choices of a type for parameter `a`. In Figure 11.1 we apply `map-vec` to a vector of integers, a choice of `Integer` for `a`, but we could have just as well applied `map-vec` to a vector of Booleans (and a function on Booleans).

Figure 11.2 defines the concrete syntax of $R_{\texttt{Poly}}$ and Figure 11.3 defines the abstract syntax. We add a second form for function definitions in which a type declaration comes before the `define`. In the abstract syntax, the

```
(: map-vec (All (a) ((a -> a) (Vector a a) -> (Vector a a))))
(define (map-vec f v)
  (vector (f (vector-ref v 0)) (f (vector-ref v 1))))

(define (add1 [x : Integer]) : Integer (+ x 1))

(vector-ref (map-vec add1 (vector 0 41)) 1)
```

Figure 11.1: The `map-vec` example using parametric polymorphism.

$$
\begin{array}{lll}
type & ::= & \ldots \,|\, (\texttt{All} \ (var \ldots) \ type) \,|\, var \\
def & ::= & (\texttt{define} \ (var \ [var\!:\!type] \ldots) : type \ exp) \\
 & | & (:\ var \ type) \\
 & & (\texttt{define} \ (var \ var \ldots) \ exp) \\
R_{\texttt{Poly}} & ::= & def \ldots \ exp
\end{array}
$$

Figure 11.2: The concrete syntax of $R_{\texttt{Poly}}$, extending $R_{\texttt{While}}$ (Figure 9.1).

$$
\begin{array}{lll}
type & ::= & \ldots \,|\, (\texttt{All} \ (var \ldots) \ type) \,|\, var \\
def & ::= & (\texttt{Def} \ var \ ([var\!:\!type] \ldots) \ type \ \texttt{'()} \ exp) \\
 & | & (\texttt{Decl} \ var \ type) \\
 & & (\texttt{Def} \ var \ (var \ldots) \ \texttt{'Any} \ \texttt{'()} \ exp) \\
R_{\texttt{Poly}} & ::= & (\texttt{ProgramDefsExp} \ \texttt{'()} \ (def \ldots) \ exp)
\end{array}
$$

Figure 11.3: The abstract syntax of $R_{\texttt{Poly}}$, extending $R_{\texttt{While}}$ (Figure 9.2).

return type in the Def is Any, but that should be ignored in favor of the return type in the type declaration. (The Any comes from using the same parser as in Chapter 8.) The presence of a type declaration enables the use of an All type for a function, thereby making it polymorphic. The grammar for types is extended to include polymorphic types and type variables.

By including polymorphic types in the *type* non-terminal we choose to make them first-class which has interesting repercussions on the compiler. Many languages with polymorphism, such as C++ [94] and Standard ML [76], only support second-class polymorphism, so it is useful to see an example of first-class polymorphism. In Figure 11.4 we define a function apply-twice whose parameter is a polymorphic function. The occurrence of a polymorphic type underneath a function type is enabled by the normal recursive structure of the grammar for *type* and the categorization of the All type as a *type*. The body of apply-twice applies the polymorphic function to a Boolean and to an integer.

The type checker for $R_{\texttt{Poly}}$ in Figure 11.7 has three new responsibilities (compared to $R_{\texttt{While}}$). The type checking of function application is extended to handle the case where the operator expression is a polymorphic function. In that case the type arguments are deduced by matching the type of the parameters with the types of the arguments. The match-types auxiliary function carries out this deduction by recursively descending through a parameter type pt and the corresponding argument type at, making sure that they are equal except when there is a type parameter on the left (in the parameter type). If it's the first time that the type parameter has been en-

```
(: apply-twice ((All (b) (b -> b)) -> Integer))
(define (apply-twice f)
  (if (f #t) (f 42) (f 777)))

(: id (All (a) (a -> a)))
(define (id x) x)

(apply-twice id)
```

Figure 11.4: An example illustrating first-class polymorphism.

countered, then the algorithm deduces an association of the type parameter to the corresponding type on the right (in the argument type). If it's not the first time that the type parameter has been encountered, the algorithm looks up its deduced type and makes sure that it is equal to the type on the right. Once the type arguments are deduced, the operator expression is wrapped in an `Inst` AST node (for instantiate) that records the type of the operator, but more importantly, records the deduced type arguments. The return type of the application is the return type of the polymorphic function, but with the type parameters replaced by the deduced type arguments, using the `subst-type` function.

The second responsibility of the type checker is extending the function `type-equal?` to handle the `All` type. This is not quite a simple as equal on other types, such as function and vector types, because two polymorphic types can be syntactically different even though they are equivalent types. For example, `(All (a) (a -> a))` is equivalent to `(All (b) (b -> b))`. Two polymorphic types should be considered equal if they differ only in the choice of the names of the type parameters. The `type-equal?` function in Figure 11.8 renames the type parameters of the first type to match the type parameters of the second type.

The third responsibility of the type checker is making sure that only defined type variables appear in type annotations. The `check-well-formed` function defined in Figure 11.9 recursively inspects a type, making sure that each type variable has been defined.

The output language of the type checker is $R_{\texttt{Inst}}$, defined in Figure 11.5. The type checker combines the type declaration and polymorphic function into a single definition, using the `Poly` form, to make polymorphic functions more convenient to process in next pass of the compiler.

The output of the type checker on the polymorphic `map-vec` example is

$$
\begin{array}{lll}
type & ::= & \ldots \mid (\texttt{All}\ (var\ldots)\ type) \mid var \\
exp & ::= & \ldots \mid (\texttt{Inst}\ exp\ type\ (type\ldots)) \\
def & ::= & (\texttt{Def}\ var\ ([var\!:\!type]\ldots)\ type\ \texttt{'()}\ exp) \\
 & \mid & (\texttt{Poly}\ (var\ldots)\ (\texttt{Def}\ var\ ([var\!:\!type]\ldots)\ type\ \texttt{'()}\ exp)) \\
R_{\texttt{Inst}} & ::= & (\texttt{ProgramDefsExp}\ \texttt{'()}\ (def\ldots)\ exp)
\end{array}
$$

Figure 11.5: The abstract syntax of $R_{\texttt{Inst}}$, extending $R_{\texttt{While}}$ (Figure 9.2).

```
(poly (a) (define (map-vec [f : (a -> a)] [v : (Vector a a)]) : (Vector a a)
  (vector (f (vector-ref v 0)) (f (vector-ref v 1)))))

(define (add1 [x : Integer]) : Integer (+ x 1))

(vector-ref ((inst map-vec (All (a) ((a -> a) (Vector a a) -> (Vector a a)))
                    (Integer))
          add1 (vector 0 41)) 1)
```

Figure 11.6: Output of the type checker on the map-vec example.

listed in Figure 11.6.

11.1 Compiling Polymorphism

Broadly speaking, there are four approaches to compiling parametric polymorphism, which we describe below.

Monomorphization generates a different version of a polymorphic function for each set of type arguments that it is used with, producing type-specialized code. This approach results in the most efficient code but requires whole-program compilation (no separate compilation) and increases code size. For our current purposes monomorphization is a non-starter because, with first-class polymorphism, it is sometimes not possible to determine which generic functions are used with which type arguments during compilation. (It can be done at runtime, with just-in-time compilation.) This approach is used to compile C++ templates [94] and polymorphic functions in NESL [13] and ML [100].

Uniform representation generates one version of each polymorphic function but requires all values have a common "boxed" format, such as the tagged values of type Any in $R_{\texttt{Any}}$. Non-polymorphic code (i.e.

```
(define type-check-poly-class
  (class type-check-Rwhile-class
    (super-new)
    (inherit check-type-equal?)

    (define/override (type-check-apply env e1 es)
      (define-values (e^ ty) ((type-check-exp env) e1))
      (define-values (es^ ty*) (for/lists (es^ ty*) ([e (in-list es)])
                                 ((type-check-exp env) e)))
      (match ty
        [`(,ty^* ... -> ,rt)
         (for ([arg-ty ty*] [param-ty ty^*])
           (check-type-equal? arg-ty param-ty (Apply e1 es)))
         (values e^ es^ rt)]
        [`(All ,xs (,tys ... -> ,rt))
         (define env^ (append (for/list ([x xs]) (cons x 'Type)) env))
         (define env^^ (for/fold ([env^^ env^]) ([arg-ty ty*] [param-ty tys])
                         (match-types env^^ param-ty arg-ty)))
         (define targs
           (for/list ([x xs])
             (match (dict-ref env^^ x (lambda () #f))
               [#f (error 'type-check "type variable ~a not deduced\nin ~v"
                          x (Apply e1 es))]
               [ty ty])))
         (values (Inst e^ ty targs) es^ (subst-type env^^ rt))]
        [else (error 'type-check "expected a function, not ~a" ty)]))

    (define/override ((type-check-exp env) e)
      (match e
        [(Lambda `([,xs : ,Ts] ...) rT body)
         (for ([T Ts]) ((check-well-formed env) T))
         ((check-well-formed env) rT)
         ((super type-check-exp env) e)]
        [(HasType e1 ty)
         ((check-well-formed env) ty)
         ((super type-check-exp env) e)]
        [else ((super type-check-exp env) e)]))

    (define/override ((type-check-def env) d)
      (verbose 'type-check "poly/def" d)
      (match d
        [(Generic ts (Def f (and p:t* (list `[,xs : ,ps] ...)) rt info body))
         (define ts-env (for/list ([t ts]) (cons t 'Type)))
         (for ([p ps]) ((check-well-formed ts-env) p))
         ((check-well-formed ts-env) rt)
         (define new-env (append ts-env (map cons xs ps) env))
         (define-values (body^ ty^) ((type-check-exp new-env) body))
         (check-type-equal? ty^ rt body)
         (Generic ts (Def f p:t* rt info body^))]
        [else ((super type-check-def env) d)]))

    (define/override (type-check-program p)
      (match p
        [(Program info body)
         (type-check-program (ProgramDefsExp info '() body))]
        [(ProgramDefsExp info ds body)
         (define ds^ (combine-decls-defs ds))
         (define new-env (for/list ([d ds^])
                           (cons (def-name d) (fun-def-type d))))
         (define ds^^ (for/list ([d ds^]) ((type-check-def new-env) d)))
         (define-values (body^ ty) ((type-check-exp new-env) body))
         (check-type-equal? ty 'Integer body)
         (ProgramDefsExp info ds^^ body^)]))
    ))
```

Figure 11.7: Type checker for the $R_{\texttt{Poly}}$ language.

```
(define/override (type-equal? t1 t2)
  (match* (t1 t2)
    [(`(All ,xs ,T1) `(All ,ys ,T2))
     (define env (map cons xs ys))
     (type-equal? (subst-type env T1) T2)]
    [(other wise)
     (super type-equal? t1 t2)]))

(define/public (match-types env pt at)
  (match* (pt at)
    [('Integer 'Integer) env] [('Boolean 'Boolean) env]
    [('Void 'Void) env] [('Any 'Any) env]
    [(`(Vector ,pts ...) `(Vector ,ats ...))
     (for/fold ([env^ env]) ([pt1 pts] [at1 ats])
       (match-types env^ pt1 at1))]
    [(`(,pts ... -> ,prt) `(,ats ... -> ,art))
     (define env^ (match-types env prt art))
     (for/fold ([env^^ env^]) ([pt1 pts] [at1 ats])
       (match-types env^^ pt1 at1))]
    [(`(All ,pxs ,pt1) `(All ,axs ,at1))
     (define env^ (append (map cons pxs axs) env))
     (match-types env^ pt1 at1)]
    [((? symbol? x) at)
     (match (dict-ref env x (lambda () #f))
       [#f (error 'type-check "undefined type variable ~a" x)]
       ['Type (cons (cons x at) env)]
       [t^ (check-type-equal? at t^ 'matching) env])]
    [(other wise) (error 'type-check "mismatch ~a != a" pt at)]))

(define/public (subst-type env pt)
  (match pt
    ['Integer 'Integer] ['Boolean 'Boolean]
    ['Void 'Void] ['Any 'Any]
    [`(Vector ,ts ...)
     `(Vector ,@(for/list ([t ts]) (subst-type env t)))]
    [`(,ts ... -> ,rt)
     `(,@(for/list ([t ts]) (subst-type env t)) -> ,(subst-type env rt))]
    [`(All ,xs ,t)
     `(All ,xs ,(subst-type (append (map cons xs xs) env) t))]
    [(? symbol? x) (dict-ref env x)]
    [else (error 'type-check "expected a type not ~a" pt)]))

(define/public (combine-decls-defs ds)
  (match ds
    ['() '()]
    [`(,(Decl name type) . ,(Def f params _ info body) . ,ds^))
     (unless (equal? name f)
       (error 'type-check "name mismatch, ~a != ~a" name f))
     (match type
       [`(All ,xs (,ps ... -> ,rt))
        (define params^ (for/list ([x params] [T ps]) `[,x : ,T]))
        (cons (Generic xs (Def name params^ rt info body))
              (combine-decls-defs ds^))]
       [`(,ps ... -> ,rt)
        (define params^ (for/list ([x params] [T ps]) `[,x : ,T]))
        (cons (Def name params^ rt info body) (combine-decls-defs ds^))]
       [else (error 'type-check "expected a function type, not ~a" type) ])]
    [`(,(Def f params rt info body) . ,ds^)
     (cons (Def f params rt info body) (combine-decls-defs ds^))]))
```

Figure 11.8: Auxiliary functions for type checking $R_{\texttt{Poly}}$.

```
(match ty
  ['Integer (void)]
  ['Boolean (void)]
  ['Void (void)]
  [(? symbol? a)
   (match (dict-ref env a (lambda () #f))
     ['Type (void)]
     [else (error 'type-check "undefined type variable ~a" a)])]
  [`(Vector ,ts ...)
   (for ([t ts]) ((check-well-formed env) t))]
  [`(,ts ... -> ,t)
   (for ([t ts]) ((check-well-formed env) t))
   ((check-well-formed env) t)]
  [`(All ,xs ,t)
   (define env^ (append (for/list ([x xs]) (cons x 'Type)) env))
   ((check-well-formed env^) t)]
  [else (error 'type-check "unrecognized type ~a" ty)]))
```

Figure 11.9: Well-formed types.

monomorphic code) is compiled similarly to code in a dynamically typed language (like $R_{\mathtt{Dyn}}$), in which primitive operators require their arguments to be projected from **Any** and their results are injected into **Any**. (In object-oriented languages, the projection is accomplished via virtual method dispatch.) The uniform representation approach is compatible with separate compilation and with first-class polymorphism. However, it produces the least-efficient code because it introduces overhead in the entire program, including non-polymorphic code. This approach is used in the implementation of CLU [70, 69], ML [19, 7], and Java [14].

Mixed representation generates one version of each polymorphic function, using a boxed representation for type variables. Monomorphic code is compiled as usual (as in $R_{\mathtt{While}}$) and conversions are performed at the boundaries between monomorphic and polymorphic (e.g. when a polymorphic function is instantiated and called). This approach is compatible with separate compilation and first-class polymorphism and maintains the efficiency for monomorphic code. The tradeoff is increased overhead at the boundary between monomorphic and polymorphic code. This approach is used in compilers for variants of ML [67] and starting in Java 5 with the addition of autoboxing.

```
(define (map-vec [f : (Any -> Any)] [v : (Vector Any Any)])
                 : (Vector Any Any)
  (vector (f (vector-ref v 0)) (f (vector-ref v 1))))

(define (add1 [x : Integer]) : Integer (+ x 1))

(vector-ref ((cast map-vec
                   ((Any -> Any) (Vector Any Any) -> (Vector Any Any))
                   ((Integer -> Integer) (Vector Integer Integer)
                            -> (Vector Integer Integer)))
             add1 (vector 0 41)) 1)
```

Figure 11.10: The polymorphic `map-vec` example after type erasure.

Type passing uses the unboxed representation in both monomorphic and
   polymorphic code. Each polymorphic function is compiled to a single
   function with extra parameters that describe the type arguments. The
   type information is used by the generated code to direct access of the
   unboxed values at runtime. This approach is used in compilers for
   the Napier88 language [79] and ML [48]. This approach is compatible
   with separate compilation and first-class polymorphism and maintains
   the efficiency for monomorphic code. There is runtime overhead in
   polymorphic code from dispatching on type information.

   In this chapter we use the mixed representation approach, partly because
of its favorable attributes, and partly because it is straightforward to imple-
ment using the tools that we have already built to support gradual typing.
To compile polymorphic functions, we add just one new pass, `erase-types`,
to compile $R_{\texttt{Inst}}$ to $R_{\texttt{cast}}$.

11.2   Erase Types

We use the `Any` type from Chapter 8 to represent type variables. For exam-
ple, Figure 11.10 shows the output of the `erase-types` pass on the poly-
morphic `map-vec` (Figure 11.1). The occurrences of type parameter `a` are
replaced by `Any` and the polymorphic `All` types are removed from the type
of `map-vec`.

   This process of type erasure creates a challenge at points of instantiation.
For example, consider the instantiation of `map-vec` in Figure 11.6. The type
of `map-vec` is

```
(All (a) ((a -> a) (Vector a a) -> (Vector a a)))
```

and it is instantiated to

```
((Integer -> Integer) (Vector Integer Integer)
   -> (Vector Integer Integer))
```

After erasure, the type of `map-vec` is

```
((Any -> Any) (Vector Any Any) -> (Vector Any Any))
```

but we need to convert it to the instantiated type. This is easy to do in the target language R_{cast} with a single `cast`. In Figure 11.10, the instantiation of `map-vec` has been compiled to a `cast` from the type of `map-vec` to the instantiated type. The source and target type of a cast must be consistent (Figure 10.4), which indeed is the case because both the source and target are obtained from the same polymorphic type of `map-vec`, replacing the type parameters with `Any` in the former and with the deduced type arguments in the later. (Recall that the `Any` type is consistent with any type.)

To implement the `erase-types` pass, we recommend defining a recursive auxiliary function named `erase-type` that applies the following two transformations. It replaces type variables with `Any`

x
$\Rightarrow$
`Any`

and it removes the polymorphic `All` types.

(`All` $xs\ T_1$)
$\Rightarrow$
T_1'

Apply the `erase-type` function to all of the type annotations in the program.

Regarding the translation of expressions, the case for `Inst` is the interesting one. We translate it into a `Cast`, as shown below. The type of the subexpression e is the polymorphic type (`All`xsT). The source type of the cast is the erasure of T, the type T'. The target type T'' is the result of substituting the arguments types ts for the type parameters xs in T followed by doing type erasure.

(`Inst` e (`All` $xs\ T$) ts)
$\Rightarrow$
(`Cast` $e'\ T'\ T''$)

where $T'' = $ (`erase-type` (`subst-type` $s\ T$)) and $s = $ (`map cons` $xs\ ts$).

Finally, each polymorphic function is translated to a regular functions in which type erasure has been applied to all the type annotations and the body.

(Poly ts (Def f ([x_1 : T_1] ...) T_r *info* e))
$\Rightarrow$
(Def f ([x_1 : T_1'] ...) T_r' *info* e')

Exercise 38. Implement a compiler for the polymorphic language $R_{\texttt{Poly}}$ by extending and adapting your compiler for $R_?$. Create 6 new test programs that use polymorphic functions. Some of them should make use of first-class polymorphism.

Figure 11.11 provides an overview of all the passes needed for the compilation of $R_{\texttt{Poly}}$.
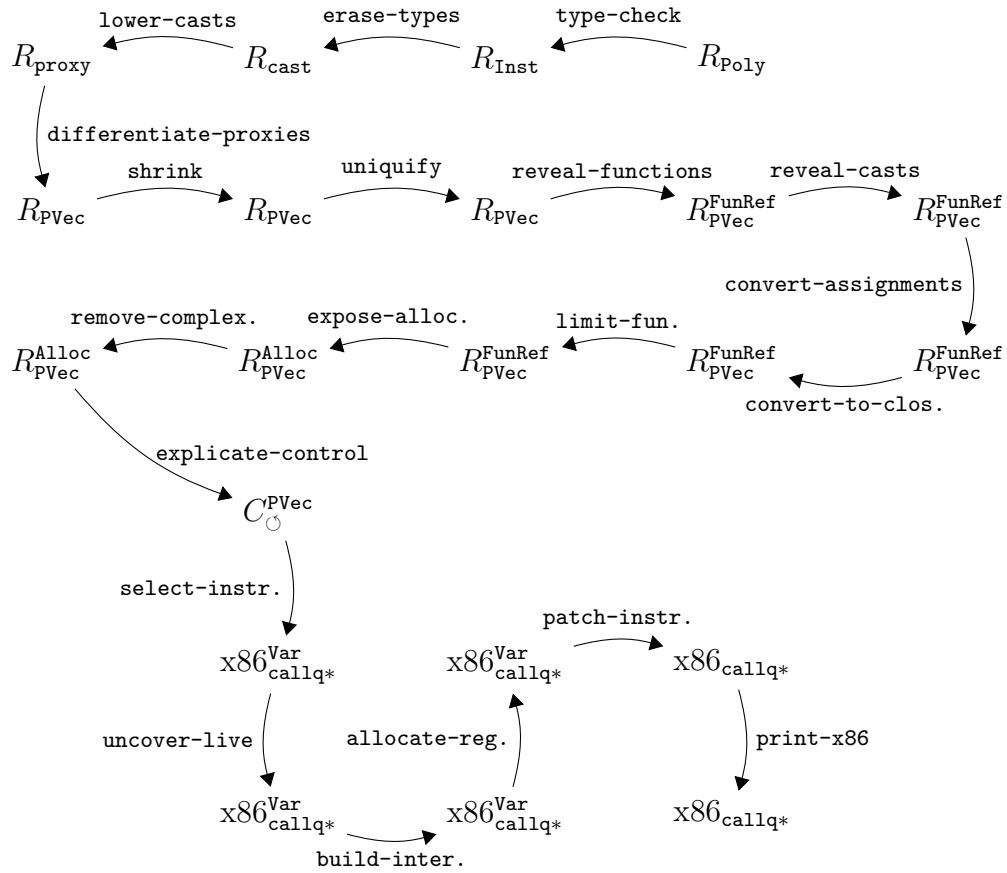
Figure 11.11: Diagram of the passes for $R_{\texttt{Poly}}$ (parametric polymorphism).

12

Appendix

12.1 Interpreters

We provide interpreters for each of the source languages R_{Int}, R_{Var}, ... in the files `interp-Rint.rkt`, `interp-Rvar.rkt`, etc. The interpreters for the intermediate languages C_{Var} and C_{If} are in `interp-Cvar.rkt` and `interp-C1.rkt`. The interpreters for C_{Vec}, C_{Fun}, pseudo-x86, and x86 are in the `interp.rkt` file.

12.2 Utility Functions

The utility functions described in this section are in the `utilities.rkt` file of the support code.

interp-tests The `interp-tests` function runs the compiler passes and the interpreters on each of the specified tests to check whether each pass is correct. The `interp-tests` function has the following parameters:

name (a string) a name to identify the compiler,

typechecker a function of exactly one argument that either raises an error using the `error` function when it encounters a type error, or returns `#f` when it encounters a type error. If there is no type error, the type checker returns the program.

passes a list with one entry per pass. An entry is a list with four things:

    1. a string giving the name of the pass,

2. the function that implements the pass (a translator from AST to AST),

3. a function that implements the interpreter (a function from AST to result value) for the output language,

4. and a type checker for the output language. Type checkers for the R and C languages are provided in the support code. For example, the type checkers for $R_{\mathtt{Var}}$ and $C_{\mathtt{Var}}$ are in `type-check-Rvar.rkt` and `type-check-Cvar.rkt`. The type checker entry is optional. The support code does not provide type checkers for the x86 languages.

source-interp an interpreter for the source language. The interpreters from Appendix 12.1 make a good choice.

test-family (a string) for example, `"r1"`, `"r2"`, etc.

tests a list of test numbers that specifies which tests to run. (see below)

The `interp-tests` function assumes that the subdirectory `tests` has a collection of Racket programs whose names all start with the family name, followed by an underscore and then the test number, ending with the file extension `.rkt`. Also, for each test program that calls `read` one or more times, there is a file with the same name except that the file extension is `.in` that provides the input for the Racket program. If the test program is expected to fail type checking, then there should be an empty file of the same name but with extension `.tyerr`.

compiler-tests runs the compiler passes to generate x86 (a `.s` file) and then runs the GNU C compiler (gcc) to generate machine code. It runs the machine code and checks that the output is 42. The parameters to the `compiler-tests` function are similar to those of the `interp-tests` function, and consist of

- a compiler name (a string),

- a type checker,

- description of the passes,

- name of a test-family, and

- a list of test numbers.

compile-file   takes a description of the compiler passes (see the comment for `interp-tests`) and returns a function that, given a program file name (a string ending in `.rkt`), applies all of the passes and writes the output to a file whose name is the same as the program file name but with `.rkt` replaced with `.s`.

read-program   takes a file path and parses that file (it must be a Racket program) into an abstract syntax tree.

parse-program   takes an S-expression representation of an abstract syntax tree and converts it into the struct-based representation.

assert   takes two parameters, a string (`msg`) and Boolean (`bool`), and displays the message `msg` if the Boolean `bool` is false.

lookup   takes a key and an alist, and returns the first value that is associated with the given key, if there is one. If not, an error is triggered. The alist may contain both immutable pairs (built with `cons`) and mutable pairs (built with `mcons`).

12.3   x86 Instruction Set Quick-Reference

Table 12.1 lists some x86 instructions and what they do. We write $A \rightarrow B$ to mean that the value of A is written into location B. Address offsets are given in bytes. The instruction arguments A, B, C can be immediate constants (such as `$4`), registers (such as `%rax`), or memory references (such as `-4(%ebp)`). Most x86 instructions only allow at most one memory reference per instruction. Other operands must be immediates or registers.

Instruction	Operation
addq A, B	$A + B \to B$
negq A	$-A \to A$
subq A, B	$B - A \to B$
callq L	Pushes the return address and jumps to label L
callq $*A$	Calls the function at the address A.
retq	Pops the return address and jumps to it
popq A	$*\texttt{rsp} \to A; \texttt{rsp} + 8 \to \texttt{rsp}$
pushq A	$\texttt{rsp} - 8 \to \texttt{rsp}; A \to *\texttt{rsp}$
leaq A,B	$A \to B$ (B must be a register)
cmpq A, B	compare A and B and set the flag register (B must not be an immediate)
je L	Jump to label L if the flag register matches the
jl L	condition code of the instruction, otherwise go to the
jle L	next instructions. The condition codes are e for
jg L	"equal", l for "less", le for "less or equal", g for
jge L	"greater", and ge for "greater or equal".
jmp L	Jump to label L
movq A, B	$A \to B$
movzbq A, B	$A \to B$, where A is a single-byte register (e.g., al or cl), B is a 8-byte register, and the extra bytes of B are set to zero.
notq A	$\sim A \to A$ \quad (bitwise complement)
orq A, B	$A|B \to B$ \quad (bitwise-or)
andq A, B	$A\&B \to B$ \quad (bitwise-and)
salq A, B	$B \ll A \to B$ (arithmetic shift left, where A is a constant)
sarq A, B	$B \gg A \to B$ (arithmetic shift right, where A is a constant)
sete A setl A setle A setg A setge A	If the flag matches the condition code, then $1 \to A$, else $0 \to A$. Refer to je above for the description of the condition codes. A must be a single byte register (e.g., al or cl).

Table 12.1: Quick-reference for the x86 instructions used in this book.

$$
\begin{array}{lll}
type & ::= & \texttt{Integer} \mid \texttt{Boolean} \mid \texttt{(Vector} \; type\dots\texttt{)} \mid \texttt{Void} \\
      & \mid & \texttt{(}type\dots \; \texttt{->} \; type\texttt{)} \mid \texttt{Any} \\
ftype & ::= & \texttt{Integer} \mid \texttt{Boolean} \mid \texttt{Void} \mid \texttt{(Vector Any}\dots\texttt{)} \\
      & \mid & \texttt{(Any}\dots \; \texttt{-> Any)} \\
exp & ::= & \dots\texttt{(inject} \; exp \; ftype\texttt{))} \mid \texttt{(project} \; exp \; ftype\texttt{)} \\
    & \mid & \texttt{(any-vector-length} \; exp\texttt{)} \mid \texttt{(any-vector-ref} \; exp \; exp\texttt{)} \\
    & \mid & \texttt{(any-vector-set!} \; exp \; exp \; exp\texttt{)} \\
    & \mid & \texttt{(boolean?} \; exp\texttt{)} \mid \texttt{(integer?} \; exp\texttt{)} \mid \texttt{(void?} \; exp\texttt{)} \\
    & \mid & \texttt{(vector?} \; exp\texttt{)} \mid \texttt{(procedure?} \; exp\texttt{)} \\
def & ::= & \texttt{(define} \; (var \; [var\!:\!type]\dots) : type \; exp\texttt{)} \\
R_{\texttt{Any}} & ::= & def\dots \; exp
\end{array}
$$

Figure 12.1: The concrete syntax of $R_{\texttt{Any}}$, extending R_λ (Figure 7.4) with `Any`.

$$
\begin{array}{lll}
atm & ::= & int \mid var \\
exp & ::= & atm \mid \texttt{(read)} \mid \texttt{(-} \; atm\texttt{)} \mid \texttt{(+} \; atm \; atm\texttt{)} \\
stmt & ::= & var \; \texttt{=} \; exp\texttt{;} \\
tail & ::= & \texttt{return} \; exp\texttt{;} \mid stmt \; tail \\
C_{\texttt{Var}} & ::= & (label\!: \; tail)\dots
\end{array}
$$

Figure 12.2: The concrete syntax of the $C_{\texttt{Var}}$ intermediate language.

12.4 Concrete Syntax for Intermediate Languages

The concrete syntax of $R_{\texttt{Any}}$ is defined in Figure 12.1.

The concrete syntax for $C_{\texttt{Var}}$, $C_{\texttt{If}}$, $C_{\texttt{Vec}}$ and $C_{\texttt{Fun}}$ is defined in Figures 12.2, 12.3, 12.4, and 12.5, respectively.

$$
\begin{array}{rcl}
atm & ::= & int \mid var \mid bool \\
cmp & ::= & \texttt{eq?} \mid \texttt{<} \\
exp & ::= & atm \mid (\texttt{read}) \mid (\texttt{-}\ atm) \mid (\texttt{+}\ atm\ atm) \\
    &   & \mid \quad (\texttt{not}\ atm) \mid (cmp\ atm\ atm) \\
stmt & ::= & var\ \texttt{=}\ exp; \\
tail & ::= & \texttt{return}\ exp; \mid stmt\ tail \mid \texttt{goto}\ label; \\
    &   & \mid \quad \texttt{if}\ (cmp\ atm\ atm)\ \texttt{goto}\ label;\ \texttt{else goto}\ label; \\
C_{\texttt{If}} & ::= & (label\texttt{:}\ tail)\dots
\end{array}
$$

Figure 12.3: The concrete syntax of the $C_{\texttt{If}}$ intermediate language.

$$
\begin{array}{rcl}
atm & ::= & int \mid var \mid bool \\
cmp & ::= & \texttt{eq?} \mid \texttt{<} \\
exp & ::= & atm \mid (\texttt{read}) \mid (\texttt{-}\ atm) \mid (\texttt{+}\ atm\ atm) \\
    &   & \mid \quad (\texttt{not}\ atm) \mid (cmp\ atm\ atm) \\
    &   & \mid \quad (\texttt{allocate}\ int\ type) \\
    &   & \mid \quad (\texttt{vector-ref}\ atm\ int) \mid (\texttt{vector-set!}\ atm\ int\ atm) \\
    &   & \mid \quad (\texttt{global-value}\ var) \mid (\texttt{void}) \\
stmt & ::= & var\ \texttt{=}\ exp; \mid (\texttt{collect}\ int) \\
tail & ::= & \texttt{return}\ exp; \mid stmt\ tail \mid \texttt{goto}\ label; \\
    &   & \mid \quad \texttt{if}\ (cmp\ atm\ atm)\ \texttt{goto}\ label;\ \texttt{else goto}\ label; \\
C_{\texttt{Vec}} & ::= & (label\texttt{:}\ tail)\dots
\end{array}
$$

Figure 12.4: The concrete syntax of the $C_{\texttt{Vec}}$ intermediate language.

$$
\begin{array}{rcl}
atm & ::= & int \mid var \mid \texttt{\#t} \mid \texttt{\#f} \\
cmp & ::= & \texttt{eq?} \mid \texttt{<} \\
exp & ::= & atm \mid (\texttt{read}) \mid (\texttt{-}\ atm) \mid (\texttt{+}\ atm\ atm) \mid (\texttt{not}\ atm) \mid (cmp\ atm\ atm) \\
    &   & \mid \quad (\texttt{allocate}\ int\ type) \mid (\texttt{vector-ref}\ atm\ int) \\
    &   & \mid \quad (\texttt{vector-set!}\ atm\ int\ atm) \mid (\texttt{global-value}\ name) \mid (\texttt{void}) \\
    &   & \mid \quad (\texttt{fun-ref}\ label) \mid (\texttt{call}\ atm\ atm\dots) \\
stmt & ::= & (\texttt{Assign}\ var\ exp) \mid (\texttt{Return}\ exp) \mid (\texttt{collect}\ int) \\
tail & ::= & (\texttt{Return}\ exp) \mid (\texttt{seq}\ stmt\ tail) \\
    &   & \mid \quad (\texttt{goto}\ label) \mid (\texttt{If}\ (cmp\ atm\ atm)\ (\texttt{goto}\ label)\ (\texttt{goto}\ label)) \\
    &   & \mid \quad (\texttt{tail-call}\ atm\ atm\dots) \\
def & ::= & (\texttt{define}\ (label\ [var\texttt{:}type]\dots)\texttt{:}type\ ((label\ \texttt{.}\ tail)\dots)) \\
C_{\texttt{Fun}} & ::= & def\dots
\end{array}
$$

Figure 12.5: The $C_{\texttt{Fun}}$ language, extending $C_{\texttt{Vec}}$ (Figure 12.4) with functions.

Index

Bibliography

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996. ISBN 0262011530.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

[3] Hussein Al-Omari and Khair Eddin Sabri. New graph coloring algorithms. *Journal of Mathematics and Statistics*, 2(4), 2006.

[4] Frances E. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, 1970.

[5] Christopher Anderson and Sophia Drossopoulou. BabyJ - from object based to class based programming via types. In *WOOD '03*, volume 82. Elsevier, 2003.

[6] Andrew W. Appel. Runtime tags aren't necessary. *LISP and Symbolic Computation*, 2(2):153–162, 1989. ISSN 0892-4635. doi: 10.1007/BF01811537. URL http://dx.doi.org/10.1007/BF01811537.

[7] Andrew W. Appel and David B. MacQueen. *A standard ML compiler*, pages 301–324. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987. ISBN 978-3-540-47879-9. doi: 10.1007/3-540-18317-5_17. URL http://dx.doi.org/10.1007/3-540-18317-5_17.

[8] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2003. ISBN 052182060X.

[9] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic

language algol 60. *Commun. ACM*, 3(5):299–314, May 1960. ISSN 0001-0782. doi: 10.1145/367236.367262. URL `http://doi.acm.org/10.1145/367236.367262`.

[10] J. Baker, A. Cunei, T. Kalibera, F. Pizlo, and J. Vitek. Accurate garbage collection in uncooperative environments revisited. *Concurr. Comput. : Pract. Exper.*, 21(12):1572–1606, August 2009. ISSN 1532-0626. doi: 10.1002/cpe.v21:12. URL `http://dx.doi.org/10.1002/cpe.v21:12`.

[11] V. K. Balakrishnan. *Introductory Discrete Mathematics*. Dover Publications, Incorporated, 1996. ISBN 0486691152.

[12] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 25–36, New York, NY, USA, 2004. ACM. ISBN 1-58113-873-3. doi: 10.1145/1005686.1005693. URL `http://doi.acm.org/10.1145/1005686.1005693`.

[13] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 102–111, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915895. doi: 10.1145/155332.155343. URL `https://doi.org/10.1145/155332.155343`.

[14] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the java programming language. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 183–200, New York, NY, USA, 1998. ACM. ISBN 1-58113-005-8. doi: http://doi.acm.org/10.1145/286936.286957. URL `http://doi.acm.org/10.1145/286936.286957`.

[15] Daniel Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, 1979. ISSN 0001-0782.

[16] Randal E. Bryant and David R. O'Hallaron. *x86-64 Machine-Level Programming*. Carnegie Mellon University, September 2005.

[17] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective.* Addison-Wesley Publishing Company, USA, 2nd edition, 2010. ISBN 0136108040, 9780136108047.

[18] Luca Cardelli. The functional abstract machine. Technical Report TR-107, AT&T Bell Laboratories, 1983.

[19] Luca Cardelli. Compiling a functional language. In *ACM Symposium on LISP and Functional Programming*, LFP '84, pages 208–217. ACM, 1984.

[20] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985. ISSN 0360-0300.

[21] C. J. Cheney. A nonrecursive list compacting algoirthm. *Communications of the ACM*, 13(11), 1970.

[22] George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, December 1960. ISSN 0001-0782. doi: 10.1145/367487.367501. URL `https://doi.org/10.1145/367487.367501`.

[23] Keith Cooper and Linda Torczon. *Engineering a Compiler.* Morgan Kaufmann, 2nd edition, 2011.

[24] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2001. ISBN 0070131511.

[25] Cody Cutler and Robert Morris. Reducing pause times with clustered collection. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 131–142, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3589-8. doi: 10.1145/2754169.2754184. URL `http://doi.acm.org/10.1145/2754169.2754184`.

[26] Olivier Danvy. Three steps for the CPS transformation. Technical Report CIS-92-02, Kansas State University, December 1991.

[27] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM. ISBN 1-58113-945-4.

doi: 10.1145/1029873.1029879. URL `http://doi.acm.org/10.1145/1029873.1029879`.

[28] E. W. Dijkstra. Why numbering should start at zero. Technical Report EWD831, University of Texas at Austin, 1982.

[29] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 273–282, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9. doi: 10.1145/143095.143140. URL `http://doi.acm.org/10.1145/143095.143140`.

[30] R. Kent Dybvig. *The SCHEME Programming Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987. ISBN 0-13-791864-X.

[31] R. Kent Dybvig. The development of chez scheme. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 1–12, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3. doi: 10.1145/1159803.1159805. URL `http://doi.acm.org/10.1145/1159803.1159805`.

[32] R. Kent Dybvig and Andrew Keep. P523 compiler assignments. Technical report, Indiana University, 2010.

[33] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine and the lambda-calculus. pages 193–217, 1986.

[34] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, Cambridge, MA, USA, 2001. ISBN 0-262-06218-6.

[35] Matthias Felleisen, M.D. Barski Conrad, David Van Horn, and Eight Students of Northeastern University. *Realm of Racket: Learn to Program, One Game at a Time!* No Starch Press, San Francisco, CA, USA, 2013. ISBN 1593274912, 9781593274917.

[36] Cormac Flanagan. Hybrid type checking. In *POPL 2006: The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, Charleston, South Carolina, January 2006.

[37] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Conference on Programming Language Design and Implementation*, PLDI, pages 502–514, June 1993.

[38] Matthew Flatt and PLT. The Racket reference 6.0. Technical report, PLT Inc., 2014. `http://docs.racket-lang.org/reference/index.html`.

[39] Matthew Flatt, Robert Bruce Findler, and PLT. The racket guide. Technical Report 6.0, PLT Inc., 2014.

[40] Daniel P. Friedman and Matthias Felleisen. *The Little Schemer (4th Ed.)*. MIT Press, Cambridge, MA, USA, 1996. ISBN 0-262-56099-2.

[41] Daniel P. Friedman and David S. Wise. Cons should not evaluate its arguments. Technical Report TR44, Indiana University, 1976.

[42] Ben Gamari and Laura Dietz. Alligator collector: A latency-optimized garbage collector for functional programming languages. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2020, pages 87–99, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375665. doi: 10.1145/3381898.3397214. URL `https://doi.org/10.1145/3381898.3397214`.

[43] Assefaw Hadish Gebremedhin. *Parallel Graph Coloring*. PhD thesis, University of Bergen, 1999.

[44] Abdulaziz Ghuloum. An incremental approach to compiler construction. In *Scheme and Functional Programming Workshop*, 2006.

[45] Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 691–704, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335492. doi: 10.1145/2837614.2837631. URL `https://doi.org/10.1145/2837614.2837631`.

[46] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*,

PLDI '91, pages 165–176, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. doi: 10.1145/113445.113460. URL `http://doi.acm.org/10.1145/113445.113460`.

[47] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[48] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141. ACM Press, 1995. ISBN 0-89791-692-1.

[49] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 150–156, New York, NY, USA, 2002. ACM. ISBN 1-58113-539-4. doi: 10.1145/512429.512449. URL `http://doi.acm.org/10.1145/512429.512449`.

[50] Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

[51] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.

[52] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167–189, 2010.

[53] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, 3C and 3D*, December 2015.

[54] Nicholas Jacek and J. Eliot B. Moss. Learning when to garbage collect with random forests. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2019, pages 53–63, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367226. doi: 10.1145/3315573.3329983. URL `https://doi.org/10.1145/3315573.3329983`.

[55] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-020249-5.

[56] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, Inc., New York, NY, USA, 1996. ISBN 0-471-94148-4.

[57] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman & Hall/CRC, 1st edition, 2011. ISBN 1420082795, 9781420082791.

[58] Andrew W. Keep. *A Nanopass Framework for Commercial Compiler Development.* PhD thesis, Indiana University, December 2012.

[59] Andrew W. Keep, Alex Hearn, and R. Kent Dybvig. Optimizing closures in O(0)-time. In *Proceedings of the 2012 Workshop on Scheme and Functional Programming*, Scheme '12, 2012.

[60] R. Kelsey, W. Clinger, and J. Rees (eds.). Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

[61] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language.* Prentice Hall Press, Upper Saddle River, NJ, USA, 1988. ISBN 0-13-110362-8.

[62] Gary A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM Press, 1973.

[63] S. Kleene. Introduction to Metamathematics, 1952.

[64] Donald E. Knuth. Backus normal form vs. backus naur form. *Commun. ACM*, 7(12):735–736, December 1964. ISSN 0001-0782. doi: 10.1145/355588.365140. URL `http://doi.acm.org/10.1145/355588.365140`.

[65] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4.

[66] Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. Toward efficient gradual typing for structural types via coercions. In *Conference on Programming Language Design and Implementation*, PLDI. ACM, June 2019.

[67] Xavier Leroy. Unboxed objects and polymorphic typing. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 177–188, New York, NY, USA, 1992. ACM Press. ISBN 0-89791-453-8.

[68] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983. ISSN 0001-0782. doi: 10.1145/358141.358147. URL `http://doi.acm.org/10.1145/358141.358147`.

[69] Barbara Liskov. A history of clu. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 133–147, New York, NY, USA, 1993. ACM. ISBN 0-89791-570-4.

[70] Barbara Liskov, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler, and Alan Snyder. CLU reference manual. Technical Report LCS-TR-225, MIT, October 1979.

[71] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2007.

[72] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface, AMD64 Architecture Processor Supplement*, October 2013.

[73] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960. ISSN 0001-0782.

[74] Microsoft. x64 architecture. `https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture`, March 2018.

[75] Microsoft. x64 calling convention. `https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention`, July 2020.

[76] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, 1990. ISBN 0-262-63132-6.

[77] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 271–283, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3. doi: http://doi.acm.org/10.1145/237721.237791. URL `http://doi.acm.org/10.1145/237721.237791`.

[78] E.F. Moore. The shortest path through a maze. In *Proceedings of an International Symposium on the Theory of Switching*, April 1959.

[79] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Trans. Program. Lang. Syst.*, 13(3):342–371, July 1991. ISSN 0164-0925. doi: 10.1145/117009.117017. URL `http://doi.acm.org/10.1145/117009.117017`.

[80] Erik Österlund and Welf Löwe. Block-free concurrent gc: Stack scanning and copying. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 1–12, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343176. doi: 10.1145/2926697.2926701. URL `https://doi.org/10.1145/2926697.2926701`.

[81] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[82] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM Annual Conference*, pages 717–740. ACM Press, 1972.

[83] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 2002. ISBN 0072474777.

[84] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 201–212. ACM Press, 2004. ISBN 1-58113-905-5.

[85] Rifat Shahriyar, Stephen M. Blackburn, Xi Yang, and Kathryn M. McKinley. Taking off the gloves with reference counting immix. In *OOPSLA '13: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, oct 2013. doi: http://dx.doi.org/10.1145/2509136.2509527.

[86] Jonathan Shidal, Ari J. Spilo, Paul T. Scheid, Ron K. Cytron, and Krishna M. Kavi. Recycling trash in cache. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 118–130, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3589-8. doi: 10.1145/2754169.2754183. URL http://doi.acm.org/10.1145/2754169.2754183.

[87] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 164–174, New York, NY, USA, 1988. ACM.

[88] Fridtjof Siebert. *Compiler Construction: 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2–6, 2001 Proceedings*, chapter Constant-Time Root Scanning for Deterministic Garbage Collection, pages 304–318. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-45306-2. doi: 10.1007/3-540-45306-7_21. URL http://dx.doi.org/10.1007/3-540-45306-7_21.

[89] Jeremy G. Siek and Bor-Yuh Evan Chang. A problem course in compilation: From python to x86 assembly. Technical report, Univesity of Colorado, 2012.

[90] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

[91] Jeremy G. Siek, Peter Thiemann, and Philip Wadler. Blame and coercion: Together again for the first time. In *Conference on Programming Language Design and Implementation*, PLDI, June 2015.

[92] Michael Sperber, R. KENT DYBVIG, MATTHEW FLATT, ANTON VAN STRAATEN, ROBBY FINDLER, and JACOB MATTHEWS. Revised[6] report on the algorithmic language scheme. *Journal of Functional Programming*, 19:1–301, 8 2009. ISSN 1469-7653. doi: 10.1017/S0956796809990074. URL http://journals.cambridge.org/article_S0956796809990074.

[93] Guy L. Steele, Jr. Data representations in pdp-10 maclisp. AI Memo 420, MIT Artificial Intelligence Lab, September 1977.

[94] Bjarne Stroustrup. Parameterized types for C++. In *USENIX C++ Conference*, October 1988.

[95] Gerald Jay Sussman and Jr. Guy L. Steele. Scheme: an interpreter for extended lambda calculus. Technical Report AI Memo No. 349, MIT, December 1975.

[96] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: the continuously concurrent compacting collector. In *Proceedings of the international symposium on Memory management*, ISMM '11, pages 79–88, New York, NY, USA, 2011. ACM. doi: http://doi.acm.org/10.1145/1993478. 1993491.

[97] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium*, 2006.

[98] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 157–167, New York, NY, USA, 1984. ACM. ISBN 0-89791-131-8. doi: 10. 1145/800020.808261. URL `http://doi.acm.org/10.1145/800020. 808261`.

[99] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, ESOP, pages 1–16, March 2009.

[100] Stephen Weeks. Whole-program compilation in mlton. In *Proceedings of the 2006 Workshop on ML*, ML '06, page 1, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934839. doi: 10.1145/1159876.1159877. URL `https://doi.org/10.1145/ 1159876.1159877`.

[101] Paul Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer Berlin / Heidelberg, 1992. URL `http://dx.doi.org/10.1007/BFb0017182. 10.1007/BFb0017182`.