编译基础

增量的微通道方法

Jeremy G. Siek
印第安纳大学

贡献者:
Carl Factora
Andre Kuhlenschmidt
Ryan R. Newton
Ryan Scott
Cameron Swords
Michael M. Vitousek
Michael Vollmer

吴华均 译

May 15, 2021

这本书是献给印第安纳大学的编程语言专家。

目录

插图目录

前言

当程序员按下“运行”按钮，软件就开始运行，这是一个神奇的时刻。不知何故，用某种高级语言编写的程序只能以某种方式在能位转换的计算机上运行。下面我们就来揭示让这一刻成为可能的魔法。从 20 世纪 50 年代 Backus 及其同事的突破性工作开始，计算机科学家发现构造程序的技术，称为 *编译器*，，它可以自动将高级程序转换为机器代码。

我们为一门小而强大的语言构建自己的编译器，带你踏上征程。在此过程中，我们解释编译器基础的基本概念、算法和数据结构。帮助你理解程序是如何映射到计算机硬件上，这有助于在硬件和软件之间的连接点进行推理，如执行时间、软件错误和安全漏洞。对于那些对编译器构造感兴趣的人，我们的目标是为高级主题 (如即时编译、程序分析和程序优化) 提供一个跳板。对于那些对设计和实现自己的编程语言感兴趣的人，将语言设计选择与它们对编译器产生的代码的影响联系起来。

编译器通常被组织成一系列的阶段，逐步地把程序翻译成在硬件上运行的代码。我们将编译器划分为大量的 *微通道*，每个微通道执行一项任务，从而达到极致。这使我们能够独立地测试每个通道的输出，而且，允许我们集中精力使编译器更容易理解。

最常见的描述编译器的方法是每一章进行一次编译。这样做的问题是，它混淆了语言特性如何在编译器中激发设计选择。我们采用一种 *增量*的方法，在每一章中构建一个完整的编译器，从算术和变量开始，并在随后的章节中添加新的特性。

我们对语言特性的选择是为了引出编译器中使用的基本概念和算法。

- 第 1 章和第 2 章，从整数算术和局部变量开始，介绍编译器构造的基

本工具: *抽象语法树*和 *递归函数*。

- 第 3 章中，用 *图着色*给机器寄存器变量赋值。
- 第 4 章增加 `if` 表达式，这激发一个优雅的递归算法，用于将表达式映射到 *控制流图*。
- 第 5 章增加堆分配元组，促进 *垃圾收集*。
- 第 6 章添加一等变量，但没有词法作用域，这与 C 编程语言 [72] 类似，只是我们生成有效的尾部调用。读取器了解过程调用堆栈、*调用约定*以及它们与寄存器分配和垃圾收集的交互。
- 第 7 章增加具有词法作用域的匿名函数，即 *lambda 抽象*。学习 *闭包转换*，在闭包转换中，lambdas 被转换为函数和元组的组合。
- 第 8 章增加 *动态类型*。在此之前，输入语言是静态类型的。读取器使用 `Any` 类型扩展静态类型语言，作为编译动态类型语言的目标。
- 第 9 章通过添加循环和可变变量来充实对命令式编程语言的支持。这些增加引出在寄存分配器中进行 *数据流分析*的需要。
- 第 10 章使用第 8 章的 `Any` 类型来实现一种 *逐渐类型化语言*，在这种语言中，程序的不同区域可以是静态类型的，也可以是动态类型的。读取器为 *代理*实现运行时支持，以允许值在区域之间安全移动。
- 第 11 章利用在第 8 章和第 10 章中开发的 `Any` 类型和类型强制转换，增加带有自动装箱功能的 *泛型*。

有许多语言特性我们没有包括在内。我们的选择是在特性附带的复杂性和它所暴露的基本概念之间进行权衡。例如，我们包含元组而不包含记录，因为它们都引出对堆分配和垃圾回收的研究，但是记录带有更多的附带复杂性。

自 2016 年起，这本书作为印第安纳大学编译课程的教材，这是一门专为高年级本科生和一年级研究生开设的 16 周课程。在这门课程之前，学生学习命令式语言和函数式语言编程，学习数据结构和算法，并学习离散数学。课程开始时，学生以 2-4 人为一组。每个小组每两周完成一章，从第 2 章开

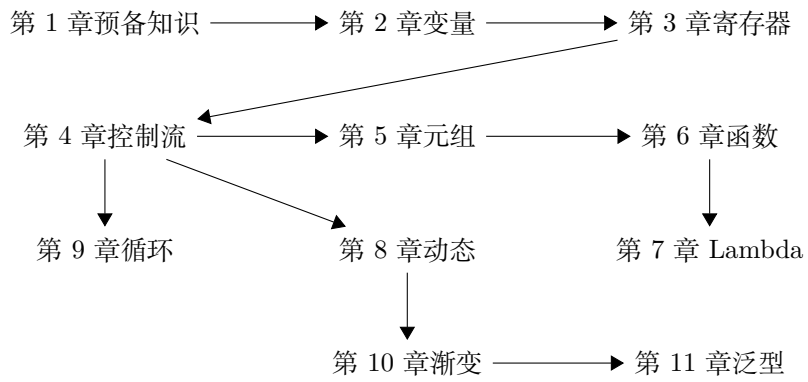


图 1: 章节依赖关系图。

始，到第 8 章结束。许多章节包括我们分配给研究生的一个具有挑战性的问题。本课程的最后两周包括一个期末专题，在这个专题中，学生设计并实现他们所选择的编译器扩展。第 9 章、第 10 章和第 11 章可以用来支持这些项目，或者它们可以取代前面的一些章节。例如，一门强调静态类型命令式语言的课程会跳过第 8 章，而选择第 9 章。图 1 描述章节之间的依赖关系。

本书还在加利福尼亚州立理工大学，罗斯-霍尔曼技术学院和麻萨诸塞州洛厄尔大学的编译器课程中使用过。

我们在编译器的实现和输入语言中都使用 Racket 语言，所以读者应该精通 Racket 或 Scheme 。有很多很好的学习 Scheme 和 Racket 的资源 [37, 1, 47, 41, 42, 46] 。本书的支持代码在 github 存储库中，网址如下：

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

该编译器的目标是 x86 汇编语言 [63] ，所以对于已经上过计算机系统课程 [19] 的读者来说是有帮助的，但不是必需的。本书介绍需要的 x86-64 汇编语言的部分。我们遵循 System V 调用约定 [18, 83] ，所以当我们生成的汇编代码在 Linux 和 MacOS 操作系统上使用 GNU C 编译器 (`gcc`) 编译时，它与运行时系统一起工作 (用 C 编写)。在 Windows 操作系统上，`gcc` 使用 Microsoft x64 调用约定 [85, 86] 。因此，我们生成的汇编代码 不能与 Windows 上的运行时系统一起工作。一种解决方案是使用 Linux 虚拟机作为客户操作系统。

致谢

编译器构造在印第安纳大学的传统可以追溯到 Daniel Friedman 在 20 世纪 70 年代和 80 年代的编程语言研究和课程。他的一个学生 Kent Dybvig 实现了 Chez Scheme [39] ，这是一个针对 Scheme 的高质量、高效的编译器。在 20 世纪 90 年代和 2000 年代，Dybvig 教授编译器课程，并继续开发 Chez Scheme 。编译器课程发展到融合新颖的教学思想，同时也包括有效的现实世界编译器的元素。Friedman 的一个想法是把编译器分成许多小的通道。另一个被称为“游戏”的想法是测试每一个解释器的通道生成的代码。

在他的学生 Dipanwita Sarkar 和 Andrew Keep 的帮助下，Dybvig 开发支持这种方法的基础设施，并发展使用更小微通道的课程 [97, 68]。本书中的许多编译器设计决策都受到 Dybvig and Keep [40] 的赋值描述的启发。在 2000 年中期，Dybvig 的一个名叫 Abdulaziz Ghuloum 的学生注意到，课程的前后结构让学生很难理解编译器设计的基本原理。Ghuloum 提出了增量方法 [52] 。

我们感谢 Bor-Yuh Chang 、John Clements 、Jay McCarthy 、Joseph Near 、Nate Nystrom 和 Michael Wollowski ，他们基于早期草稿教授课程。

我们感谢 Ronald Garcia 在 2000 年早期与 Jeremy 学习编译器课程时成为他们的合作伙伴，尤其感谢他找到让垃圾收集器白费力气的 bug !

Jeremy G. Siek
Bloomington, Indiana

1

预备知识

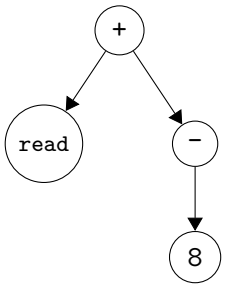
在本章中，回顾实现编译器所需的基本工具。程序通常是由程序员以文本形式输入的，也就是一串字符。程序即文本表示称为 *具体语法*。我们用具体的语法来简明地写和讨论程序。在编译器内部，使用 *抽象语法树* (AST)，以一种有效支持编译器要执行的操作方式来表示程序。 从具体语法到抽象语法的转换过程称为 *解析* [2] 。在本书中，不涉及解析的理论和实现。在支持代码中提供一个解析器，将具体语法转换为抽象语法。

在编译器中，AST 可以以多种不同的方式表示，这取决于编写编译器所用的编程语言。使用 Racket 的 `struct` 特性来表示 AST (第 1.1 节) 。使用语法来定义编程语言的抽象语法 (第 1.2 节)，使用模式匹配来检查 AST 中的单个节点 (第 1.3 节) 。使用递归函数来构造和解构 AST (第 1.4 节) 。本章简要介绍这些概念。

1.1 抽象语法树和 Racket 结构

编译器使用抽象语法树来表示程序，因为他们经常需要问这样的问题：对于程序的给定部分，它是什么样的语言特性? 它的子部分是什么? 考虑左边的程序和右边的 AST 。这个程序是一个加法操作，它有两个子部分，一个读操作和一个求反。否定还有另一个子部分，整数常数 `8` 。通过使用树来表示程序，可以很容易地按照链接从程序的一个部分走到它的子部分。

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



(1.1)

使用树的标准术语来描述 AST：上面的每个圆称为节点。箭头将节点连接到它的 子节点 (它们也是节点)。最上面的节点是 根节点。除了根节点之外，每个节点都有其父节点 (其父节点是子节点)。如果一个节点没有子节点，它就是一个叶节点。否则就是内部节点。

为每种节点定义一个 Racket `struct`。本章只需要两种节点：一种用于整型常量，另一种用于基本操作。下面是整型常量的 `struct` 的定义。

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

整数节点只包括一个东西：整数值。为整数 8 创建 AST 节点，写下 `(Int 8)`。

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

`(Int 8)` 创建的值是 `Int` 结构体的一个实例。

下面是原语操作的 `struct` 定义。

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

一个原语操作节点包括一个操作符符号 `op` 和一组子 `args`。例如，要创建一个对数字 8 求反的 AST，可以这样写 `(Prim '- (list eight))`。

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

原始操作可以有 0 个或多个子操作。`read` 操作符有 0 个子操作符：

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

而加法运算符有两个子操作符：

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

我们对 `Prim` 的结构做出设计选择。与其为许多不同的操作 (`read` 、`+` 和 `-`) 使用一个结构，不如为每个操作定义一个结构，如下所示。

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

选择只使用一种结构的原因是，在编译器的许多部分中，不同原语操作符的代码是相同的，所以我们不妨只编写一次代码，这是通过使用单一结构来实现的。

当编译像 (1.1) 这样的程序时，需要知道与根节点相关的操作是加法，并且要能访问它的两个子节点。Racket 提供模式匹配来支持这类查询，如在 1.3 节中看到的。

在这本书中，通常写下程序的具体语法，即使我们脑子里已经有 AST，因为具体语法更简洁。我们建议，在头脑中，始终把程序看作是抽象语法树。

1.2 语法

编程语言可以被看作是程序的集合。集合通常是无限的 (人们总是可以创建越来越大的程序)，所以不能简单地通过列出语言中的所有程序来描述一种语言。相反，我们写下一套规则，一套 *语法*，来构建程序。语法通常用来定义一种语言的具体语法，但它们也可以用来描述抽象语法。我们用 Backus-Naur 形式 (BNF) [9, 75] 。例如，描述一种名为 $R_{\texttt{Int}}$ 的小语言，它由整数和算术运算组成。

$R_{\texttt{Int}}$ 的抽象语法的第一个语法规则是 `Int` 结构的一个实例是一个表达式：

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

每条规则都有左手边和右手边。读取规则的方法是，如果有一个与右手边匹配的 AST 节点，那么可以根据左手边对它进行分类。语法规则定义的名称，如 *exp* ，是*非终端符*。名称 *int* 也是一个非终端符，但我们没有使用语法规则来定义它，而是使用以下解释来定义它。我们做出简化设计的决定，即本书中所有的语言都只处理机器可表示的整数。在大多数现代机器上，这对应

于用 64 位表示的整数，也就是 -2^{63} 到 $2^{63}-1$ 的范围。我们进一步限制这个范围，以匹配 Racket `fixnum` 数据类型，它允许 64 位机器上的 63 位整数。因此，*int* 是一个十进制序列 (0 到 9)，可能从 − (对于负整数) 开始，因此十进制序列表示范围为 -2^{62} 到 $2^{62}-1$ 的整数。

第二条语法规则是 `read` 操作，它从程序的用户那里接收一个输入整数。

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

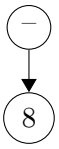
第三条规则是，给定一个 *exp* 节点，该节点的否定也是一个 *exp* 。

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

打字机字体中的符号，如 `-` 和 `read` ，都是 *终端*符号，必须按字面意思出现在程序中，规则才能适用。

可以用这些规则对 $R_{\texttt{Int}}$ 语言中的 AST 进行分类。例如，根据规则 (1.2) `(Int 8)` 是 *exp* ，那么根据规则 (1.4) 下面的 AST 是 *exp* 。

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



(1.5)

下一条语法规则是关于加法表达式的：

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

现在可以证明 AST (1.1) 是 $R_{\texttt{Int}}$ 中的 *exp* 。我们知道 `(Prim 'read '())` 是规则 (1.3) 的 *exp* ，并且已经将 `(Prim '- (list (Int 8)))` 归类为 *exp* ，根据规则 (1.6) 表明

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

是 $R_{\texttt{Int}}$ 语言中的 *exp* 。

如果有一个不适用上述规则的 AST ，那么这个 AST 就不是 $R_{\texttt{Int}}$ 。例如，程序 `(- (read) (+ 8))` 不在 $R_{\texttt{Int}}$ 中，因为对于只有一个参数的 `+` 有规则，对于有两个参数的 `-` 也没有规则。当我们用语法定义一种语言时，这种语言只包括那些符合规则的程序。

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

图 1.1: $R_{\texttt{Int}}$ 的具体语法。

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

图 1.2: $R_{\texttt{Int}}$ 的抽象语法。

$R_{\texttt{Int}}$ 的最后一条语法规则是有一个 `Program` 节点来标记整个程序的顶部:

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

`Program` 结构的定义如下

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

这里的 `body` 是一个表达式。在后面的章节中，`info` 部分将被用来存储辅助信息，但目前它只是一个空列表。

很多语法规则左手边相同，右手边不同，这是很常见的，比如 $R_{\texttt{Int}}$ 语法中的 *exp* 规则。作为一种简写，竖条可以用来将几个右手边组合成一条规则。

我们收集图 1.2 中 $R_{\texttt{Int}}$ 抽象语法的所有语法规则。$R_{\texttt{Int}}$ 的具体语法在图 1.1 中定义。

支持代码 `utilities.rkt` 中提供的 `read-program` 函数从文件 (Racket 具体语法中的字符序列) 中读取程序，并将其解析为一个抽象语法树。更多细节请参阅附录 12.2 中对 `read-program` 的描述。

1.3 模式匹配

正如 1.1 节中提到的，编译器经常需要访问 AST 节点的各个部分。Racket 提供 `match` 形式，以访问一个结构的各个部分。考虑下面的示例和

右边的输出。

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

在上面的例子中 match 形式接受一个 AST (1.1)，并将它的各个部分绑定到三个模式变量 op 、child1 和 child2 ，然后打印出操作符。一般来说，一个匹配子句由 *pattern* 和 *body* 组成。模式被递归地定义为模式变量、结构名后面跟着每个结构参数的模式，或者 S-表达式 (符号、列表等)。(关于 match 的完整描述，请参阅《Racket 指南》[1] 第 12 章和《Racket 参考》[2] 第 9 章。) 匹配子句的正文可以包含任意的 Racket 代码。模式变量可以在主体的作用域中使用，例如 (print op) 中的 op 。

一个 match 形式可能包含几个子句，如下面的函数 leaf? 识别 $R_{\texttt{Int}}$ 节点何时是 AST 中的叶子。match 按顺序遍历子句，检查模式是否能匹配输入的 AST 。匹配的第一个子句的主体将被执行。右侧显示多个 AST 的 leaf? 的输出。

```
(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
```

写 match 时, 我们指的是用语法定义来确定预匹配哪些非终结符, 然后我们确保: 1) 对于非终结符的每个选项, 都有一个子句; 2) 每个子句中的模式对应于语法规则的右手边。对于 leaf? 函数中的 match 我们参考图 1.2

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

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

中的 $R_{\texttt{Int}}$ 语法。*exp* 非终结符有 4 个选项，所以 `match` 有 4 个子句。每个子句中的模式对应于语法规则的右侧。例如，模式 `(Prim '+ (list e1 e2))` 对应右边的 `(Prim + (`*exp exp*`))` 。当从语法转换为模式时，用选择的模式变量 (例如 `e1` and `e2`) 替换非终结符，例如 *exp* 。

1.4 递归函数

程序本质上是递归的。例如，$R_{\texttt{Int}}$ 表达式通常由更小的表达式组成。因此，处理整个程序的自然方法是使用递归函数。作为这种递归函数的第一个例子，我们定义 `exp?` ，它取一个任意的值，并且判定它是否是一个 $R_{\texttt{Int}}$ 表达式。当使用与语法相对应的匹配子句序列定义函数时，我们说函数是通过*结构递归*定义的，并且每个子句的主体对每个子节点进行递归调用。[3] 下面还定义第二个函数，名为 `Rint?` ，它确定 AST 是否是一个 $R_{\texttt{Int}}$ 程序。一般来说，可以编写一个递归函数来处理语法中的每个非终结符。

[3]在 *How to Design Programs* `http://www.ccs.neu.edu/home/matthias/HtDP2e/`中提倡了根据数据定义来构造代码的这一原理。

```
(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))
(Rint? (Program '()
      (Prim '- (list (Prim 'read '())
                     (Prim '+ (list (Num 8)))))))
```

```
#t
#f
```

你可能会试图将两个函数合并为一个，像这样：

```
(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]))
```

有时这样的技巧可以节省几行代码，特别是当涉及到 `Program` 包装器时。然而，这种风格通常 不推荐使用，因为它会带来麻烦。例如，上面的函数微妙地错误：`(Rint? (Program '() (Program '() (Int 3))))` 应该返回 false 却返回 true 。

```
(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)]))
```

图 1.3: $R_{\texttt{Int}}$ 语言的解释器。

1.5 解释器

一般来说，程序的预期行为是由语言规范来定义的。例如，Scheme 语言在报告中由 [104] 定义。Racket 语言在它的参考手册 [45] 中有定义。在本书中，使用解释器来指定我们所考虑的每种语言。被指定为语言定义的解释器称为 定义解释器 [95] 。通过为 $R_{\texttt{Int}}$ 语言创建定义解释器进行预热，这是结构递归的第二个示例。`interp-Rint` 函数在图 1.3 中定义。函数体对输入程序进行匹配，然后对 `interp-exp` 辅助函数进行调用，该函数对每个 $R_{\texttt{Int}}$ 表达式的语法规则都有一个匹配子句。

让我们解释几个 $R_{\texttt{Int}}$ 程序的结果。下面的程序将两个整数相加。

```
(+ 10 32)
```

结果是 42 , 生命、宇宙和一切的答案：42![4]. 用具体的语法编写上面的程序，而解析的抽象语法是：

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

下一个示例演示表达式可以相互嵌套，在本例中嵌套几个加法和否定。

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

上述程序的结果是什么?

正如前面提到的，$R_{\texttt{Int}}$ 语言不支持任意大的整数，而只支持 63 位整数，因此我们在 Racket 中使用 fixnum 算术来解释 $R_{\texttt{Int}}$ 的算术操作。假设

$$n = 999999999999999999$$

它确实适合 63 位。当我们在解释器中运行以下程序时会发生什么?

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

它会产生一个错误:

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

我们建立这样一种约定: 如果在程序上运行定义解释器产生一个错误，那么该程序的意义是 不确定的 ，除非该错误是一个 trapped-error 。该语言的编译器没有义务关注具有未指定行为的程序; 它不必生成可执行文件，如果生成，可执行文件就可以做任何事情。另一方面，如果错误是一个 trapped-error ，那么编译器必须生成一个可执行文件，并且必须报告错误发生的情况。要发出错误信号，退出时返回代码为 255 。第 8 章和第 10 章的解释器使用 trapped-error 。

接下来是 $R_{\texttt{Int}}$ 语言的最后一个特性，read 操作提示程序的用户是否需要一个整数。回想一下程序 (1.1) 执行一个 read ，然后减去 8 。所以如果我们运行

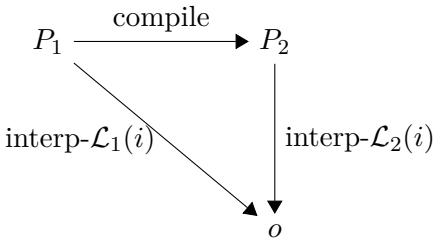
```
(interp-Rint (Program '() ast1.1))
```

如果输入是 50 ，结果是 42 。

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

我们将 read 操作包含在 $R_{\texttt{Int}}$ 中，因此聪明的学生无法为 $R_{\texttt{Int}}$ 实现编译器，该编译器仅在编译期间运行解释器以获取输出，然后生成琐碎的代码来生成输出。(是的，一个聪明的学生在本课程的第一部分中做到了这一点。)

编译器的工作是将一种语言的程序翻译成另一种语言的程序，使输出程序的行为与输入程序的行为相同。这个想法如下图所示。假设有两种语言，$\mathcal{L}_1$ 和 $\mathcal{L}_2$，每种语言都有一个定义解释器。给定一个编译器将语言 $\mathcal{L}_1$ 翻译成语言 $\mathcal{L}_2$，并且给定语言 $\mathcal{L}_1$ 的任何程序 P_1，编译器必须将其翻译成程序 P_2，以便在各自的解释器上输入相同的 i 来解释 P_1 和 P_2，产生相同的输出 o。



(1.7)

在下一节中，将看到第一个编译器示例。

1.6 编译器示例: 部分求值器

在本节中，考虑一种编译器，它可以将 $R_{\texttt{Int}}$ 程序翻译成更高效的 $R_{\texttt{Int}}$ 程序，也就是说，这种编译器是一种优化器。这个优化器急切地计算程序中不依赖于任何输入的部分，这个过程称为 *部分求值* [65]。例如，给定如下程序

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

我们的编译器将把它翻译成程序

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

图 1.4 给出 $R_{\texttt{Int}}$ 语言的一个简单的部分求值程序的代码。部分求值器的输出是一个 $R_{\texttt{Int}}$ 程序。在图 1.4中，pe-exp 函数捕获关于 *exp* 的结构递归，而部分计算否定和加法运算的代码则被分解成两个独立的辅助函数: pe-neg 和 pe-add。这些辅助函数的输入是对子函数部分求值的输出。

```
(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))]))
```

图 1.4: $R_{\texttt{Int}}$ 的部分求值程序。

`pe-neg` 和 `pe-add` 函数检查它们的参数是否为整数，如果是，则执行适当的算术。否则，它们将为算术操作创建一个 AST 节点。

为了获得部分求值器是正确的一些信心，可以测试它是否生成与输入程序得到相同结果的程序。也就是说，可以测试它是否满足图 1.7 。下面的代码在几个示例上运行部分求值程序，并测试输出程序。`parse-program` 和 `assert` 函数在附录 12.2中定义。

```
(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

整数和变量

本章是关于编译 Racket 的子集到 x86-64 汇编代码 [63] 。名为 $R_{\texttt{Var}}$ 的子集包括整数算术和局部变量绑定。通常把 x86-64 简单地称为 x86。本章首先介绍 $R_{\texttt{Var}}$ 语言（2.1 节），然后介绍 x86 汇编语言（2.2 节）。x86 汇编语言非常大，只讨论编译 $R_{\texttt{Var}}$ 所需的指令。我们将在后面的章节中介绍更多的 x86 指令。在介绍 $R_{\texttt{Var}}$ 和 x86 之后，我们反思它们的区别，并列出一个计划，将 $R_{\texttt{Var}}$ 到 x86 的转换分解为几个步骤（2.3 节)。本章的其余部分给出每个步骤的详细提示（2.4 至 2.9 节)。我们希望给出足够的提示，让准备充分的读者和一些朋友能够在几周内实现从 $R_{\texttt{Var}}$ 到 x86 的编译器。为了让读者了解第一个编译器的规模，$R_{\texttt{Var}}$ 编译器的指导解决方案大约是 500 行代码。

2.1 $R_{\texttt{Var}}$ 语言

$R_{\texttt{Var}}$ 语言通过变量定义扩展 $R_{\texttt{Int}}$ 语言。$R_{\texttt{Var}}$ 语言的具体语法由图 2.1 中的语法定义，抽象语法由图 2.2 定义。非终端 *var* 可以是任何 Racket 标识符。和 $R_{\texttt{Int}}$ 一样，`read` 是一个空操作符，`-` 是一个一元操作符，`+` 是一个二元操作符。与 $R_{\texttt{Int}}$ 类似，$R_{\texttt{Var}}$ 的抽象语法包含标记程序顶部的 `Program` 结构体。尽管 $R_{\texttt{Var}}$ 语言非常简单，但它的丰富程度足以展示几种编译技术。

让我们进一步研究 $R_{\texttt{Var}}$ 语言的语法和语义。`let` 特性定义一个变量，

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

图 2.1: $R_{\texttt{Var}}$ 的具体语法。

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

图 2.2: $R_{\texttt{Var}}$ 的抽象语法。

以便在其主体中使用，并使用表达式的值初始化该变量。`let` 的抽象语法在图 2.2 中定义。`let` 的具体语法是

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

例如，下面的程序将 `x` 初始化为 32 ，然后对 `(+ 10 x)` 求值，生成 42 。

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

当同一个变量有多个 `let` 时，使用最接近的 `let` 。也就是说，变量定义掩盖先前的定义。考虑下面的程序，它有两个 `let` ，定义名为 `x` 的变量。你能算出结果吗?

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

为了描述哪个变量的使用对应于哪个定义，下面显示用下标标注的 `x` ，以区分它们。仔细检查您对上述问题的回答是否与本程序注释版本的回答相同。

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

初始化表达式总是在 `let` 函数体求值之前，因此如下 `x` 的 `read` 函数会在 `y` 的 `read` 函数体之前执行。给定输入 52 ，然后是 10 ，下面的结果是 42 (不是 -42)。

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

2.1.1 通过方法覆盖的可扩展解释器

为了准备讨论 $R_{\texttt{Var}}$ 的解释器，需要解释为什么选择使用面向对象编程实现解释器，也就是说，将其作为类中的方法集合。在本书中，定义许多解释器，学习的每种语言都有一个解释器。因为每种语言都建立在前一种语言的基础上，所以它们的解释器之间有很多共性。这些公共部分只需要写下一次而不是很多次。例如，一种简单的方法是让 $R_{\texttt{If}}$ 的解释器处理该语言的所有新特性，然后有一个默认的 case 分派给 $R_{\texttt{Var}}$ 的解释器。下面的代码说明这个想法。

```
(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)]))
```

这种方法的问题是，它不能处理 $R_{\texttt{If}}$ 特性，如 `If` 嵌套在 $R_{\texttt{Var}}$ 的情况；就像下面的程序中的 `-` 操作符一样。

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

如果我们在这个程序上调用 `interp-Rif` ，它会分派给 `interp-Rvar` 来处理 `-` 运算符，但是它会在 `-` 的参数（即一个 `If` ）上再次递归地调用 `interp-Rvar` 。但是在 `interp-Rvar` 中没有 `If` 的情况，所以得到一个错误!

为了使解释器具有可扩展性，我们需要一种称为 *开放递归* 的方法，在这种方法中，递归结的捆绑延迟到函数组合时进行。面向对象的语言通过覆盖方法的后期绑定提供开放递归。下面的代码概述使用 Racket 的 `class` 特征来解释 $R_{\texttt{Var}}$ 和 $R_{\texttt{If}}$ 的想法。我们为每种语言定义一个类，并定义一个方法来解释每个类中的表达式。$R_{\texttt{If}}$ 的类继承自 $R_{\texttt{Var}}$ 的类，$R_{\texttt{If}}$ 中的 `interp-exp` 方法覆盖 $R_{\texttt{Var}}$ 中的 `interp-exp` 方法。注意，$R_{\texttt{If}}$ 中 `interp-exp` 的默认情况是使用 `super` 来调用 `interp-exp` ，并且由于 $R_{\texttt{If}}$ 继承自 $R_{\texttt{Var}}$ ，因此将分派到 $R_{\texttt{Var}}$ 中的 `interp-exp` 。

```
(define interp-Rvar-class
  (class object%
    (define/public (interp-exp e)
      (match e
        [(Prim '- (list e))
         (fx- 0 (interp-exp e))]
        ...))
    ...))
```

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

回到那个麻烦的例子，在这里重复：

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

可以在这个表达式上调用 $R_{\texttt{If}}$ 的 `interp-exp` 方法，方法是创建 $R_{\texttt{If}}$ 类的一个对象，并将 `interp-exp` 方法和参数 `e0` 发送给它。

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

$R_{\texttt{If}}$ 中的 `interp-exp` 的默认情况是通过分配给 $R_{\texttt{Var}}$ 中的 `interp-exp` 方法来处理它，该方法处理 `-` 操作符。但是对于递归方法调用，它将分派回 $R_{\texttt{If}}$ 中的 `interp-exp` ，在那里 `If` 被正确处理。因此，方法覆盖，提供以可扩展的方式实现解释器所需的开放递归。

2.1.2 $R_{\texttt{Var}}$ 的定义解释器

在证明如何使用类和方法来实现解释器之后，我们转向图 2.3 中 $R_{\texttt{Var}}$ 的定义解释器。它与 $R_{\texttt{Int}}$ 的解释器相似，但是为变量和 `let` 添加两个新的 `match` 用例。对于 `let`，我们需要一种方式来将绑定到变量的值传递给变量的所有引用。为了实现这一点，我们维护一个从变量到值的映射。在整个编译过程中，我们经常需要将变量映射到有关它们的信息。将这些映射称为 环境 。[1] 为简单起见，使用关联列表 (alist) 来表示环境。右边的侧边栏简要介绍 alists 和 `racket/dict` 包。`interp-exp` 函数将当前环境 `env` 作为一个额外参数。当解释器遇到一个变量时，它会使用 `dict-ref` 函数找到相应的值。当解释器遇到 `Let` 时，它计算初始表达式，使用 `dict-set` 将结果值绑定到变量，扩展环境，然后计算 `Let` 的主体。

作为字典的关联列表

关联列表 (alist) 是键-值对的列表。例如，可以通过一个列表来映射人们的年龄。

```
(define ages
  '((jane . 25) (sam . 24) (kate . 45)))
```

词典接口用于将键映射到值。每个列表都实现这个接口。`racket/dict` 包提供许多使用字典的功能。以下是其中的一些：

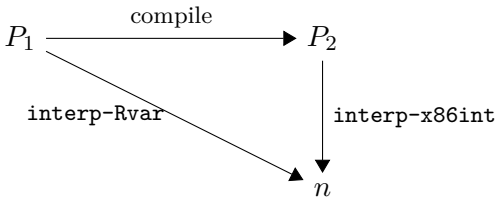
(dict-ref *dict key*) 返回与给定 *key* 相关联的值。

(dict-set *dict key val*) 返回一个新的字典，它将 *key* 映射到 *val* ，但在其他方面与 *dict* 相同。

(in-dict *dict*) 返回 *dict* 中的键和值的 序列 。例如，下面的代码创建一个新列表，其中年龄是递增的。

```
(for/list ([(k v) (in-dict ages)])
  (cons k (add1 v)))
```

本章的目标是实现一个编译器，它可以将任何用 $R_{\texttt{Var}}$ 语言编写的程序 P_1 翻译成一个 x86 汇编程序 P_2 ，使 P_2 在计算机上运行时表现出与通过 `interp-Rvar` 解释的 P_1 程序相同的行为。也就是说，它们输出的是相同的整数 n 。在下图中描述这个正确性标准。



在下一节中，介绍 x86 的 $\text{x86}_{\texttt{Int}}$ 子集，它足以编译 $R_{\texttt{Var}}$ 。

[1] 在编译器文献中，环境的另一个常见术语是 符号表 。

```
(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))
```

图 2.3: $R_{\texttt{Var}}$ 语言的解释器。

reg	::=	`rsp` | `rbp` | `rax` | `rbx` | `rcx` | `rdx` | `rsi` | `rdi` |
		`r8` | `r9` | `r10` | `r11` | `r12` | `r13` | `r14` | `r15`
arg	::=	`$`*int* | `%`*reg* | *int*`(%`*reg*`)`
instr	::=	`addq` *arg*`,`*arg* | `subq` *arg*`,`*arg* | `negq` *arg* | `movq` *arg*`,`*arg* |
		`callq` *label* | `pushq` *arg* | `popq` *arg* | `retq` | `jmp` *label*
		label`:` *instr*
x86$_{\text{Int}}$	::=	`.globl main`
		`main:` *instr*...

图 2.4: x86$_{\text{Int}}$ 汇编语言的语法 (AT&T 语法)。

2.2 x86$_{\text{Int}}$ 汇编语言

图 2.4 定义 x86$_{\text{Int}}$ 的具体语法。我们使用 GNU 汇编程序所期望的 AT&T 语法。程序以 `main` 标签开始，后面跟着一系列指令。`globl` 指令说 `main` 过程是外部可见的，这是操作系统可以调用它的必要条件。在语法中，省略如 ... 用于指示一个项目序列，例如，*instr*... 是指令序列。x86 程序存储在计算机的内存中。就我们的目的而言，计算机的内存是 64 位地址到 64 位值的映射。计算机在 `rip` 寄存器中存储一个 *程序计数器* (PC)，它指向下一条要执行的指令的地址。对于大多数指令，程序计数器在指令执行后递增，因此它指向内存中的下一条指令。大多数 x86 指令有两个操作数，每个操作数要么是一个整型常量 (称为 *立即数*)，要么是一个 *寄存器*，要么是一个内存位置。

寄存器是一种特殊的变量。每一个都有一个 64 位的值；计算机中有 16 个通用寄存器，它们的名称在图 2.4 中给出。写入寄存器时，`%` 后跟寄存器名，比如 `%rax`。

立即数使用符号 `$`n 写入，其中 n 是整数。对内存的访问是使用语法 n(`%`r) 指定的，它获得存储在寄存器 r 中的地址，然后给地址加 n 个字节。生成的地址用于加载或存储到内存中，这取决于它是作为指令的源参数还是目标参数出现。

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

图 2.5: 一个相当于 (+ 10 32) 的 x86 程序。

算术指令，比如 addq s, d 从源 s 和目标 d 读取数据，应用算术运算，然后将结果写回目标 d 。移动指令 movq s, d 从 s 中读取并将结果存储在 d 中。callq *label* 指令跳转到由 label 指定的过程，而 retq 则从过程返回到它的调用者。在本章后面和第 6 章中更详细地讨论过程调用。指令 jmp *label* 将程序计数器更新到指定 label 之后的指令地址。

附录 12.3 包含这本书中使用的所有 x86 指令的快速参考。

图 2.5 描述一个相当于 (+ 10 32) 的 x86 程序。指令 movq $10, %rax 将 10 放入寄存器 rax ，然后 addq $32, %rax 将 32 加到 rax 中的 10 ，并将结果 42 放回 rax 。最后一条指令，retq ，通过将 rax 中的整数返回给操作系统来完成 main 函数。操作系统将这个整数解释为程序的退出码。按照惯例，退出码为 0 表示程序成功完成，所有其他退出码表示各种错误。然而，在本书中，返回程序的结果作为退出码。

x86 汇编语言有几种不同的方式，这取决于它是在什么操作系统中组装的。这里显示的代码示例在 Linux 和大多数类 unix 平台上是正确的，但在 Mac OS X 上组装时，像 main 这样的标签必须加上下划线前缀，就像 _main 一样。

在下一个示例中，演示如何使用内存存储中间结果。图 2.6 列出一个相当于 (+ 52 (- 10)) 的 x86 程序。这个程序使用一个称为 过程调用栈 (或简称 栈) 的内存区域。 堆栈由每个过程调用的独立 帧 组成。单个帧的内存布局如图 2.7 所示。寄存器 rsp 被称为 堆栈指针 ，它指向堆栈顶部的项。堆栈在内存中向下增长，通过减去堆栈指针来增加堆栈的大小。在过程调用的上下文中，返回地址 是调用方调用指令之后的指令。函数调用指令 callq 在跳转到过程之前将返回地址压入堆栈。寄存器 rbp 是 基指针 ，用于访问

```
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
```

图 2.6: 相当于 (+ 52 (- 10)) 的 x86 程序。

存储在当前过程调用框架中的变量。调用者的基指针在返回地址之后被压入堆栈，然后基指针被设置为旧基指针的位置。在图 2.7中，对变量编号，从 1 到 n 。变量 1 存储在地址 -8(%rbp) ，变量 2 存储在地址 -16(%rbp) ，等等。

回到图 2.6 中的程序，考虑控制是如何从操作系统转移到 main 函数的。操作系统发出一个 callq main 指令，将其返回地址推送到堆栈上，然后跳转到 main 。在 x86-64 中，堆栈指针 rsp 在执行任何 callq 指令之前必须被 16 个字节整除，因此当控制到达 main 时，rsp 偏离对齐 8 个字节 (因为 callq 推入返回地址)。前三条指令是一个程序的典型 前奏 。指令 pushq %rbp 将调用者的基指针保存到堆栈上，并从堆栈指针中减去 8 。第二个指令 movq %rsp, %rbp 改变基指针，使其指向旧基指针的位置。指令 subq $16, %rsp 将堆栈指针向下移动，以便为存储变量留出足够的空间。这个程序需

位置	内容
8(%rbp)	return address
0(%rbp)	old rbp
-8(%rbp)	variable 1
-16(%rbp)	variable 2
...	...
0(%rsp)	variable n

图 2.7: 帧的内存布局。

要一个变量 (8 个字节)，但是四舍六入到 16 个字节，这样 rsp 是 16 字节对齐的，就可以调用其他函数。前奏的最后一条指令是 jmp start，它将控制转移到由 Racket 表达式 (+ 52 (- 10)) 生成的指令。

start 标签下的第一条指令是 movq $10, -8(%rbp)，它将 10 存储在变量 1 中。指令 negq -8(%rbp) 将变量 1 更改为 -10 。下一条指令将变量 1 中的 -10 移动到 rax 寄存器中。最后，addq $52, %rax 将 52 添加到 rax 中的值，将其值更新为 42 。

conclusion 标签下的三个指令是一个程序的典型 结论 。前两条指令将 rsp 和 rbp 寄存器恢复到过程开始时的状态。指令 addq $16, %rsp 将栈指针移回指向旧的基指针。然后 popq %rbp 返回 rbp 的旧基指针，并在堆栈指针上加 8 。最后一条指令 retq 返回到调用它的过程，并将 8 添加到堆栈指针。

编译器需要一个方便的表示来操作 x86 程序，所以在图 2.8 中为 x86 定义一个抽象语法。将这种语言称为 $\text{x86}_{\texttt{Int}}$ 。与 $\text{x86}_{\texttt{Int}}$ (图 2.4) 的具体语法相比，主要区别在于不允许在每个指令前都添加标签。相反，指令被分组成块 ，每个块都有一个标签，这就是为什么 X86Program 结构包含一个列表，将标签映射到块。在第 4 章中，当我们引入条件分支时，这种组织的原因就变得很明显。Block 结构包含一个 *info* 字段，这个字段本章不需要，但在第 3 章会很有用。现在，*info* 字段应该包含一个空列表。同样，关于 callq 的抽象语法，Callq 结构包括一个整数，代表函数的有效性，即参数的数量，这在寄存器分配过程中会很有帮助（第 3 章）。

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*`...))`
$\text{x86}_{\texttt{Int}}$	::=	`(X86Program` *info* `((`*label* `.` *block*`)...))`

图 2.8: $\text{x86}_{\texttt{Int}}$ 程序集的抽象语法。

2.3 通过 $C_{\texttt{Var}}$ 语言计划 x86 之旅

要将一种语言编译成另一种语言，关注两种语言之间的差异会有所帮助，因为编译器需要弥合这些差异。$R_{\texttt{Var}}$ 和 x86 汇编有什么区别? 以下是一些最重要的:

(a) x86 算术指令通常有两个参数，并在适当的地方更新第二个参数。相反，$R_{\texttt{Var}}$ 算术操作接受两个参数并产生一个新值。一个 x86 指令最多可以有一个内存访问参数。此外，一些指令对它们的参数进行特殊的限制。

(b) $R_{\texttt{Var}}$ 操作符的参数可以是一个嵌套很深的表达式，而 x86 指令将它们的参数限制为整数、常量、寄存器和内存地址。

(c) x86 中的执行顺序在语法中是明确的：指令序列并跳转到标记的位置，而在 $R_{\texttt{Var}}$ 中，评估顺序是抽象语法树从左到右的深度优先遍历。

(d) $R_{\texttt{Var}}$ 中的程序可以有任意数量的变量，而 x86 有 16 个寄存器和过程调用堆栈。

(e) $R_{\texttt{Var}}$ 中的变量可以覆盖同名的其他变量。在 x86 中，寄存器有唯一的名称，内存位置有唯一的地址。

通过将问题分解为几个步骤，逐个处理上面的差异，减轻从 $R_{\texttt{Var}}$ 编译到 x86 的挑战。每一个步骤都被称为编译器的 通道。 这个术语来自于程序的每个步骤通过 AST 的方式。首先勾画出如何实现每个通道，并给它们命名。然后，计算出通道的顺序以及每个通道的输入/输出语言。第一个通道使用 $R_{\texttt{Var}}$ 作为输入语言，最后一个使用 $\text{x86}_{\texttt{Int}}$ 作为输出语言。在这两者之间，可以选择最方便表示每个通道的输出语言，无论是 $R_{\texttt{Var}}$ 、$\text{x86}_{\texttt{Int}}$ ，还是我们设计的新 中间语言。最后，要实现每个通道，需要在通道输入语言的语法中为每个非终端编写一个递归函数。

select-instructions 处理 $R_{\texttt{Var}}$ 操作和 x86 指令之间的区别。这个通道将每个 $R_{\texttt{Var}}$ 操作转换为完成相同任务的短指令序列。

remove-complex-opera* 确保原语操作的每个子表达式都是变量或整数，即 原子表达式。将非原子表达式称为 复合表达式。这一通道引入临时变量来保存复合子表达式的结果。[2]

explicate-control 使程序以显式方式执行。它将抽象语法树表示形式转换为控制流图，其中每个节点包含一系列语句，节点之间的边缘表示哪些节点包含到其他节点的跳转。

assign-homes 用 x86 中的寄存器或堆栈地址替换 $R_{\texttt{Var}}$ 中的变量。

uniquify 通过为每个变量重命名一个唯一的名称来处理变量的隐藏。

下一个问题是: 应该以什么顺序应用这些通道? 这个问题很有挑战性，因为很难提前知道哪种排序更好 (更容易实现，生成更高效的代码，等等)，所以经常需要反复试验。不过，可以试着提前计划，在排序方面做出明智的选择。

相对于 `uniquify` , `explicate-control` 的顺序应该是什么? `uniquify` 应该放在第一位，因为 `explicate-control` 将所有 `let` 绑定的变量都变成作用域为整个程序的局部变量,这将使同名的变量产生混淆。将 `remove-complex-opera*`

[2]操作的子表达式通常被称为操作符和操作数，这就解释为什么在这个通道中会有 `opera*` 。

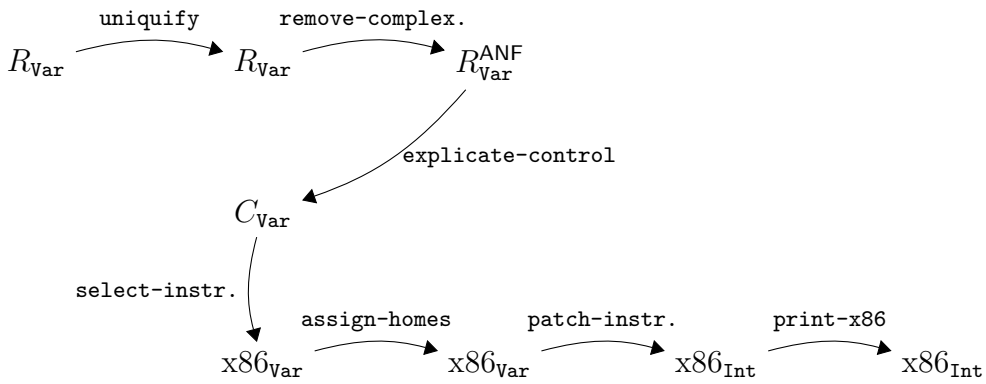


图 2.9: 编译 $R_{\texttt{Var}}$ 的通道示意图

放在 `explicate-control` 之前,因为后者删除 `let` 形式,但在 `remove-complex-opera*` 的输出中使用 `let` 是很方便的。`uniquify` 相对于 `remove-complex-opera*` 的顺序无关紧要，所以我们随意选择 `uniquify` 排在第一位。

最后考虑 `select-instructions` 和 `assign-homes` 。这两个通道是交织在一起的。在第 6 章中，了解到，在 x86 中，寄存器是用来传递参数给函数的，最好是将参数赋值给相应的寄存器。另一方面，通过先选择指令，可能会在 `assign-homes` 中陷入死胡同。回想一下，x86 指令只有一个参数可能是内存访问，但是 `assign-homes` 可能无法将它们中的任何一个分配到寄存器。一种复杂的方法是迭代地重复两个通道，直到找到解决方案。然而，为了降低实现的复杂性，推荐一种更简单的方法，首先是 `select-instructions` ，然后是 `assign-homes` ，然后是第三个名为 `patch-instructions` 的方法，它使用一个保留的寄存器来修复未解决的问题。

图 2.9 显示编译器通道的顺序，并标识每个通道的输入和输出语言。最后一个通道，`print-x86` ，将 $\text{x86}_{\texttt{Int}}$ 的抽象语法转换为具体语法。在接下来的两节中，将讨论 $C_{\texttt{Var}}$ 中间语言和 x86 的 $\text{x86}_{\texttt{Var}}$ 方言。本章的其余部分给出图 2.9 中关于每个编译器通道的实现的提示。

2.3.1 $C_{\texttt{Var}}$ 中间语言

`explicate-control` 的输出类似于 C 语言 [72] ，它为表达式和语句提供单独的语法类别，因此将其命名为 $C_{\texttt{Var}}$ 。图 2.10 定义 $C_{\texttt{Var}}$ 的抽象语法。

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

图 2.10: 中间语言 $C_{\texttt{Var}}$ 的抽象语法。

($C_{\texttt{Var}}$ 的具体语法见附录，图 12.2 。) $C_{\texttt{Var}}$ 语言支持与 $R_{\texttt{Var}}$ 相同的操作符，但操作符的参数仅限于原子表达式。代替 `let` 表达式，$C_{\texttt{Var}}$ 有赋值语句，可以使用 `Seq` 形式按顺序执行。语句序列总是以 `Return` 结尾，这是 *tail* 语法规则中的保证。这个非终端符的命名来自术语*尾部位置* ，它指的是函数中最后一个执行的表达式。

一个 $C_{\texttt{Var}}$ 程序由一个控制流图组成，表示为一个映射标签到尾部的列表。这对于本章来说是不必要的，因为我们还没有引入跳转到标签的 `goto` ，但它节省我们在第 4 章中更改语法的时间。现在只有一个标签 `start` ，整个程序就是它的尾部。`CProgram` 形式的 *info* 字段，在 `explicate-control` 传递之后，包含从符号 `locals` 到变量列表的映射，也就是程序中使用的所有变量列表。在程序开始时，这些变量未初始化；它们在第一次赋值时开始初始化。

$C_{\texttt{Var}}$ 的定义解释器在支持代码 `interp-Cvar.rkt` 文件中。

2.3.2 $\text{x86}_{\texttt{Var}}$ 方言

$\text{x86}_{\texttt{Var}}$ 语言是通道 `select-instructions` 的输出。它用无数个程序作用域变量扩展 $\text{x86}_{\texttt{Int}}$ ，并消除关于指令参数的限制。

2.4 Uniquify 变量

`uniquify` 通道将 $R_{\texttt{Var}}$ 程序编译为 $R_{\texttt{Var}}$ 程序，在 $R_{\texttt{Var}}$ 程序中，每个 `let` 都绑定一个唯一的变量名。例如，`uniquify` 通道应该将左边的程序转

换为右边的程序。

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

下面是另一个翻译示例，这一次是一个程序，在另一个 `let` 的初始化表达式中嵌套了一个 `let` 。

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

建议通过创建一个名为 `uniquify-exp` 的结构递归函数来实现 `uniquify` ，该函数主要只是复制一个表达式。但是，当遇到 `let` 时，它应该为变量生成唯一的名称，并将旧名称与列表中的新名称关联起来。[3] 当 `uniquify-exp` 函数得到一个变量引用时，需要访问这个列表，因此为列表的 `uniquify-exp` 添加一个参数。

`uniquify-exp` 函数的框架如图 2.11 所示。该函数经过咖喱处理，因此可以方便地将其部分应用于一个列表，然后将其应用于不同的表达式，如图 2.11 中最后一种原始操作所示。Racket 的 `for/list` 形式对于转换列表的每个元素以生成新列表很有用。

Exercise 1. 通过填写图 2.11 中的空白来完成 `uniquify` 传递，即在支持代码的文件 `compiler.rkt` 中实现变量和 `let` 形式的用例。

Exercise 2. 创建五个 $R_{\texttt{Var}}$ 程序，使用 `uniquify` 通道中最有趣的部分，也就是说，程序应该包含 `let` 形式、变量和相互遮掩的变量。这五个程序应该放在名为 `tests` 的子目录中，文件名应该以 `var_test_` 开头，后跟一个唯一的整数，以文件扩展名`.rkt` 结束。支持代码中的 `run-tests.rkt` 脚本检查输出程序是否产生与输入程序相同的结果。该脚本使用 `utilities.rkt` 中的 `interp-tests` 函数 (附录 12.2) 来测试示例程序的 `uniquify` 通道。`interp-tests` 的 `passes` 参数是一个列表，在你的编译器中，每个通道都应该有一个条目。现在，`passes` 定义只包含 `uniquify` 的一个条目，如下所示。

[3] Racket 函数 `gensym` 可以方便地生成唯一的变量名。

```
(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))]))
```

图 2.11: `uniquify` 通道的框架。

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

运行支持代码中的 `run-tests.rkt` 脚本，检查输出程序是否产生与输入程序相同的结果。

2.5 删除复杂的操作数

`remove-complex-opera*` 通道将 $R_{\texttt{Var}}$ 程序编译成一种受限的形式，在这种形式中，操作的参数是原子表达式。换句话说，这个传递删除复杂的操作数，例如下面程序中的表达式 `(- 10)`。这是通过引入一个新的 `let` 绑定变量来实现的，将复杂操作数绑定到新变量，然后使用新变量来代替复杂操作数，如右边的 `remove-complex-opera*` 的输出所示。

```
(+ 52 (- 10))        ⇒   (let ([tmp.1 (- 10)])
                           (+ 52 tmp.1))
```

图 2.12 给出该通道输出的语法，即语言 $R_{\texttt{Var}}^{\mathsf{ANF}}$。唯一的区别是操作符参数被限制为由 atm 非终端定义的原子表达式。特别是整数常量和变量

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

图 2.12: $R_{\texttt{Var}}^{\texttt{ANF}}$ 是 $R_{\texttt{Var}}$ 的管理标准形式 (ANF)。

是原子的。在文献中，将参数限制为原子表达式称为 管理规范形式，简称 ANF [33, 44] 。

建议使用两个相互递归的函数：`rco-atom` 和 `rco-exp` 来实现这个过程。其思想是将 `rco-atom` 应用于需要成为原子的子表达式，并将 `rco-exp` 应用于不需要原子的子表达式。这两个函数都接受 $R_{\texttt{Var}}$ 表达式作为输入。`rco-exp` 函数返回一个表达式。`rco-atom` 函数返回两个东西：原子表达式和将临时变量映射到复合子表达式的列表。你可以用 Racket `values` 形式从函数中返回多个东西，也可以用 `define-values` 形式从函数调用中接收多个东西。如果你不熟悉这些特性，请查阅 Racket 文档。此外，在函数返回多个值的情况下，`for/lists` 形式用于将函数应用到列表的每个元素。

回到示例程序 `(+ 52 (- 10))`，应该使用 `rco-atom` 函数处理子表达式 `(- 10)` 因为它是 `+` 的一个参数，因此需要变成原子的。`rco-atom` 应用于 `(- 10)` 的输出如下。

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

要特别注意下列程序，它们将变量绑定到原子表达式。应该保持这些变量绑定不变，如右边的程序所示

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

如果不小心实现 `rco-exp` 和 `rco-atom`，可能会产生以下带有不必要临时变量的输出。

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

Exercise 3. 在 `compiler.rkt` 实现 `remove-complex-opera*` 函数。创建三个新的 $R_{\texttt{Int}}$ 程序，执行 `remove-complex-opera*` 通道中的有趣代码 (遵循与前面相同的文件名准则。)。在 `run-tests.rkt` 脚本中，将以下条目添加到 `passes` 列表中，然后运行该脚本以测试编译器。

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

在调试编译器时，查看每个通道输出的中间程序通常是很有用的。要打印中间程序，请在 `run-tests.rkt` 中的 `interp-tests` 调用之前放置以下内容。

```
(debug-level 1)
```

2.6 解释控制

`explicate-control` 通道将 $R_{\texttt{Var}}$ 程序编译为 $C_{\texttt{Var}}$ 程序，$C_{\texttt{Var}}$ 程序在语法中明确了执行顺序。现在，这相当于将 `let` 构造成一个赋值语句序列。例如，考虑下面的 $R_{\texttt{Var}}$ 程序。

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

前面的通道和 `explicate-control` 的输出如下所示。回想一下，`let` 的右边在它的主体之前执行，所以这个程序的求值顺序是把 `20` 赋值给 `x.1` ，把 `22` 赋值给 `x.2` ，把 `(+ x.1 x.2)` 赋值给 `y` ，然后返回 `y` 。实际上，`explicate-control` 的输出使这种排序变得明确。

```
(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) ___]))
```

图 2.13: `explicate-control` 通道的框架。

```
(let ([y (let ([x.1 20])
           (let ([x.2 22])
             (+ x.1 x.2)))])
  y)
```

⇒

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

这个通道的组织方式取决于我们之前提到的尾部位置的概念。形式上，$R_{\texttt{Var}}$ 上下文中的 *尾部位置* 由以下两条规则递归定义。

1. 在 (`Program` () e), 中，表达式 e 位于尾部位置。
2. 如果 (`Let` x e_1 e_2) 在尾部位置，那么 e_2 也是。

建议使用两个相互递归的函数,即 `explicate-tail` 和 `explicate-assign`

来实现 `explicate-control`，如图 2.13 的框架代码中所建议。`explicate-tail` 函数应用于位于尾部位置的表达式，而 `explicate-assign` 函数应用于出现在 `let` 右侧的表达式。`explicate-tail` 函数接受 $R_{\texttt{Var}}$ 中的 *exp* 作为输入，并在 $C_{\texttt{Var}}$ 中生成一个 *tail*（见图 2.10）。`explicate-assign` 函数在 *exp* 中接受 $R_{\texttt{Var}}$，也就是要被赋值的变量，在 *tail* 中接受 $C_{\texttt{Var}}$，表示赋值之后的代码。`explicate-assign` 函数在 $C_{\texttt{Var}}$ 中返回一个 *tail*。

`explicate-assign` 函数采用累加器传递方式，因为 `cont` 形参用于累加输出。读者可能会试图用一种更直接的方式组织 `explicate-assign`，而不使用 `cont` 参数，可能会使用 `append` 来组合状态。我们对这种替代提出警告，因为累加器通道样式是第 4 章中为条件表达式生成高质量代码的关键。

Exercise 4. 在 `compiler.rkt` 中实现 `explicate-control` 函数。创建三个新的 $R_{\texttt{Int}}$ 程序来执行 `explicate-control` 中的代码。在 `run-tests.rkt` 脚本中，将以下条目添加到 `passes` 列表中，然后运行脚本来测试编译器。

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

2.7 选择指令

在 `select-instructions` 中，开始将 $C_{\texttt{Var}}$ 转换为 $\text{x86}_{\texttt{Var}}$。此通道的目标语言是 x86 的一种变体，该变体仍使用变量，因此将形式为 `(Var` *var*`)` 的 AST 节点添加到 $\text{x86}_{\texttt{Int}}$ 抽象语法的 *arg* 非终端中 (Figure 2.8)。建议使用三个辅助函数来实现 `select-instructions`，分别针对 $C_{\texttt{Var}}$ 的非终端：*atm*、*stmt* 和 *tail*。

atm 的情况很简单，变量保持不变，整数常量更改为立即数：`(Int` n`)` 更改为 `(Imm` n`)`。

接下来考虑 *stmt* 的情况，从算术运算开始。例如，考虑加法操作。可以使用 `addq` 指令，但它执行一个就地更新。所以可以把 arg_1 移到左边的 *var* 中，然后把 arg_2 加到 *var* 中。

$$\textit{var} \texttt{ = (+ } \textit{arg}_1 \ \textit{arg}_2\texttt{);} \quad \Rightarrow \quad \begin{array}{l}\texttt{movq } \textit{arg}_1\texttt{, } \textit{var} \\ \texttt{addq } \textit{arg}_2\texttt{, } \textit{var}\end{array}$$

还有一些情况需要特别注意，以避免生成不必要的复杂代码。例如，如果加法的一个参数与赋值的左边的变量相同，那么就不需要额外的 move 指令。赋值语句可以被翻译成一个 `addq` 指令，如下所示。

var `= (+` arg_1 *var*`);` $\Rightarrow$ `addq` arg_1`,` *var*

`read` 操作在 x86 汇编中没有直接对应的，所以用 C 语言编写的文件 `runtime.c` 中的函数 `read_int` 来提供这个功能 [72] 。通常，将该文件中的所有功能称为 *运行时系统* ，或简称为 *运行时*。编译生成的 x86 汇编代码时，需要将 `runtime.c` 编译为 `runtime.o` (使用 `gcc` 选项 `-c` 的 "object file") ，并将其链接到可执行文件中。对于我们的代码生成目的，要做的就是将 `read` 的赋值转换为对 `read_int` 函数的调用，然后将其从 `rax` 移至左侧变量。(回想一下，函数的返回值进入 `rax` 。)

var `= (read);` $\Rightarrow$
```
callq read_int
movq %rax, var
```

tail 非终端有两种情况：`Return` 和 `Seq` 。关于 `Return` ，我们建议将其视为对 `rax` 寄存器的赋值，然后跳转到程序的结尾 (所以结论需要标注)。对于 `(Seq` s t`)` ，可以递归地翻译语句 s 和尾部 t ，然后附加产生的指令。

Exercise 5. 实现 `compiler.rkt` 中的 `select-instructions` 通道。创建三个新的示例程序，它们被设计来练习本步骤中所有有趣的情况。在 `run-tests.rkt` 脚本中，将以下条目添加到 `passes` 列表中，然后运行脚本来测试编译器。

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

2.8 分配空间

`assign-homes` 将 $\text{x86}_{\texttt{Var}}$ 程序编译成不再使用程序变量的 $\text{x86}_{\texttt{Var}}$ 程序。因此，`assign-homes` 通道负责将所有程序变量放入寄存器或堆栈中。为了提高运行时的效率，最好是将变量放在寄存器中，但是由于只有 16 个寄存器，一些程序必须求助于将一些变量放在堆栈中。在本章中，重点讨论在堆栈中放置变量的机制。在第 3 章中研究一种在寄存器中放置变量的算法。

再次考虑第 2.5 节中的 $R_{\texttt{Var}}$ 程序。

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

select-instructions 的输出显示在左边，assign-homes 的输出显示在右边。在本例中，将变量 a 分配给堆栈位置 -8(%rbp) ，将变量 b 分配给堆栈位置 -16(%rbp) 。

```
locals-types:
    a : Integer, b : Integer
start:
    movq $42, a
    movq a, b
    movq b, %rax
    jmp conclusion
```

⇒

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

X86Program 节点 *info* 中的 locals-types 条目是一个将程序中的所有变量映射到它们类型 (目前仅为 Integer) 的列表。assign-homes 通道应将所有这些变量的使用替换为堆栈位置。顺便说一句，locals-types 条目由支持代码中的 type-check-Cvar 计算，将其安装在 *info* 节点的 CProgram 字段中，应传播到 X86Program 节点。

在为堆栈位置赋值变量的过程中，可以方便地在 X86Program 节点的 *info* 字段中计算和存储帧的大小 (以字节为单位)，使用键 stack-space ，稍后生成 main 过程的结论时需要这个键。x86-64 标准要求帧大小是 16 字节的倍数。

Exercise 6. 实现 compiler.rkt 中的 assign-homes 通道，为非终端 *arg* 、*instr* 和 *block* 定义辅助函数。建议辅助函数使用一个额外的参数，该参数是将变量名称映射到空间（现在为堆栈位置）的清单。在 run-tests.rkt 脚本中，将以下条目添加到 passes 列表中，然后运行脚本来测试编译器。

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

2.9 补丁说明

patch-instructions 通道从 $\text{x86}_{\texttt{Var}}$ 编译到 $\text{x86}_{\texttt{Int}}$ 方法是确保每条指令都遵守一个限制：一条指令最多只能有一个参数是一个内存引用。

我们回到下面的例子。

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

assign-homes 通道为该程序生成以下输出。

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

第二个 movq 指令有问题，因为两个参数都是堆栈位置。建议从源位置移动到寄存器 rax ，然后从 rax 移动到目标位置来解决这个问题，如下所示。

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

Exercise 7. 在 patch-instructions 中实现 compiler.rkt 通道。创建三个新的示例程序，它们被设计来练习本步骤中所有有趣的情况。在 run-tests.rkt 脚本中，将以下条目添加到 passes 列表中，然后运行脚本来测试编译器。

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

2.10 打印 x86

编译器从 $R_{\texttt{Var}}$ 到 x86 的最后一步是将 $\text{x86}_{\texttt{Int}}$ AST (在图 2.8 中定义) 转换为字符串表示 (在图 2.4 中定义) 。Racket 的 format 和 string-append 函数在这方面很有用。这个步骤需要完成的主要工作是创建 main 函数及其前奏和结论的标准指令，如 2.2 节的图 2.6 所示。您需要知道堆栈帧所需

的空间大小，这可以从 X86Program 节点的 *info* 字段中的 stack-space 条目中获得。

在 Mac OS X 上运行时，编译器应该在 main 这样的标签前加上下划线前缀。Racket 调用 (system-type 'os) 用于判断编译器运行在哪个操作系统上。它返回 'macosx 、'unix 或 'windows 。

Exercise 8. 在 print-x86 中实现 compiler.rkt 通道。在 run-tests.rkt 脚本中，将以下条目添加到 passes 列表中，然后运行脚本来测试编译器。

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

取消对 compiler-tests 函数 (附录 12.2) 的调用的注释，该函数通过执行生成的 x86 代码来测试完整的编译器。编译提供的 runtime.c 文件到 runtime.o ，使用 gcc. 运行脚本来测试编译器。

2.11 挑战：$R_{\texttt{Var}}$ 的部分求值器

本节描述一些可选的挑战练习，这些练习涉及到调整和改进第 1.6 节中介绍的 $R_{\texttt{Int}}$ 的部分求值器。

Exercise 9. 调整 1.6 节 (图 1.4) 中的部分求值程序，使其适用于 $R_{\texttt{Var}}$ 程序，而不是 $R_{\texttt{Int}}$ 程序。回想一下，$R_{\texttt{Var}}$ 将 let 绑定和变量添加到 $R_{\texttt{Int}}$ 语言中，所以你需要在 pe-exp 函数中为它们添加用例。完成后，将部分求值通道添加到编译器前端，并确保编译器仍然通过所有测试。

下一个练习以练习 9 为基础。

Exercise 10. 用更了解算术的函数替换 pe-neg 和 pe-add 辅助函数，改进部分求值器。例如，部分求值程序应该将

```
(+ 1 (+ (read) 1))    翻译为    (+ 2 (read))
```

为此，pe-exp 函数应该以以下语法的 *residual* 非终端的形式产生输出。其思想是，当处理一个加法表达式时，我们总是可以产生：1) 一个整数常数；

2) 左边有一个整数常数，而右边没有整数常数的加法表达式；3) 一个子表达式中没有一个是整数常数的加法表达式。

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

`pe-add` 和 `pe-neg` 函数可以假设它们的输入是 *residual* 表达式，它们应该返回 *residual* 表达式。一旦改进完成，确保编译器仍然通过所有的测试。毕竟，如果产生不正确的结果，快速的代码是无用的!

3

寄存器分配

在第 2 章中，学习如何在栈中存储变量。在本章中，学习如何通过在寄存器中放置一些变量来提高生成代码的性能。CPU 可以在一个周期内访问寄存器，而访问堆栈需要 10 到 100 个周期。图 3.1 中的程序是一个运行示例。源程序在左侧，指令选择的输出在右侧。这个程序几乎是用 x86 汇编语言编写的，但它仍然使用变量。

寄存器分配的目标是将尽可能多的变量放入寄存器中。有些程序的变量比寄存器多，所以我们不能总是将每个变量映射到不同的寄存器。幸运的是，在程序执行期间的不同时间需要不同的变量是很常见的，在这种情况下，几个变量可以映射到同一个寄存器。考虑图 3.1 中的变量 `x` 和 `z` 。变量 `x` 移动到 `z` 后就不再需要它了。另一方面，变量 `z` 只在这一点之后才使用，因此 `x` 和 `z` 可以共享相同的寄存器。第 3.2 节的主题是如何计算哪里需要变量。一旦有了这些信息，就计算同时需要哪些变量，即哪些变量相互 干扰，并将这种关系表示为一个无向图，顶点是变量，边表示两个变量干扰 (第 3.3 节) 。然后将寄存器分配模型化为一个图着色问题 (第 3.4 节)。

如果用尽所有的寄存器，就把剩余的变量放在堆栈上，就像在第 2 章中所做的那样。通常使用动词 溢出将变量赋值给堆栈位置。溢出变量的决定将作为图形着色过程的一部分进行处理（第 3.4 节)。

做一个简化的假设，即每个变量都被分配到一个位置 (寄存器或堆栈地址)。更复杂的方法是将变量分配到程序不同区域中的一个或多个位置。例

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

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

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
```

图 3.1: 一个寄存器分配的运行示例。

如，如果一个变量在短序列中被多次使用，然后在许多其他指令之后才再次使用，那么在初始序列期间将该变量赋值给一个寄存器，然后在其剩余的生命周期中将其移动到堆栈中可能会更有效。建议感兴趣的读者向 Cooper and Torczon [29] 了解有关该方法的更多信息。

3.1 寄存器和调用约定

当执行寄存器分配时，需要注意在 x86 中的控制函数调用如何执行 *调用约定* 。即使 $R_{\texttt{Var}}$ 不包括程序员定义的函数，生成的代码也包含一个由操作系统调用的 `main` 函数，并且生成的代码包含对 `read_int` 函数的调用。

函数调用需要两段代码之间的协调，这两段代码可能由不同的程序员编写或由不同的编译器生成。这里我们遵循 Linux 和 MacOS 上的 GNU C 编译器使用的 System V 调用约定 [18, 83] 。调用约定包括关于函数如何共享寄存器使用的规则。特别是，调用方负责在函数调用之前释放一些寄存器，供被调用方使用。这些被称为 *调用者保存寄存器* ，它们是

```
rax rcx rdx rsi rdi r8 r9 r10 r11
```

另一方面，被调用方负责保留 *调用者保存寄存器*，即

```
rsp rbp rbx r12 r13 r14 r15
```

我们可以从调用方视图和被调用方视图两个角度来考虑这个调用方/被调用方约定:

- 调用方应该假定所有调用方保存的寄存器都被调用方用任意值重写。另一方面，调用者可以安全地假设所有被调用者保存的寄存器在调用后包含与调用前相同的值。
- 被调用方可以自由地使用任何调用方保存的寄存器。但是，如果被调用方希望使用被调用方保存的寄存器，被调用方必须在返回给调用方之前安排将原始值放回寄存器中。这可以通过将值保存到函数前奏中的堆栈中，并在函数的结论中恢复值来实现。

在 x86 中，寄存器还用于向函数传递参数和返回值。特别地，函数的前六个参数按以下顺序，在以下六个寄存器中传递。

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

如果有六个以上的参数，则约定是将调用方框架上的空间用于其余参数。然而，在第 6 章中，将不需要超过 6 个参数。目前，关心的唯一函数是 `read_int`，它接受零参数。寄存器 `rax` 用于函数的返回值。

下一个问题是这些调用约定如何影响寄存器分配。考虑图 3.2 中的 $R_{\texttt{Var}}$ 程序。首先从调用者的角度分析这个例子，然后从被调用者的角度分析。

该程序两次调用 `read` 函数。另外，变量 `x` 在第二次调用 `read` 时被使用，所以需要确保 `x` 中的值不会在调用 `read` 时被意外地删除。一种明显的方法是，在每次函数调用之前，将调用方保存的寄存器中的所有值保存到堆栈中，并在每次调用之后恢复它们。这样，如果寄存器分配器选择将 `x` 分配给调用方保存的寄存器，那么它的值将在调用 `read` 的过程中被保留。然而，保存和恢复到堆栈相对较慢。如果 `x` 没有被多次使用，那么最好首先将 `x` 分配给一个堆栈位置。或者更好的是，如果可以将 `x` 放置在被调用者保存的寄存器中，那么它就不需要在函数调用期间保存和恢复。

对于在函数调用期间使用的变量，推荐的方法是将它们赋值给被调用方保存的寄存器，或者将它们溢出到堆栈中。另一方面，对于在函数调用期间没有使用的变量，按顺序尝试以下替代方法：1) 寻找一个可用的调用者保存的寄存器 (为被调用者保存的寄存器中的其他变量留出空间)；2) 查找被调用者保存的寄存器；3) 将变量溢出到堆栈中。

在一个图着色寄存器分配器中实现这种方法很简单。首先，我们知道每个函数调用过程中使用了哪些变量，因为我们为每条指令计算了这些信息 (第 3.2 节)。其次，当构建干扰图 (第 3.3节) 时，我们可以在这些变量和干扰图中的调用者保存的寄存器之间放置一条边。这防止图着色算法将这些变量分配给调用者保存的寄存器。

回到图 3.2 中的示例，分析右边生成的 x86 代码，重点关注 `start` 块。注意变量 `x` 被赋值给 `rbx`，一个被调用者保存的寄存器。因此，在第二次调用 `read_int` 时，它已经在一个安全的地方。接下来，注意变量 `y` 被赋值给 `rcx`，一个调用者保存的寄存器，因为在块的其余部分没有函数调用。

接下来从调用者的角度来分析这个例子，重点是 `main` 函数的前奏和结论。像往常一样，前奏从将 `rbp` 寄存器保存到堆栈并将 `rbp` 设置为当前堆

栈指针开始处。现在我们知道为什么需要保存 `rbp`：它是一个被调用者保存的寄存器。然后，前奏将 `rbx` 推入堆栈，因为：1) `rbx` 是被调用者保存的寄存器；2) `rbx` 赋值给变量 (`x`) 。其他被调用者保存的寄存器没有保存在前奏中，因为它们没有被使用。前奏从 `rsp` 中减去 8 个字节以使其对齐 16 个字节，然后跳转到 `start` 块。将注意力转移到 `conclusion` 上，看到 `rbx` 用 `popq` 指令从堆栈中恢复。

$R_{\texttt{Var}}$ 示例程序:

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

生成的 x86 汇编:

```
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
```

图 3.2: 一个函数调用的例子

3.2 活性分析

`uncover-live` 通道执行 *活性分析*，也就是说，它发现程序的不同区域中哪些变量在使用。如果一个变量或寄存器的当前值在程序的后面某个点被使用，那么这个变量或寄存器在程序的某个点就是 *活*的。把变量和寄存器统称为 *位置*。考虑下面的代码片段，其中有两次对 `b` 的写操作。`a` 和 `b` 是同时活的吗?

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

答案是否定的，因为 `a` 从 1 到 3 行是活的，`b` 从 4 到 5 行是活的。在第 2 行写入 `b` 的整数从未被使用，因为它在下一次读取 (第 5 行) 之前被覆盖 (第 4 行)。

Racket 集合包

set 是没有重复的无序元素集合。

(set v ...**)** 构造一个包含指定元素的集合。

(set-union set_1 set_2**)** 返回两个集合的并集。

(set-subtract set_1 set_2**)** 返回两个集合的差值。

(set-member? set v**)** v 是 set 中的元素吗?

(set-count set**)** set 集合中有多少个元素?

(set->list set**)** 集合转换成列表。

活动位置可以通过将指令序列往回遍历 (即按执行顺序向后遍历) 来计算。设 $I_1, \ldots, I_n$ 为指令序列。我们在指令 I_k 之后为活动位置集写为 $L_{\mathsf{after}}(k)$，在指令 I_k 之前为活动位置集写为 $L_{\mathsf{before}}(k)$。一条指令之后的活动位置总是与下一条指令之前的活动位置相同。

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

开始时，在最后一条指令之后没有实时位置，所以

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

然后我们重复应用以下规则，将指令序列从头到尾遍历。

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

其中 $W(k)$ 是指令 I_k 写入的位置，$R(k)$ 是指令 I_k 读取的位置。

对于 `jmp` 指令有一个特殊情况。在 `jmp` 之前存在的位置应该是 $L_{\texttt{before}}$ 中跳转目标的位置。因此，建议维护一个名为 `label->live` 的列表，该列表针对其块中的第一条指令将每个标签映射到 $L_{\texttt{before}}$ 。目前，$\text{x86}_{\texttt{Var}}$ 程序中唯一的 `jmp` 是结尾的那个。(例如，如图 3.1 所示。) 结论从 `rax` 和 `rsp` 读取，所以列表应该将结论映射到集合 $\{\texttt{rax}, \texttt{rsp}\}$ 。

回顾一下上面的例子，从第 5 行开始应用这些公式。在图 3.3 中收集答案。`addq b, c` 指令的 L_{after} 是 $\emptyset$ ，因为它是最后一条指令 (公式 3.2)。这个指令的 L_{before} 是 $\{\texttt{b}\texttt{c}\}$ ，因为它从变量 `b` 和 `c` (公式 3.3) 中读取数据，即

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

继续移动第 4 行指令 `movq $10, b` ，将第 5 行中的 live-before 集合复制为该指令的 live-after 集合 (公式 3.1)。

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

这个 move 指令写入 `b` ，而不从任何变量中读取，所以有以下 live-before 集合 (公式 3.3)。

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

`movq a, c` 的 live-before 是 $\{\texttt{a}\}$ ，因为它向 $\{\texttt{c}\}$ 写入并从 $\{\texttt{a}\}$ 读取 (公式 3.3) 。`movq $30, b` 的 live-before 是 $\{\texttt{a}\}$ ，因为它写入一个非活动的变量，不从变量中读取。最后，`movq $5, a` 的 live-before 是 $\emptyset$ ，因为它写入变量 `a` 。

Exercise 11. 对图 3.1 中运行的示例执行活性分析，计算每个指令的 live-before 和 live-after 集合。将您的答案与图 3.4 中所示的解决方案进行比较。

Exercise 12. 实现 `uncover-live` 通道。将 live-after 集合序列存储在 `Block` 结构的 *info* 字段中。建议创建一个辅助函数，接受一个指令列表和一个初始的 l live-after 集合 (通常为空)，并返回 live-after 集合列表。还建议创建辅助函数：1) 计算出现在 *arg* 的位置集；2) 计算指令读取的位置 (R 函数)；

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

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

图 3.3: 在一个简短的示例上输出活性分析的示例。

3) 指令所写的位置 (W 函数)。`callq` 指令应该在其写集 W 中包含所有调用者保存的寄存器，因为调用约定说，这些寄存器可以在函数调用期间写入。同样，`callq` 指令应该在其读集 R 中包含适当的参数传递寄存器，这取决于被调用函数的种类。(这就是为什么 `callq` 的抽象语法包含细节。)

```
                {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
```

图 3.4: 这个正在运行的例子说明 live-after 集合。

3.3 构建干涉图

根据活体分析，我们知道每个地方都在哪里。但是，在分配寄存器时，需要回答具体形式的问题：u 和 v 位置是同时存在的吗？(因此不能分配给同一个寄存器。) 为了更有效地回答这个问题，创建一个显式的数据结构，*干涉图* 。干涉图是一个在两个位置之间有一条边的无向图，如果它们同时存在，也就是说，如果它们相互干扰。

计算干涉图的一种明显方法是查看每个指令和下一个指令之间的活动位置集合，并为同一集合中的每一对变量在图中添加一条边。这种方法并不理想，原因有二：首先，它很昂贵，因为在 n 活动位置的集合中，每对花费的时间为 $O(n^2)$ ；第二，在特殊情况下，两个位置拥有相同的值 (因为一个被分配给另一个)，它们可以同时存在，而不会相互干扰。

> **Racket 图形库**
>
> 图是顶点和边的集合，其中每条边连接两个顶点。如果图的每条边都从源指向目标，那么图就是 *有向的*。否则，图是 *无向的*。
>
> **(undirected-graph** *edges***)** 从一列边构造一个无向图。每条边都由一个包含两个顶点的列表表示。
>
> **(add-vertex!** *graph vertex***)** 在图中插入一个顶点。
>
> **(add-edge!** *graph source target***)** 在两个顶点之间插入一条边到图中。
>
> **(in-neighbors** *graph vertex***)** 返回给定顶点的所有邻近点的序列。
>
> **(in-vertices** *graph***)** 返回图形中所有顶点的序列。

计算干涉图的更好方法是关注写操作 [8] 。指令执行的写操作不能覆盖活动位置中的内容。因此，对于每条指令，在被写入的位置和活动的位置之间创建一条边。(除了一个位置不能创造自交边。) 注意，对于 `callq` 指令，我们将所有被写入的调用者保存的寄存器都考虑在内，因此在每个活变量和每个调用者保存的寄存器之间添加一条边。对于 `movq` ，处理上述特殊情况时，如果 v 与源匹配，则不在活动变量 v 和目标变量之间添加一条边。

1. 如果指令 I_k 是一个移动，如 `movq` s`,` d ，则对每一个 $v \in L_{\mathsf{after}}(k)$ 添加边 (d, v) ，除非 $v = d$ 或 $v = s$ 。
2. 对于任何其他指令 I_k ，对于每一个 $d \in W(k)$ ，为 $v \in L_{\mathsf{after}}(k)$ 增加一条边 (d, v) ，除非 $v = d$ 。

`movq $1, v`	`v` 干扰 `rsp`,
`movq $42, w`	`w` 干扰 `v` 和 `rsp`,
`movq v, x`	`x` 干扰 `w` 和 `rsp`,
`addq $7, x`	`x` 干扰 `w` 和 `rsp`,
`movq x, y`	`y` 干扰 `w` 和 `rsp` ， `x` 除外,
`movq x, z`	`z` 干扰 `w` 、`y` 和 `rsp`,
`addq w, z`	`z` 干扰 `y` 和 `rsp`,
`movq y, t`	`t` 干扰 `z` 和 `rsp`,
`negq t`	`t` 干扰 `z` 和 `rsp`,
`movq z, %rax`	`rax` 干扰 `t` 和 `rsp`,
`addq t, %rax`	`rax` 干扰 `rsp`.
`jmp conclusion`	没有干扰。

图 3.5: 运行示例的干扰结果。

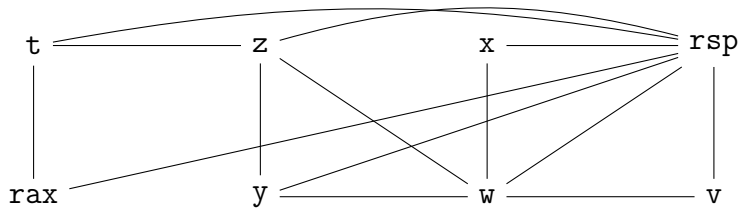


图 3.6: 示例程序的干涉图。

从图 3.4 的顶部到底部，将上述规则应用于每个指令。强调一些说明。第一个指令是 `movq $1, v` 和 live-after 集合是 {`v`, `rsp`} 。规则 1 适用，所以 `v` 干扰 `rsp` 。第四个指令是 `addq $7, x` 和 live-after 集合是 {`w`, `x`, `rsp`} 。规则 2 适用于 `x` 干扰 `w` 和 `rsp` 。下一个指令是 `movq x, y` 和 live-after 集合是 {`w`, `x`, `y`, `rsp`} 。规则 1 适用，所以 `y` 会干扰 `w` 和 `rsp` ，但不会干扰 `x` ，因为 `x` 是移动的来源，因此 `x` 和 `y` 保持相同的值。图 3.5 列出所有指令的干扰结果，得到的干扰图如图 3.6 所示。

Exercise 13. 根据上面建议的算法实现名为 `build-interference` 的编译

器通道。建议使用 `graph` 包来创建和检查干涉图。这个通道的输出图应该存储在程序的 *info* 字段中，在 `conflicts` 键下。

3.4 通过数独进行图形着色

我们来看看主要事件，将变量映射到寄存器和堆栈位置。相互干扰的变量必须映射到不同的位置。就干涉图而言，这意味着相邻的顶点必须映射到不同的位置。如果把位置看作颜色，那么寄存器分配问题就变成图着色问题 [12, 96] 。

读者可能比他或她意识到的更熟悉图形着色问题；流行的数独游戏就是图着色问题的一个例子。下面描述如何从最初的数独板构建一个图。

- 图中每个数独方格都有一个顶点。
- 如果对应的正方形在同一行、同一列，或者正方形在相同的 3×3 区域，则两个顶点之间存在一条边。
- 选择九种颜色对应数字 1 到 9 。
- 在数独棋盘上初始分配数字的基础上，给图中相应的顶点分配相应的颜色。

如果可以用这 9 种颜色给图中剩余的顶点上色，那么你也解决相应的数独游戏。图 3.7 显示一个初始数独游戏板和相应的带有彩色顶点的图形。将数独数字 1 映射为蓝色，2 映射为黄色，3 映射为红色。只对采样的顶点 (有颜色的顶点) 显示边，因为显示所有顶点的边会使图形不可读。

结果表明，玩数独游戏的一些技术对应于图着色算法中使用的启发式。例如，数独游戏的一个基本技巧是铅笔标记。这个方法是用消元法来确定哪些数在方框中不存在，然后把这些数写在方框中 (写得很小)。例如，如果数字 1 被分配给一个正方形，那么在同一行、同列和同区域的所有正方形上用铅笔写上 1 。铅笔标记技术对应于由于 [16] 而产生的 *饱和度* 概念。顶点的饱和，用数独的术语来说，就是一组不再可用的数字。在图形术语中，有以下定义：

$$(u) = \{c \mid \exists v.v \in (u) \text{ 和} (v) = c\}$$

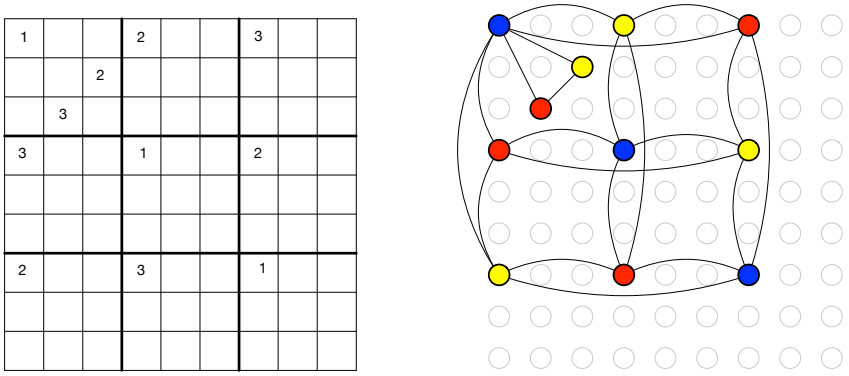


图 3.7: 一个数独游戏的棋盘和相应的彩色图表。

其中 (u) 是与 u 共享一条边的顶点集合。

使用铅笔标记技术可以得到一个简单的数字填充策略: 如果一个方格中只剩下一个可能的数字，那么就选择那个数字! 但如果没有方格，只剩下一种可能呢? 一种蛮力方法是尝试所有的方法: 选择第一个，如果它最终能引出一个解决方案，那很好。如果没有，就回头选择下一个可能性。铅笔标记的一个优点是它减少搜索树的分支程度。然而，回溯可能非常耗时。减少回溯数量的一种方法是使用最受约束优先启发式。也就是说，当选择一个方块时，总是选择一个剩余可能性最小的 (饱和度最高的顶点)。这个想法是，尽早选择高约束的方块，因为较低约束在高度饱和的方块中可能没有任何剩余的可能性。

然而，寄存器分配比数独更容易，因为寄存器分配器可以在寄存器耗尽时将变量映射到堆栈位置。因此，用贪婪搜索取代回溯是有意义的：在当做出最佳选择并继续前进。仍然希望最小化所需的颜色数量，因此在贪婪搜索中使用最受约束优先启发式。图 3.8 给出基于饱和和最受约束优先启发式的寄存器分配的简单贪婪算法的伪代码。它大致相当于 DSATUR 算法 [16, 50, 3] 。就像在数独游戏中一样，这个算法用整数表示颜色。从 0 到 $k-1$ 的整数对应于寄存器分配的 k 个寄存器。整数 k 及以上对应的是堆栈位置。不用于寄存器分配的寄存器 (如 `rax`) 被分配给负整数。特别地，把

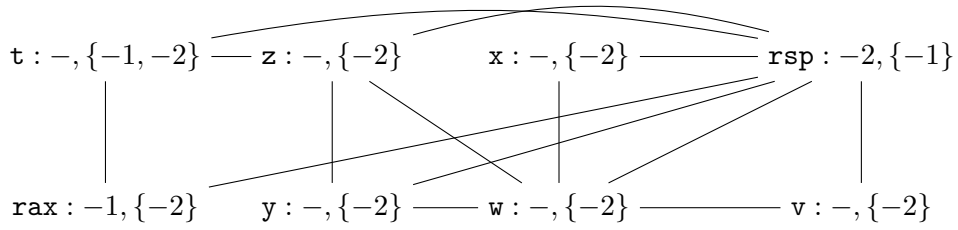
算法： DSATUR
输入： 图 G
输出： 每个顶点 $v \in G$ 的赋值 $[v]$

$W \leftarrow (G)$
while $W \neq \emptyset$ **do**
    从 W 中选择一个饱和度最高的顶点 u ，随机打破束缚
    找到不属于 $\{[v] : v \in (u)\}$ 的最低颜色 c
    $[u] \leftarrow c$
    $W \leftarrow W - \{u\}$

图 3.8: 基于饱和的贪心图着色算法。

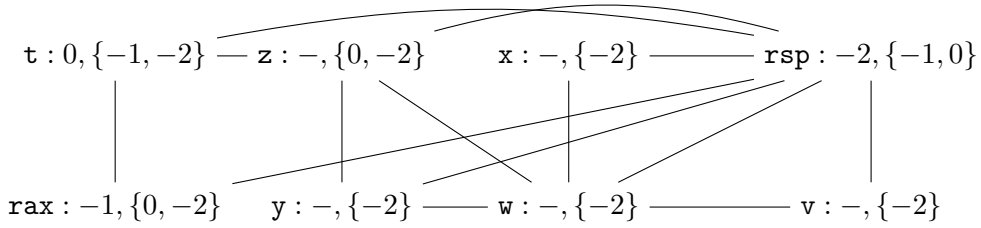
-1 赋给 `rax` ，把 -2 赋给 `rsp` 。

有了 DSATUR 算法之后，回到正在运行的示例，考虑如何给图 3.6 中的干涉图上色。首先将寄存器节点分配给它们自己的颜色。例如，`rax` 被赋予颜色 -1 ，`rsp` 被赋予颜色 -2 。这些变量还没有着色，所以用破折号对它们进行注释。然后更新与寄存器相邻的顶点的饱和度，得到以下注释图。例如，`t` 的饱和度是 $\{-1, -2\}$ ，因为它干扰 `rax` 和 `rsp` 。

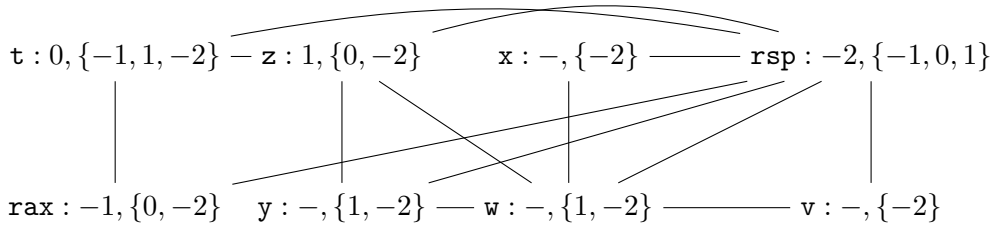


算法要求选择一个最大饱和顶点。我们取 `t` ，用第一个整数，也就是 0 ，给

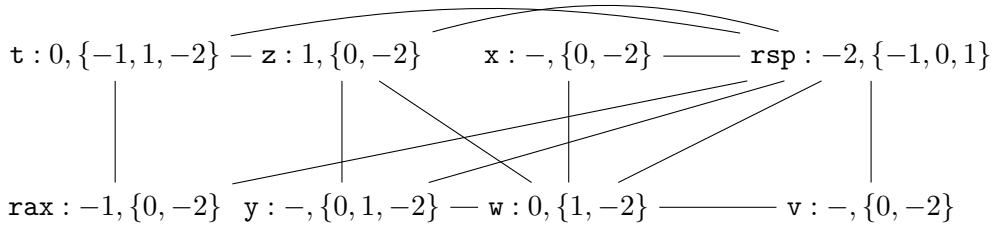
它上色。我们标记：0 不再适用于 z 、rax 和 rsp ，因为它们与 t 相干扰。



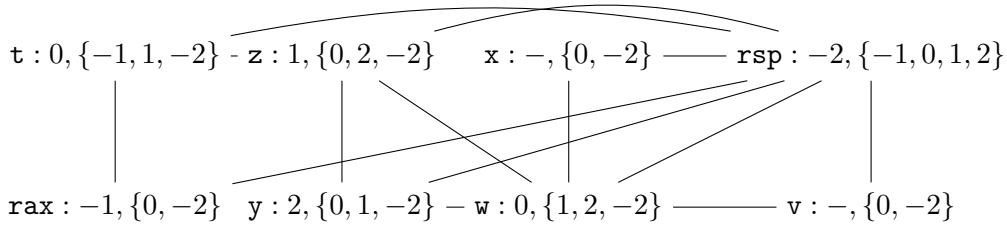
重复这个过程，选择下一个最大饱和顶点，也就是 z ，然后用第一个可用的数字，也就是 1 给它上色。给相邻的顶点 t 、y 、w 和 rsp 的饱和度加 1 。



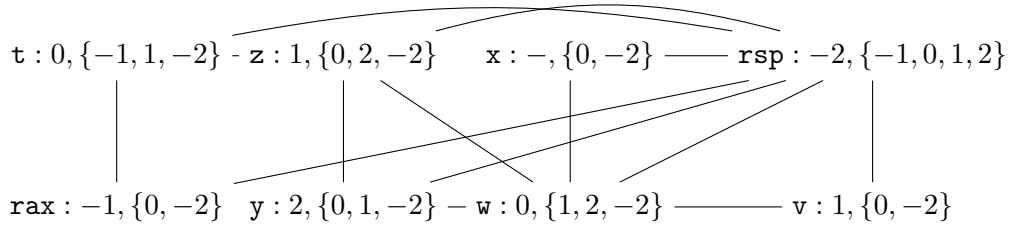
现在最饱和的顶点是 w 和 y 。用第一个可用的颜色为 w 上色，即 0 。



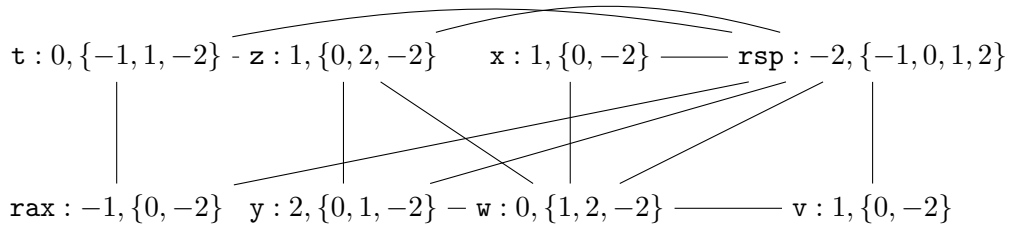
现在顶点 y 是最饱和的，所以用 y 给 2 上色。不能选择 0 或 1 ，因为这些数字在 y 的饱和集合中。实际上，y 会干扰 w 和 z ，它们的颜色分别为 0 和 1 。



现在 x 和 v 是最饱和的，所以 v 的颜色是 1 。



在算法的最后一步，用 1 给 x 上色。



建议创建一个名为 `color-graph` 的辅助函数，它接受一个干涉图和程序中所有变量的列表。这个函数应该返回变量与其颜色 (用自然数表示) 的映射。通过创建这个辅助函数，在第 6 章中添加对函数的支持时重用它。

为了优先处理 `color-graph` 函数中高度饱和的节点，建议使用优先队列数据结构 (参见右边的边栏)。此外，还需要在优先队列中维护一个从变量到它们的 “句柄” 的映射，这样当它们的饱和度发生变化时，您就可以通知优先队列。

着色完成后，就完成对寄存器和堆栈位置变量的赋值。将前 k 种颜色映射到 k 个寄存器，其余的颜色映射到堆栈位置。假设现在只有一个用于寄存器分配的寄存器 `rcx` 。然后有以下从颜色到位置的地图。

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

优先排队

优先排队是项的集合，其中项的移除由优先级控制。在 “min” 队列中，首先删除优先级较低的项。一个实现在支持代码 `priority_queue.rkt` 中。

(make-pqueue *cmp*) 构造一个空优先级队列，该队列使用 *cmp* 谓词来确定其第一个参数的优先级是否低于或等于其第二个参数。

(pqueue-count *queue*) 返回队列中的项数。

(pqueue-push! *queue item*) 将项插入队列中并返回队列中项的句柄。

(pqueue-pop! *queue*) 返回具有最低优先级的项。

(pqueue-decrease-key! *queue handle*) 通知队列与给定句柄关联的项的优先级已降低。

将这个映射与着色组合在一起，得到以下将变量赋值到位置的方法。

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

改写 `assign-homes` 通道的的代码（2.8 节），用它们所分配的位置替换变量。将上面的赋值应用到左边正在运行的示例中，将生成右边的程序。

```
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. 实现编译器通道 `allocate-registers` 。创建五个程序来执行所有的寄存器分配算法，包括将变量溢出到堆栈。将 `run-tests.rkt` 脚本中 `passes` 列表中 `assign-homes` 替换为三个新的通道：`uncover-live` 、`build-interference` 和 `allocate-registers` 。暂时从通道列表中删除 `print-x86` 通道和对 `compiler-tests` 的调用。运行脚本来测试寄存器分配器。

3.5 补丁说明

编译到 x86 的剩余步骤是确保指令最多有一个参数，即内存访问。在运行的示例中，`movq -8(%rbp), -16(%rbp)` 指令有问题。修正是先将 `-8(%rbp)` 移动到 `rax` ，然后将 `rax` 移动到 `-16(%rbp)` 。从 `-8(%rbp)` 到 `-8(%rbp)` 的两次移动也有问题，但可以通过简单地删除它们来修复。一般来说，建议删除所

有源和目标位置相同的琐碎移动。下面是运行示例的 patch-instructions 的输出。

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

⇒

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

Exercise 15. 实现 patch-instructions 编译器通道。将其插入 run-tests.rkt 脚本中的 passes 列表中的 allocate-registers 之后。运行脚本来测试 patch-instructions 是否通过。

3.6 打印 x86

回想一下，print-x86 通道生成的前奏和结论指令满足 x86 调用约定 (第 3.1 节)。随着寄存器分配器的增加，寄存器分配器使用的被调用保存寄存器必须保存在前奏中，并在结论中恢复。在 allocate-registers 通道中，将一个条目添加到名为 used-callee 的 X86Program 的 *info* 中，该条目存储分配给变量的被调用者保存的寄存器集。然后 print-x86 通道可以访问此信息，以决定需要保存和恢复哪些被调用保存的寄存器。在前奏中计算调整 rsp 的帧的大小时，确保考虑到用于保存被调用保存寄存器的空间。另外，不要忘记帧必须是 16 字节的倍数!

图 3.9 显示所有涉及到寄存器分配的通道的概览。

Exercise 16. 按照本节的描述更新 print-x86 通道。在 run-tests.rkt 脚本中，恢复通道列表中的 print-x86 和对 compiler-tests 的调用。运行

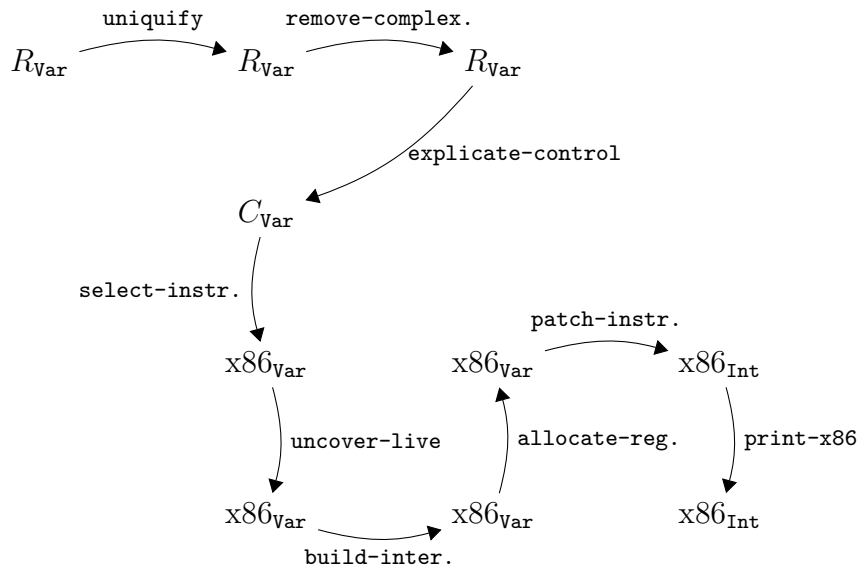


图 3.9: 带寄存器分配的 $R_{\texttt{Var}}$ 通道图。

脚本来测试执行寄存器分配的 $R_{\texttt{Var}}$ 的完整编译器。

3.7 挑战：移动偏压

本节描述对注册表分配器的改进，供那些寻求额外挑战或对注册表分配有更深兴趣的学生使用。

为了激发对移动偏置的需求，我们回到运行的例子，但这次使用所有通用寄存器。所以我们有以下颜色数字到寄存器的映射。

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

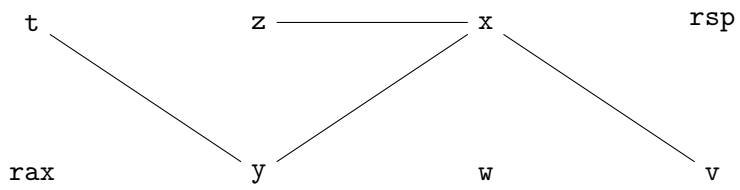
使用上一节中描述的寄存器分配器生成的相同的对颜色数字的变量赋值，得到以下程序。

```
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
```

在上面的输出代码中，有两个 `movq` 指令可以删除，因为它们的源和目标是相同的。但是，如果将 `t` 、`v` 、`x` 和 `y` 放入同一个寄存器中，则可以删除三 `movq` 指令。可以通过考虑哪些变量和哪些其他变量出现在 `movq` 指令中来实现这一点。

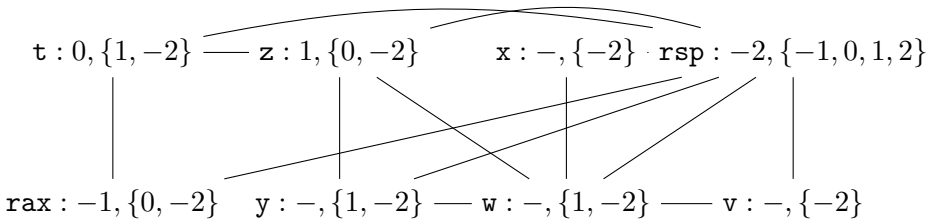
我们说，如果两个变量 p 和 q 一起参与 `movq` 指令，即 `movq` p, q 或 `movq` q, p ，则它们是 *移动相关* 。当寄存器分配器为一个变量选择颜色时，它应该选择一个已经为一个与移动相关的变量使用过的颜色 (假设它们不会干扰)。当然，这个首选项不应该覆盖寄存器优先于堆栈位置的首选项。当在寄存器之间进行选择或在堆栈位置之间进行选择时，他的偏好应该被用作平局打破者。

建议用图形表示移动关系，就像表示干涉一样。下面是运行示例的 *移动图*。

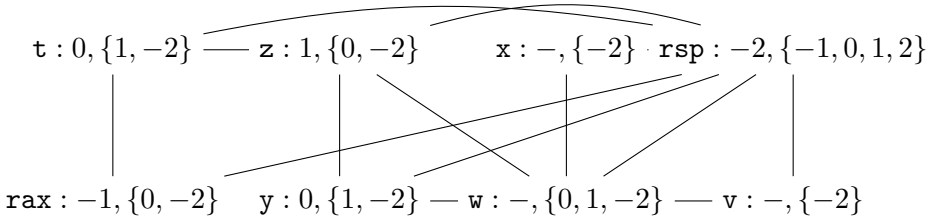


现在回放图形着色，暂停看 `y` 的着色。回想一下下面的配置。饱和的顶

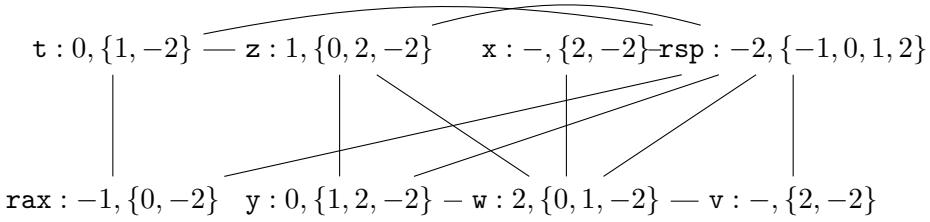
点是 w 和 y 。



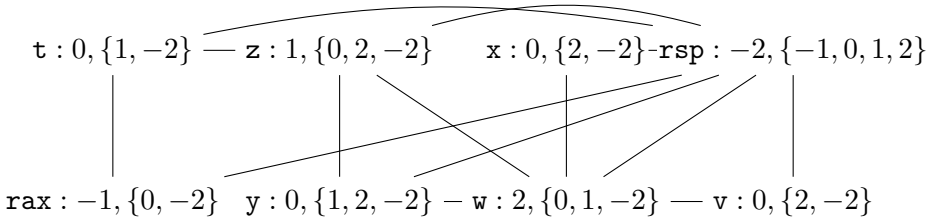
上次用 0 给 w 色。但是这次看到，w 和任何顶点都没有关系，但是 y 和 t 有关系。所以用和 t, 0 一样的颜色来表示 y 。



现在 w 是最饱和的，所以把它涂成 2 。



在这一点上，顶点 x 和 v 是最饱和的，但是 x 是与 y 和 z 相关的，所以给 x 上色为 0 来匹配 y 。最后，把 v 涂成 0 。



对寄存器进行如下的变量赋值。

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

将这个寄存器赋值应用到左边正在运行的示例，以获得中间的代码。然后，patch-instructions 删除三个简单的步骤，以获得右边的代码。

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

Exercise 17. 改变你的 allocate-registers 的实现来考虑移动偏置。创建两个新的测试，其中至少包含一个移动偏置的机会，并直观地检查输出的 x86 程序，以确保你的移动偏置工作正常。确保编译器仍然通过所有测试。

图 3.10 显示为运行示例 (图 3.1) 生成的带有寄存器分配和移动偏置的 x86 代码。为了演示寄存器和堆栈的使用，将寄存器分配器限制为仅使用两个寄存器：rbx 和 rcx 。在 main 函数的前奏中，将 rbx 压入堆栈，因为它是一个被调用保存的寄存器，并且它被寄存器分配器分配给变量。在前奏的最后从 rsp 中减去 8 ，为溢出的变量保留空间。在这个减法之后，rsp 对齐到 16 字节。

移到 start 块，看到寄存器是如何分配的。变量 v 、x 和 y 被赋给 rbx ，变量 z 被赋给 rcx 。变量 w 溢出到堆栈位置 -16(%rbp) 。回想一下前奏曲将被调用保存寄存器 rbx 保存到堆栈中。溢出的变量必须放置在堆栈上低于保存的被调用者保存寄存器的位置，因此在这种情况下 w 被放置在 -16(%rbp) 。

在 conclusion 中，撤销在前奏中所做的工作。将堆栈指针向上移动 8 个字节 (用于溢出变量的空间)，然后弹出 rbx 和 rbp (被调用者保存的寄存器) 的旧值，并以 retq 结束，将控制权交还给操作系统。

```
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
```

图 3.10: 运行示例中的 x86 输出 (图 3.1)。

3.8 延伸阅读

早期的寄存器分配算法是在 20 世纪 50 年代为 Fortran 编译器开发的 [62, 10] 。图着色的使用始于 20 世纪 70 年代末和 80 年代初，当时 Chaitin et al. [24] 在 PL/I 的优化编译器上工作。该算法基于 19 世纪 70 年代对 Kempe [71] 的观察。如果一个图 G 有一个顶点 v 的度小于 k ，那么 G 是 k 可着色的，如果 G 去掉 v 的子图也是 k 可着色的。假设子图 k 是可着色的。在最坏的情况下，v 的邻居被分配不同的颜色，但是因为它们的数量少于 k ，所以 G 中会剩下一种或更多的颜色来给 v 。

Chaitin et al. [24] 的算法从图中移除度小于 k 的顶点 v ，并递归地为图的其余部分着色。当从递归返回时，它用一个可用的颜色给 v 上色并返回。Chaitin [23] 对该算法进行如下扩充，以处理溢出。如果没有度数小于 k 的顶点，则随机选择一个顶点，溢出它，从图中移除它，然后递归地为图的其余部分着色。

在着色之前，，Chaitin et al. [24] 合并与移动相关且互不干扰的变量，这个过程被称为 *合并*。虽然合并减少移动的次数，但它会使图形更难着色。Briggs et al. [17] 提出 *保守合并*，即只有当两个变量的高阶邻居小于 k 时，才会合并。George and Appel [51] 观察到，保守的合并有时过于保守，通过迭代合并，移除低阶顶点，使其更加激进。从不同的角度来解决这个问题，Briggs et al. [17] 还提出 *偏色*，即一个变量被赋予与另一个与移动相关的变量相同的颜色，如第 3.7 节所述。Chaitin et al. [24] 算法及其成功子算法迭代地执行合并、图着色和溢出码插入，直到所有变量都被分配到一个位置。

Briggs et al. [17] 观察到，Chaitin [23] 有时会溢出一些不必要的变量；如果一个高度变量的许多邻居被赋予相同的颜色，那么它就可以被着色。Briggs et al. [17] 提出 *乐观着色法*，即高度顶点不会立即溢出。相反，决策被推迟到递归调用之后，这时就可以清楚地看到是否存在可用的颜色。我们观察到，如果将前 k 个颜色作为寄存器，其余的颜色作为堆栈位置，那么这个算法等价于最小最后排序算法 [82] 。印第安纳大学 [40] 编译课程的早期版本是基于 Briggs et al. [17] 的算法。

最小最后排序算法是众多 *贪婪着色算法*中的一种。贪婪着色算法以特定的顺序访问所有顶点，并为每个顶点分配第一个可用的颜色。*离线贪婪算*

法在分配颜色之前预先选择排序。Chaitin et al. [24]算法应该被认为是离线的，因为顶点排序不依赖于分配的颜色，所以算法可以分为两个阶段。其他的命令也是可能的。例如，Chow and Hennessy [26] 的顺序变量根据运行时成本的估计。

*在线*贪婪着色算法使用关于当前颜色分配的信息来影响其余顶点着色的顺序。本章所描述的基于饱和的算法就是这样一种算法。我们选择使用饱和着色是因为通过数独引入图形着色很有趣。

寄存器分配器可以选择将每个变量仅映射到一个位置，如 Chaitin et al. [24]，也可以选择将一个变量映射到一个或多个位置。后者可以通过 *活动范围分割*来实现，其中一个变量被多个变量替换，每个变量处理其活动范围的一部分 [26, 17, 30] 。

Palsberg [92] 观察到 JoeQ 编译器中 Java 程序产生的许多干涉图都是*弦状*的，也就是说，每个有 4 条或更多边的循环有一条边不是循环的一部分，但它连接循环上的两个顶点。这样的图可以通过贪婪算法最优着色，顶点排序由最大基数搜索确定。

在编译时间非常重要的情况下，例如在即时编译器中，图着色算法可能过于昂贵，而 Poletto and Sarkar [94] 的线性扫描可能更合适。

4

布尔和控制流

$R_{\texttt{Int}}$ 和 $R_{\texttt{Var}}$ 语言只有一种值：整数。在本章中，添加第二种值，布尔值，来创建 $R_{\texttt{If}}$ 语言。布尔值 *true* 和 *false* 在 Racket 中分别表示为 `#t` 和 `#f` 。$R_{\texttt{If}}$ 语言包括几个涉及布尔值 (`and` 、`not` 、`eq?` 、`<` 等) 和条件式 `if` 表达式。通过添加 `if` ，程序可以具有重要的控制流，这将影响显式控制和动态分析。另外，因为现在有两种类型的值，所以需要处理对错误类型的值应用操作的程序，比如 `(not 1)` 。

对于这种情况，有两种语言设计选项。一种选择是发出错误信号，另一种选择是提供对操作的更广泛的解释。Racket 语言混合使用这两种选项，这取决于操作和值的类型。例如，在 Racket 中 `(not 1)` 的结果是 `#f` ，因为 Racket 将非零整数视为 `#t` 。另一方面，`(car 1)` 在 Racket 中导致运行时错误，因为 `car` 期望一对。

类型 Racket 的设计选择与 Racket 相似，只是大部分错误检测发生在编译时而不是运行时。类型 Racket 接受并运行 `(not 1)` ，产生 `#f` 。但是在 `(car 1)` ，类型 Racket 报告一个编译时错误，因为类型 Racket 期望参数的类型是 `(Listof T)` 或 `(Pairof T1 T2)` 的形式。

$R_{\texttt{If}}$ 语言在编译期间执行类型检查，就像类型 Racket 。在第 **??** 章中，研究另一种选择，即像 Racket 这样的动态类型语言。$R_{\texttt{If}}$ 语言是类型化 Racket 的子集；对于某些操作，我们有更多的限制，例如，拒绝 `(not 1)` 。

本章组织如下。首先定义 $R_{\texttt{If}}$ 语言的语法和解释器 (第 4.1 节)。然后介

绍类型检查的概念，并为 $R_{\texttt{If}}$ 构建一个类型检查器 (第 4.2 节)。为了编译 $R_{\texttt{If}}$ ，需要将中间语言 $C_{\texttt{Var}}$ 放大为 $C_{\texttt{If}}$ (4.3 节) ，将 $\text{x86}_{\texttt{Int}}$ 放大为 $\text{x86}_{\texttt{If}}$ (4.4 节) 。本章的其余部分将讨论编译器如何更改通道以适应布尔值和条件控制流。还有一个名为 `shrink` 的新通道，它将一些操作符转换为其他操作符，从而减少在以后的通道中需要处理的操作符的数量。最大的变化发生在 `explicate-control` ，将 `if` 表达式转换为控制流图 (第 4.8 节)。关于寄存器分配，现在活性分析有多个基本块需要处理，并且存在一个有趣的问题，即如何处理条件跳转。

4.1 $R_{\texttt{If}}$ 语言

图 4.1 定义 $R_{\texttt{If}}$ 语言的具体语法，图 4.2 定义抽象语法。$R_{\texttt{If}}$ 语言包括所有的 $R_{\texttt{Var}}$ (以灰色显示) 、布尔文字 `#t` 和 `#f` ，以及条件 `if` 表达式。将操作符扩展为包括

1. 整数减法；
2. 逻辑运算符 `and` 、`or` 和 `not` ；
3. `eq?` 操作用于比较两个整数或两个布尔值；
4. `<` 、`<=` 、`>` 和 `>=` 用于比较整数的操作。

重新组织图 4.2中基本操作的抽象语法，对所有这些操作只使用一条语法规则。这意味着语法不再检查操作符的数量是否与实参的数量匹配。这个职责被转移到 $R_{\texttt{If}}$ 的类型检查器上，我们将在 4.2 节中介绍它。

图 4.3 定义 $R_{\texttt{If}}$ 的解释器，它继承自 $R_{\texttt{Var}}$ 的解释器 (图 2.3) 。字面量 `#t` 和 `#f` 计算为相应的布尔值。条件表达式 (`if` *cnd thn els*) 计算 *cnd* ，然后根据 *cnd* 生成的是 `#t` 还是 `#f` ，计算 *thn* 或 *els* 。逻辑操作 `not` 和 `and` 的行为与可能期望的一样，但请注意 `and` 操作是短路的。也就是说，给定表达式 (`and` $e_1\, e_2$) ，如果 e_1 计算为 `#f` ，则不计算表达式 e_2 。

随着原语操作数量的增加，解释器将变得重复而无需多加注意。我们重构 `Prim` 的案例，将每个操作不同的代码移到图 `interp-op` 中所示的 4.4 方法中。分开处理 `and` 操作，因为它的短路行为。

bool	::=	`#t` | `#f`
cmp	::=	`eq?` | `<` | `<=` | `>` | `>=`
exp	::=	*int* | `(read)` | `(-` *exp*`)` | `(+` *exp exp*`)` | `(-` *exp exp*`)`
	|	*var* | `(let ([`*var exp*`]) `*exp*`)`
	|	*bool* | `(and` *exp exp*`)` | `(or` *exp exp*`)` | `(not` *exp*`)`
	|	`(`*cmp exp exp*`)` | `(if` *exp exp exp*`)`
$R_{\texttt{If}}$	::=	*exp*

图 4.1: $R_{\texttt{If}}$ 的具体语法，用布尔值和条件符扩展 $R_{\texttt{Var}}$ (图 2.1)

bool	::=	`#t` | `#f`
cmp	::=	`eq?` | `<` | `<=` | `>` | `>=`
op	::=	*cmp* | `read` | `+` | `-` | `and` | `or` | `not`
exp	::=	`(Int` *int*`)` | `(Var` *var*`)` | `(Let` *var exp exp*`)`
	|	`(Prim` *op* `(`*exp*`...))`
	|	`(Bool` *bool*`)` | `(If` *exp exp exp*`)`
$R_{\texttt{If}}$	::=	`(Program '()` *exp*`)`

图 4.2: $R_{\texttt{If}}$ 的抽象语法。

```
(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))
```

图 4.3: $R_{\texttt{If}}$ 语言的解释器。(请参见图 4.4 中的 `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")]))
```

图 4.4: R_{If} 语言中原语操作符的解释器。

4.2 类型检查 $R_{\texttt{If}}$ 程序

用两种互补的方式来考虑类型检查是很有帮助的。类型检查器预测程序中每个表达式将产生的值的类型。对于 $R_{\texttt{If}}$ ，只有两种类型，Integer 和 Boolean 。所以类型检查器应该预测

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

产生一个 Integer ，而

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

生成一个 Boolean 。

另一种考虑类型检查的方式是，它强制一组规则，规定哪些操作符可以应用于哪些类型的值。例如，$R_{\texttt{If}}$ 类型检查器会为下面的表达式发出错误信号

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

子表达式 (+ 10 (- (+ 12 20))) 的类型为 Integer ，但类型检查器强制规定 not 的参数必须是 Boolean 。

我们使用类和方法实现类型检查，因为它们提供重用代码所需的开放递归，将在后面的章节扩展类型检查器，类似于为解释器使用类和方法(第 2.1.1 节)。

将 $R_{\texttt{Var}}$ 片段的类型检查器分离到它自己的类中，如图 4.5 所示。$R_{\texttt{If}}$ 的类型检查器如图 4.6 所示，它继承自 $R_{\texttt{Var}}$ 的类型检查器。这些类型检查器在支持代码的 type-check-Rvar.rkt 和 type-check-Rif.rkt 文件中。每个类型检查器都是 AST 上的结构递归函数。给定一个输入表达式 e ，类型检查器要么发出错误信号，要么返回一个表达式及其类型 (Integer 或 Boolean) 。它返回一个表达式，因为在某些情况下我们想要更改或更新表达式。

接下来，讨论图 4.5 中的 type-check-exp 的 match 用例。整型常量的类型为 Integer 。要处理变量，类型检查器使用环境 env 将变量映射到类型。考虑一下 let 的情况。对初始化表达式进行类型检查以获得其类型 T ，然后将类型 T 与用于 let 主体类型检查的环境中的变量 x 关联起来。因此，当类型检查器遇到变量 x 的使用时，它可以在环境中找到它的类型。对于基

元操作符，递归地分析实参，然后调用 type-check-op 来检查是否允许实参类型。

在类型检查器中使用几种辅助方法。operator-types 方法定义一个将操作符名称映射到它们的形参和返回类型的字典。type-equal? 方法确定两种类型是否相等，目前它只是分派给 equal? (深度相等) 。如果两种类型不相等，check-type-equal? 方法会触发错误。type-check-op 方法在 operator-types 字典中查找操作符，然后检查实参类型是否等于形参类型。结果是操作符的返回类型。

接下来讨论图 4.6 中 R_{If} 的类型检查器。操作符 eq? 要求两个参数具有相同的类型。布尔常量的类型是 Boolean 。if 的条件必须是 Boolean 类型，并且两个分支必须具有相同的类型。operator-types 函数为其他新操作符添加字典条目。

Exercise 18. 在 R_{If} 中创建 10 个新的测试程序。一半的程序应该有一个类型错误。对于这些程序，创建一个具有相同基名但文件扩展名为 .tyerr 的空文件。例如，如果测试 cond_test_14.rkt 预计会出错，则创建一个名为 cond_test_14.tyerr 的空文件。这指示 interp-tests 和 compiler-tests 预期会出现类型错误。另一半的测试程序不应该有类型错误。

在 run-tests.rkt 脚本中，将 interp-tests 和 compiler-tests 的第二个参数更改为 type-check-Rif ，这将导致类型检查器在编译器通过之前运行。暂时将 passes 更改为空列表并运行脚本，从而检查新的测试程序是否如预期的那样进行类型检查。

4.3 C_{If} 中间语言

图 4.7 定义 C_{If} 中间语言的抽象语法。(具体的语法见附录，图 12.3 。) 与 C_{Var} 相比，C_{If} 语言将逻辑运算符和比较运算符添加到 *exp* 非终端中，将文字 #t 和 #f 添加到 *arg* 非终端中。

对于控制流，C_{If} 将 goto 和 if 语句添加到 *tail* 非终端中。if 语句的条件是比较操作，分支是 goto 语句，这使得将 if 语句编译到 x86 非常简单。

```
(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))
```

图 4.5: $R_{\texttt{Var}}$ 语言的类型检查器。

4.4 x86$_{\texttt{If}}$ 语言

为了实现新的逻辑操作、比较操作和 `if` 表达式，需要进一步研究 x86 语言。图 4.8 和 4.9 定义 x86 的 x86$_{\texttt{If}}$ 子集的具体和抽象语法，其中包括用于逻辑操作、比较和条件跳转的指令。

一个挑战是 x86 没有提供直接实现逻辑否定的指令 (在 $R_{\texttt{If}}$ 和 $C_{\texttt{If}}$ 中是 `not`) 。但是，`xorq` 指令可以用来编码 `not` 。`xorq` 指令接受两个参数，对其参数的每一位执行一个成对异或 (XOR) 操作，并将结果写入第二个参数。回想一下异或的真值表：

	0	1
0	0	1
1	1	0

例如，对二进制数 0011 和 0101 的每一位应用 XOR ，结果是 0110 。注意，在表中第 1 位的行中，结果与第 2 位相反。因此，`not` 操作可以由 `xorq` 以 1 作为第一个参数来实现：

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

```
(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))
```

图 4.6: $R_{\texttt{If}}$ 语言的类型检查器。

```
atm    ::= (Int int) | (Var var) | (Bool bool)
cmp    ::= eq? | <
exp    ::= atm | (Prim read ())
         | (Prim - (atm)) | (Prim + (atm atm))
         | (Prim 'not (atm)) | (Prim 'cmp (atm atm))
stmt   ::= (Assign (Var var) exp)
tail   ::= (Return exp) | (Seq stmt tail) | (Goto label)
         | (IfStmt (Prim cmp (atm atm)) (Goto label) (Goto label))
C_If   ::= (CProgram info ((label . tail) ...))
```

图 4.7: $C_{\texttt{If}}$ 的抽象语法，$C_{\texttt{Var}}$ 的扩展 (图 2.10)。

```
bytereg  ::= ah | al | bh | bl | ch | cl | dh | dl
arg      ::= $int | %reg | int(%reg) | %bytereg
cc       ::= e | l | le | g | ge
instr    ::= addq arg,arg | subq arg,arg | negq arg | movq arg,arg |
             callq label | pushq arg | popq arg | retq | jmp label
             label: instr | xorq arg, arg | cmpq arg, arg |
             setcc arg | movzbq arg, arg | jcc label
x86_If   ::= .globl main
             main: instr ...
```

图 4.8: x86$_{\texttt{If}}$ 的具体语法 (扩展图 2.4 中的 x86$_{\texttt{Int}}$)。

```
bytereg  ::= ah | al | bh | bl | ch | cl | dh | dl
arg      ::= (Imm int) | (Reg reg) | (Deref reg int) | (ByteReg bytereg)
cc       ::= e | l | le | g | ge
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)
           | (Instr xorq ( arg arg)) | (Instr cmpq ( arg arg))
           | (Instr set ( cc arg)) | (Instr movzbq ( arg arg))
           | (JmpIf cc label)
block    ::= (Block info (instr ...))
x86If    ::= (X86Program info ((label . block) ...))
```

图 4.9: $\text{x86}_{\texttt{If}}$ 的抽象语法 (扩展图 2.8 中的 $\text{x86}_{\texttt{Int}}$)。

接下来，我们考虑与编译比较操作相关的 x86 指令。`cmpq` 指令比较它的两个参数来确定一个参数是小于、等于还是大于另一个参数。`cmpq` 指令在参数的顺序和结果放置的位置上是不同寻常的。参数顺序颠倒：如果你想测试是否 $x < y$ ，那么写 `cmpq` y, x 。`cmpq` 的结果被放置在特殊的 EFLAGS 寄存器中。这个寄存器不能直接访问，但可以通过许多指令进行查询，包括 `set` 指令。指令 `set`cc d 根据条件代码 cc (`e` 表示等于，`l` 表示小于，`le` 表示不等于，`g` 表示大于，`ge` 表示大于等于) ，将 `1` 或 `0` 放入目标 d 中。`set` 指令有一个令人讨厌的怪癖，因为它的目标参数必须是单字节寄存器，例如 `al` (低位为 L) 或 `ah` (高位为 H)，它们是 `rax` 寄存器的一部分。值得庆幸的是，可以使用 `movzbq` 指令将单个字节寄存器移到普通 64 位寄存器。`set` 指令的抽象语法不同于具体语法，因为它将指令名与条件代码分开。

x86 条件跳转指令与 `if` 表达式的编译有关。指令 `j`cc *label* 更新程序计数器指向 *label* 之后的指令，这取决于 EFLAGS 寄存器中的结果是否与条件代码 cc 匹配，否则跳转指令将跳转到下一条指令。与 `set` 的抽象语法一样，条件跳转的抽象语法将指令名与条件代码分开。例如，`(JmpIf le foo)` 对应于 `jle foo` 。因为条件跳转指令依赖于 EFLAGS 寄存器，所以通常它前面会立即有一个 `cmpq` 指令来设置 EFLAGS 寄存器。

4.5 缩小 $R_{\texttt{If}}$ 语言

$R_{\texttt{If}}$ 语言包含几个操作符，可以很容易地用其他操作符表示。例如，减法可以用加法和否定来表示。

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

一些比较操作可以使用小于和逻辑否定来表示。

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

在上面的转换中需要 `let` ，以确保表达式 e_1 在 e_2 之前求值。

通过在编译器前端执行这些转换，编译器后面的通道不需要处理这些操作符，使得通道更短。

Exercise 19. 通过将它们转换为 $R_{\texttt{If}}$ 中的其他结构，实现 `shrink` 通道来从语言中删除减法、`and` 、`or` 、`<=` 、`>` 和 `>=` 。创建 6 个包含这些操作符的测试程序。在 `run-tests.rkt` 脚本中，将下面的 `shrink` 条目添加到通道列表中 (此时它应该是唯一的传递)。

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

这指示 `interp-tests` 在 `shrink` 输出上运行解释器 `interp-Rif` 和类型检查器 `type-check-Rif` 。运行该脚本在所有测试程序上测试编译器。

4.6 统一变量

将个案添加到 `uniquify-exp` 以处理布尔常量和 `if` 表达式。

Exercise 20. 更新 $R_{\texttt{If}}$ 的 `uniquify-exp` ,并将以下条目添加到 `run-tests.rkt` 脚本中的 `passes` 列表中。

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

运行脚本来测试编译器。

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

图 4.10: $R_{\texttt{if}}^{\texttt{ANF}}$ 是管理范式 (ANF) 中的 $R_{\texttt{If}}$ 。

4.7 去除复杂的操作数

此通道的输出语言是 $R_{\texttt{if}}^{\texttt{ANF}}$ (图 4.10) ，$R_{\texttt{If}}$ 的管理标准形式。Bool 形式是一个原子表达式，但 If 不是。If 的三个子表达式都可以是复杂表达式，但 not 的操作数和比较操作数必须是原子。

在 rco-exp 和 rco-atom 函数中添加 Bool 和 If 的情况，根据输出是需要 *exp* 还是在 $R_{\texttt{if}}^{\texttt{ANF}}$ 语法中指定的 *atm* 。关于 If ，特别重要的是 **not** 将其条件替换为临时变量，因为这干扰在 explicate-control 通道中生成高质量的输出。

Exercise 21. 在 compiler.rkt 中的 rco-atom 和 rco-exp 函数中添加布尔常量和 if 的大小写。创建三个新的 $R_{\texttt{Int}}$ 程序来执行本文中有趣的代码。在 run-tests.rkt 脚本中，将以下条目添加到 passes 列表中，然后运行该脚本来测试编译器。

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

4.8 说明控制

回想一下，explicate-control 的目的是使程序语法中的求值顺序显式。加上 if 这个会更有趣。

下面的程序有一个 if 表达式嵌套在另一个 if 的谓词中，这是一个有启发性的例子。

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

编译 `if` 和比较的简单方法是孤立地处理它们中的每一个，而不考虑它们的上下文。每次比较都将被转换为一个 `cmpq` 指令，后面跟着两个指令，将结果从 EFLAGS 寄存器移动到一个通用寄存器或堆栈位置。每个 `if` 都将被翻译成一个 `cmpq` 指令，后面跟着一个条件跳转。上面例子中为内部 `if` 生成的代码如下所示。

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

但是，如果考虑到上下文，可以做得更好，并减少使用 `cmpq` 指令来访问 EFLAG 寄存器。

我们的目标是编译 `if` 表达式，以便相关的比较指令直接出现在条件跳转之前。例如，希望为内部 `if` 生成以下代码。

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

实现这一点的一种方法是在 $R_{\texttt{If}}$ 级别上重新组织代码，将外部的 `if` 推到内部的 `if` 中，产生以下代码。

```
(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)))))
```

不幸的是，这种方法从外部的 `if` 复制两个分支，编译器绝对不能复制代码!

需要一种方法来执行上述转换，但又不复制代码。也就是说，需要一种方法让程序的不同部分引用同一段代码。在 x86 汇编的层次上，这是很简单的，因为可以为每个分支标记代码，并在需要执行分支的所有地方插入跳转。在中间语言中，需要远离抽象的语法 *树*，而使用 *图*。特别是，使用一种称为 *控制流图* (CFG) 的标准程序表示，这是由 Frances Elizabeth Allen [4] 提出的。每个顶点是一个标记的代码序列，称为 *基本块*，每条边表示到另一个块的跳转。$C_{\texttt{Var}}$ 的 $C_{\texttt{If}}$ 的 `CProgram` 结构包含一个控制流图，表示为一个将标签映射到基本块的列表。每个基本块都由 *tail* 非终端表示。

图 4.11 显示示例程序中 `remove-complex-opera*` 通道和 `explicate-control` 通道的输出。遍历输出程序，然后讨论算法。按照 `remove-complex-opera*` 输出中的求值顺序，首先对 `(read)` 进行两次调用，然后对内部 `if` 谓词中的比较 `(< x 1)` 进行调用。在 `explicate-control` 的输出中，在标记为 `start` 的块中，是两个赋值语句，后面跟着一个 `if` 语句，该语句分支到 `block40` 或 `block41` 。与这些标签相关的块分别包含代码 `(eq? x 0)` 和 `(eq? x 2)` 的翻译。特别地，以比较 `(eq? x 0)` 开始 `block40` ，然后分支到 `block38` 或 `block39` ，这是外部 `if` 的两个分支，即 `(+ y 2)` 和 `(+ y 10)` 。`block41` 的情况也类似。

回想一下，在 2.6 节中，使用两个相互递归的函数，`explicate-tail` 和 `explicate-assign` 来实现 $R_{\texttt{Var}}$ 的 `explicate-control` 。前一个函数在尾部位置转换表达式，而后一个函数在 `let` 的右侧转换表达式。随着在 $R_{\texttt{If}}$ 中添加 `if` 表达式，我们有一种新的位置需要处理：`if` 语句的谓语位

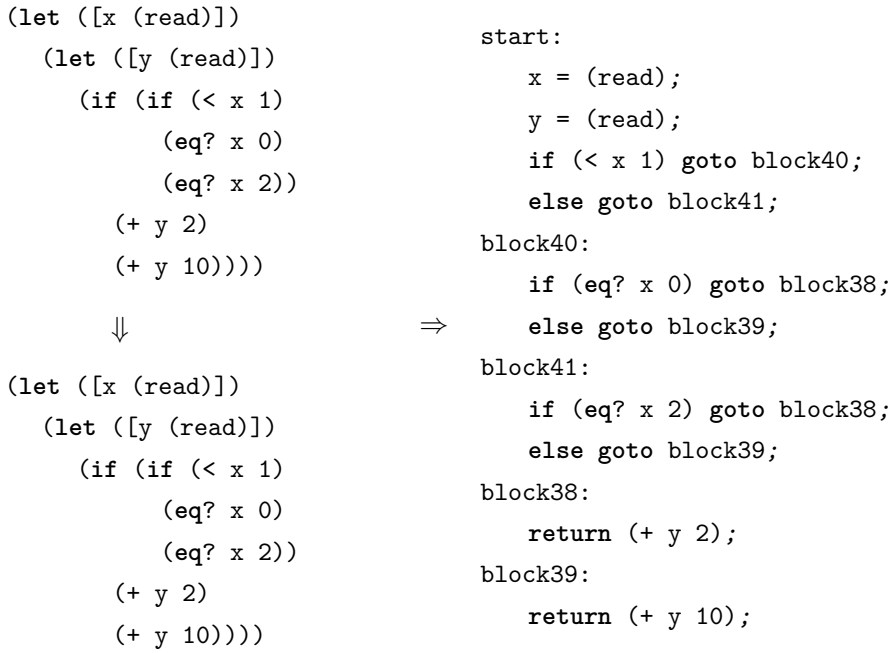


图 4.11: 通过 `explicate-control` 将 $R_{\texttt{If}}$ 转换为 $C_{\texttt{If}}$ 。

```
(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)]))
```

图 4.12: explicate-pred 辅助功能的骨架。

置。我们需要另一个函数，explicate-pred 它需要一个 $R_{\texttt{If}}$ 表达式，然后为 then-branch 和 else-branch 分配两个块。explicate-pred 的输出是一个块。在下面的段落中，讨论 explicate-pred 函数的具体情况，以及 explicate-tail 和 explicate-assign 函数的补充。

explicate-pred 函数的框架如图 4.12 所示。对于每一个可以有 Boolean 类型的表达式，它都有大小写。在这里详细介绍几个案例，其余的留给读者。这个函数的输入是一个表达式和两个块，thn 和 els ，用于封闭 if 的两个分支。考虑图 4.12 中布尔常量的情况。执行一种部分计算 并根据常数是真还是假输出 thn 或 els 分支。这个例子说明有时会丢弃输入到 explicate-pred 中的 thn 或 els 块。

explicate-pred 中 if 的例子特别具有启发性，因为它处理上面讨论的关于嵌套的 if 表达式的挑战 (图 4.11) 。if 的 thn^ 和 els^ 分支从当前上下文继承它们的上下文，即谓词上下文。所以应该递归应用 explicate-pred 到 thn^ 和 els^ 分支。对于这两个递归调用，传递 thn 和 els 作为额外参数。因此，thn 和 els 可以使用两次，每次在递归调用中使用一次。如前所述，为了避免代码重复，需要将它们添加到控制流图中，以便可以通过名称引用它们并使用 goto 执行它们。然而，正如在上面的布尔常量案例中所看到的，块 thn 和 els 可能根本不会被使用，如果它们最终被丢弃，不想过

早地将它们添加到控制流图中。

这个难题的解决方案是使用 惰性求值[48] 来延迟将块添加到控制流图中，直到我们知道将使用它们的点。Racket 通过 racket/promise 包提供对惰性求值的支持。表达式 (delay $e_1 \ldots e_n$) 创建一个 承诺 ，其中表达式的求值被推迟。当 (force p) 第一次应用于承诺 p 时，表达式 $e_1 \ldots e_n$ 被计算，e_n 的结果被缓存在承诺中并返回。如果 force 再次应用于相同的承诺，则返回缓存的结果。如果将 force 应用于不是承诺的参数，则 force 只会返回该参数。

对函数 explicate-pred 和 explicate-assign 的输入和输出块以及 explicate-tail 的输出块使用惰性求值。所以它们不是拿回方块，而是拿回承诺。此外，当遇到一个区块可能被多次使用的情况时，例如在 explicate-pred 的 if 情况下，将承诺转换为一个新的承诺，该新的承诺会将区块添加到控制流图并返回 goto 。下面的辅助函数 block->goto 将完成此任务。它以 delay 开始，以创造一个承诺。当这个承诺被强迫的时候，这个承诺就会强迫原来的承诺。如果返回 goto (因为块已经添加到控制流图中) ，则返回 goto 。否则，使用另一个名为 add-node 的辅助函数将块添加到控制流图中。该函数返回新块的标签，用它来创建 goto 。

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

回到 explicate-pred 的讨论 (图 4.12) ，考虑比较运算符的情况。这是递归函数的基本情况之一，所以我们将比较转换为 if 语句。将 block->goto 应用于 thn 和 els 以获得两个承诺，这两个承诺将把 then 添加到控制流图中，可以立即进行 force 获得 if 语句分支中的两个 goto 。

explicate-tail 和 explicate-assign 函数需要额外的布尔常量和 if 。在 if 的情况下，两个分支继承当前上下文，所以在 explicate-tail 中它们处于尾部位置，在 explicate-assign 中它们处于赋值位置。在两次递归调用中都使用 explicate-assign 的 cont 参数，所以请确保使用

block->goto 。

shrink 通过转换逻辑操作的方式,如 and 和 or 可以影响由 explicate-control 生成的代码的质量。例如，考虑下面的程序。

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

and 操作应该转换成一些东西，explicate-pred 函数仍然可以分析和下降，以达到潜在的 eq? 条件。理想情况下，explicate-control 通道应该为上面的程序生成类似如下的代码。

```
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. 通过在 explicate-tail 和 explicate-assign 中添加布尔常量和 if 来实现 explicate-control 的通道。为谓词上下文实现辅助函数 explicate-pred 。创建测试用例，测试代码中的所有新用例。将以下条目添加到 run-tests.rkt 中的 passes 列表中，然后运行此脚本来测试编译器。

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

4.9 选择指令

`select-instructions` 通道将 $C_{\texttt{If}}$ 转换为 $\text{x86}^{\texttt{Var}}_{\texttt{If}}$ 。回想一下，使用三个辅助函数来实现这个通道，分别是非终端 *atm* 、*stmt* 和 *tail* 。

对于 *atm* ，有一些新的布尔值案例。通常的方法将它们编码为整数，true 为 1，false 为 0。

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

对于 *stmt* ，讨论几个案例。正如在本节开始时讨论的那样，`not` 操作可以通过 `xorq` 实现。给定赋值 *var* `= (not` *atm*`);` ，如果左边的 *var* 与 *atm* 相同，那么只要 `xorq` 就足够了。

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

否则，需要一个 `movq` 来适应 x86 的就地更新语义。将 *arg* 设为 *atm* 转换为 x86 的结果。然后我们有

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

接下来考虑 `eq?` 和小于比较的情况。由于上面讨论的 `cmpq` 指令的不寻常性质，将这些操作转换到 x86 稍微有些困难。建议将 `eq?` 中的赋值翻译成以下三个指令序列。

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

关于 *tail* 非终端，有两个新的例子：`goto` 和 `if` 语句。两者转换到 x86 都很简单。一个 `goto` 变成一个跳转指令。

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

`if` 语句变成一个比较指令，后面跟着一个条件跳转 (对于” then ” 分支来说)，而跌落是一个常规跳转 (对于” else ” 分支来说)。

`if (eq?` atm_1 atm_2`) goto` ℓ_1`;` `else goto` ℓ_2`;`	$\Rightarrow$	`cmpq` arg_2`,` arg_1 `je` ℓ_1 `jmp` ℓ_2

Exercise 23. 扩展你的 `select-instructions` 通道来处理 $R_{\texttt{If}}$ 语言的新特性。将以下条目添加到 `run-tests.rkt` 中的 `passes` 列表中

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

运行该脚本在所有测试程序上测试编译器。

4.10 寄存器分配

$R_{\texttt{If}}$ 所需的变化影响活性分析、构建干扰图和分配域，但图着色算法本身不改变。

4.10.1 活性分析

回想一下，对于 $R_{\texttt{Var}}$，我们为单个基本块实现活性分析 (第 3.2 节)。通过在 $R_{\texttt{If}}$ 中添加 `if` 表达式，`explicate-control` 生成许多基本块，这些块被安排在一个控制流图中。建议创建一个新的辅助函数，名为 `uncover-live-CFG`，将活性分析应用到控制流图。

第一个问题是：应该以什么顺序处理控制流图中的基本块？回想一下，要在一个基本块上执行活性分析，需要知道它的 live-after 集合。如果一个基本块没有继任者 (即在控制流图中没有外边)，那么它就有一个空的 live-after 集，可以立即对它应用 live-after 分析。如果一个基本块有一些后继体，那么需要先完成这些块的活性分析。在图论中，如果一个节点的每个顶点在其后续顶点之前，那么这个节点序列就是 *拓扑顺序* 。我们需要反义词，这样就可以在计算拓扑顺序之前转置这个图。用 Racket `graph` 包的 `tsort` 和 `transpose` 函数来实现这一点。另外，拓扑顺序只有在图中不包含任何循环时才保证存在。这确实是我们从 $R_{\texttt{If}}$ 程序生成的控制流图的情况。然而，在第 9 章中，在 $R_{\texttt{While}}$ 中添加循环，并学习如何在控制流图中处理循环。

您需要构造一个有向图来表示控制流图。不要使用 graph 包中的 directed-graph，因为它最多只允许在每一对顶点之间有一条边，但是一个控制流图可能在一对顶点之间有多条边。支持代码中的 multigraph.rkt 文件实现一种图表示，允许一对顶点之间有多条边。

下一个问题是如何分析跳跃指令。回想一下，在第 3.2 节中，我们维护一个名为 label->live 的列表，它将每个标签映射到其块开始处的 live 位置集。使用 label->live 来确定每个（Jmp *label*）指令的 live-before 设置。现在有许多基本的块，label->live 需要在处理这些块时进行更新。特别是，在对一个块执行活性分析之后，取其第一个指令的 live-before 集合，并将其与 label->live 中的块的标签相关联。

在 $\text{x86}^{\texttt{Var}}_{\texttt{If}}$ 中，还需要处理条件跳转（JmpIf *cc label*）。这条指令的动态分析特别有趣，因为在编译期间，不知道条件跳转将朝哪个方向发展。因此，不知道是否使用 live-before 设置为下面的指令或 live-before 设置为 *label* 。然而，如果将更多的位置划分为实时位置，而不是在特定的指令执行过程中实际运行的位置，则不会对编译器的正确性造成损害。因此，可以从下面的指令和 label->live 中的 *label* 映射中获得 live-before 集合的并集。

用于计算指令参数中的变量和用于计算指令从 (R) 或写入 (W) 的变量的辅助函数需要被更新，以处理 $\text{x86}^{\texttt{Var}}_{\texttt{If}}$ 中的新类型的参数和指令。

Exercise 24. 更新 uncover-live 通道，并实现 uncover-live-CFG 辅助函数，将活性分析应用到控制流图中。将以下条目添加到 run-tests.rkt 脚本中的 passes 列表中。

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

4.10.2 构建干涉图

$\text{x86}^{\texttt{Var}}_{\texttt{If}}$ 中的许多新指令可以用与 $\text{x86}_{\texttt{Var}}$ 中的指令相同的方式处理。因此，如果代码已经相当通用，那么就不需要更改代码来处理新的指令。如果代码不够通用，建议将代码更改为更通用的代码。例如，可以将每种指令的读写集计算分解为两个辅助函数。

注意，与 movq 指令类似，movzbq 指令需要一些特殊的注意事项。参见 3.3 节规则 1。

Exercise 25. 更新 $\text{x86}_{\texttt{If}}^{\texttt{Var}}$ 的 `build-interference` 通道，并将以下条目添加到 `run-tests.rkt` 的 `passes` 列表中。

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

运行该脚本在所有的 $R_{\texttt{If}}$ 测试程序上测试编译器。

4.11 补丁说明

`cmpq` 指令的第二个参数不能是立即值 (例如整数)。因此，如果正在比较两个即时对象，建议插入一个 `movq` 指令，将第二个参数放入 `rax` 中。另外，记住，指令最多只能有一个内存引用。`movzbq` 的第二个参数必须是一个寄存器。对于跳转指令没有特殊的限制。

Exercise 26. 更新 $\text{x86}_{\texttt{If}}^{\texttt{Var}}$ 的 `patch-instructions` 通道。将以下条目添加到 `run-tests.rkt` 的 `passes` 列表中，然后运行此脚本来测试编译器。

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

图 4.13 列出编译 $R_{\texttt{If}}$ 所需的所有步骤。

4.12 一个例子的翻译

图 4.14 是一个简单的 $R_{\texttt{If}}$ 翻译成 x86 的示例程序,显示 `explicate-control`, `select-instructions` 的结果，以及最终的 x86 汇编代码。

4.13 挑战: 删除跳转

有一个优化跳转的机会，这在图 4.14 的例子中很明显。`start` 块以跳转到 `block7953` 结束，在程序的其余部分中没有其他跳转到 `block7953` 。在这种情况下，可以通过合并 `block7953` 到前面的块来避免运行时开销，在这种情况下是 `start` 块。图 4.15 左边显示 `select-instructions` 的输出，右边显示优化的结果。

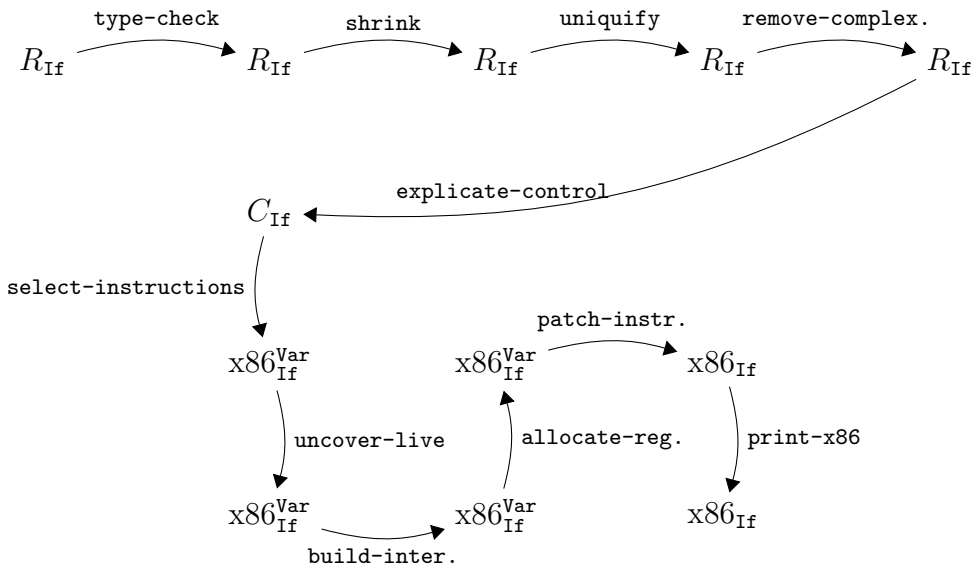


图 4.13: 带条件的 $R_{\texttt{If}}$ 语言的通道图。

Exercise 27. 实现一个名为 `remove-jumps` 的通道, 当只有一个前面的块时, 它将基本块合并到其前面的基本块中。这个通道应该从 $\text{x86}^{\texttt{Var}}_{\texttt{If}}$ 转换到 $\text{x86}^{\texttt{Var}}_{\texttt{If}}$ 。在 `run-tests.rkt` 脚本中, 将以下条目添加到 `allocate-registers` 和 `patch-instructions` 之间的 `passes` 列表中。

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

运行此脚本来测试编译器。检查 `remove-jumps` 是否完成在几个测试程序中合并基本块的目标。

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

⇓

```
start:
   tmp7951 = (read);
   if (eq? tmp7951 1)
     goto block7952;
   else
     goto block7953;
block7952:
   return 42;
block7953:
   return 0;
```

⇓

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

⇒

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

       .globl main
main:
       pushq  %rbp
       movq   %rsp, %rbp
       pushq  %r13
       pushq  %r12
       pushq  %rbx
       pushq  %r14
       subq   $0, %rsp
       jmp start
conclusion:
       addq   $0, %rsp
       popq   %r14
       popq   %rbx
       popq   %r12
       popq   %r13
       popq   %rbp
       retq
```

图 4.14: 将 `if` 表达式编译到 x86 的示例。

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

⇒

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

图 4.15: 通过删除不必要的跳转来合并基本块。

5

元组和垃圾收集

在本章中，研究可变元组 (在 Racket 中称为向量) 的实现。这个语言特性是第一个使用计算机 堆 的特性，因为 Racket 元组的生存期是不确定的，也就是说，从程序员的角度来看，元组是永远存在的。当然，从实现者的角度来看，当不再需要与元组相关的空间时，回收空间是很重要的，这就是为什么在本章也要学习 垃圾回收技术。

5.1 节介绍 $R_{\texttt{Vec}}$ 语言，包括它的解释器和类型检查器。$R_{\texttt{Vec}}$ 语言通过向量和 Racket 的 `void` 值扩展第 4 章的 $R_{\texttt{If}}$ 语言。包含后者的原因是 `vector-set!` 操作返回一个类型为 `Void`[1] 的值。

第 5.2 节描述一种基于在堆的两个部分之间来回复制活动对象的垃圾收集算法。垃圾收集器需要与编译器协调，以便它可以看到所有的 *root* 指针，即寄存器或过程调用堆栈上的指针。

第 5.4 至 5.9 节讨论对编译器通道的所有必要修改和添加，包括名为 `expose-allocation` 新的编译器通道。

[1]Racket 的 `Void` 类型对应于编程语言文献中所谓的 `Unit` 类型。Racket 的 `Void` 类型由一个单独的值 `void` 组成，在文献 [93] 中对应于 `unit` 或 `()`。

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

图 5.1: $R_{\texttt{Vec}}$ 的具体语法，扩展 $R_{\texttt{If}}$ (图 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))
```

图 5.2: 创建元组并从中读取的示例程序

5.1 $R_{\texttt{Vec}}$ 语言

图 5.1 定义 $R_{\texttt{Vec}}$ 的具体语法，图 5.3 定义抽象语法。$R_{\texttt{Vec}}$ 语言包括三种新形式：`vector` 用于创建元组，`vector-ref` 用于读取元组中的元素，`vector-set!` 用于写入元组中的元素。图 5.2 中的程序显示元组在 Racket 中的使用。创建一个 3 元组 `t` 和一个 1 元组，它们存储在 3 元组的下标 2 处，证明元组是一等值。`t` 的下标 1 处的元素是 `#t` ，因此取 `if` 的 “then” 分支。`t` 下标 0 处的元素是 `40` ，再加上 `2` ，即 1 元组下标 0 处的元素。所以这个程序的结果是 `42` 。

元组是我们第一次遇到堆分配的数据，这引发几个有趣的问题。首先，变量绑定在处理元组时执行浅拷贝，这意味着不同的变量可以引用相同的元

$$
\begin{array}{lcl}
\mathit{op} & ::= & \ldots \mid \texttt{vector} \mid \texttt{vector-length} \\
\mathit{exp} & ::= & \texttt{(Int } \mathit{int}\texttt{)} \mid \texttt{(Var } \mathit{var}\texttt{)} \mid \texttt{(Let } \mathit{var}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
& \mid & \texttt{(Prim } \mathit{op}\ \texttt{(}\mathit{exp}\ldots\texttt{))} \mid \texttt{(Bool } \mathit{bool}\texttt{)} \mid \texttt{(If } \mathit{exp}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
& \mid & \texttt{(Prim vector-ref (} \mathit{exp}\ \texttt{(Int } \mathit{int}\texttt{)))} \\
& \mid & \texttt{(Prim vector-set! (} \mathit{exp}\ \texttt{(Int } \mathit{int}\texttt{)}\ \mathit{exp}\texttt{))} \\
& \mid & \texttt{(Void)} \mid \texttt{(HasType } \mathit{exp}\ \mathit{type}\texttt{)} \\
R_{\texttt{Vec}} & ::= & \texttt{(Program '()} \ \mathit{exp}\texttt{)}
\end{array}
$$

图 5.3: $R_{\texttt{Vec}}$ 的抽象语法。

组，也就是说，不同的变量可以是相同实体的 *别名*。考虑下面的例子，其中 `t1` 和 `t2` 都引用同一个元组。因此，在从 `t1` 引用元组时，`t2` 的变化是可见的，因此这个程序的结果是 `42` 。

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

下一个问题涉及元组的生命周期。当然，它们是由 `vector` 形式创造的，但是它们的生命周期何时结束呢？注意，$R_{\texttt{Vec}}$ 不包含删除元组的操作。此外，元组的生存期不与任何静态作用域的概念相关联。例如，下面的程序返回 `42` ，即使变量 `w` 在 `vector-ref` (从绑定到它的向量中读取) 之前超出了作用域。

```
(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))))
```

从程序员可观察行为的角度来看，元组永远存在。当然，如果它们真的

永远存在，那么许多程序将耗尽内存。[2] 因此，Racket 实现必须执行自动垃圾收集。

图 5.4 显示 $R_{\texttt{Vec}}$ 语言的定义解释器。根据相应的 Racket 操作为 r $R_{\texttt{Vec}}$ 定义 `vector` 、`vector-length` 、`vector-ref` 和 `vector-set!` 操作。一个微妙的点是，`vector-set!` 操作返回 `#<void>` 值。`#<void>` 值可以像 $R_{\texttt{Vec}}$ 程序中的其他值一样传递，`#<void>` 值可以与另一个 `#<void>` 值进行相等比较。然而，在 $R_{\texttt{Vec}}$ 中没有其他特定于 `#<void>` 值的操作。相反，Racket 定义 `void?` 谓词，当应用于 `#<void>` 时返回 `#t` ，否则返回 `#f` 。

图 5.5 显示 $R_{\texttt{Vec}}$ 的类型检查器，它需要一些解释。分配向量时，我们需要知道向量的哪些元素是指针（即也是向量）。可以在类型检查中获得这些信息。图 5.5 的类型检查器不仅计算表达式的类型，还将每个创建的 `vector` 包装为 (`HasType` $e\ T$) 形式，其中 T 是向量的类型。要为图 5.5 中的 `Vector` 类型创建 s 表达式，使用 unquote-splicing operator `,@` 插入列表 `t*` ，而不使用通常的开始和结束括号。

5.2 垃圾回收

这里研究一个相对简单的垃圾收集算法，它是最先进的垃圾收集器的基础 [78, 110, 66, 34, 39, 108] 。特别地，描述一个使用 Cheney 算法执行复制 [25] 的两空间复制收集器 [113] 。 图 5.6 给出两空间收集器的粗粒度描述，显示两个时间步骤，在垃圾收集之前 (在上面) 和之后 (在下面)。在一个两空间收集器中，堆分为两个部分，分别命名为 FromSpace 和 ToSpace 。最初，所有分配都转到 FromSpace，直到没有足够的空间用于下一个分配请求。这时，垃圾收集器开始工作以腾出更多空间。

垃圾收集器必须小心，不要回收将来将被程序使用的元组。当然，通常不可能预测一个程序将做什么，但是我们可以通过保留所有在给定当前计算机状态下可以被 任何程序访问的元组来过度近似将要被使用的元组。程序可以访问地址在寄存器或过程调用堆栈上的任何元组。这些地址称为 根集。

[2] $R_{\texttt{Vec}}$ 语言没有循环或递归函数，所以在 $R_{\texttt{Vec}}$ 中编写一个耗尽内存的程序几乎是不可能的!

```
(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))
```

图 5.4: 用于 $R_{\texttt{Vec}}$ 语言的解释器。

```
(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))
```

图 5.5: $R_{\texttt{Vec}}$ 语言的类型检查器。

此外，程序可以访问从根集传递可达的任何元组。因此，垃圾收集器可以安全地回收无法通过这种方式访问的元组。

所以垃圾收集器的目标有两个：

1. 保留所有可通过指针路径从根集到达的元组，即 *活元组*；

2. 回收其他所有东西的内存，也就是 *垃圾*。

复制收集器通过将所有活动对象从 FromSpace 复制到 ToSpace 来实现这一点，然后执行一种巧妙的操作，将 ToSpace 当作新的 FromSpace，将旧的 FromSpace 当作新的 ToSpace。在图 5.6 的例子中，根集中有三个指针，一个在寄存器中，两个在堆栈中。所有活动对象都以保留指针关系的方式复制到 ToSpace(图 5.6 的右侧)。例如，寄存器中的指针仍然指向一个二元组，它的第一个元素是一个三元组，第二个元素是一个二元组。有 4 个元组不能从根集到达，因此不能复制到 ToSpace 中。

在 $R_{\texttt{Vec}}$ 中，一个类型良好的程序不能创建图 5.6 中的确切情况，因为它包含一个循环。然而，一旦我们到达 $R_{\texttt{Any}}$ ，创建循环将成为可能。我们首先设计垃圾收集器来处理循环，因此不需要重新讨论这个问题。

当涉及到垃圾收集时，除了复制收集器 (及其更大的兄弟，分代收集器) 外，还有许多替代方法，例如标记-清除 [84] 和引用计数 [28] 。复制收集器的优点是分配速度快 (只需要比较和指针增量)，没有碎片，回收循环垃圾，收集的时间复杂性只取决于实时数据量，而不是垃圾量 [113] 。两空间复制收集器的主要缺点是它使用大量空间，并且执行复制需要很长时间，尽管这些问题在分代收集器中得到改善。Racket 和 Scheme 程序倾向于分配许多小对象并生成大量垃圾，因此复制和分代收集器是一个很好的选择。垃圾收集是一个活跃的研究课题，特别是并发垃圾收集 [108] 。研究人员正在不断开发新技术，并重新考虑旧的权衡 [13, 67, 98, 32, 99, 91, 64, 49] 。每年研究人员都会在国际记忆管理研讨会上发表这些发现。

5.2.1 通过切尼算法复制图形

让我们仔细看看活动对象的复制。分配的对象和指针可以被视为一个图，需要复制从根集可以到达的部分图。为了确保复制图中所有可到达的顶

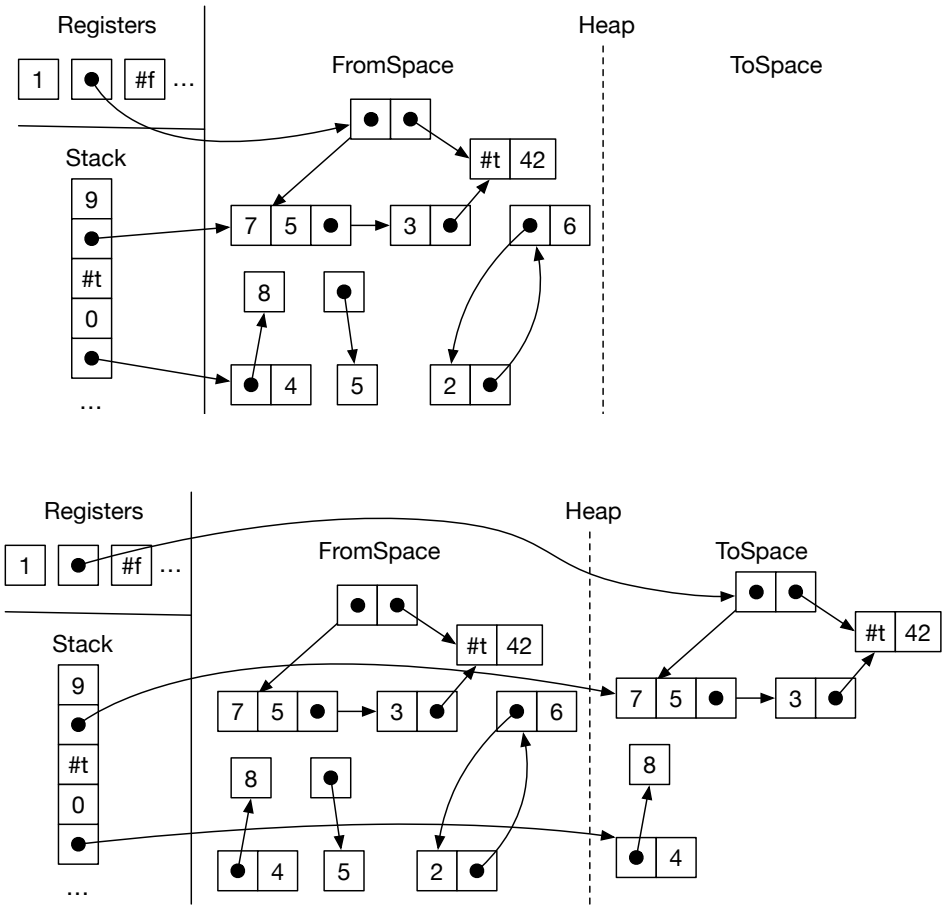


图 5.6: 操作中的复制收集器。

点，需要一个穷举图遍历算法，例如深度优先搜索或宽度优先搜索 [89, 31] 。回想一下，这样的算法通过标记哪些顶点已经被访问来考虑循环的可能性，从而确保算法的终止。这些搜索算法还使用堆栈或队列等数据结构作为待办事项列表，以跟踪需要访问的顶点。使用宽度优先搜索和 Cheney [25] 的一个技巧，用于同时表示队列和将元组复制到 ToSpace 中。

图 5.7 显示在复制过程中 ToSpace 的几个快照。队列由 ToSpace 开头的一块连续内存表示，使用两个指针跟踪队列的前端和后部。该算法首先将所有可以立即从根集到达的元组复制到 ToSpace 中，以形成初始队列。当复制一个元组时，将旧元组标记为已访问过它。在第 5.2.2节讨论如何完成这个标记。注意，队列中复制的元组内的任何指针仍然指向 FromSpace。一旦创建初始队列，算法将进入一个循环，在此循环中它将重复处理队列前面的元组，并将其从队列中弹出。为了处理一个元组，该算法将从该元组中直接可达的所有元组复制到 ToSpace，将它们放在队列的后面。然后算法更新弹出元组中的指针，使它们指向新复制的元组。

回到图 5.7 ，在第一步中，将第二个元素为 42 的元组复制到队列的后面。另一个指针指向一个已经复制的元组，因此不需要再次复制它，但需要更新指向新位置的指针。这可以通过在旧元组中存储一个指向新位置的 *转发指针*来实现，这可以追溯到最初将元组复制到 ToSpace 中的时候。这完成算法的一个步骤。该算法以这种方式继续，直到队列的前端为空，也就是说，直到前端赶上后端。

5.2.2 数据表示法

垃圾收集器对编译器使用的数据表示提出一些要求。首先，垃圾收集器需要区分指针和其他类型的数据。有几种方法可以实现这一点。

1. 在每个对象上附加一个标签来识别它是什么类型的对象 [84] 。

2. 在不同的区域存储不同类型的对象 [106] 。

3. 使用来自程序的类型信息来生成用于收集的特定类型代码，或者生成可以指导收集器的表 [6, 54, 36] 。

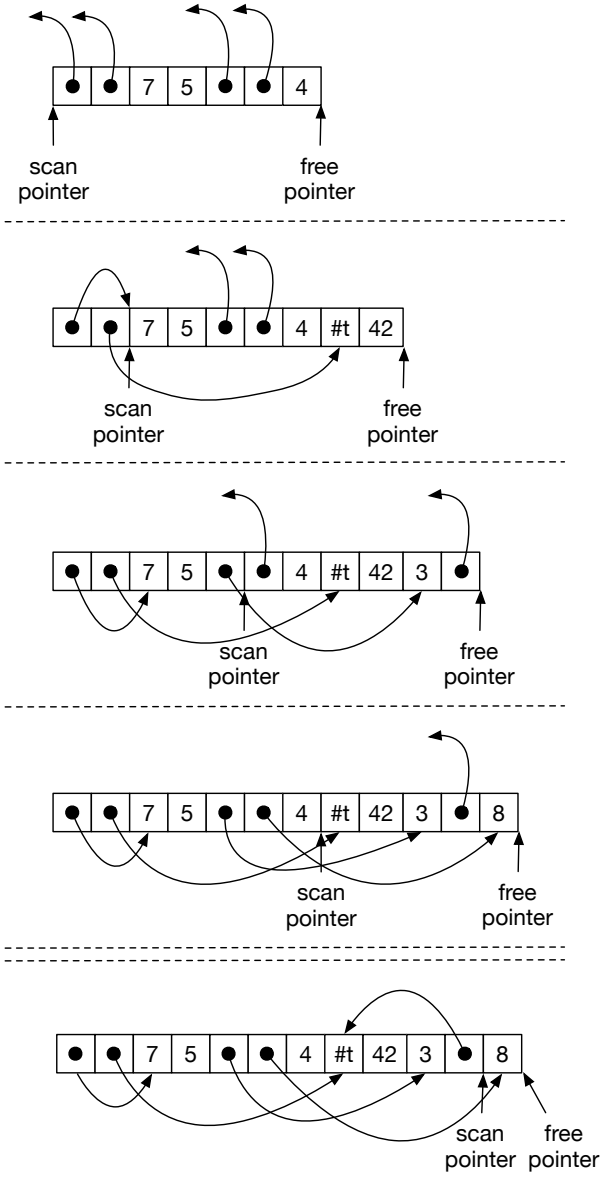


图 5.7: 复制活元组的 Cheney 算法的描述。

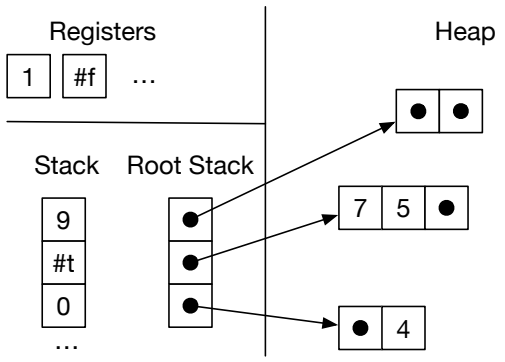


图 5.8: 维护根堆栈以方便垃圾收集。

动态类型语言，如 Lisp，需要标记对象，因此选项 1 是这些语言的自然选择。然而，$R_{\texttt{Vec}}$ 是一种静态类型语言，因此在每个对象上都需要标记是很不幸的，特别是像整数和布尔值这样的小而普遍的对象。选项 3 是静态类型语言的最佳性能选择，但是其实现复杂性相对较高。为了保证本章在 2 周内完成，我们建议将选项 1 和选项 2 结合起来，对堆栈和堆使用不同的策略。

关于堆栈，建议为指针使用一个单独的堆栈，我们称之为 *根堆栈* (也称为 “影子堆栈”) [101, 58, 11] 。也就是说，当一个局部变量需要被溢出并且类型为 (`Vector` $type_1 \ldots type_n$) 时，将其放在根堆栈上，而不是普通的过程调用堆栈上。此外，如果向量类型的变量在调用收集器期间是活的，总是溢出它们，从而确保在集合期间没有指针在寄存器中。图 5.8 再现图 5.6 中的示例，并使用根堆栈将其与数据布局进行对比。根堆栈包含两个来自常规堆栈的指针，以及第二个寄存器中的指针。

指针和其他类型的数据之间的区别问题也出现在堆上的每个元组中。通过给每个元组附加一个额外的 64 位标签来解决这个问题。图 5.9 放大了图 5.6 中示例中的两个元组的标记。请注意，我们以大端方式绘制了位，从右到左，位位置 0(最低有效位) 在最右边，它对应着 x86 移动指令 `salq` (左移) 和 `sarq` (右移) 的方向。每个标记的一部分用于指定元组中哪些元素是指针，标记为 “指针掩码” 的部分。在指针掩码中，1 位表示有一个指针，0 位表示其他类型的数据。指针掩码从位 7 开始。将元组的最大大小限制为

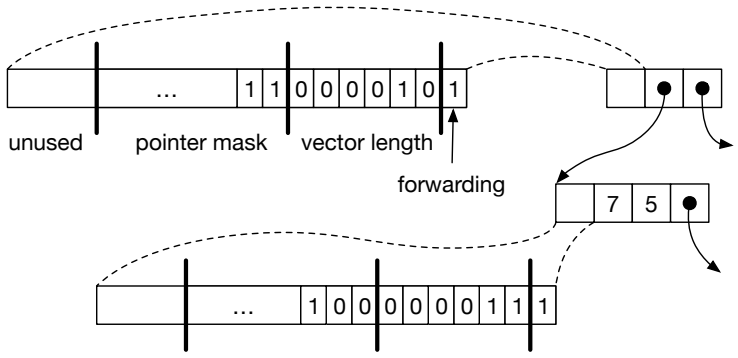


图 5.9: 堆中元组的表示。

50 个元素，因此指针掩码只需要 50 位。标签还包含另外两部分信息。元组的长度 (元素的数量) 存储在位 1 到 6 的位置。最后，位置 0 的位表示元组是否还没有被复制到 ToSpace。如果位值为 1，则该元组还没有被复制。如果位值为 0，则整个标签是一个转发指针。(指针的低 3 位总是 0，因为我们的元组是 8 字节对齐的。)

5.2.3 垃圾收集器的实现

`runtime.c` 文件中提供复制收集器的实现。图 5.10 定义编译器使用的垃圾收集器的接口。`initialize` 函数创建 FromSpace、ToSpace 和根堆栈，应该在 `main` 函数的前奏中调用。`initialize` 的参数是根堆栈大小和堆大小。两者都需要是 64 的倍数，16384 对两者来说都是不错的选择。`initialize` 函数将 FromSpace 开头的地址放入全局变量 `free_ptr` 中。全局变量 `fromspace_end` 指向的地址是 FromSpace 的最后一个元素的 1 位。(我们使用半开的间隔来表示内存块 [35] 。) `rootstack_begin` 变量指向根堆栈的第一个元素。

只要 FromSpace 中还有剩余空间，生成的代码就可以通过向前移动 `free_ptr` 来分配元组。FromSpace 中剩余的空间是 `fromspace_end` 和 `free_ptr` 之间的差值。当 FromSpace 中没有足够的空间留给下一次分配时，应该调用 `collect` 函数。`collect` 函数接受一个指向根堆栈当前顶部的指针 (位于被推入的最后一项的后面) 和需要分配的字节数。`collect` 函数执行复制收

```
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;
```

图 5.10: 编译器到垃圾收集器的接口。

集，并使堆处于下一次分配将成功的状态。

垃圾收集的引入对我们的编译器通道有重要的影响。我们引入一个新的编译器通道，名为 `expose-allocation` 。对 `select-instructions` 、`build-interference` 、`allocate-registers` 和 `print-x86` 做了显著的改变，并在更多的通道中做微小的改变。下面的程序将作为运行示例。它创建两个元组，一个嵌套在另一个中。两个元组的长度都是 1。程序通过两个向量引用来访问内部元组中的元素。

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

5.3 收缩

回想一下，`shrink` 通道将原语操作符转换为更小的原语集合。因为这个通道是在类型检查之后，但是在需要 `HasType` AST 节点中的类型信息的通道之前，`shrink` 通道必须被修改以将 `HasType` 包裹在它生成的每个 AST 节点上。

5.4 公开分配

`expose-allocation` 通道将 `vector` 创建形式降低为对收集器的条件调用，然后是分配。选择将 `expose-allocation` 通道置于 `remove-complex-opera*` 之前，因为 `expose-allocation` 生成的代码包含复杂操作数。还把 `expose-allocation`

放在 `explicate-control` 之前，因为 `expose-allocation` 使用 `let` 引入新的变量，但 `let` 在 `explicate-control` 之后。

`expose-allocation` 的输出是一种语言 $R_{\texttt{Alloc}}$，它扩展 $R_{\texttt{Vec}}$，并使用在 `vector` 形式的翻译中使用的三种新形式。

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

(`collect` n) 表单运行垃圾收集器,请求 n 个字节。它将成为 `select-instructions` 中的 `runtime.c` 中的 `collect` 函数的调用。(`allocate` $n\,T$) 形式创建一个包含 n 个元素的元组。T 形参是元组的类型：(`Vector` $type_1 \ldots type_n$)，其中 $type_i$ 是元组中第 i 个元素的类型。(`global-value` $name$) 形式读取全局变量的值，比如 `free_ptr` 。

下面，将 `vector` 形式转化为：1) 初始化表达式的 let 绑定序列；2) 对 `collect` 的有条件调用；3) 调用 `allocate` ；4) 向量的初始化。在下面，len 指的是向量的长度，而 $bytes$ 是需要为向量分配的总字节数，即 8(标签) 加上 len 乘以 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) ... )))) ...)
```

在上面，为了可读性，在输出中禁止所有的 `has-type` 形式。初始化表达式 $e_0, \ldots, e_{n-1}$ 在 `allocate` 在 `vector-set!` 序列之前的位置很重要，因为这些表达式可能会触发垃圾收集，并且在收集期间, 堆上不能有一个已分配但未初始化的元组。

图 5.11 显示运行的示例 `expose-allocation` 通道的输出。

```
(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)
```

图 5.11: `expose-allocation` 的输出，减去所有的 `has-type` 形式。

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

图 5.12: $R_{\texttt{Vec}}^{\texttt{ANF}}$ 是管理范式 (ANF) 中的 $R_{\texttt{Vec}}$ 。

5.5 去除复杂的操作数

新的表单 `collect` 、`allocate` 和 `global-value` 都应该被视为复杂操作数。图 5.12 显示该通道的输出语言 $R_{\texttt{Vec}}^{\texttt{ANF}}$ 的语法, 即管理规范形式的 $R_{\texttt{Vec}}$ 。

5.6 解释控制和 $C_{\texttt{Vec}}$ 语言

`explicate-control` 的输出是一个中间语言 $C_{\texttt{Vec}}$ 的程序，其抽象语法定义在图 5.13 中。(具体语法在附录的图 12.4 中定义。) $C_{\texttt{Vec}}$ 的新形式包括 `allocate` 、`vector-ref` 、`vector-set!` 、`global-value` 表达式和 `collect` 语句。`explicate-control` 通道可以像对待我们已经遇到的其他表达式形式一样对待这些新形式。

5.7 选择指令和 $\text{x86}_{\texttt{Global}}$ 语言

在本文中,为编译元组所需的大多数新操作生成 x86 代码,包括 `Allocate` 、`Collect` 、`vector-ref` 、`vector-set!` 和 `void` 。将 `GlobalValue` 编译为 `Global` ， 是因为后者有不同的具体语法 (参见图 5.14 和 5.15)。

`vector-ref` 和 `vector-set!` 形式可转换为 `movq` 指令。(偏移量中的 + 1 是通过元组表示开头的标记。)

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

图 5.13: $C_{\texttt{Vec}}$ 的抽象语法，扩展 $C_{\texttt{If}}$ (图 4.7)。

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*′

lhs′ 、*vec*′ 和 *arg*′ 是通过将 *vec* 和 *arg* 转换到 x86 来获得的。将 *vec*′ 移动到寄存器 `r11` 确保偏移表达式 $-8(n+1)$`(%r11)` 包含一个寄存器操作数。这需要通过寄存器分配将 `r11` 从考虑中删除。

为什么不用 `rax` 代替 `r11`？假设用 `rax` 代替。那么生成的 `vector-set!` 的代码将是

`movq` *vec*′`, %rax`

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

接下来，假设 arg' 最终是一个堆栈位置，所以 `patch-instructions` 会像下面这样通过 `rax` 插入一次移动。

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

但是上面的指令序列不起作用，因为我们试图同时为两个不同的值（vec' 和 arg'）使用 `rax`！

将 `allocate` 表单编译为 `free_ptr` 上的操作，如下所示。`free_ptr` 中的地址是 FromSpace 中的下一个空闲地址，因此将其复制到 `r11` 中，然后将其向前移动足够的空间以分配元组，即 $8(len + 1)$ 个字节，因为每个元素为 8 个字节（64 位），使用 8 个字节作为标记。然后初始化 tag，最后将 `r11` 中的地址复制到左边。参见图 5.9 来查看标签是如何组织的。建议在编译期间使用 Racket 操作 `bitwise-ior` 和 `arithmetic-shift` 来计算标记。`vector` 形式的类型注释用于确定标记的指针掩码区域。

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

`collect` 表单被编译为运行时对 `collect` 函数的调用。`collect` 的参数是：1) 根堆栈的顶部；2) 需要分配的字节数。使用另一个专用寄存器 `r15` 来存储指向根堆栈顶部的指针。因此，`r15` 不能被寄存器分配器使用。

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

arg	::=	$\$int \mid \%reg \mid int(\%reg) \mid \%bytereg \mid var$(%rip)
$\text{x86}_{\texttt{Global}}$	::=	.globl main main: *instr* ...

图 5.14: $\text{x86}_{\texttt{Global}}$ 的具体语法 (扩展图 4.8 中的 $\text{x86}_{\texttt{If}}$)。

arg	::=	(Int *int*) | (Reg *reg*) | (Deref *reg int*) | (ByteReg *reg*)
	|	(Global *var*)
$\text{x86}_{\texttt{Global}}$	::=	(X86Program *info* ((*label* . *block*) ...))

图 5.15: $\text{x86}_{\texttt{Global}}$ 的抽象语法 (扩展图 4.9 中的 $\text{x86}_{\texttt{If}}$)。

图 5.14 和图 5.15 定义 $\text{x86}_{\texttt{Global}}$ 语言的具体和抽象语法。它与 $\text{x86}_{\texttt{If}}$ 的不同之处只是增加全局变量的形式。图 5.16 显示运行示例 `select-instructions` 通道的输出。

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

图 5.16: `select-instructions` 通道的输出。

5.8 寄存器分配

正如本章前面所讨论的，垃圾收集器需要访问根集中的所有指针，也就是说，所有的向量变量。寄存器分配器有责任确保：

1. 根堆栈用于溢出向量类型的变量；

2. 如果 vector 类型变量在调用收集器期间处于活动状态，则必须对其进行溢出，以确保收集器可以看到它。

后面的责任可以在干涉图的构造过程中处理，通过在调用实时矢量类型变量和所有被调用保存的寄存器之间添加干涉边。(它们已经干扰调用者保存的寄存器。) 变量的类型信息是 `Program` 形式的，所以建议向 `build-interference` 函数添加另一个参数来传达这个列表。

当选择如何将颜色 (整数) 分配给寄存器和堆栈位置时，可以在图着色之后处理向根堆栈溢出的向量类型变量。这个通道的 `Program` 输出也会改变，以记录到根堆栈的溢出次数。

5.9 打印 x86

图 5.17 显示运行示例上 `print-x86` 通道的输出。在 `main` 函数的前奏和结尾部分，将根栈和普通栈处理得非常相似，移动根栈指针 (`r15`) ，为根栈的溢出腾出空间，但根栈是向上而不是向下增长的。对于正在运行的示例，只有一个溢出，因此将 `r15` 增加 8 个字节。在结论部分，`r15` 减了 8 个字节。

需要特别注意的一个问题是，在对根堆栈中的所有变量进行初始化赋值之前，可能会调用 `collect` 。不希望垃圾收集器意外地认为某个未初始化的变量是一个需要跟随的指针。因此，在 `main` 的前奏中，将根堆栈上的所有位置归零。在图 5.17 中，指令 `movq $0, (%r15)` 完成了这个任务。垃圾收集器会在对每个根进行解引用之前测试它是否为空。

图 5.18 给出编译 $R_{\texttt{Vec}}$ 所需的所有通道的概览。

```
block35:
       movq   free_ptr(%rip), %rcx
       addq   $16, free_ptr(%rip)
       movq   %rcx, %r11
       movq   $131, 0(%r11)
       movq   %rcx, %r11
       movq   -8(%r15), %rax
       movq   %rax, 8(%r11)
       movq   $0, %rdx
       movq   %rcx, %r11
       movq   8(%r11), %rcx
       movq   %rcx, %r11
       movq   8(%r11), %rax
       jmp conclusion
block36:
       movq   $0, %rcx
       jmp block35
block38:
       movq   free_ptr(%rip), %rcx
       addq   $16, free_ptr(%rip)
       movq   %rcx, %r11
       movq   $3, 0(%r11)
       movq   %rcx, %r11
       movq   %rbx, 8(%r11)
       movq   $0, %rdx
       movq   %rcx, -8(%r15)
       movq   free_ptr(%rip), %rcx
       addq   $16, %rcx
       movq   fromspace_end(%rip), %rdx
       cmpq   %rdx, %rcx
       jl block36
       movq   %r15, %rdi
       movq   $16, %rsi
       callq  collect
       jmp block35
block39:
       movq   $0, %rcx
       jmp block38
start:
       movq   $42, %rbx
       movq   free_ptr(%rip), %rdx
       addq   $16, %rdx
       movq   fromspace_end(%rip), %rcx
       cmpq   %rcx, %rdx
       jl block39
       movq   %r15, %rdi
       movq   $16, %rsi
       callq  collect
       jmp block38

       .globl main
main:
       pushq  %rbp
       movq   %rsp, %rbp
       pushq  %r13
       pushq  %r12
       pushq  %rbx
       pushq  %r14
       subq   $0, %rsp
       movq $16384, %rdi
       movq $16384, %rsi
       callq initialize
       movq rootstack_begin(%rip), %r15
       movq $0, (%r15)
       addq $8, %r15
       jmp start
conclusion:
       subq $8, %r15
       addq   $0, %rsp
       popq   %r14
       popq   %rbx
       popq   %r12
       popq   %r13
       popq   %rbp
       retq
```

图 5.17: `print-x86` 通道的输出。

5.10 挑战：简单的结构

图 5.19 定义 $R_{\texttt{Vec}}^{\texttt{Struct}}$ 的具体语法，它扩展 $R_{\texttt{Vec}}$ ，支持简单的结构。回想一下，类型化 Racket 中的 `struct` 是用户定义的数据类型，它包含命名字段，是堆分配的，类似于向量。下面是一个结构定义的示例，在本例中是 `point` 类型的定义。

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

结构的实例是使用函数调用语法创建的，结构的名称位于函数位置

```
(point 7 12)
```

函数调用语法也用于读取结构字段中的值。函数名由结构名、破折号和字段名组成。下面的示例使用 `point-x` 和 `point-y` 访问两个点实例的 `x` 和 `y` 字段。

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

类似地，要写入结构的字段，请使用它的 set 函数，该函数的名称以 `set-` 开头，接着是结构名，然后是破折号，然后是字段名，最后是感叹号。下面的示例使用 `set-point-x!` 将 `x` 字段从 `7` 更改为 `42` 。

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

Exercise 28. 扩展编译器，支持简单的结构，将 $R_{\texttt{Vec}}^{\texttt{Struct}}$ 编译为 x86 汇编代码。创建五个使用结构并测试编译器的新测试用例。

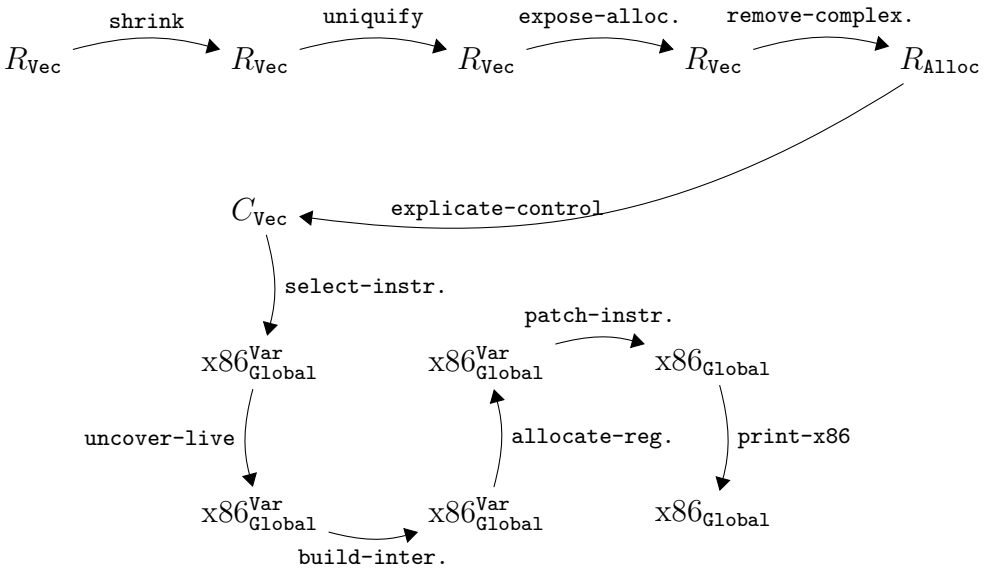


图 5.18: 带有元组的语言 $R_{\texttt{Vec}}$ 的通道图

type	::=	`Integer` | `Boolean` | (`Vector` *type* ...) | `Void` | *var*
cmp	::=	`eq?` | `<` | `<=` | `>` | `>=`
exp	::=	*int* | (`read`) | (`-` *exp*) | (`+` *exp exp*) | (`-` *exp exp*)
	|	*var* | (`let` ([*var exp*]) *exp*)
	|	`#t` | `#f` | (`and` *exp exp*) | (`or` *exp exp*) | (`not` *exp*)
	|	(*cmp exp exp*) | (`if` *exp exp exp*)
	|	(`vector` *exp* ...) | (`vector-ref` *exp int*)
	|	(`vector-set!` *exp int exp*)
	|	(`void`) | (*var exp* ...)
def	::=	(`struct` *var* ([*var* : *type*] ...) `#:mutable`)
$R^{\texttt{Struct}}_{\texttt{Vec}}$	::=	*def* ... *exp*

图 5.19: $R^{\texttt{Struct}}_{\texttt{Vec}}$ 的具体语法，扩展 $R_{\texttt{Vec}}$ (图 5.1)。

5.11 挑战：分代收集

第 5.2 节中描述的复制收集器会产生大量的运行时开销，因为 `collect` 调用所花费的时间与所有实时数据成比例。减少这种开销的一种方法是减少每次 `collect` 调用中被检查的数据量。特别是，研究人员观察到，最近分配的数据比之前经过一次或多次 `collect` 的数据更有可能成为垃圾。这种洞察力激发 *分代垃圾收集器* 的诞生：1) 根据年龄将数据隔离为两个或更多代；2) 为年轻的代分配更少的空间，因此收集它们的速度更快，为年老的代分配更多的空间；3) 对年轻代的收集比对老代的更频繁 [113] 。

对于这个挑战分配，目标是使 `runtime.c` 中实现的复制收集器使用两代，一代用于年轻数据，一代用于旧数据。每一代都由 FromSpace 和 ToSpace 组成。下面是如何调整 `collect` 功能来使用这两代人的草图。

1. 复制年轻一代的 FromSpace 到它的 ToSpace，然后切换 ToSpace 和 FromSpace 的角色
2. 如果年轻的 FromSpace 中有足够的空间容纳请求的字节数，那么从 `collect` 返回。
3. 如果年轻的 FromSpace 中没有足够的空间来存放请求的字节，则按照以下步骤将数据从年轻代移动到老代：
   (a) 如果旧的 FromSpace 中有足够的空间，将年轻的 FromSpace 复制到旧的 FromSpace 中，然后返回。
   (b) 如果旧的 FromSpace 中没有足够的空间，那么通过将旧的 FromSpace 复制到旧的 ToSpace 来收集旧的代，并交换旧的 FromSpace 和 ToSpace 的角色。
   (c) 如果现在有足够的空间，将年轻的 FromSpace 复制到旧的 FromSpace 并返回。否则，为旧代分配更大的 FromSpace 和 ToSpace。将年轻的 FromSpace 和旧的 FromSpace 复制到年老代的较大的 FromSpace 中，然后返回。

建议你一般化 `cheney` 函数，这样它就可以用于上面提到的所有副本：年轻的 FromSpace 和老的 FromSpace 之间，年轻的 FromSpace 和老的

FromSpace 之间。这可以通过向 `cheney` 添加参数来完成，以替换全局变量 `fromspace_begin` 、`fromspace_end` 、`tospace_begin` 和 `tospace_end` 。

请注意，年轻代的集合不会遍历老代。这就带来一个潜在的问题：可能有一些年轻的数据只能通过老一代的指针访问。如果不考虑这些指针，收集器可能会丢弃实时的年轻数据！一种称为 *指针记录*的解决方案是维护从老代到新代的所有指针的集合，并将该集合视为根集的一部分。为了维护这个向量集，编译器必须在每个 `vector-set!` 周围插入额外的指令。如果被修改的向量属于旧的一代，并且被写入的值是指向新一代的指针，则必须将该指针添加到集合中。此外，如果被覆盖的值是指向新一代的指针，则应该从集合中删除该指针。

Exercise 29. 调整 `runtime.c` 中的 `collect` 函数来实现分代垃圾收集，如本节所述。更新 `vector-set!` 的代码生成，以实现指针记录。确保新编译器和运行时通过测试套件。

6

函数

本章研究类似于 C 语言的函数的编译。这对应于一个只允许顶级函数定义的类型化 Racket 子集。这类函数是实现词汇作用域函数的重要基石，即 `lambda` 抽象，这是第 7 章的主题。

6.1 $R_{\texttt{Fun}}$ 语言

函数定义和函数应用程序的具体和抽象语法如图 6.1 和 6.2 所示，其中我们定义 $R_{\texttt{Fun}}$ 语言。$R_{\texttt{Fun}}$ 中的程序以零个或多个函数定义开始。这些定义中的函数名在整个程序的作用域内，包括所有其他函数定义 (因此函数定义的顺序无关紧要)。函数应用 的具体语法是 (*exp exp*...)，其中第一个表达式必须计算为一个函数，其余的是参数。函数应用程序的抽象语法是 `(Apply` *exp exp*...`)`。

函数是一等函数，因为函数指针 是数据，可以存储在内存中，也可以作为参数传递给另一个函数。因此，介绍一个函数类型

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

对于 n 个形参类型为 $type_1$ 到 $type_n$ 且返回类型为 $type_r$ 的函数。这些函数 (相对于 Racket 函数而言) 的主要限制是它们没有词法范围。也就是说，唯一可以从函数体内部引用的外部实体是其他全局定义的函数。$R_{\texttt{Fun}}$ 的语法阻止函数之间的嵌套。

type	::=	`Integer` | `Boolean` | (`Vector` *type* ...) | `Void` | (*type* ... `->` *type*)
cmp	::=	`eq?` | `<` | `<=` | `>` | `>=`
exp	::=	*int* | (`read`) | (`-` *exp*) | (`+` *exp exp*) | (`-` *exp exp*)
	|	*var* | (`let` (`[`*var exp*`]`) *exp*)
	|	`#t` | `#f` | (`and` *exp exp*) | (`or` *exp exp*) | (`not` *exp*)
	|	(*cmp exp exp*) | (`if` *exp exp exp*)
	|	(`vector` *exp* ...) | (`vector-ref` *exp int*)
	|	(`vector-set!` *exp int exp*) | (`void`) | (`has-type` *exp type*)
	|	(*exp exp* ...)
def	::=	(`define` (*var* `[`*var*`:`*type*`]` ...) `:` *type exp*)
$R_{\texttt{Fun}}$	::=	*def* ... *exp*

图 6.1: $R_{\texttt{Fun}}$ 扩展 $R_{\texttt{Vec}}$ 的具体语法 (图 5.1) 。

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* ...)
def	::=	(`Def` *var* (`[`*var*`:`*type*`]` ...) *type* `'()` *exp*)
$R_{\texttt{Fun}}$	::=	(`ProgramDefsExp` `'()` (*def* ...)) *exp*)

图 6.2: $R_{\texttt{Fun}}$ 的抽象语法扩展 $R_{\texttt{Vec}}$ (图 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)
```

图 6.3: 在 $R_{\texttt{Fun}}$ 中使用函数的例子。

图 6.3 中的程序是在 $R_{\texttt{Fun}}$ 中定义和使用函数的一个典型例子。定义一个函数 `map-vec` ，它将其他函数 `f` 应用于向量的两个元素，并返回一个包含结果的新向量。还定义一个函数 `add1`。程序应用 `map-vec` 到 `add1` 和 `(vector 0 41)` 。结果是 `(vector 1 42)` ，返回 `42` 。

$R_{\texttt{Fun}}$ 的定义解释器如图 6.4 所示。`ProgramDefsExp` 表单的案例负责设置顶级函数定义之间的相互递归。使用经典的回补 方法 [70] ，即使用可变变量，并对函数定义进行两个通道。在第一个通道中，为每个函数定义使用可变 cons 单元格来设置顶层环境。注意，每个函数的 `lambda` 值是不完整的；它还不包括环境因素。一旦构建顶层环境，我们就对其进行迭代，并更新 `lambda` 值以使用顶层环境。

$R_{\texttt{Fun}}$ 的类型检查器如图 6.5 所示。

6.2 x86 中的函数

已经看到，x86 提供标签，以便可以引用指令的位置，这是跳转指令所需要的。标签也可以用来标记函数指令的开头。更进一步，可以通过使用 `leaq` 指令和 PC 相对寻址来获得一个标签的地址。例如，下面将 `add1` 标签的地址放入 `rbx` 寄存器中。

```
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))
```

图 6.4: $R_{\texttt{Fun}}$ 语言的解释器。

指令指针寄存器 rip（也称为程序计数器）始终指向要执行的下一条指令。当与一个标签结合时，如 add1(%rip)，链接器计算 add1 的地址与 rip 此刻的位置之间的距离 d，然后将 add1(%rip) 更改为 d(%rip)，运行时将计算 add1 的地址。

在第 2.2 节中，使用 callq 指令跳转到一个函数，该函数的位置由标签给出。在本章中，为了支持函数调用，将跳转到一个由寄存器中的地址给出的函数，也就是说，需要进行一个 *间接函数调用*。x86 的语法是 callq 指令，但是在寄存器名之前有一个星号。

```
callq *%rbx
```

6.2.1 调用约定

callq 指令为实现函数提供了部分支持：它将返回地址压入堆栈，然后跳转到目标函数。但是，callq 不处理

1. 参数传递；
2. 将帧压入过程调用堆栈并取出它们；

```
(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))
```

图 6.5: R_{Fun} 语言的类型检查器。

3. 确定不同函数如何共享寄存器。

关于 (1) 参数传递，请记住以下 6 个寄存器用于按此顺序将参数传递给函数。

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

如果有 6 个以上的参数，那么约定是在调用者的框架上为其余的参数预留空间。但是，为了简化有效的尾部调用的实现 (第 6.2.2节)，将参数设置为不超过 6 个。还记得寄存器 rax 是用于函数的返回值的。

关于 (2) 帧和过程调用堆栈，回顾第 2.2 节，堆栈向下增长，每个函数调用使用一块称为帧的空间。调用者将堆栈指针寄存器 rsp 设置为帧中的最后一个数据项。被调用方不能改变调用方框架内的任何内容，即在堆栈指针上或上面的任何内容。被调用方可以自由使用堆栈指针下面的位置。

回想一下，我们是在根堆栈上存储向量类型的变量。所以前奏需要将根堆栈指针 r15 向上移动，而结论需要将根堆栈指针向下移动。同样，前奏必须将这帧在根栈中的插槽初始化为 0 ，以向垃圾收集器发出这些插槽还不包含指向向量的指针的信号。否则，垃圾收集器将把这些槽中的垃圾位解释为内存地址，并试图遍历它们，从而造成严重的破坏!

关于 (3) 不同函数之间的寄存器共享，回顾一下第 3.1 节，寄存器被分为两组，调用者保存的寄存器和被调用者保存的寄存器。调用方应该假设所有调用方保存的寄存器被调用方用任意值覆盖。这就是为什么在第 3.1 节中建议，在函数调用期间活动的变量，不应该被分配给调用方保存的寄存器。

另一方面，如果被调用方想要使用被调用方保存的寄存器，则被调用方必须将这些寄存器的内容保存在它们的堆栈帧上，然后在返回给调用方之前将它们放回去。这就是为什么在第 3.1节中建议，如果寄存器分配器将一个变量赋给被调用者保存的寄存器，那么 main 函数的前奏必须将该寄存器保存到堆栈中，而 main 函数的结论必须恢复它。这个建议现在适用于所有函数。

还记得基指针 rbp 寄存器被用作帧内的一个引用点，这样每个局部变量都可以在基指针的固定偏移量处被访问 (第 2.2节)。图 6.6 显示调用者和被调用者框架的总体布局。

调用者视角	调用者视角	内容	框架
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	

图 6.6: 调用者和被调用者帧的内存布局。

6.2.2 高效的尾部调用

通常，程序所使用的堆栈空间的大小是由最长的嵌套函数调用链决定的。也就是说，如果函数 f_1 调用 f_2 ，f_2 调用 $f_3, \ldots$ ，f_{n-1} 调用 f_n ，那么堆栈空间的大小是 $O(n)$ 。在递归或相互递归函数的情况下，深度 n 可以增长相当大。然而，在某些情况下，可以安排只使用常数空间，即 $O(1)$ ，而不是 $O(n)$ 。

如果函数调用是函数体中的最后一个操作，则该调用称为 *尾部调用*。例如，在下面的程序中，对 `tail-sum` 的递归调用就是尾部调用。

```
(define (tail-sum [n : Integer] [r : Integer]) : Integer
  (if (eq? n 0)
      r
      (tail-sum (- n 1) (+ n r))))

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

在尾部调用时，不再需要调用方的帧，因此可以在进行尾部调用之前弹出调用方的帧。使用这种方法，只进行尾部调用的递归函数将只使用 $O(1)$ 堆栈空间。像 Racket 这样的函数式语言通常严重依赖递归函数，所以通常保证所有尾部调用都以这种方式优化。

但是，在尾部调用中传递参数时需要注意。如上所述，对于第 6 个以上的参数，约定是在调用者的框架中使用空间传递参数。但是对于一个尾部调用，弹出调用者的帧，不能再使用它。另一种选择是在被调用方的框架中使用空间传递参数然而，这个选项也有问题，因为调用者和被调用者的帧在内存中重叠。当我们开始从它们在调用者的帧中的源中复制参数时，被调用者帧中的目标位置可能会与后面的参数的源重叠！解决这个问题的方法是不使用堆栈传递超过 6 个参数，而是使用堆，如第 6.5 节所述。

如上所述，对于尾部调用，在进行尾部调用之前弹出调用方的帧。弹出帧的指令是我们通常放在函数结尾的指令。因此，还需要在每个尾部调用之前立即放置这样的代码。这些指令包括恢复被调用者保存的寄存器，因此传递参数的寄存器都是调用者保存的寄存器是很好的。

关于使用哪个指令进行尾部调用的最后一个注意事项。当被调用方完成时，它不应该返回当前函数，但应该返回调用当前函数的函数。因此，堆栈上已经存在的返回地址就是正确的，不应该使用 `callq` 来进行尾部调用，因为那样会不必要地覆盖返回地址。相反，可以简单地使用 `jmp` 指令。与间接函数调用类似，使用带有星号前缀的寄存器编写 *间接跳转* 。建议使用 `rax` 来保存跳转目标，因为前面的结论几乎覆盖所有其他内容。

```
jmp *%rax
```

6.3 缩小 R_{Fun}

`shrink` 通道执行一个小的修改，以减轻后面的通道。这个函数引入一个显式的 `main` 函数，并将顶部的 `ProgramDefsExp` 形式更改为 `ProgramDefs` ，如下所示。

(`ProgramDefsExp` *info* (*def* ...) *exp*)
⇒ (`ProgramDefs` *info* (*def* ... *mainDef*))

其中，*mainDef* 是

(`Def 'main '() 'Integer '()` *exp′*)

6.4 揭示函数和 R_{FunRef} 语言

R_{Fun} 的语法在编译时很不方便: 它合并函数名和局部变量的使用。这是一个问题，因为需要将函数名的使用与局部变量的使用进行不同的编译；需要使用 `leaq` 将函数名 (x86 中的标签) 转换为寄存器中的地址。因此，创建一个新的通道将函数引用从符号 f 更改为 (`FunRef` f) 是一个好主意。这个通道被命名为 `reveal-functions` ，输出语言 R_{FunRef} 在图 6.7 中定义。函数引用的具体语法是 (`fun-ref` f) 。

将这个通道放在 `uniquify` 之后将确保没有共享相同名称的局部变量和函数。另一方面，`reveal-functions` 需要在 `explicate-control` 通道之前，因为该通道帮助我们将 `FunRef` 表单编译成赋值语句。

exp	::=	... | (FunRef *var*)
def	::=	(Def *var* ([*var*:*type*] ...) *type* '() *exp*)
$R_{\texttt{FunRef}}$	::=	(ProgramDefs '() (*def* ...))

图 6.7: 抽象语法 $R_{\texttt{FunRef}}$ ， $R_{\texttt{Fun}}$ 的扩展 (图 6.2)。

6.5 极限函数

回想一下，我们希望将函数参数的数量限制为 6 个，这样就不需要使用堆栈来传递参数，这使得实现有效的尾部调用更容易。但是，由于输入语言 $R_{\texttt{Fun}}$ 支持任意数量的函数参数，我们需要做一些工作!

此过程将涉及六个以上参数的函数和函数调用转换为照常传递前五个参数，但它将其余参数打包到向量中，并将其作为第六个参数传递。

对于包含太多参数的每个函数定义进行如下转换。

(Def f ([x_1:T_1] ... [x_n:T_n]) T_r *info* *body*)
⇒
(Def f ([x_1:T_1] ... [x_5:T_5] [vec : (Vector $T_6 \ldots T_n$)]) T_r *info* $body'$)

通过将后面出现的参数替换为向量引用，将 *body* 转换为 $body'$ 。

(Var x_i) ⇒ (Prim 'vector-ref (list vec (Int $(i-6)$)))

对于带有太多参数的函数调用，`limit-functions` 通道按以下方式转换它们。

(e_0 e_1 ... e_n) ⇒ (e_0 $e_1 \ldots e_5$ (vector $e_6 \ldots e_n$))

6.6 去除复杂的操作数

要做的主要决定是将 FunRef 和 Apply 分类为原子表达式还是复杂表达式。回想一下，一个简单的表达式最终只是 x86 指令的一个直接参数。函数应用将被翻译成一个指令序列，因此 Apply 必须归类为复杂表达式。另一方面，Apply 的参数应该是原子表达式。如上所述，对于 FunRef ，需要使用 leaq 指令将函数标签转换为地址。因此，尽管 FunRef 看起来相当简

$$
\begin{array}{rcl}
\mathit{atm} & ::= & \texttt{(Int } \mathit{int}\texttt{)} \mid \texttt{(Var } \mathit{var}\texttt{)} \mid \texttt{(Bool } \mathit{bool}\texttt{)} \mid \texttt{(Void)} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{(Prim read ())} \\
& \mid & \texttt{(Prim -} \ (\mathit{atm})\texttt{)} \mid \texttt{(Prim +} \ (\mathit{atm}\ \mathit{atm})\texttt{)} \\
& \mid & \texttt{(Let } \mathit{var}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
& \mid & \texttt{(Prim 'not} \ (\mathit{atm})\texttt{)} \\
& \mid & \texttt{(Prim } \mathit{cmp}\ (\mathit{atm}\ \mathit{atm})\texttt{)} \mid \texttt{(If } \mathit{exp}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
& \mid & \texttt{(Collect } \mathit{int}\texttt{)} \mid \texttt{(Allocate } \mathit{int}\ \mathit{type}\texttt{)} \mid \texttt{(GlobalValue } \mathit{var}\texttt{)} \\
& \mid & \texttt{(FunRef } \mathit{var}\texttt{)} \mid \texttt{(Apply } \mathit{atm}\ \mathit{atm} \ldots\texttt{)} \\
\mathit{def} & ::= & \texttt{(Def } \mathit{var}\ ([\mathit{var}\texttt{:}\mathit{type}] \ldots)\ \mathit{type}\ \texttt{'()}\ \mathit{exp}\texttt{)} \\
R_4^{\dagger} & ::= & \texttt{(ProgramDefs '()} \ \mathit{def}\texttt{)}
\end{array}
$$

图 6.8: $R_{\texttt{Fun}}^{\texttt{ANF}}$ 是行政范式 (ANF) 中的 $R_{\texttt{Fun}}$ 。

单，但需要将它归类为一个复杂表达式，以便生成一个左侧可作为 `leaq` 目标的赋值语句。图 6.8 定义这个通道的输出语言 $R_{\texttt{Fun}}^{\texttt{ANF}}$ 。

6.7 解释控制和 $C_{\texttt{Fun}}$ 语言

图 6.9 定义 $C_{\texttt{Fun}}$ 的抽象语法，即 `explicate-control` 的输出。(附录的图 12.5 给出具体的语法。) 赋值和尾部上下文的辅助功能应使用 `Apply` 和 `FunRef` 的用例进行更新，谓词上下文的功能应针对 `Apply` 而不是 `FunRef` 进行更新。(`FunRef` 不能是布尔值。) 在赋值和谓词上下文中，`Apply` 变成 `Call` ，而在尾部位置 `Apply` 变成 `TailCall` 。建议定义一个新的辅助函数来处理函数定义。这段代码类似于 $R_{\texttt{Vec}}$ 中的 `Program` 。然后，处理 $R_{\texttt{Fun}}$ `ProgramDefs` 形式的顶级 `explicate-control` 功能，可以将此新功能应用于所有功能定义。

6.8 选择指令和 $\text{x86}_{\texttt{callq*}}$ 语言

选择指令的输出是一个使用 $\text{x86}_{\texttt{callq*}}$ 语言的程序，其语法在图 6.11 中定义。

$$
\begin{array}{lcl}
\mathit{atm} & ::= & \texttt{(Int } \mathit{int}\texttt{)} \mid \texttt{(Var } \mathit{var}\texttt{)} \mid \texttt{(Bool } \mathit{bool}\texttt{)} \\
\mathit{cmp} & ::= & \texttt{eq?} \mid \texttt{<} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{(Prim read ())} \\
 & \mid & \texttt{(Prim -} \ (\mathit{atm})\texttt{)} \mid \texttt{(Prim +} \ (\mathit{atm}\ \mathit{atm})\texttt{)} \\
 & \mid & \texttt{(Prim not} \ (\mathit{atm})\texttt{)} \mid \texttt{(Prim} \ \mathit{cmp}\ (\mathit{atm}\ \mathit{atm})\texttt{)} \\
 & \mid & \texttt{(Allocate} \ \mathit{int}\ \mathit{type}\texttt{)} \\
 & \mid & \texttt{(Prim 'vector-ref} \ (\mathit{atm}\ \texttt{(Int} \ \mathit{int}\texttt{)))} \\
 & \mid & \texttt{(Prim 'vector-set!} \ \texttt{(list} \ \mathit{atm}\ \texttt{(Int} \ \mathit{int}\texttt{)}\ \mathit{atm}\texttt{))} \\
 & \mid & \texttt{(GlobalValue} \ \mathit{var}\texttt{)} \mid \texttt{(Void)} \\
 & \mid & \texttt{(FunRef} \ \mathit{label}\texttt{)} \mid \texttt{(Call} \ \mathit{atm}\ \texttt{(}\mathit{atm}\ldots\texttt{))} \\
\mathit{stmt} & ::= & \texttt{(Assign (Var} \ \mathit{var}\texttt{)}\ \mathit{exp}\texttt{)} \mid \texttt{(Collect} \ \mathit{int}\texttt{)} \\
\mathit{tail} & ::= & \texttt{(Return} \ \mathit{exp}\texttt{)} \mid \texttt{(Seq} \ \mathit{stmt}\ \mathit{tail}\texttt{)} \mid \texttt{(Goto} \ \mathit{label}\texttt{)} \\
 & \mid & \texttt{(IfStmt (Prim} \ \mathit{cmp}\ (\mathit{atm}\ \mathit{atm})\texttt{) (Goto} \ \mathit{label}\texttt{) (Goto} \ \mathit{label}\texttt{))} \\
 & \mid & \texttt{(TailCall} \ \mathit{atm}\ \mathit{atm}\ldots\texttt{)} \\
\mathit{def} & ::= & \texttt{(Def} \ \mathit{label}\ \texttt{(}[\mathit{var}\texttt{:}\mathit{type}]\ldots\texttt{)}\ \mathit{type}\ \mathit{info}\ \texttt{((}\mathit{label}\ \texttt{.}\ \mathit{tail}\texttt{)}\ldots\texttt{))} \\
C_{\texttt{Fun}} & ::= & \texttt{(ProgramDefs} \ \mathit{info}\ \texttt{(}\mathit{def}\ldots\texttt{))}
\end{array}
$$

图 6.9: $C_{\texttt{Fun}}$ 的抽象语法扩展 $C_{\texttt{Vec}}$ (图 5.13)。

$$
\begin{array}{lcl}
\mathit{arg} & ::= & \texttt{\$}\mathit{int} \mid \texttt{\%}\mathit{reg} \mid \mathit{int}\texttt{(\%}\mathit{reg}\texttt{)} \mid \texttt{\%}\mathit{bytereg} \mid \mathit{var}\texttt{(\%rip)} \mid \texttt{(fun-ref} \ \mathit{label}\texttt{)} \\
\mathit{cc} & ::= & \texttt{e} \mid \texttt{l} \mid \texttt{le} \mid \texttt{g} \mid \texttt{ge} \\
\mathit{instr} & ::= & \ldots \mid \texttt{callq *}\mathit{arg} \mid \texttt{tailjmp} \ \mathit{arg} \mid \texttt{leaq} \ \mathit{arg}\texttt{,} \ \texttt{\%}\mathit{reg} \\
\mathit{block} & ::= & \mathit{instr}\ldots \\
\mathit{def} & ::= & \texttt{(define (}\mathit{label}\texttt{) ((}\mathit{label}\ \texttt{.}\ \mathit{block}\texttt{)}\ldots\texttt{))} \\
\text{x86}_{\texttt{callq*}} & ::= & \mathit{def}\ldots
\end{array}
$$

图 6.10: $\text{x86}_{\texttt{callq*}}$ 的具体语法 (扩展图 5.14 中的 $\text{x86}_{\texttt{Global}}$)。

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* ...))

图 6.11: x86$_{\texttt{callq*}}$ 的抽象语法 (扩展图 5.15 中的 x86$_{\texttt{Global}}$)。

将函数引用赋值给变量成为 load-effective-address 指令，如下所示：

lhs = (fun-ref *f*); ⇒ leaq (fun-ref *f*), *lhs*′

关于函数定义，需要删除参数，并使用第 6.2节中讨论的约定来执行参数传递。也就是说，参数是在寄存器中传递的。建议将形参转换为局部变量，并在函数开始时生成指令，将传递寄存器的实参转移到这些局部变量。

(Def f '([x_1 : T_1] [x_2 : T_2] ...) T_r *info* G)
⇒
(Def f '() 'Integer *info*′ G')

G' 控制流图与 G 一样；除了 start 块被修改为添加从参数寄存器移动到参数变量的指令。所以左边显示的 G 的 start 块被更改为右边的代码。

```
start:
  instr_1
  ⋮
  instr_n
```

⇒

```
start:
  movq %rdi, x_1
  movq %rsi, x_2
  ⋮
  instr_1
  ⋮
  instr_n
```

通过将参数更改为局部变量，让寄存器分配器控制使用哪个寄存器或堆栈位置。如果你实现了移动偏置挑战 (第 3.7节)，寄存器分配器尝试将参数变量

分配给相应的实参寄存器，在这种情况下，patch-instructions 通道将删除 movq 指令。这在第 6.12 节的图 6.13 中的 add 函数的示例转换中发生。另外，请注意，寄存器分配器将对这个移动指令序列进行动态分析，并构建干涉图。例如，x_1 将被标记为干扰 rsi ，这将阻止将 x_1 赋值给 rsi ，这很好，因为这会覆盖需要移到 x_2 的参数。

接下来，考虑函数调用的编译。在处理函数定义的形参的镜像中，需要将实参移动到实参传递寄存器中。函数调用本身是通过间接函数调用来执行的。函数的返回值存储在 rax 中，因此需要将其移动到 *lhs* 中。

lhs = (call *fun* arg_1 $arg_2 \ldots$));
⇒
movq arg_1, %rdi
movq arg_2, %rsi
⋮
callq **fun*
movq %rax, *lhs*

IndirectCallq AST 节点包含一个整数来表示函数的数量，即参数的数量。该信息在 uncover-live 通道中非常有用，可以用来确定调用期间可能读取哪些参数传递寄存器。

对于尾部调用，传递的参数与非尾部调用相同：生成指令将参数移动到参数传递寄存器中。之后，需要从过程调用堆栈中弹出帧。然而，还不知道这个框架有多大；这是在寄存器分配期间决定的。因此，没有在这里生成这些指令，而是发明一个新指令，意思是 “弹出帧，然后进行间接跳转”，将其命名为 TailJmp 。该指令的抽象语法包括一个参数 (指定跳转到哪里) 和一个整数 (表示被调用函数的数量)。

回想一下，在第 2.6 节中，建议在程序的初始块中使用标签 start ，在第 2.7 节中，建议用 conclusion 标记程序的结尾，这样 (Return *arg*) 就可以被编译为对 rax 的赋值，然后跳转到 conclusion 。通过添加函数定义，每个函数都有一个开始块和结论，但是它们的标签必须是唯一的。建议将函数名分别放在 start 和 conclusion 前面，以获得唯一的标签。(或者，可以使用 gensym 标签作为开始和结束，并将它们存储在函数定义的 *info* 字段中。)

6.9 寄存器分配

6.9.1 活性分析

`IndirectCallq` 指令应该像 `Callq` 一样对待其写入位置 W，因为它们应该包括所有调用者保存的寄存器。回想一下，这样做的原因是强制将调用活动变量分配给被调用保存的寄存器，或者将其溢出到堆栈中。

对于读位置集合 R，`TailJmp` 和 `IndirectCallq` 的属性字段决定这些指令应该读取多少传递参数的寄存器。

6.9.2 建立干涉图

通过添加函数定义，计算每个函数的干涉图 (而不是整个程序的干涉图)。

回想一下，在第 5.8 节中，讨论在 `collect` 调用期间溢出 vector 类型变量的必要性。在语言中添加函数之后，需要重新讨论这个问题。许多函数执行分配，因此在它们内部有对收集器的调用。因此，不仅应该在 `collect` 调用期间泄漏 vector 类型的变量，而且应该在任何函数调用期间泄漏该变量。因此，在 `build-interference` 通道中，建议在调用实时向量类型的变量和被调用者保存的寄存器之间添加干扰边 (除了通常在调用实时变量和被调用者保存的寄存器之间添加边)。

6.9.3 配置寄存器

`allocate-registers` 的主要变化是添加一个辅助函数来处理函数定义 (图 6.11 中的 *def* 非终端) 。其逻辑与第 3 章中描述的相同，除了现在寄存器分配被多次执行，为每个函数定义一次，而不是为整个程序一次。

6.10 补丁说明

在 `patch-instructions` 中，你应该处理 x86 的特性，即 `leaq` 的目标参数必须是一个寄存器。此外，应该确保 `TailJmp` 的参数是我们的保留寄

存器 *rax*—这是为了使代码生成更方便，因为在尾部调用之前破坏了许多寄存器 (如下一节所述)。

6.11 打印 x86

对于 `print-x86` 通道，`FunRef` 和 `IndirectCallq` 的情况很简单：输出它们的具体语法。

(`FunRef` *label*) ⇒ *label*(`%rip`)
(`IndirectCallq` *arg int*) ⇒ `callq *`*arg*`'`

`TailJmp` 节点需要一些工作。`TailJmp` 的一个简单的翻译是 `jmp *`*arg* ，但在跳转之前，需要弹出当前帧。这个指令序列与函数的结尾代码相同，除了 `retq` 被替换为 `jmp *`*arg* 。

关于函数定义，需要为每个函数生成一个前奏和结论。这段代码类似于在第 5 章中生成的 `main` 函数的前奏和结论。回顾一下，每个函数的前奏应执行以下步骤。

1. 以 `.global` 和 `.align` 指令开头，后面跟着函数的标签。(示例见图 6.13 。)
2. 将 `rbp` 推入堆栈，并将 `rbp` 设置为当前堆栈指针。
3. 将所有被调用者保存的用于寄存器分配的寄存器推入堆栈。
4. 将堆栈指针 `rsp` 向下移动到此函数的堆栈帧的大小，这取决于常规溢出的数量。(对齐到 16 字节)
5. 将根堆栈指针 `r15` 向上移动到这个函数的根堆栈帧的大小，这取决于溢出向量的数量。
6. 将根堆栈框架中的所有条目初始化为零。
7. 跳到开始块上。

`main` 函数的前段有一个附加任务：调用 `initialize` 函数来设置垃圾收集器，并在 `r15` 中移动全局 `rootstack_begin` 的值。这应该发生在上面的第5步之前，这取决于 `r15` 。

每个函数的结论应该做以下工作。

1. 将堆栈指针按此函数的堆栈帧的大小向上移动。
2. 通过将被调用保存的寄存器从堆栈中取出来恢复它们。
3. 将根堆栈指针向下移动到此函数的根堆栈框架的大小。
4. 通过将 `rbp` 从堆栈中取出来恢复它。
5. 使用 `retq` 指令返回到调用者。

Exercise 30. 如本章所述，展开你的编译器来处理 $R_{\texttt{Fun}}$ 。创建 5 个使用函数的新程序，包括从其他函数传递函数和返回函数的例子，递归函数，创建向量的函数，以及进行尾部调用的函数。在这些新程序和之前创建的所有测试程序上测试编译器。

图 6.12 概述将 $R_{\texttt{Fun}}$ 编译到 x86 的过程。

6.12 一个例子的翻译

图 6.13 显示一个简单函数在 $R_{\texttt{Fun}}$ 到 x86 转换的示例。图中还包括 `explicate-control` 和 `select-instructions` 通道的结果。

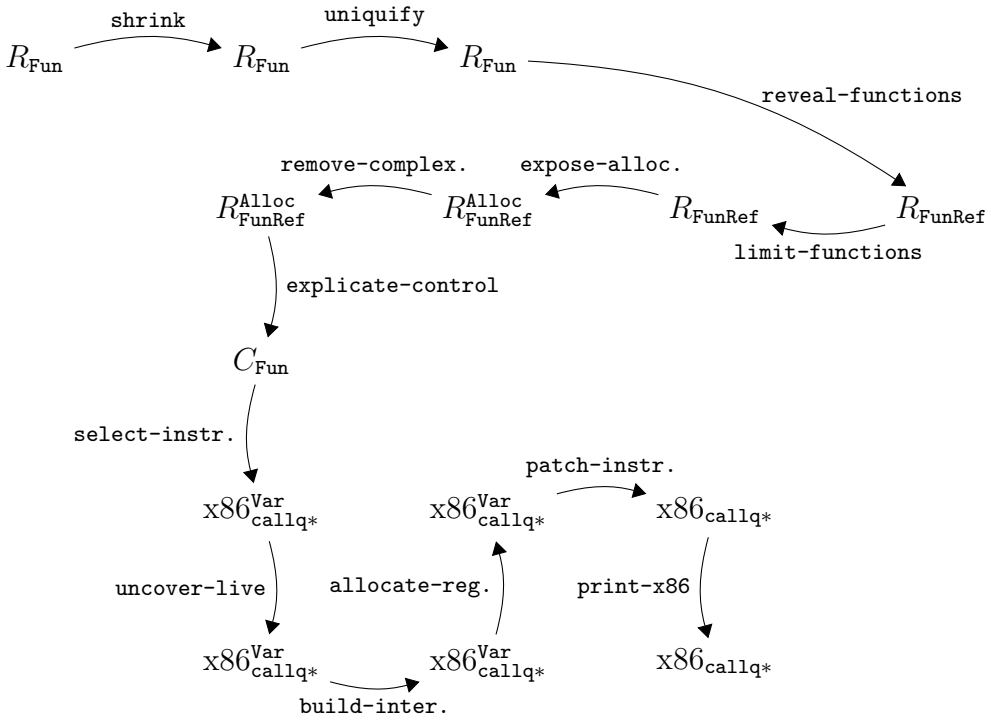


图 6.12: 具有函数的语言 $R_{\texttt{Fun}}$ 的通道图

```
(define (add [x : Integer] [y : Integer])
  : Integer
  (+ x y))
(add 40 2)
```

⇓

```
(define (add86 [x87 : Integer]
               [y88 : Integer]) : Integer
  add86start:
    return (+ x87 y88);
  )
(define (main) : Integer ()
  mainstart:
    tmp89 = (fun-ref add86);
    (tail-call tmp89 40 2)
  )
```

⇒

```
(define (add86) : Integer
  add86start:
    movq %rdi, x87
    movq %rsi, y88
    movq x87, %rax
    addq y88, %rax
    jmp add11389conclusion
  )
(define (main) : Integer
  mainstart:
    leaq (fun-ref add86), tmp89
    movq $40, %rdi
    movq $2, %rsi
    tail-jmp tmp89
  )
```

⇓

```
        .globl add86
        .align 16
add86:
        pushq  %rbp
        movq   %rsp, %rbp
        jmp    add86start
add86start:
        movq   %rdi, %rax
        addq   %rsi, %rax
        jmp add86conclusion
add86conclusion:
        popq   %rbp
        retq
```

```
        .globl main
        .align 16
main:
        pushq  %rbp
        movq   %rsp, %rbp
        movq   $16384, %rdi
        movq   $16384, %rsi
        callq  initialize
        movq   rootstack_begin(%rip), %r15
        jmp    mainstart
mainstart:
        leaq   add86(%rip), %rcx
        movq   $40, %rdi
        movq   $2, %rsi
        movq   %rcx, %rax
        popq   %rbp
        jmp    *%rax
mainconclusion:
        popq   %rbp
        retq
```

图 6.13: 一个简单函数在 x86 上的编译示例。

7

词法作用域的函数

本章研究在像 Racket 这样的函数语言中出现的词法作用域函数。词法作用域是指函数体可以引用绑定位置在函数外部、在封闭作用域中的变量。考虑图 7.1 中用 R_λ 编写的例子，它使用 `lambda` 形式扩展 $R_{\texttt{Fun}}$ 的匿名函数。`lambda` 的体引用三个变量：`x` 、`y` 和 `z` 。`x` 和 `y` 的结合位点在 `lambda` 之外。变量 `y` 以外围 `let` 为界，`x` 是函数 `f` 的参数。`lambda` 是从函数 `f` 中返回。程序的主表达式包括两次对 `f` 的调用，使用 `x` 的不同参数，先是 `5` ，然后是 `3` 。从 `f` 返回的函数被绑定到变量 `g` 和 `h` 。尽管这两个函数是由同一个 `lambda` 创建的，但它们实际上是不同的函数，因为它们对 `x` 使用不同的值。对 `11` 加 `g` 得到 `20` ，而对 `15` 加 `h` 得到 `22` 。这个程序的结果是 `42` 。

```
(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))))
```

图 7.1: 一个词法作用域函数的示例。

实现词法作用域函数的方法是将它们编译为顶级函数定义，从 R_{λ} 转换为 $R_{\texttt{Fun}}$ 。但是，编译器需要对图 7.1 中的 `lambda` 中的 `x` 和 `y` 等变量进行特殊处理。毕竟，$R_{\texttt{Fun}}$ 函数不能引用外部定义的变量。为了识别这些变量的出现，回顾自由变量的标准概念。

Definition 31. *如果一个变量出现在 e 中，但在 e 中没有封闭绑定，那么它在表达式 e 中是 自由 的。*

例如，在表达式 `(+ x (+ y z))` 中，变量 `x` 、`y` 和 `z` 都是自由的。另一方面，在下面的表达式中只有 `x` 和 `y` 是自由的，因为 `z` 受 `lambda` 的限制。

```
(lambda: ([z : Integer]) : Integer
  (+ x (+ y z)))
```

所以，`lambda` 的自由变量是需要特殊处理的。需要安排某种方式，在运行时将这些变量的值从创建 `lambda` 的地方传输到应用 `lambda` 的地方。由于 Cardelli [20] ，这个问题的一个有效解决方案是将自由变量的值与 lambda 代码的函数指针捆绑到一个向量中，这种安排称为 *扁平闭包* (将其缩短为“闭包”) 。幸运的是，已经具备闭包的所有要素，第 5 章给出向量，第 6 章给出函数指针。函数指针驻留在索引 0 处，自由变量的值将填充向量的其余部分。

回顾一下图 7.1 中的示例，看看闭包是如何工作的。分三步。程序首先调用函数 `f` ，它为 `lambda` 创建一个闭包。闭包是一个向量，它的第一个元素是一个指针，指向我们将为 `lambda` 生成的顶级函数，第二个元素是 `x` 的值，即 `5`，第三个元素是 `4` ，即 `y` 的值。闭包不包含 `z` 的元素，因为 `z` 不是 `lambda` 的自由变量。创建结束是第一步。闭包从 `f` 返回并绑定到 `g` ，如图 7.2 所示。对 `f` 的第二次调用创建另一个闭包，这一次 `3` 在第二个槽中(用于 `x`) 。这个闭包也从 `f` 返回，但绑定到 `h` ，如图 7.2 所示。

继续这个示例，考虑图 7.1 中 `g` 对 `11` 的应用。要应用闭包，我们获取闭包第一个元素中的函数指针并调用它，传入闭包本身，然后传入常规参数，在本例中是 `11` 。这种应用闭包的技巧是第二步。但是对于参数 `z` ，这个 `lambda` 不是只有一个参数吗？第三步也是最后一步是为 `lambda` 生成顶层函数。为闭包添加一个附加参数，并在每个自由变量的函数开始处插入 `let` ，以便将这些变量绑定到闭包参数中的适当元素。这种三步被称为 *闭合转*

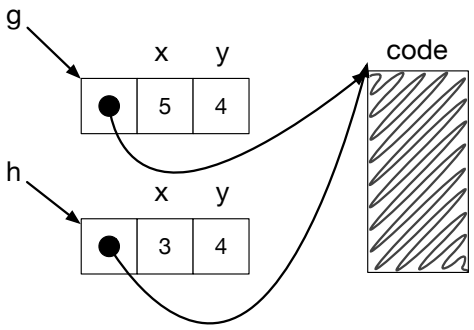


图 7.2: 7.1 中的 lambda 闭包表示示例。

换。在第 7.3 节讨论闭包转换的细节，并从第 7.4 节的示例中生成代码。但是首先在 7.1 节中定义 R_λ 的语法和语义。

7.1 R_λ 语言

图 7.3 和 7.4 定义 R_λ 的具体和抽象语法，这是一种具有匿名函数和词法作用域的语言。它将 lambda 表单添加到 $R_{\texttt{Fun}}$ 的语法中，$R_{\texttt{Fun}}$ 已经有用于函数应用程序的语法。

图 7.5 显示 R_λ 的定义解释程序。lambda 的大小写将当前环境保存在返回的 lambda 中。Apply 的例子在解释 lambda 的主体时使用 lambda 中的环境，即 lam-env 。lam-env 环境通过参数到参数值的映射进行扩展。

图 7.6 显示如何键入检查新的 lambda 表单。lambda 的主体是在包含当前环境 (因为它是词法范围内的) 的环境中检查的，并且还包括 lambda 的参数。我们要求主体的类型与声明的返回类型匹配。

7.2 揭示函数和 F_2 语言

为了支持 procedure-arity 操作符，需要将函数的 arity 传达给闭包创建点。可以通过将 (FunRef *var*) 结构体替换为一个具有第二个属性的结构体来实现这一点：(FunRefArity *var int*) 。这个通道的输出是 F_2 语言，其语法在图 7.7 中定义。

```
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
```

图 7.3: R_λ 的具体语法，用 `lambda` 扩展 $R_{\texttt{Fun}}$ (图 6.1) 。

```
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)
```

图 7.4: R_λ 的抽象语法，扩展 $R_{\texttt{Fun}}$ (图 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))
```

图 7.5: R_λ 的翻译。

```
(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)])]
      ...
      )))
```

图 7.6: 类型检查 lambda' 中的 R_λ 。

exp	::=	... | (FunRefArity *var int*)
def	::=	(Def *var* ([*var* : *type*] ...) *type* '() *exp*)
F_2	::=	(ProgramDefs '() (*def* ...))

图 7.7: 抽象语法 F_2 ， R_λ 的扩展 (图 7.4) 。

7.3 闭包转换

将词汇作用域的函数编译为顶级函数定义是在 `convert-to-closures` 中完成的，该函数位于 `reveal-functions` 之后和 `limit-functions` 之前。

通常，将通道作为 AST 上的递归函数来实现。所有的操作都在 `Lambda` 和 `Apply` 的情况下。将 `Lambda` 表达式转换为创建闭包的表达式，即一个向量，其第一个元素是函数指针，其余元素是 `Lambda` 的自由变量。在这里使用结构 `Closure` 而不是 `vector` ，以便可以在第 7.8 节中将闭包与 vector 区别开来，并记录 Arity。在下面生成的代码中，*name* 是用来识别函数的唯一符号，*arity* 是参数的数量（*ps* 的长度）。

```
(Lambda ps rt body)
⇒
(Closure arity (cons (FunRef name) fvs))
```

除了将每个 `Lambda` 转换为一个 `Closure` 之外，还为每个 `Lambda` 创建一个顶层函数定义，如下所示。

```
(Def name ([clos : (Vector _ fvts ...)] ps′ ...) rt′
  (Let fvs_1 (Prim 'vector-ref (list (Var clos) (Int 1)))
    ...
    (Let fvs_n (Prim 'vector-ref (list (Var clos) (Int n)))
      body′)...))
```

`clos` 参数指向闭包。转换 *ps* 中的类型注释和返回类型 *rt* ，如下一段所讨论的，以获得 *ps′* 和 *rt′* 。*fvts* 类型是 lambda 中的自由变量的类型，而下划线 `_` 是我们使用的虚拟类型，因为在闭包类型中给函数指定类型相当困难。[1] 在类型检查期间，虚拟类型被认为与任何其他类型相等。`Let` 形式的序列将自由变量绑定到从闭包中获得的值上。

闭包转换将函数转换为向量，因此程序中的类型注释也必须进行转换。建议为此定义一个辅助递归函数。函数类型应按如下方式进行翻译。

```
(T_1, ..., T_n -> T_r)
⇒
(Vector ((Vector _) T′_1, ..., T′_n -> T′_r))
```

[1] 要给闭包一个准确的类型，需要向类型检查器添加存在类型 [88]。

上面的类型表示向量中的第一件事是函数指针。函数指针的第一个参数是一个向量 (一个闭包)，其余的参数来自原始函数，类型为 $T'_1 \ldots T'_n$ 。闭包的 `Vector` 类型省略自由变量的类型，因为：1) 这些类型在此上下文中不可用；2) 不需要在为函数应用程序生成的代码中使用它们。

将函数应用程序转换为代码，从闭包中检索函数指针，然后调用该函数，将闭包作为第一个参数传入。将 e' 绑定到一个临时变量，以避免代码重复。

```
(Apply e es)
⇒
(Let tmp e'
  (Apply (Prim 'vector-ref (list (Var tmp) (Int 0))) (cons tmp es')))
```

还有一个问题是如何处理引用顶级函数定义。为了维护函数应用程序的统一翻译，将函数引用转换为闭包。

```
(FunRefArity f n)  ⇒  (Closure n (FunRef f) '())
```

顶层函数定义也需要更新，以接受一个额外的闭包参数。

7.4 一个例子的翻译

图 7.8 显示 `reveal-functions` 和 `convert-to-closures` 的结果，这个示例程序演示在本章开始时讨论的词法作用域。

Exercise 32. 扩展你的编译器来处理 R_λ 在这一章概述。创建 5 个使用 `lambda` 函数并利用词法作用域的新程序。在这些新程序和之前创建的所有测试程序上测试编译器。

7.5 公开分配

编译 `(Closure` *arity* `(`*exp*`...))` 形式转换为代码，用于分配和初始化 vector，类似于第 5.4 节中 `vector` 操作符的转换。唯一的区别是使用 `(AllocateClosure` *len type arity*`)` 替换 `(Allocate` *len type*`)` 。

```
(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)))))
```

⇒

```
(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)))))
```

图 7.8: 闭包转换的示例。

$$
\begin{array}{lcl}
\mathit{exp} & ::= & \ldots \mid \texttt{(AllocateClosure}\ \mathit{int}\ \mathit{type}\ \mathit{int}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{(Assign (Var}\ \mathit{var}\texttt{)}\ \mathit{exp}\texttt{)} \mid \texttt{(Collect}\ \mathit{int}\texttt{)} \\
\mathit{tail} & ::= & \texttt{(Return}\ \mathit{exp}\texttt{)} \mid \texttt{(Seq}\ \mathit{stmt}\ \mathit{tail}\texttt{)} \mid \texttt{(Goto}\ \mathit{label}\texttt{)} \\
 & \mid & \texttt{(IfStmt (Prim}\ \mathit{cmp}\ \texttt{(}\mathit{atm}\ \mathit{atm}\texttt{)) (Goto}\ \mathit{label}\texttt{) (Goto}\ \mathit{label}\texttt{))} \\
 & \mid & \texttt{(TailCall}\ \mathit{atm}\ \mathit{atm}\ldots\texttt{)} \\
\mathit{def} & ::= & \texttt{(Def}\ \mathit{label}\ \texttt{(}[\mathit{var}\texttt{:}\mathit{type}]\ldots\texttt{)}\ \mathit{type}\ \mathit{info}\ \texttt{((}\mathit{label}\ \texttt{.}\ \mathit{tail}\texttt{)}\ldots\texttt{))} \\
C_{\texttt{Clos}} & ::= & \texttt{(ProgramDefs}\ \mathit{info}\ \texttt{(}\mathit{def}\ldots\texttt{))}
\end{array}
$$

图 7.9: $C_{\texttt{Clos}}$ 的抽象语法，扩展 $C_{\texttt{Fun}}$ (图 6.9)。

7.6 解释控制和 $C_{\texttt{Clos}}$

`explicate-control` 的输出语言是 $C_{\texttt{Clos}}$ ，其抽象语法在图 7.9 中定义。与 $C_{\texttt{Fun}}$ 的唯一区别是在 *exp* 语法中添加 `AllocateClosure` 形式。`explicate-control` 通道中的 `AllocateClosure` 的处理类似于对其他表达式 (如原语操作符) 的处理。

7.7 选择指令

编译 `(AllocateClosure` *len type arity*`)` 的方式几乎与 `(Allocate` *len type*`)` 形式 (第 5.7 节) 相同。唯一的区别是，应该将 *arity* 放在存储在向量 0 位置的标记中。回想一下，在第 5.7 节中没有使用 64 位标记的一部分。从 58 号位置开始，把它存储在 5 位中。

将 `procedure-arity` 操作符编译为一个指令序列，该指令从向量的位置 0 访问标记，并从标记中提取从位置 58 开始的 5 位。

图 7.10 提供编译 R_λ 所需的所有通道的概述。

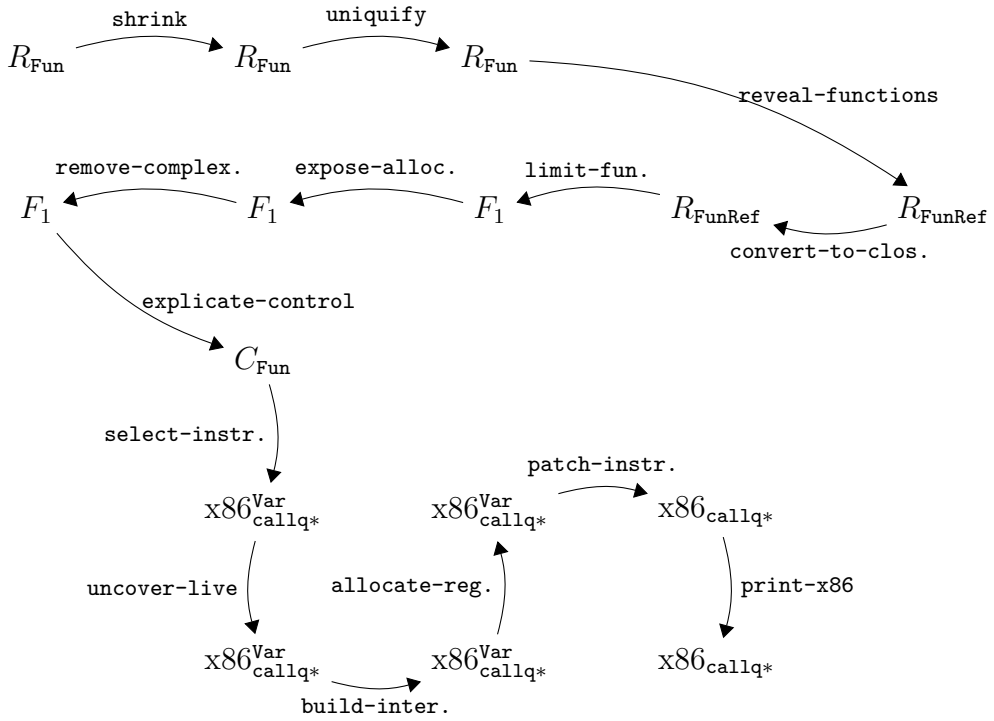


图 7.10: R_λ 的通道图，这是一种具有词汇作用域的函数的语言。

7.8 挑战：优化闭包

在本章中，将词汇作用域的函数编译成一个相对有效的表示：扁平闭包。然而，即使是这种表示也会带来一些开销。例如，考虑下面的程序，它的函数 tail-sum 没有任何自由变量，并且 tail-sum 所有用法都在我们知道仅应用 tail-sum 的应用程序中（而不是其他任何函数）。

```
(define (tail-sum [n : Integer] [r : Integer]) : Integer
  (if (eq? n 0)
      r
      (tail-sum (- n 1) (+ n r))))

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

如本章所述，统一地对所有函数应用闭包转换，得到这个程序的如下输出。

```
(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))
```

在前一章中，程序中没有分配，对 tail-sum 的调用将是直接调用。相比之下，上面的程序为每个 closure 分配内存，对 tail-sum 的调用是间接的。这两个差异在这样的程序中产生相当大的开销，在这种程序中，分配和间接调用发生在一个紧密循环中。

有人可能认为这个问题是微不足道的解决：不能只是识别形式的调用 ((fun-ref f) $e_1 \ldots e_n$)，并编译它们直接调用 ((fun-ref f) $e'_1 \ldots e'_n$)，而不是把它作为一个闭包的调用？还将删除 tail_sum1 的 fvs5 参数。然而，这个问题并不是那么简单，因为全局函数可能会“转义”，并在也涉及

闭包的应用程序中出现。考虑下面的例子，应用程序 (f 41) 需要被编译成一个闭包应用程序，因为 lambda 可能被绑定到 f ，但 add1 函数也可能被绑定到 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)))
```

如果全局函数名不是直接调用中的操作符，而是以其他方式使用，则称该函数 转义。如果全局函数没有转义，则不需要对该函数执行闭包转换。

Exercise 33. 实现一个辅助函数来检测哪个全局函数转义。使用该函数，实现闭包转换的改进版本，不对闭包转换进行转义，而是将它们作为常规函数编译的全局函数。创建几个新的测试用例，检查是否正确地检测到全局函数是否转义。

到目前为止，已经减少调用全局函数的开销，但是当我们可以在编译时确定将调用哪个 lambda 时，减少调用 lambda 的开销也会很好。我们称此类调用为 已知调用。考虑下面的例子，在这个例子中，lambda 被限定为 f ，然后应用。

```
(let ([y (read)])
  (let ([f (lambda: ([x : Integer]) : Integer
             (+ x y))])
    (f 21)))
```

闭包转换 (f 21) 编译为一个间接调用：

```
(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))))
```

但是可以将应用程序 `(f 21)` 编译成对 `lambda5` 的直接调用：

```
(define (main) : Integer
  (let ([y2 (read)])
    (let ([f4 (Closure 1 (list (fun-ref lambda5) y2))])
      ((fun-ref lambda5) f4 21))))
```

一般来说，确定某个特定应用程序将调用哪个 lambda 的问题相当具有挑战性，这是需要进行大量研究的主题 [100, 53] 。对于下面的练习，建议您将应用程序编译为直接调用，当操作符是一个变量并且变量被 `let` 绑定到一个闭包时。这可以通过维护一个将 `let` 绑定变量映射到函数名的环境来实现。当在 `let` 的右边遇到一个闭包时，扩展环境，将 `let` 绑定变量映射到闭包的全局函数名。

Exercise 34. 实现一个名为 `optimize-known-calls` 的编译器通道，将已知调用编译为直接调用。在几个示例程序上验证编译器在这方面是成功的。

这些练习只是触及闭包优化的表面。对感兴趣的读者来说，下一步是看看 Keep et al. [69] 的著作。

7.9 延伸阅读

词法上限定匿名函数的概念比现代计算机早大约十年。它们是由 Church [27] 发明的，他提出 λ 演算作为逻辑的基础。匿名函数包含在 LISP [84] 编程语言中，但最初是动态作用域的。LISP 的 Scheme 方言采用词法作用域，Steele [105] 演示如何高效地编译 Scheme 程序。但是，环境被表示为链表，所以变量查找在环境的大小中是线性的。在本章中，使用扁平闭包来表示环境，扁平闭包是 Cardelli [20, 21] 为编译 ML 语言 [55, 87] 而发明的。对于扁平闭包，变量查找的时间是固定的，但是创建闭包的时间与它的自由变量的

数量成正比。扁平闭包是由 Dybvig [38] 在他的博士论文中重新发明的，并在 Chez Scheme 版本 1 [39] 中使用。

8

动态类型

本章讨论 $R_{\texttt{Dyn}}$ 的编译，是一种动态类型语言，是 Racket 的一个子集。这是与前面的章节形成对比，其中已经研究类型化 Racket 的编译。在 $R_{\texttt{Dyn}}$ 等动态类型语言中，给定的表达式每次执行时可能产生不同类型的值。考虑下面的示例，该示例带有条件 `if` 表达式，它可能根据程序的输入返回一个布尔值或整数。

```
(not (if (eq? (read) 1) #f 0))
```

允许表达产生不同值的语言被称为 *多态*，这个词由希腊语词根“poly”(意为“许多”) 和“morph”(意为“形状”) 组成。在编程语言中有几种多态性，例如子类型多态性和参数多态性 [22] 。在本章中研究的多态性没有一个特殊的名称，但它是动态类型语言中出现的一种。

动态类型语言的另一个特征是基元操作，比如 `not` ，通常被定义为对许多不同类型的值进行操作。实际上，在 Racket 中，`not` 操作符为任何类型的值产生结果：给定 `#f` 返回 `#t` ，给定其他任何值返回 `#f` 。此外，即使原语操作将其输入限制为某种类型的值，也会在运行时而不是编译期间强制执行这种限制。例如，下面的向量引用导致运行时契约违反，因为索引必须是整数，而不是像 `#t` 这样的布尔值。

```
(vector-ref (vector 42) #t)
```

Racket 子集的 $R_{\texttt{Dyn}}$ 的具体和抽象语法在图 8.1 和 8.2 中定义。$R_{\texttt{Dyn}}$ 没有类型检查器，因为它不是静态类型语言 (它是动态类型的！)。

$$
\begin{array}{rcl}
\mathit{cmp} & ::= & \texttt{eq?} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\mathit{exp} & ::= & \mathit{int} \mid \texttt{(read)} \mid \texttt{(-}\ \mathit{exp}\texttt{)} \mid \texttt{(+}\ \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(-}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
& \mid & \mathit{var} \mid \texttt{(let ([}\mathit{var}\ \mathit{exp}\texttt{])}\ \mathit{exp}\texttt{)} \\
& \mid & \texttt{\#t} \mid \texttt{\#f} \mid \texttt{(and}\ \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(or}\ \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(not}\ \mathit{exp}\texttt{)} \\
& \mid & \texttt{(}\mathit{cmp}\ \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(if}\ \mathit{exp}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
& \mid & \texttt{(vector}\ \mathit{exp}\ldots\texttt{)} \mid \texttt{(vector-ref}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
& \mid & \texttt{(vector-set!}\ \mathit{exp}\ \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(void)} \\
& \mid & \texttt{(}\mathit{exp}\ \mathit{exp}\ldots\texttt{)} \mid \texttt{(lambda (}\mathit{var}\ldots\texttt{)}\ \mathit{exp}\texttt{)} \\
& \mid & \texttt{(boolean?}\ \mathit{exp}\texttt{)} \mid \texttt{(integer?}\ \mathit{exp}\texttt{)} \\
& \mid & \texttt{(vector?}\ \mathit{exp}\texttt{)} \mid \texttt{(procedure?}\ \mathit{exp}\texttt{)} \mid \texttt{(void?}\ \mathit{exp}\texttt{)} \\
\mathit{def} & ::= & \texttt{(define (}\mathit{var}\ \mathit{var}\ldots\texttt{)}\ \mathit{exp}\texttt{)} \\
R_{\texttt{Dyn}} & ::= & \mathit{def}\ldots\ \mathit{exp}
\end{array}
$$

图 8.1: $R_{\texttt{Dyn}}$ 的语法，一种非类型化语言 (Racket 的子集)。

$$
\begin{array}{rcl}
\mathit{exp} & ::= & \texttt{(Int}\ \mathit{int}\texttt{)} \mid \texttt{(Var}\ \mathit{var}\texttt{)} \mid \texttt{(Let}\ \mathit{var}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
& \mid & \texttt{(Prim}\ \mathit{op}\ \texttt{(}\mathit{exp}\ldots\texttt{))} \\
& \mid & \texttt{(Bool}\ \mathit{bool}\texttt{)} \mid \texttt{(If}\ \mathit{exp}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
& \mid & \texttt{(Void)} \mid \texttt{(Apply}\ \mathit{exp}\ \mathit{exp}\ldots\texttt{)} \\
& \mid & \texttt{(Lambda (}\mathit{var}\ldots\texttt{) 'Any}\ \mathit{exp}\texttt{)} \\
\mathit{def} & ::= & \texttt{(Def}\ \mathit{var}\ \texttt{(}\mathit{var}\ldots\texttt{) 'Any '()}\ \mathit{exp}\texttt{)} \\
R_{\texttt{Dyn}} & ::= & \texttt{(ProgramDefsExp '() (}\mathit{def}\ldots\texttt{)}\ \mathit{exp}\texttt{)}
\end{array}
$$

图 8.2: $R_{\texttt{Dyn}}$ 的抽象语法。

$R_{\texttt{Dyn}}$ 的定义解释器如图 8.3 所示，其辅助函数在图 8.4 中定义。考虑 `(Int n)` 的匹配情况。代替简单地返回整数 `n` (如图 2.3 中 $R_{\texttt{Var}}$ 解释器)，$R_{\texttt{Dyn}}$ 的解释器创建一个 *标记值* ，该值将基础值与标识其类型的标记结合在一起。定义以下结构来表示标记值。

```
(struct Tagged (value tag) #:transparent)
```

标签是 `Integer` 、`Boolean` 、`Void` 、`Vector` 和 `Procedure` 。标记与类型密切相关，但并不总是捕获类型所做的所有信息。例如，一个类型为 `(Vector Any Any)` 的向量被标记为 `Vector` ，一个类型为 `(Any Any -> Any)` 的过程被标记为 `Procedure` 。

接下来考虑向量 `vector-ref` 的匹配情况。`check-tag` 辅助函数 (图 8.4) 用于确保第一个参数是一个向量，第二个参数是一个整数。如果不是，将引发 `trapped-error` 。回想一下第 1.5 节，当定义解释器引发 `trapped-error` 错误时，编译后的代码也必须通过退出返回代码 `255` 发出错误信号。如果索引不小于向量的长度，也会引发 `trapped-error` 。

```
(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)]))]
```

图 8.3: $R_{\texttt{Dyn}}$ 语言的解释器。

```
(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)))
```

图 8.4: $R_{\texttt{Dyn}}$ 解释器的辅助函数。

8.1 标记值的表示

$R_{\texttt{Dyn}}$ 的解释器引入一种新的值，标记值。要将 $R_{\texttt{Dyn}}$ 编译为 x86，必须决定如何在位级别表示标记值。因为 $R_{\texttt{Dyn}}$ 中的几乎每个操作都涉及到操作标记值，所以表示必须是高效的。回想一下，所有的值都是 64 位的。取最右边的 3 位来编码标签。使用 001 来标识整数，100 用于布尔值，010 用于向量，011 用于过程，101 用于空值。定义以下用于将类型映射到标记代码的辅助函数。

$$\begin{aligned} \mathit{tagof}(\texttt{Integer}) &= 001 \\ \mathit{tagof}(\texttt{Boolean}) &= 100 \\ \mathit{tagof}((\texttt{Vector}\ldots)) &= 010 \\ \mathit{tagof}((\ldots\texttt{->}\ldots)) &= 011 \\ \mathit{tagof}(\texttt{Void}) &= 101 \end{aligned}$$

这种取 3 位的行为是有代价的：整数被减少到 -2^{60} 到 2^{60} 之间。这种窃取并不会对向量和过程产生负面影响，因为这些值是地址，而的地址是 8 字节对齐的，所以最右边的 3 位是没有用的，它们总是 000 。因此，通过用标签覆盖最右边的 3 位，不会丢失信息，而且可以简单地使标签归零以恢复原始地址。

为了将标记值放入第一类实体中，可以给它们一个名为 `Any` 的类型，并定义诸如 `Inject` 和 `Project` 之类的操作来创建和使用它们，从而生成 $R_{\texttt{Any}}$ 中间语言。在第 8.3 节中描述如何将 $R_{\texttt{Dyn}}$ 编译为 $R_{\texttt{Any}}$ ，但首先将更详细地描述 $R_{\texttt{Any}}$ 语言。

8.2 $R_{\texttt{Any}}$ 语言

$R_{\texttt{Any}}$ 的抽象语法在图 8.5 中定义。($R_{\texttt{Any}}$ 的具体语法见附录图 12.1。) (`Inject` e T) 形式将 T 类型的表达式 e 产生的值转换为标记值。(`Project` e T) 形式将表达式 e 产生的标记值转换为类型 T 的值，或者如果类型标记不等同于 T ，则停止程序。请注意，在 `Inject` 和 `Project` 中，类型 T 都被限制为平面类型 *ftype* ，这简化实现，并与编译 $R_{\texttt{Dyn}}$ 所需的类型相对应。

type	::=	... | `Any`
op	::=	... | `any-vector-length` | `any-vector-ref` | `any-vector-set!`
	|	`boolean?` | `integer?` | `vector?` | `procedure?` | `void?`
exp	::=	... | `(Prim` *op* `(`*exp*`...))`
	|	`(Inject` *exp ftype*`)` | `(Project` *exp ftype*`)`
def	::=	`(Def` *var* `([`*var*`:`*type*`]...)` *type* `'()` *exp*`)`
R_{Any}	::=	`(ProgramDefsExp '() (`*def*`...)` *exp*`)`

图 8.5: R_{Any} 的抽象语法，扩展 R_λ (图 7.4)。

`any-vector` 操作符适用于 vector 操作，因此它们可以应用于 `Any` 类型的值。它们还推广向量操作，即索引在语法中不被限制为一个字面整数，但允许为任何表达式。

像 `boolean?` 这样的类型谓词期望它们的参数产生一个标记值；如果标记对应于谓词，则返回 `#t` ，否则返回 `#f` 。

R_{Any} 的类型检查器如图 8.6 和 8.7 所示，并使用图 8.8 中的辅助函数。R_{Any} 的解释器在图 8.9 中，辅助函数 `apply-inject` 和 `apply-project` 在图 8.10 中。

```
(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)]
```

图 8.6: R_{Any} 语言的类型检查器，第 1 部分。

```
     [(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)]))]

))
```

图 8.7: $R_{\texttt{Any}}$ 语言的类型检查器，第 2 部分。

```
(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]))
```

图 8.8: 类型检查 R_{Any} 的辅助方法。

```
(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))
```

图 8.9: $R_{\texttt{Any}}$ 解释器。

```
(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)]))
```

图 8.10: 用于注入和投影的辅助功能。

8.3 强制转换插入：将 R_{Dyn} 编译为 R_{Any}

cast-insert 通道从 R_{Dyn} 编译到 R_{Any} 。图 8.11 显示将许多 R_{Dyn} 表单编译成 R_{Any} 的过程。此通道的一个重要不变式是，给定 R_{Dyn} 程序中的子表达式 e ，通道将产生 R_{Any} 中的表达式 e' ，该表达式的类型为 Any 。例如，图 8.11 中的第一行显示布尔 #t 的编译，必须注入它才能产生类型 Any 的表达式。图 8.11 的第二行是加法的编译，它代表许多基本操作的编译: 参数的类型为 Any ，并且必须在加法执行之前被映射为 Integer 。

lambda 的编译 (图 8.11 的第三行) 显示当我们需要产生类型注释时发生了什么：只需使用 Any 。if 和 eq? 的编译说明这个通道如何解释 R_{Dyn} 和 R_{Any} 之间行为上的一些差异。R_{Dyn} 语言比 R_{Any} 更宽容，可以在不同的地方使用什么样的值。例如，if 的条件不一定是布尔值。对于 eq? ，参数不需要是相同类型的 (在这种情况下结果是 #f)。

8.4 显示演员表

在 reveal-casts 通道中，建议将 project 编译成 if 表达式，用于检查值的标记是否与目标类型匹配; 如果是，则通过删除标记将值转换为目标类型的值; 如果没有，程序退出。为了执行这些操作，需要一个新的基本操作，tag-of-any ，以及两个新的表单，ValueOf 和 Exit 。tag-of-any 操作从 Any 类型的标记值中检索类型标记。ValueOf 表单从标记值中检索基础值。ValueOf 表单包含类型检查器使用的基础值的类型。最后，Exit 表单结束程序的执行。

如果投影的目标类型是 Boolean 或 Integer ，那么 Project 可以翻译如下。

`#t`	⇒	`(inject #t Boolean)`
`(+` e_1 e_2`)`	⇒	`(inject (+ (project` e_1' `Integer) (project` e_2' `Integer)) Integer)`
`(lambda (`$x_1 \ldots x_n$`)` e`)`	⇒	`(inject (lambda: ([`x_1`:Any]...[`x_n`:Any]):Any` e'`) (Any...Any -> Any))`
`(`e_0 $e_1 \ldots e_n$`)`	⇒	`((project` e_0' `(Any...Any -> Any))` $e_1' \ldots e_n'$`)`
`(vector-ref` e_1 e_2`)`	⇒	`(any-vector-ref` e_1' e_2'`)`
`(if` e_1 e_2 e_3`)`	⇒	`(if (eq?` e_1' `(inject #f Boolean))` e_3' e_2'`)`
`(eq?` e_1 e_2`)`	⇒	`(inject (eq?` e_1' e_2'`) Boolean)`
`(not` e_1`)`	⇒	`(if (eq?` e_1' `(inject #f Boolean)) (inject #t Boolean) (inject #f Boolean))`

图 8.11: 演员表插入

```
(Project e ftype)
⇒
(Let tmp e′
  (If (Prim 'eq? (list (Prim 'tag-of-any (list (Var tmp)))
                       (Int tagof(ftype))))
    (ValueOf tmp ftype)
    (Exit)))
```

如果投影的目标类型是向量或函数类型，则需要做更多的工作。对于向量，检查向量类型的长度是否与向量的长度匹配 (使用 `vector-length` 原语)。对于函数，检查函数类型中的参数数量是否与函数的属性匹配 (使用 `procedure-arity`)。

关于 `inject` ，建议将其编译为一个稍低级别的原语操作 `make-any` 。该操作接受一个标记而不是类型。

```
(Inject e ftype)
⇒
(Prim 'make-any (list e′ (Int tagof(ftype))))
```

类型谓词 (`boolean?` 等) 可以被翻译成 `tag-of-any` 和 `eq?` 的用法，类似于 `Project` 的翻译。

`any-vector-ref` 和 `any-vector-set!` 操作结合投影操作和向量操作。另外，读和写操作允许索引的任意表达式，因此 $R_{\texttt{Any}}$ 的类型检查器 (图 8.6) 不能保证索引在范围内。因此，插入代码在运行时执行边界检查。`any-vector-ref` 的转换如下，其他两个操作以类似的方式进行转换。

```
(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)))))
```

$$
\begin{array}{rcl}
\mathit{exp} & ::= & \ldots \mid \texttt{(Prim 'any-vector-ref (}\mathit{atm}\ \mathit{atm}\texttt{))} \\
& \mid & \texttt{(Prim 'any-vector-set! (list }\mathit{atm}\ \mathit{atm}\ \mathit{atm}\texttt{))} \\
& \mid & \texttt{(ValueOf }\mathit{exp}\ \mathit{ftype}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{(Assign (Var }\mathit{var}\texttt{) }\mathit{exp}\texttt{)} \mid \texttt{(Collect }\mathit{int}\texttt{)} \\
\mathit{tail} & ::= & \texttt{(Return }\mathit{exp}\texttt{)} \mid \texttt{(Seq }\mathit{stmt}\ \mathit{tail}\texttt{)} \mid \texttt{(Goto }\mathit{label}\texttt{)} \\
& \mid & \texttt{(IfStmt (Prim }\mathit{cmp}\ \texttt{(}\mathit{atm}\ \mathit{atm}\texttt{)) (Goto }\mathit{label}\texttt{) (Goto }\mathit{label}\texttt{))} \\
& \mid & \texttt{(TailCall }\mathit{atm}\ \mathit{atm}\ldots\texttt{)} \mid \texttt{(Exit)} \\
\mathit{def} & ::= & \texttt{(Def }\mathit{label}\ \texttt{([}\mathit{var}\texttt{:}\mathit{type}\texttt{]}\ldots\texttt{)}\ \mathit{type}\ \mathit{info}\ \texttt{((}\mathit{label}\ .\ \mathit{tail}\texttt{)}\ldots\texttt{))} \\
C_{\texttt{Clos}} & ::= & \texttt{(ProgramDefs }\mathit{info}\ \texttt{(}\mathit{def}\ldots\texttt{))}
\end{array}
$$

图 8.12: $C_{\texttt{Any}}$ 的抽象语法，扩展 $C_{\texttt{Clos}}$ (图 7.9)。

8.5 去除复杂的操作数

`ValueOf` 和 `Exit` 表单都是复杂表达式。`ValueOf` 的子表达式必须是原子的。

8.6 解释控制和 $C_{\texttt{Any}}$

`explicate-control` 的输出是 $C_{\texttt{Any}}$ 语言，其语法在图 8.12 中定义。添加到 $R_{\texttt{Any}}$ 中的 `ValueOf` 形式仍然是一个表达式，而 `Exit` 表达式变成 *tail*。另外，注意 `vector-ref` 和 `vector-set!` 的索引参数是 *atm* 而不是整数，就像 $C_{\texttt{Vec}}$ (图 5.13)。

8.7 选择指令

在 `select-instructions` 中，将 `Any` 类型上的基本操作转换为 x86 指令，该指令涉及操作标记值的 3 个标记位。

Make-any 如果标签是 `Integer` 或 `Boolean` ,建议按如下方式编译 `make-any` 原语。`salq` 指令将目的地向左移动指定其源参数的位数 (在本例中为 3 ，标记的长度)，并保留整数的符号。使用 `orq` 指令将标记和值组合起来，形成

标记值。

```
(Assign lhs (Prim 'make-any (list e (Int tag))))
⇒
movq e′, lhs′
salq $3, lhs′
orq $tag, lhs′
```

对于向量和过程的指令选择是不同的，因为它们不需要将它们向左移动。最右边的 3 位已经是 0，就像本章开头描述的那样。因此，只需使用 `orq` 将值和标记结合起来。

```
(Assign lhs (Prim 'make-any (list e (Int tag))))
⇒
movq e′, lhs′
orq $tag, lhs′
```

Tag-of-any 回想一下，`tag-of-any` 操作从 `Any` 类型的值中提取类型标记。类型标签是最下面的三位，所以通过取位和值的 111 (十进制数为 7) 来获得标签。

```
(Assign lhs (Prim 'tag-of-any (list e)))
⇒
movq e′, lhs′
andq $7, lhs′
```

ValueOf 与 `make-any` 一样，`ValueOf` 的指令也因 T 类型是指针 (向量或过程) 或非指针 (整数或布尔) 而不同。下面显示整数和布尔的指令选择。通过向右移动 3 位来产生未标记的值。

```
(Assign lhs (ValueOf e T))
⇒
movq e′, lhs′
sarq $3, lhs′
```

对于向量和程序，没有必要进行转移。相反，只需要去掉最右边的 3 位。通过创建位模式 ...0111 (十进制 7) 并应用 `bitwise-not` 来获得 ...11111000 (十进制 -8) ，将其 `movq` 放入目标 *lhs* 中来实现这一点。然后，应用 `andq` 与标记值以获得所需的结果。

```
(Assign lhs (ValueOf e T))
⇒
movq $−8, lhs′
andq e′, lhs′
```

Any-vector-length

```
(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 索引可以是一个任意的原子，所以不是在编译时计算偏移量，而是需要生成指令在运行时计算偏移量，如下所示。注意新指令 `imulq` 的使用。

```
(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! `any-vector-set!` 的代码生成类似于其他 `any-vector` 操作。

8.8 R_{Any} 的寄存器分配

标记值和垃圾收集之间有一个有趣的交互，它会影响寄存器分配。类型为 `Any` 的变量可能引用一个向量，因此它可能是一个根，需要在垃圾收集期间检查和复制。因此，为了实现寄存器分配，需要以类似于 `Vector` 类型变量的方式来处理 `Any` 类型变量。尤其，

- 如果一个 `Any` 类型的变量在函数调用期间处于活动状态，那么它必须被溢出。这可以通过更改 `build-interference` 来完成，将 `callq` 之后存在的所有类型为 `Any` 的变量标记为干扰所有寄存器。
- 如果溢出类型为 `Any` 的变量，则必须将其溢出到根堆栈，而不是常规过程调用堆栈。

关于根堆栈的另一个问题是，垃圾收集器需要区分：(1) 普通旧的元组指针；(2) 指向元组的标记值；(3) 不是元组的标记值。通过选择在 *tagof* 函数中不使用标签 000 来实现这种区分。相反，该位模式被保留用于标识指向元组的普通旧指针。这样，如果设置前三个位中的一个，那么就有一个标记值，并且检查标记可以区分向量 (010) 和其他类型的值。

Exercise 35. 展开编译器以处理后面几节中讨论的 R_{Any} 。创建 5 个新程序，使用 `Any` 类型和新操作 (`inject` 、`project` 、`boolean?` 等)。在这些新程序和之前创建的所有测试程序上测试编译器。

Exercise 36. 展开编译器来处理 R_{Dyn} ，如本章所述。通过删除类型注释来调整以前的十个测试程序，为 R_{Dyn} 创建测试。添加另外 5 个测试程序，这些程序特别依赖于被动态类型化的语言。也就是说，它们不应该是静态类型语言中的合法程序，但是，它们应该是有效的 R_{Dyn} 程序，运行到完成没有错误。

图 8.13 提供编译 R_{Dyn} 所需的所有通道的概览。

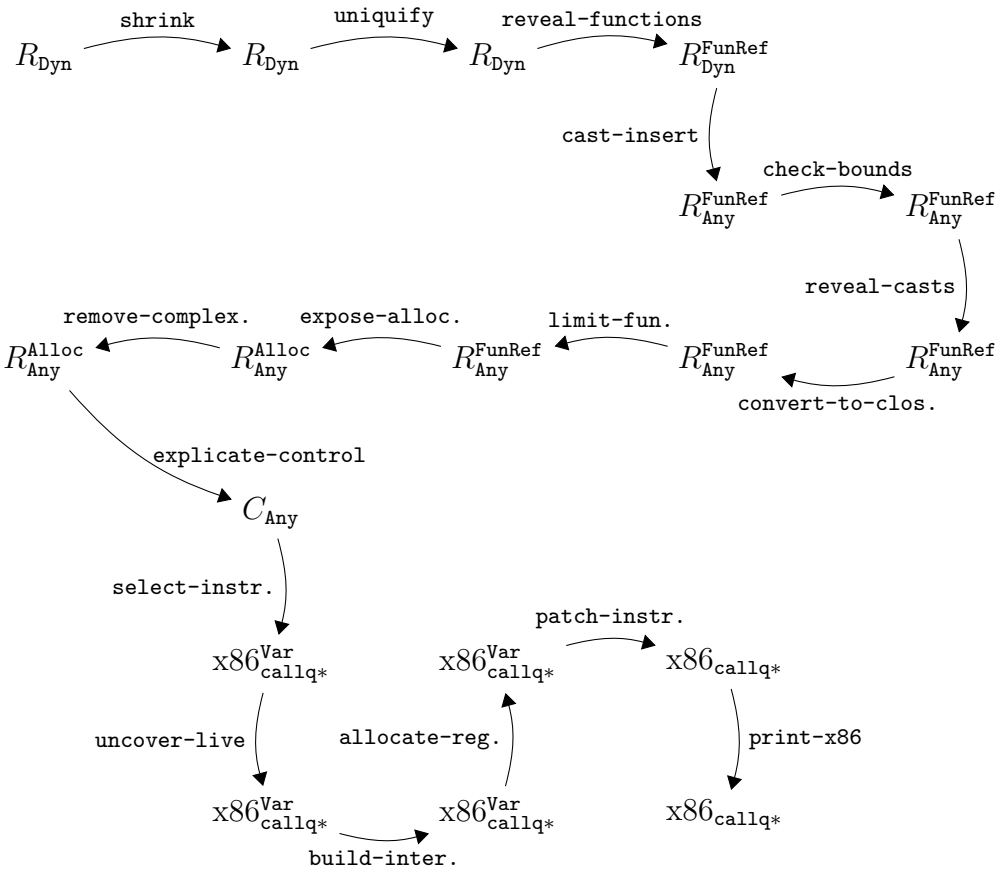


图 8.13: 动态类型语言 $R_{\texttt{Dyn}}$ 的通道图。

9

循环和赋值

在本章中，将研究命令式编程语言的两个特征: 循环和局部变量赋值。下面的示例通过计算前 5 个正整数的和来演示这些新特性。

```
(let ([sum 0])
  (let ([i 5])
    (begin
      (while (> i 0)
        (begin
          (set! sum (+ sum i))
          (set! i (- i 1))))
      sum)))
```

`while` 循环由一个条件和一个循环体组成。`set!` 由一个变量和右边的表达式组成。`while` 循环和 `set!` 的主要目的都是为了引起副作用，所以在语言特性中也包含用于排序副作用的 `begin` 表达式是很方便的。它由一个或多个从左到右求值的子表达式组成。

9.1 $R_{\texttt{While}}$ 语言

$R_{\texttt{While}}$ 的具体语法定义在图 9.1 中，抽象语法定义在图 9.2 中。$R_{\texttt{While}}$ 的定义解释器如图 9.3 所示。我们添加 `SetBang` 、`WhileLoop` 和 `Begin` 的三个新情况，并对变量 `Var` 、`Let` 和 `Apply` 的情况进行更改。为了支持对

```
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)
           | (set! var exp) | (begin exp ... exp) | (while exp exp)
def      ::= (define (var [var:type] ...) : type exp)
R_While  ::= def ... exp
```

图 9.1: $R_{\texttt{While}}$ 的具体语法，扩展 $R_{\texttt{Any}}$ (图 12.1)。

```
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)
           | (SetBang var exp) | (Begin (exp ...) exp) | (WhileLoop exp exp)
def      ::= (Def var ([var:type] ...) type '() exp)
R_While  ::= (ProgramDefsExp '() (def ...) exp)
```

图 9.2: $R_{\texttt{While}}$ 的抽象语法，扩展 $R_{\texttt{Any}}$ (图 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))
```

图 9.3: $R_{\texttt{While}}$ 解释器。

变量的赋值并使它们的生存期不确定 (见 9.2 节中的第二个例子)，将绑定到每个变量 (在 `Let` 中) 的值和函数参数 (在 `Apply` 中) 的值框起来。`Var` 的情况取消该值的框。现在来讨论一下新的案例。对于 `SetBang` ，在环境中查找变量以获得装箱值，然后使用 `set-box!` 将其更改为右侧求值的结果。`SetBang` 的结果值是 `void` 。对于 `WhileLoop` ，我们重复：1) 计算条件，如果结果为真；2) 对主体求值。`while` 循环的结果值也是 `void` 。最后，(`Begin` *es body*) 表达式计算子表达式 *es* 的效果，然后计算并返回 *body* 的结果。

$R_{\texttt{While}}$ 的类型检查器在图 9.4 中定义。对于 `SetBang` ，变量的类型和

右侧必须一致。结果类型为 Void 。对于 WhileLoop ，条件必须是 Boolean 。结果类型也是 Void 。对于 Begin ，结果类型是其最后一个子表达式的类型。

乍一看，将这些语言特性转换到 x86 似乎很简单，因为 $C_{\texttt{Fun}}$ 中间语言已经支持我们需要的所有元素：赋值、goto 、条件分支和排序。但是，出现了两个复杂问题，将在下两个部分中讨论。之后，引入一个新的编译器传递，并对现有传递进行必要的更改。

9.2 赋值函数和词法作用域函数

添加赋值给实现词汇作用域函数的方法带来一个问题。考虑下面的例子，在这个例子中，函数 f 有一个自由变量 x ，它在创建 f 之后，但在调用 f 之前发生改变。

```
(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))))))
```

本例的正确输出是 42 ，因为调用 f 需要使用 x 的当前值 (即 10) 。不幸的是，闭包转换通道 (第 7.3 节) 为 lambda 生成将 x 的旧值复制到闭包的代码。因此，如果简单地添加对当前编译器赋值的支持，这个程序的输出将是 32。

解决这个问题的第一个尝试是在闭包中保存一个指向 x 的指针，并将 x 在 lambda 中的出现更改为对该指针的解引用。当然，这需要将 x 赋值给堆栈，而不是寄存器。然而，问题更严重一些。考虑下面的例子，在这个例子中，通过创建一对共享自由变量 x 的函数来创建一个反抽象。

```
(define (f [x : Integer]) : (Vector ( -> Integer) ( -> Void))
  (vector
```

```
(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))
```

图 9.4: 在 $R_{\texttt{While}}$ 中键入检查 SetBang 、WhileLoop 和 Begin 的类型。

```
  (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)))))
```

在这个例子中，x 的生命周期超过调用 f 的生命周期。因此，如果要在堆栈帧中存储 x 用于调用 f ，那么当调用 inc 和 get 时，x 就会消失，只留下 x 的悬空指针。这个例子演示了当变量在 lambda 中自由出现时，它的生存期变得不确定。因此，变量的值需要存在于堆上。动词“box”通常用于在堆上分配单个值，产生一个指针，而“unbox”用于对指针解引用。

建议通过“boxing”在交集的局部变量来解决这些问题: 1) 出现在 set! 左边的变量；2) 在 lambda 中自由出现的变量。将在第 9.4 节中引入一个名为 convert-assignments 的新过程来执行这个转换。但是在深入讨论编译器传递之前，还要讨论另一个问题。

9.3 循环控制流与数据流分析

到目前为止，在 explicate-control 中生成的控制流图保证是无循环的。然而，每个 while 循环在控制流图中引入一个循环。但这有关系吗? 事实上确实如此。回想一下，对于寄存器分配，编译器执行动态分析，以确定哪些变量可以共享相同的寄存器。在第 4.10.1 节中，以逆拓扑顺序分析了控制流图，但拓扑顺序仅对无环图进行定义。

让回到计算前 5 个正整数的和的例子。这是在指令选择之后但在寄存器分配之前的程序。

```
(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
)
```

回想一下，活性分析是反向的，从每个函数的末尾开始。对于这个例子，可以从 `block8` 开始，因为我们知道在结论的开头是什么，就是 `rax` 和 `rsp` 。因此 `block8` 的 live-before 设置为 {`rsp`, `sum1`} 。接下来，可能会尝试分析 `block5` 或 `block7` ，但 `block5` 跳转到 `block7` ，反之亦然，所以我们似乎被卡住了。

走出这一僵局的方法来自于这样一种认识: 可以从一个空的生存后集合开始进行生存分析，以计算一个低于生存前集合的近似。所谓的 *欠近似*，是指该集合只包含真正存在的变量，但可能会遗漏一些变量。接下来，每个块的欠近似可以通过以下方法加以改进: 1) 使用来自其他块的近似的 live-before 集更新每个块的 live-after 集; 2) 再次对每个街区进行动态分析。事实上，通过迭代这个过程，欠近似最终会成为正确的解! 这种迭代分析控制流图的方法适用于许多静态分析问题，称为 *数据流分析* 。它是 Kildall [73] 在华盛顿大学的博士论文中发明的。

将此方法应用于上面的示例。对于每个块，使用空集合作为初始的 live-before 集合。假设 m_0 是从标签名到一组位置 (变量和寄存器) 的映射。

```
mainstart: {}
block5: {}
```

```
block7: {}
block8: {}
```

使用上述 live-before 逼近，确定每个区块的 live-after，然后对每个区块应用 live-after 分析。这就产生 live-before 集合的下一个近似 m_1 。

```
mainstart: {}
block5: {i2}
block7: {i2, sum1}
block8: {rsp, sum1}
```

对于第二轮，`mainstart` 的 live-after 是 `block5` 的 live-before，即 `{i2}` 。因此，`mainstart` 的活性分析计算的是空集合。`block5` 的 live-after 是 `block7` 和 `block8` 的 live- before 集的并集，即 `{i2 , rsp, sum1}`。因此，对 `block5` 的活性分析计算 `{i2 , rsp, sum1}` 。`block7` 的 live-after 是 `block5` (来自上一个迭代) 的 live-before，即 `{i2}`。因此，`block7` 的活性分析仍然是 `{i2, sum1}` 。这些组合在一起产生以下近似 m_2 的 live-before 集合。

```
mainstart: {}
block5: {i2, rsp, sum1}
block7: {i2, sum1}
block8: {rsp, sum1}
```

在前面的迭代中，只有 `block5` 发生了变化，所以可以将注意力限制在 `mainstart` 和 `block7`，这两个跳转到 `block5` 的块。因此，`mainstart` 和 `block7` 的 live-before 集合被更新为包含 `rsp` ，产生如下近似 m_3 。

```
mainstart: {rsp}
block5: {i2, rsp, sum1}
block7: {i2, rsp, sum1}
block8: {rsp, sum1}
```

因为 `block7` 改变，再次分析 `block5` ，但是它的 live-before 集合仍然是 `{ i2, rsp, sum1 }` 。在这一点上，我们的近似收敛了，所以 m_3 是解。

通过关于格上函数的一般定理 Kleene 不动点定理，该迭代过程保证收敛到一个解 [74] 。粗略地说，格是对其元素进行部分排序' $\sqsubseteq$ 、最小元素 $\bot$ (发音为 bottom) 和连接操作符 $\sqcup$ 的集合。[1] 当两个元素被排序为 $m_i \sqsubseteq m_j$ ，这意味着 m_j 包含的信息至少和 m_i 一样多，所以可以认为 m_j 是一个比 m_i 更好或相等的近似值。底部的元素 $\bot$ 表示完全缺乏信息，即最糟糕的近似。连接运算符取两个格元并组合它们的信息，也就是说，它产生这两个格元的最小上界。

数据流分析通常涉及两个格: 一个格表示抽象状态，另一个格聚集控制流图中所有块的抽象状态。对于活性分析，抽象状态是一组位置。将晶格 L 的元素取为位置集合，其顺序为集合包含 ($\subseteq$)，底部为空集合，连接算子为集合并集，从而构成晶格 L 。通过把它的元素从块标签映射到位置集合 (L 的元素) 来形成第二个格 M 。使用 L 的顺序逐点排列映射。因此给定任意两个映射 m_i 和 m_j ， $m_i \sqsubseteq_M m_j$ ，当 $m_i(\ell) \subseteq m_j(\ell)$ 对应程序中的每个块标号 ℓ ' 时。M 的底部元素是将每个标号发送到空集合的映射 $\bot_M$ ，即 $\bot_M(\ell) = \emptyset$ 。

可以把活性分析的一次迭代看成是晶格 M 上的函数 f 。它将一个映射作为输入，并计算一个新的映射。

$$f(m_i) = m_{i+1}$$

接下来，思考一下 m_s 的最终解决方案应该是什么样的。如果使用 m_s 作为输入进行活性分析，应该再次得到 m_s 作为输出。也就是说，解应该是函数 f 的一个 *不动点*。

$$f(m_s) = m_s$$

此外，该解决方案应该只包括通过对程序执行活性分析而强制出现的位置，因此该解决方案应该是 *最小固定点*。

Kleene 不动点定理指出，如果一个函数 f 是单调的 (更好的输入产生更好的输出)，那么 f 的最小不动点就是从 $\bot$ 开始，按照下面的方法迭代 f

[1] 从技术上讲，我们将使用连接半格。

得到的 上升 *Kleene* 链的最小上界。

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \cdots \sqsubseteq f^n(\bot) \sqsubseteq \cdots$$

当一个晶格只包含有限长的上升链时，每个 Kleene 链在 f 的多次迭代后在某个不动点上达到顶点。所以不动点也是链的最小上界。

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \cdots \sqsubseteq f^k(\bot) = f^{k+1}(\bot) = m_s$$

活性分析确实是一个单调函数，格 M 只有有限长的上升链，因为程序中只有有限数量的变量和块。从而保证对程序中所有块进行活性分析迭代，最终得到最小不动点解。

接下来考虑一般的数据流分析，并讨论通用的工作列表算法 (图 9.5)。该算法有四个参数: 控制流图 `G` ，将分析应用于一个块的函数 `transfer` ，抽象状态格的 `bottom` 和 `join` 操作符。该算法首先创建由散列表表示的底部映射。然后，它将控制流图中的所有节点推入工作列表 (一个队列)。只要工作列表中还有项目，算法就重复 `while` 循环。在每次迭代中，从工作列表中弹出一个节点并进行处理。节点的 `input` 是通过接受所有前任节点的抽象状态的连接来计算的。然后应用 `transfer` 函数来获得 `output` 抽象状态。如果此块的输出与之前的状态不同，则更新此块的映射，并将其后续节点推入工作列表。

讨论由于添加对赋值和循环的支持而引起的两个复杂问题之后，转而讨论一个新的编译器通道以及对现有通道的重大更改。

9.4 转换任务

回想一下,在第 9.2 节中,学习赋值和词法作用域函数的组合要求将那些既被赋值又在 `lambda` 中显示为自由的变量框起来。`convert-assignments` 通道的目的就是执行转换。建议将此通行证放在 `uniquify` 之后，但在 `reveal-functions` 之前。

再次考虑 9.2 节中的第一个例子:

```
(let ([x 0])
```

```
(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)
```

图 9.5: 用于数据流分析的通用工作表算法

```
(let ([y 0])
  (let ([z 20])
    (let ([f (lambda: ([a : Integer]) : Integer (+ a (+ x z)))])
      (begin
        (set! x 10)
        (set! y 12)
        (f y))))))
```

变量 x 和 y 被赋值。变量 x 和 z 在 lambda 中自由出现。因此，变量 x 需要装箱，而不是 y 和 z 。x 的装箱由三个变换组成: 用向量初始化 x ，用 vector-ref 替换从 x 读取的内容，用 vector-set! 替换 x 上的每个 set! 。这个例子的 convert-assignments 输出如下所示。

```
(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 建议定义一个名为 `assigned&free` 的辅助函数，它接受一个表达式并同时进行计算：1) 一组赋值变量 A ；2) 在 Lambda 的范围内自由出现的一组变量 F ；3) 一个新版本的表达式，记录在 A 和 F 的交点上出现的边界变量。可以使用结构 `AssignedFree` 来做这个。考虑 (`Let` x rhs $body$) 的情况。假设对 rhs 的递归调用产生 rhs' 、A_r 和 F_r ，对 $body$ 的递归调用产生 $body'$ 、A_b 和 F_b 。如果 x 在 $A_b \cap F_b$ 中，那么对 `Let` 进行如下变换。

(`Let` x rhs $body$)
⇒
(`Let` (`AssignedFree` x) rhs' $body'$)

如果 x 不在 $A_b \cap F_b$ 中，则省略 `AssignedFree` 。为此 `Let` 分配的变量集为 $A_r \cup (A_b - \{x\})$ ，在 lambda 中自由的变量集为 $F_r \cup (F_b - \{x\})$ 。

(`SetBang` x rhs) 的情况很简单，但很重要。递归处理 rhs 得到 rhs' 、A_r 和 F_r 。结果是 (`SetBang` x rhs'), $\{x\} \cup A_r$ 和 F_r 。

(`Lambda` $params$ T $body$) 的情况有点复杂。设 $body'$ 、A_b 和 F_b 为递归处理 $body$ 的结果。将 $A_b \cap F_b$ 中出现的每个参数用 `AssignedFree` 包装以产生 $params'$ 。设 P 是 $params$ 中的参数名集合。结果是 (`Lambda` $params'$ T $body'$), $A_b - P$ 和 $(F_b \cup \mathrm{FV}(body)) - P$ ，其中 FV 计算表达式的自由变量 (见第 7 章)。

转换任务 接下来，将讨论 convert-assignment 通道及其用于表达式和定义的辅助函数。表达式的函数 cnvt-assign-exp 应该接受一个表达式和一组赋值和自由变量 (从 assigned&free 的结果中获得)。在 (Var x) 的情况下，如果 x 是赋值和自由，那么通过将 (Var x) 转换为 vector-ref 来取消框。

```
(Var x)
⇒
(Prim 'vector-ref (list (Var x) (Int 0)))
```

对于 (Let (AssignedFree x) rhs $body$)，递归处理 rhs 得到 rhs'。接下来，递归地处理 $body$ 以获得 $body'$，但是将 x 添加到赋值和自由变量集合中。将 let 表达式按照如下方式进行转换，将 x 绑定到一个已装箱的值。

```
(Let (AssignedFree x) rhs body)
⇒
(Let x (Prim 'vector (list rhs')) body')
```

在 (SetBang x rhs) 的情况下，递归地处理 rhs 以获得 rhs'。如果 x 在赋值和自由变量中，将 set! 转换为 vector-set!，如下所示。

```
(SetBang x rhs)
⇒
(Prim 'vector-set! (list (Var x) (Int 0) rhs'))
```

Lambda 的情况并不简单，但它与我们接下来讨论的函数定义的情况类似。

定义的辅助函数 cnvt-assign-def 将赋值转换应用于函数定义。将函数定义翻译如下。

```
(Def f params T info body1)
⇒
(Def f params' T info body4)
```

因此，仍然需要解释 $params'$ 和 $body_4$。让 $body_2$、A_b 和 F_b 是 assigned&free 在 $body_1$ 上的结果。设 P 为 $params$ 中的参数名。然后应用 cnvt-assign-exp 到 $body_2$ 获得 $body_3$，传递 $A_b \cap F_b \cap P$ 作为赋值和自由变量集。最后，通过将 $body_3$ 包装在一个序列的 let 表达式中，这个表达式参数绑定在 $A_b \cap F_b$，来获得 $body_4$。关于 $params'$，更改 $A_b \cap F_b$ 中的参数名称以保持唯一性

$$
\begin{array}{rcl}
\mathit{atm} & ::= & \texttt{(Int}\ \mathit{int}\texttt{)} \mid \texttt{(Var}\ \mathit{var}\texttt{)} \mid \texttt{(Bool}\ \mathit{bool}\texttt{)} \mid \texttt{(Void)} \\
\mathit{exp} & ::= & \ldots \mid \texttt{(Let}\ \mathit{var}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
 & \mid & \texttt{(WhileLoop}\ \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(SetBang}\ \mathit{var}\ \mathit{exp}\texttt{)} \mid \texttt{(Begin (}\mathit{exp}\ldots\texttt{)}\ \mathit{exp}\texttt{)} \\
\mathit{def} & ::= & \texttt{(Def}\ \mathit{var}\ \texttt{([}\mathit{var}\texttt{:}\mathit{type}\texttt{]}\ldots\texttt{)}\ \mathit{type}\ \texttt{'()}\ \mathit{exp}\texttt{)} \\
R_8^\dagger & ::= & \texttt{(ProgramDefs '()}\ \mathit{def}\texttt{)}
\end{array}
$$

图 9.6: $R_{\texttt{While}}^{\texttt{ANF}}$ 是管理标准形式 (ANF) 中的 $R_{\texttt{While}}$ 。

(这样 let-bound 变量可以保留原来的名称)。回想一下第 9.2 节中涉及反抽象的第二个例子。下面是函数 `f` 的赋值版本的输出。

```
(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 去除复杂的操作数

三种新的语言形式，`while` 、`set!` 和 `begin` 都是复杂表达式，它们的子表达式也允许是复杂的。图 6.8 定义这个通道的输出语言 $R_{\texttt{Fun}}^{\texttt{ANF}}$ 。

通常，当复杂表达式出现在需要是原子性的语法位置时，例如原语操作符的参数，必须引入一个临时变量并将其绑定到复杂表达式。这种方法同样适用于处理新的语言形式。例如，在下面的代码中，有两个 `begin` 表达式作为 `+` 的参数出现。`rco-exp` 的输出如下所示，其中 `begin` 表达式已经绑定到临时变量。回想一下，$R_{\texttt{While}}^{\texttt{ANF}}$ 中的 `let` 表达式被允许在其右侧表达式中有任意表达式，所以将 `begin` 放在那里是没问题的。

$$
\begin{array}{rcl}
\mathit{stmt} & ::= & \texttt{(Assign (Var } \mathit{var}\texttt{) } \mathit{exp}\texttt{)} \mid \texttt{(Collect } \mathit{int}\texttt{)} \\
& \mid & \texttt{(Call } \mathit{atm}\ \texttt{(}\mathit{atm}\ldots\texttt{))} \mid \texttt{(Prim read ())} \\
& \mid & \texttt{(Prim 'vector-set! (list } \mathit{atm}\ \texttt{(Int } \mathit{int}\texttt{) } \mathit{atm}\texttt{))} \\
\mathit{def} & ::= & \texttt{(Def } \mathit{label}\ \texttt{([}\mathit{var}\texttt{:}\mathit{type}\texttt{]}\ldots\texttt{) } \mathit{type}\ \mathit{info}\ \texttt{((}\mathit{label}\ \texttt{.}\ \mathit{tail}\texttt{)}\ldots\texttt{))} \\
C_{\circlearrowleft} & ::= & \texttt{(ProgramDefs } \mathit{info}\ \texttt{(}\mathit{def}\ldots\texttt{))}
\end{array}
$$

图 9.7: $C_{\circlearrowleft}$ 的抽象语法，扩展 $C_{\texttt{Clos}}$ (图 7.9)。

```
(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 说明控制和 $C_{\circlearrowleft}$

回想一下，在 `explicate-control` 通道中，为程序中的每种位置定义一个助手函数。对于整数和变量的 $R_{\texttt{Var}}$ 语言，需要各种位置: 赋值和尾部。$R_{\texttt{If}}$ 的 `if` 表达式引入谓词位置。对于 $R_{\texttt{While}}$ ，`begin` 表达式引入另一种位置: 效果位置。除最后一个子表达式外，`begin` 语句中的子表达式只计算它们的效果。它们的结果值将被丢弃。考虑到这一点，可以生成更好的代码。

`explicate-control` 的输出语言是 $C_{\circlearrowleft}$(图 9.7) ，它几乎等同于 $C_{\texttt{Clos}}$ 。唯一的语法区别是 `Call` 、`vector-set!` 和 `read` 也可以作为语句出现。$C_{\texttt{Clos}}$ 和 $C_{\circlearrowleft}$ 语言的最大区别是什么? 后者的控制流图可能包含循环。

新的辅助函数 `explicate-effect` 接受一个表达式 (在 effect 位置) 和

一个延续块的承诺。该函数返回一个 *tail* 的承诺，其中包括为输入表达式生成的代码，后面跟着延续块。如果表达式明显是纯的，也就是说，从不引起副作用，那么可以删除该表达式，因此结果只是延续块。(`WhileLoop` *cnd body*) 表达式是最有趣的情况。首先，需要一个新的标签 *loop* 作为循环的顶部。用 `goto` 递归处理 *body* （有效位置），以 *loop* 作为延续，产生 *body′* 。接下来，处理 *cnd* (谓词位置)，将 *body′* 作为 then-branch，并将 continuation 块作为 else-branch。结果应该添加到带有 *loop* 标签的控制流图中。整个 `while` 循环的结果是 `goto` 到 *loop* 标签。请注意，只有在确实使用循环时，才应该将循环添加到控制流图中，这可以使用 `delay` 来完成。

用于尾部、赋值和谓词位置的辅助函数需要更新。三种新的语言形式，`while` 、`set!` 和 `begin` ，可以出现在赋值和尾部位置。只有 `begin` 可以出现在谓词位置中; 另外两个结果类型为 `Void` 。

9.7 选择指令

在 `select-instructions` 通道中只需要三个小的添加就可以处理对 $C_{\circlearrowleft}$ 的更改。也就是说，`Call` 、`read` 和 `vector-set!` 现在可以作为独立语句出现，而不是只出现在赋值语句的右侧。代码生成几乎是相同的; 只要去掉将结果移到左边的指令。

9.8 寄存器分配

如第 9.3 节所述，$R_{\texttt{While}}$ 中循环的存在意味着控制流图可能包含循环，这使寄存器分配所需的动态分析变得复杂。

9.8.1 活性分析

建议使用 9.3 节末尾介绍的泛型 `analyze-dataflow` 函数来执行活性分析，替换 `uncover-live-CFG` 中按照拓扑顺序处理基本块的代码 (4.10.1 节)。

`analyze-dataflow` 函数有四个参数。

1. 第一个参数 `G` 应该是一个来自 `racket/graph` 包 (见 3.3 节侧边栏) 的有向图，它表示控制流图。

2. 第二个参数 `transfer` 是一个将活性分析应用于基本块的函数。它需要两个参数: 要分析的块的标签和该块的 live-after 设置。传递函数应该返回活动前设置块。此外，作为一个副作用，它应该用每个指令的活性信息更新块的 *info* 。要实现 `transfer` 函数，你应该能够重用代码，你已经有分析基本块。

3. `analyze-dataflow` 的第三和第四个参数是抽象状态格的 `bottom` 和 `join` ，即位置集合。格的底部是空集 (`set`) ，连接算子是 `set-union` 。

图 9.8 提供了编译 $R_{\texttt{While}}$ 所需的所有通道的概述。

9.9 挑战: 数组

在第 5 章中，学习元组，即元素序列，其长度在编译时确定，并且元组中的每个元素可能有不同的类型 (它们是异构的)。这个挑战也与序列有关，但这一次的长度是在运行时确定的，并且所有元素都具有相同的类型 (它们是同构的)。对于后面的这种序列，我们使用术语 “数组”。

Racket 语言不区分元组和数组，它们都用向量表示。但是，类型化 Racket 区分元组和数组: `Vector` 类型用于元组，`Vectorof` 类型用于数组。图 9.9 定义 $R^{\texttt{Vecof}}_{\texttt{While}}$ 的具体语法，使用 `Vectorof` 类型和 `make-vector` 原语操作符扩展 $R_{\texttt{While}}$ ，用于创建数组，其参数是数组的长度和数组中所有元素的初始值。为元组定义的 `vector-length` 、`vector-ref` 和 `vector-ref!` 操作符被重载以用于数组。还在 $R^{\texttt{Vecof}}_{\texttt{While}}$ 中包含整数乘法，因为它在许多涉及数组的例子中很有用，比如计算两个数组的内积 (图 9.10)。

$R^{\texttt{Vecof}}_{\texttt{While}}$ 的类型检查器在图 9.11 中定义。`make-vector` 的结果类型是 `(Vectorof T)` ，其中 `T` 是初始化表达式的类型。长度表达式必须具有 `Integer` 类型。操作符 `vector-length` 、`vector-ref` 和 `vector-set!` 的类型检查更新以处理向量具有 `Vectorof` 类型的情况。在这些情况下，将操

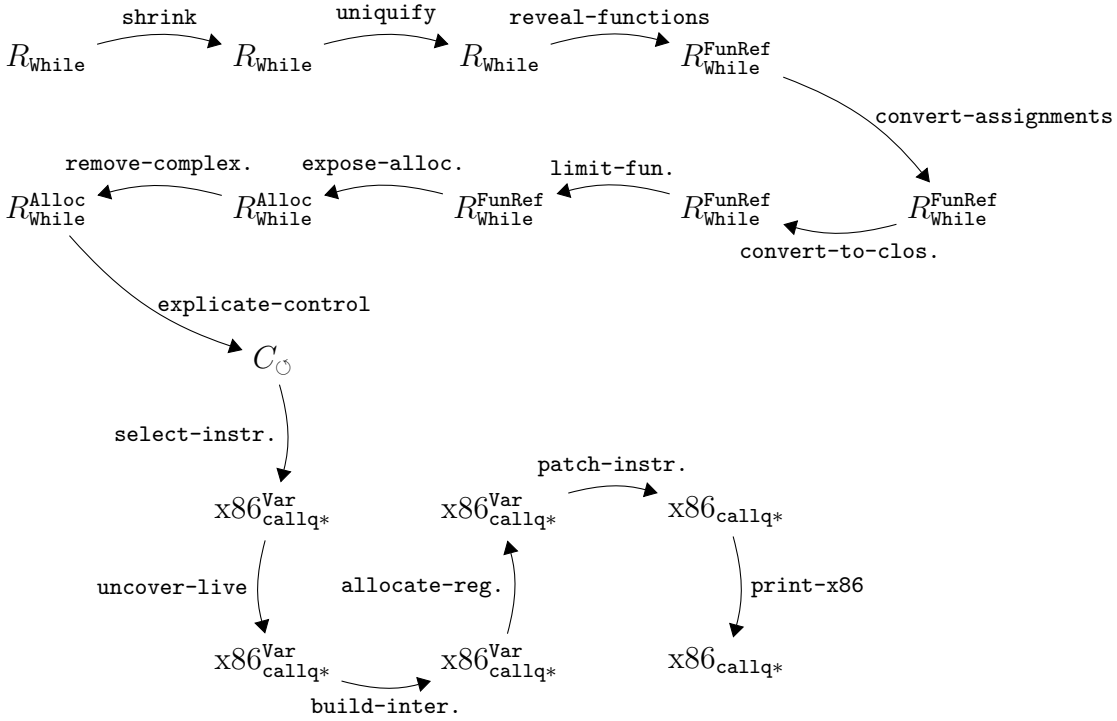


图 9.8: $R_{\texttt{While}}$ 的通道图 (循环和赋值)。

type	::=	... | (Vectorof *type*)
exp	::=	*int* | (read) | (- *exp*) | (+ *exp 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*)
	|	(set! *var exp*) | (begin *exp* ... *exp*) | (while *exp exp*)
	|	(make-vector *exp exp*)
def	::=	(define (*var* [*var*:*type*] ...) : *type exp*)
$R^{\texttt{Vecof}}_{\texttt{While}}$	::=	*def* ... *exp*

图 9.9: $R^{\texttt{Vecof}}_{\texttt{While}}$ 的具体语法，扩展 $R_{\texttt{While}}$ (图 9.1)。

```
(define (inner-product [A : (Vectorof Integer)] [B : (Vectorof Integer)]
                       [n : Integer]) : Integer
  (let ([i 0])
    (let ([prod 0])
      (begin
        (while (< i n)
          (begin
            (set! prod (+ prod (* (vector-ref A i)
                                  (vector-ref B i))))
            (set! i (+ i 1))
            ))
        prod))))


(let ([A (make-vector 2 2)])
  (let ([B (make-vector 2 3)])
    (+ (inner-product A B 2)
       30)))
```

图 9.10: 计算内积的示例程序。

作符转换为 vectorof 形式，以便稍后的通道可以很容易地区分元组和数组上的操作。重写 operator-types 方法以提供用于乘法的类型签名: 它接受两个整数并返回一个整数。为了支持 Any 类型的数组注入和投影 (8.2 节)，还覆盖 flat-ty? 谓词。

$R^{\texttt{Vecof}}_{\texttt{While}}$ 的解释器定义在图 9.12 中。make-vector 运算符是通过 Racket 的 make-vector 函数实现的，乘法是 fx* ，用于 fixnum 整数的乘法。

9.9.1 数据表示法

就像元组一样，将数组存储在堆上，这意味着垃圾收集器将需要检查数组。一个直接的想法是对数组使用与元组相同的表示。但是，将元组的长度限制为 50 ，以便它们的长度和指针掩码可以适合每个元组开始的 64 位标签 (5.2.2 节)。希望数组允许数百万个元素，所以需要更多的位来存储长度。但是，因为数组是同构的，所以指针掩码只需要 1 位，而不是每个数组元素需要 1 位。最后，垃圾收集器需要能够区分元组和数组，因此需要为此保留 1 位。因此，得到数组开头的 64 位标签的如下布局:

- 最右边的位是转发位，就像在元组中一样。0 表示是转发指针，1 表示不是。
- 左边的下一位是指针掩码。0 表示没有一个元素是指向堆的指针，1 表示所有元素都是指针。
- 接下来的 61 位存储数组的长度。
- 最左边的位区分元组 (0) 和数组 (1)。

回想一下在第 8 章中，使用 3 位标记来区分注入到 Any 类型中的值的类型。使用位模式 110 (或十进制的 6) 来表示该值是一个数组。

在下面的小节中，提供有关如何更新传递以处理数组的提示。

9.9.2 显示演员表

数组访问操作符 vectorof-ref 和 vectorof-set! 与第 8 章的 any-vector-ref 和 any-vector-set! 类似,，因为类型检查器不能判断索引是否有边界，所

```
(define type-check-Rvecof-class
  (class type-check-Rwhile-class
    (super-new)
    (inherit check-type-equal?)

    (define/override (flat-ty? ty)
      (match ty
        ['(Vectorof Any) #t]
        [else (super flat-ty? ty)]))

    (define/override (operator-types)
      (append '((* . ((Integer Integer) . Integer)))
              (super operator-types)))

    (define/override (type-check-exp env)
      (lambda (e)
        (define recur (type-check-exp env))
        (match e
          [(Prim 'make-vector (list e1 e2))
           (define-values (e1^ t1) (recur e1))
           (define-values (e2^ elt-type) (recur e2))
           (define vec-type `(Vectorof ,elt-type))
           (values (HasType (Prim 'make-vector (list e1^ e2^)) vec-type)
                   vec-type)]
          [(Prim 'vector-ref (list e1 e2))
           (define-values (e1^ t1) (recur e1))
           (define-values (e2^ t2) (recur e2))
           (match* (t1 t2)
             [(`(Vectorof ,elt-type) 'Integer)
              (values (Prim 'vectorof-ref (list e1^ e2^)) elt-type)]
             [(other wise) ((super type-check-exp env) e)])]
```

```
        [(Prim 'vector-set! (list e1 e2 e3) )
         (define-values (e-vec t-vec) (recur e1))
         (define-values (e2^ t2) (recur e2))
         (define-values (e-arg^ t-arg) (recur e3))
         (match t-vec
           [`(Vectorof ,elt-type)
            (check-type-equal? elt-type t-arg e)
            (values (Prim 'vectorof-set! (list e-vec e2^ e-arg^))
'Void)]
           [else ((super type-check-exp env) e)])]
        [(Prim 'vector-length (list e1))
         (define-values (e1^ t1) (recur e1))
         (match t1
           [`(Vectorof ,t)
            (values (Prim 'vectorof-length (list e1^)) 'Integer)]
           [else ((super type-check-exp env) e)])]
        [else ((super type-check-exp env) e)]))
   ))

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

图 9.11: $R^{\texttt{Vecof}}_{\texttt{While}}$ 语言的类型检查器。

```
(define interp-Rvecof-class
  (class interp-Rwhile-class
    (super-new)

    (define/override (interp-op op)
      (verbose "Rvecof/interp-op" op)
      (match op
        ['make-vector make-vector]
        ['* fx*]
        [else (super interp-op op)]))
    ))

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

图 9.12: $R^{\texttt{Vecof}}_{\texttt{While}}$ 解释器

以必须在运行时执行边界检查。回想一下，`reveal-casts` 通道 (第 8.4 节) 在 vector 引用周围包裹一个 `If` ，以便更新以检查索引是否小于长度。应该对 `vectorof-ref` 和 `vectorof-set!` 做同样的事情。

此外，需要更新 `reveal-casts` 的 `any-vector` 操作符的处理，以考虑注入到 `Any` 中的数组。对于 `any-vector-length` 操作符, 生成的代码应该测试标记是用于元组 (`010`) 还是数组 (`110`) ，然后分派到 `any-vector-length` 或 `any-vectorof-length` 。对于后者，在 `select-instructions` 中添加一个 case，以生成从数组头访问数组长度的适当指令。

`any-vector-ref` 和 `any-vector-set!` 操作符, 件给出代码需要检查指数小于向量的长度, 就像 `any-vector-length` 的代码, 检查标签是否使用 `any-vector-length` 或 `any-vectorof-length` 这一目的。一旦边界检查完成，生成的代码就可以使用 `any-vector-ref` 和 `any-vector-set!` ，对于元组和数组，因为用于这些操作符的指令不查看元组或数组前面的标记。

9.9.3 公开分配

这个过程应该把 make-vector 操作符转换成更低级的操作。特别是，新的 AST 节点 (AllocateArray *exp type*) 分配一个 *exp* 指定长度的数组，但不初始化数组中的元素。(类似于元组的 Allocate AST 节点。) *type* 参数必须是 (Vectorof T)，其中 T 是数组的元素类型。关于数组的初始化，建议生成一个 while 循环，使用 vector-set! 将初始值放入数组的每个元素中。

9.9.4 去除复杂的操作数

在 AllocateArray 的 rco-atom 和 rco-exp 中添加案例。特别是，AllocateArray 节点是复杂的，它的子表达式必须是原子的。

9.9.5 说明控制

为 AllocateArray 添加 explicate-tail 和 explicate-assign 案例。

9.9.6 选择指令

为 AllocateArray 生成与第 5.7 节中的 Allocate 类似的指令，除了在数组开头的标签应使用第 9.9.1 节中讨论的表示形式。

对于 vectorof-length，根据第 9.9.1 节中讨论的表示从标签中提取长度。

vectorof-ref 生成的指令不同于 vector-ref (5.7 节)，因为索引不是一个常量，所以偏移量必须在运行时计算，类似于 any-vector-of-ref (8.7 节) 生成的指令。vectorof-set! 也是如此。此外，vectorof-set! 可能会出现在赋值语句中，并作为单独的语句出现，所以要确保在此传递中处理这两种情况。

最后，any-vectorof-length 的指令应该类似于 vectorof-length 的指令，除了必须首先将数组投射到 3 位标记中

Exercise 37. 通过扩展 $R_{\texttt{While}}$ 的编译器来实现 $R^{\texttt{Vecof}}_{\texttt{While}}$ 语言的编译器。在 6 个新程序上测试编译器，包括图 9.10 中的程序和一个将两个矩阵相乘的程

序。请注意，矩阵是二维数组，但通过排列数组中的每一行，可以将它们编码为一维数组。

10

渐进的打字

这一章研究一种语言，$R_?$ ，程序员可以在程序的不同部分在静态类型检查和动态类型检查之间进行选择，从而将静态类型的 $R_{\texttt{While}}$ 语言和动态类型的 $R_{\texttt{Dyn}}$ 混合在一起。有几种方法可以混合静态和动态类型，包括多语言集成 [109, 81] 和混合类型检查 [43, 56] 。在本章中，将重点关注 *渐进式类型* ，在这种类型中，程序员通过在参数和变量上添加或删除类型注释来控制静态和动态检查的数量 [5, 102]。$R_?$ 的具体语法定义在图 10.1 中，其抽象语法定义在图 10.2 中。$R_{\texttt{While}}$ 和 $R_?$ 之间的主要语法区别是附加的 *param* 和 *ret* 非终结符，使类型注释成为可选的。返回类型在抽象语法中不是可选的; 当返回类型没有在具体语法中指定时，解析器将填充 `Any`。

$R_?$ 的类型检查器和解释器都需要一些有趣的更改来启用渐进类型，在在第 6 章的 `map-vec` 示例的上下文的下两节中讨论这一点。在图 10.3 中，修改 `map-vec` 示例，省略 `add1` 函数的类型注释。

10.1 类型检查 $R_?$ 、强制转换和 $R_{\texttt{cast}}$

$R_?$ 的类型检查器使用 `Any` 类型来缺少参数和返回类型。例如，图 10.3 中 `add1` 的 `x` 参数的类型为 `Any` ，而 `add1` 的返回类型为 `Any` 。接下来考虑 `add1` 中的 `+` 运算符。它希望两个参数都具有 `Integer` 类型，但它的第一个参数 `x` 具有 `Any` 类型。在逐渐类型语言中，只要类型 一致，这种差异是允

$$
\begin{array}{lcl}
\mathit{param} & ::= & \mathit{var} \mid \texttt{[}\mathit{var}\texttt{:}\mathit{type}\texttt{]} \\
\mathit{ret} & ::= & \epsilon \mid \texttt{:}\mathit{type} \\
\mathit{exp} & ::= & \mathit{int} \mid \texttt{(read)} \mid \texttt{(-}\ \mathit{exp}\texttt{)} \mid \texttt{(+}\ \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(-}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
 & \mid & \mathit{var} \mid \texttt{(let ([}\mathit{var}\ \mathit{exp}\texttt{])}\ \mathit{exp}\texttt{)} \\
 & \mid & \texttt{\#t} \mid \texttt{\#f} \mid \texttt{(and}\ \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(or}\ \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(not}\ \mathit{exp}\texttt{)} \\
 & \mid & \texttt{(eq?}\ \mathit{exp}\ \mathit{exp}\texttt{)} \mid \texttt{(if}\ \mathit{exp}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
 & \mid & \texttt{(vector}\ \mathit{exp}\ldots\texttt{)} \mid \texttt{(vector-ref}\ \mathit{exp}\ \mathit{int}\texttt{)} \\
 & \mid & \texttt{(vector-set!}\ \mathit{exp}\ \mathit{int}\ \mathit{exp}\texttt{)} \mid \texttt{(void)} \mid \texttt{(}\mathit{exp}\ \mathit{exp}\ldots\texttt{)} \\
 & \mid & \texttt{(procedure-arity}\ \mathit{exp}\texttt{)} \mid \texttt{(lambda: (}\mathit{param}\ldots\texttt{)}\ \mathit{ret}\ \mathit{exp}\texttt{)} \\
 & \mid & \texttt{(set!}\ \mathit{var}\ \mathit{exp}\texttt{)} \mid \texttt{(begin}\ \mathit{exp}\ldots\ \mathit{exp}\texttt{)} \mid \texttt{(while}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
\mathit{def} & ::= & \texttt{(define (}\mathit{var}\ \mathit{param}\ldots\texttt{)}\ \mathit{ret}\ \mathit{exp}\texttt{)} \\
R_? & ::= & \mathit{def}\ldots\ \mathit{exp}
\end{array}
$$

图 10.1: $R_?$ 的具体语法，扩展 $R_{\texttt{While}}$ (图 9.1)。

$$
\begin{array}{lcl}
\mathit{param} & ::= & \mathit{var} \mid \texttt{[}\mathit{var}\texttt{:}\mathit{type}\texttt{]} \\
\mathit{exp} & ::= & \texttt{(Int}\ \mathit{int}\texttt{)(Var}\ \mathit{var}\texttt{)} \mid \texttt{(Let}\ \mathit{var}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
 & \mid & \texttt{(Prim}\ \mathit{op}\ \texttt{(}\mathit{exp}\ldots\texttt{))} \\
 & \mid & \texttt{(Bool}\ \mathit{bool}\texttt{)} \mid \texttt{(If}\ \mathit{exp}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
 & \mid & \texttt{(Void)} \mid \texttt{(HasType}\ \mathit{exp}\ \mathit{type}\texttt{)} \mid \texttt{(Apply}\ \mathit{exp}\ \mathit{exp}\ldots\texttt{)} \\
 & \mid & \texttt{(Lambda (}\mathit{param}\ldots\texttt{)}\ \mathit{type}\ \mathit{exp}\texttt{)} \\
 & \mid & \texttt{(SetBang}\ \mathit{var}\ \mathit{exp}\texttt{)} \mid \texttt{(Begin (}\mathit{exp}\ldots\texttt{)}\ \mathit{exp}\texttt{)} \\
 & \mid & \texttt{(WhileLoop}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
\mathit{def} & ::= & \texttt{(Def}\ \mathit{var}\ \texttt{(}\mathit{param}\ldots\texttt{)}\ \mathit{type}\ \texttt{'()}\ \mathit{exp}\texttt{)} \\
R_? & ::= & \texttt{(ProgramDefsExp '() (}\mathit{def}\ldots\texttt{)}\ \mathit{exp}\texttt{)}
\end{array}
$$

图 10.2: $R_?$ 的抽象语法，扩展 $R_{\texttt{While}}$ (图 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)
```

图 10.3: map-vec 示例的部分类型版本。

许的，也就是说，除了存在 Any 类型的地方，它们是相等的。Any 类型与其他所有类型一致。图 10.4 定义 consistent? 谓词。

返回到图 10.3 中的 map-vec 示例，add1 函数具有类型 (Any -> Any)，但 map-vec 的参数 f 具有类型 (Integer -> Integer)。$R_?$ 的类型检查器允许这样做，因为这两种类型是一致的。特别地，-> 等于 -> ，因为 Any 与 Integer 一致。

接下来考虑一个带有错误的程序，例如将 map-vec 应用于有时返回布尔值的函数，如图 10.6 所示。$R_?$ 的类型检查器接受此程序，因为 maybe-add1 的类型与 map-vec 的参数 f 的类型一致，即 (Any -> Any) 与 (Integer -> Integer) 一致。有人可能会说，渐进式类型检查器是乐观的，因为它接受可能执行时没有运行时类型错误的程序。不幸的是，当 maybe-add1 函数返回 #t 时，使用输入 1 运行这个程序将触发一个错误。$R_?$ 在运行时执行检查以确保静态类型的完整性，例如 map-vec 的参数 f 上的 (Integer -> Integer) 注释。此运行时检查由类型检查器插入的新 Cast 表单执行。因此，类型检查器的输出是 $R_{\texttt{cast}}$ 语言中的一个程序，它将 Cast 添加到 $R_{\texttt{While}}$ 中，如图 10.5 所示。

图 10.7 显示 map-vec 和 maybe-add1 的类型检查器的输出。其思想是，每当类型检查器看到两个类型一致但不相等时，就插入 Cast 。在 add1 函数中，x 被转换为 Integer ，而 + 的结果被转换为 Any 。在对 map-vec 的调用中，add1 参数从 (Any -> Any) 转换为 (Integer -> Integer) 。

```
(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]))
```

图 10.4: 类型上的一致性谓词。

$$
\begin{array}{rcl}
\mathit{exp} & ::= & \ldots \mid \texttt{(Cast } \mathit{exp}\ \mathit{type}\ \mathit{type}\texttt{)} \\
R_{\texttt{cast}} & ::= & \texttt{(ProgramDefsExp '() (}\mathit{def}\ldots\texttt{)}\ \mathit{exp}\texttt{)}
\end{array}
$$

图 10.5: $R_{\texttt{cast}}$ 的抽象语法，扩展 $R_{\texttt{While}}$ (图 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)
```

图 10.6: 一个带有错误的 `map-vec` 示例的变体。

```
(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)
```

图 10.7: 类型检查 `map-vec` 和 `maybe-add1` 的输出。

$R_?$ 的类型检查器在图 10.8 、10.9 和 10.10 中定义。

```
(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)])]
```

图 10.8: $R_?$ 语言的类型检查器，第 1 部分。

```
[(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)]
```

图 10.9: $R_?$ 语言的类型检查器，第 2 部分。

```
[(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\nin ~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)]
)))
```

图 10.10: $R_?$ 语言的类型检查器，第 3 部分。

```
(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)]))
```

图 10.11: 用于类型检查 $R_?$ 的辅助函数

10.2 解释 $R_{\texttt{cast}}$

一阶强制转换的运行时行为很简单，即涉及简单类型 (如 `Integer` 和 `Boolean`) 的强制转换。例如，从 `Integer` 强制转换为 `Any` ，可以用 $R_{\texttt{Any}}$ 的 `Inject` 操作符完成，它将整型放入一个带标记的值 (图 8.9)。类似地，从 `Any` 到 `Integer` 的强制转换是通过 `Project` 操作符完成的，也就是说，通过检查值的标记，检索底层的整数，或者在标记不是用于整数的时候发出错误信号 (图 8.10)。对于更高阶的类型转换，也就是涉及函数或向量类型的类型转换，事情会变得更有趣。

考虑将函数 `maybe-add1` 从 `(Any -> Any)` 强制转换为 `(Integer -> Integer)` 。当函数在运行时执行此强制转换时，通常无法知道该函数是否总是返回整数。[1] 因此，$R_{\texttt{cast}}$ 解释器将延迟对转换的检查，直到应用了函数。这是通过将 `maybe-add1` 包装在一个新函数中来实现的，该函数将其参数从 `Integer` ' 转换为 `Any` ，应用 `maybe-add1` ，然后将返回值从 `Any` 转换为 `Integer` 。

把注意力转到涉及 vector 类型的强制转换上，考虑图 10.12 中的例子，它定义 `map-vec` 的部分类型版本，其形参 `v` 具有类型 `(Vector Any Any)` ，并在适当位置更新 `v` ，而不是返回一个新的 vector。因此将这个函数命名为 `map-vec!` 。将 `map-vec!` 应用于一个整数向量，因此类型检查器插入从 `(Vector Integer Integer)` 到 `(Vector Any Any)` 的强制转换。$R_{\texttt{cast}}$ t 解释器在 vector 类型之间进行类型转换的一种简单方法是构建一个新的向量，其元素是将每个原始元素转换为适当的目标类型的结果。然而，这种方法仅对不可变向量有效; 向量是可变的。在图 10.12 的例子中，如果强制转换创建一个新的向量，那么 `map-vec!` 内部的更新将发生在新的向量上，而不是原来的向量上。

相反，解释器需要创建一种新的值，一个 *向量代理*，它拦截每一个 vector 操作。在读取时，代理读取底层向量，然后对结果值应用强制转换。在写操作中，代理对参数值进行强制转换，然后对基础向量执行写操作。对于 `map-vec!` 中的第一个 `(vector-ref v 0)` ，代理将 `0` 从 `Integer` 转换为 `Any` 。对于第一个 `vector-set!` ，代理将标记 `1` 从 `Any` 转换为 `Integer`

[1] 预测函数的返回值相当于停止问题，这是无法确定的。

```
(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)))
```

图 10.12: 一个涉及向量强制转换的例子。

。

需要考虑的最后一类类型转换是 Any 类型与函数类型或向量类型之间的类型转换。图 10.13 显示 map-vec! 的一个变体，其中参数 v 没有类型注释，因此给它指定类型 Any 。在对 map-vec! 的调用中，vector 具有类型 (Vector Integer Integer) ，因此类型检查器将 (Vector Integer Integer) 强制转换到 Any 。第一个想法是使用 Inject ，但这不起作用，因为 (Vector Integer Integer) 不是一个平面类型。相反，必须先强制转换为 (Vector Any Any) (它是平面的)，然后注入到 Any 。

$R_{\texttt{cast}}$ 解释器使用一个名为 apply-cast 的辅助函数将一个值从源类型强制转换为目标类型，如图 10.14 所示。发现它处理在本节中讨论的所有类型的强制转换。

$R_{\texttt{cast}}$ 的解释器定义在图 10.15 中，其中 Cast 被分派到 apply-cast 。为了处理向量代理的添加，使用图 10.16 中的函数更新 interp-op 中的向量原语。

10.3 低投入

x86 进程的下一步是 lower-casts 通道，它将 $R_{\texttt{cast}}$ 中的 cast 转换为更低级的 Inject 和 Project 操作符，以及一个用于创建矢量代理的新操作

```
(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)))
```

图 10.13: 将向量强制转换为 Any 。

符，扩展 $R_{\texttt{While}}$ 语言来创建 $R_{\texttt{proxy}}$ 。建议创建一个名为 lower-cast 的辅助函数，它接受一个表达式 (在 $R_{\texttt{cast}}$ 中)、一个源类型和一个目标类型，并将其转换为 $R_{\texttt{proxy}}$ 中的表达式，该表达式具有与在解释器中将表达式从源类型转换为目标类型相同的行为。

lower-cast 函数的代码结构类似于 $R_{\texttt{cast}}$ 解释器中使用的 apply-cast 函数 (图 10.14) ，因为它必须处理与 apply-cast 相同的情况，并且它需要模仿 apply-cast 的行为。最有趣的情况是两种向量类型之间和两种函数类型之间的强制转换。

如第 10.2 节所述，将一种向量类型强制转换为另一种向量类型是通过创建一个代理来拦截对底层向量的操作来完成的。这里我们使用 vector-proxy 原语操作显式创建代理。它有三个参数，第一个是向量的表达式，第二个是函数的向量，用于对从向量中读取的元素进行类型转换，第三个是函数的向量，用于对写入向量的元素进行类型转换。可以使用 Lambda 创建函数。此外，正如在下一节中看到的，需要将这些向量与用户创建的向量区分开来，因此，建议使用一个名为 raw-vector 而不是 vector 的新原始运算符来创建函数的这些向量。图 10.17 显示 lower-casts 对图 10.12 中的示例的输出，该示例涉及将一个整数向量转换为 Any 向量。

通过生成参数和返回类型与目标函数类型匹配的 Lambda 来完成从一个函数类型到另一个函数类型的强制转换。Lambda 的函数体应该将形参从目

```
(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) ())]
))
```

图 10.14: `apply-cast` 辅助方法。

```
(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))
```

图 10.15: $R_{\texttt{cast}}$ 的解释器。

```
(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)]))
```

图 10.16: 被保护向量辅助函数。

```
(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)))
```

图 10.17: 图 10.12 中的示例的 `lower-casts` 的输出。

```
(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)
```

图 10.18: 在图 10.3 的例子中 `lower-casts` 。

标类型转换为源类型 (是的，backwards! 函数在形参中是逆变的)，然后调用底层函数，最后将源返回类型的结果转换为目标返回类型。图 10.18 在图 10.3 的 `map-vec` 示例中显示 `lower-casts` 通道的输出。注意，对 `map-vec` 调用中的 `add1` 参数被包装在 `lambda` 中。

10.4 区分代理

到目前为止，区分向量和向量代理的工作一直是解释器的工作。例如，$R_{\texttt{cast}}$ 的解释器使用 `guarded-vector-ref` 函数实现 `vector-ref` ，如图 10.16 所示。在 `differentiate-proxies` 通道中，将此责任转移到生成的代码。

首先设计输出语言 R_8^p 。在 $R_?$ 中，我们对实向量和向量代理都使用类型 `Vector` 。在 R_8^p 中，返回 `Vector` 类型的原始含义，作为实向量的类型，并且引入一种新的类型，`PVector` ，它的值可以是实向量或向量代理。这个新类型附带一套新的原语操作，用于创建和使用 `PVector` 类型的值。不需要引入新的类型来表示向量代理。代理由包含以下三种内容的向量表示:1) 底层的向量；2) 一个函数的向量类型，用于对从向量中读取的元素进行类型转换；3) 一个函数的向量，用于对写入向量的值进行强制转换。因此，定义以

下矢量代理类型的缩写:

$Proxy(T \ldots \Rightarrow T' \ldots) = (\texttt{Vector}\ (\texttt{PVector}\ T \ldots)\ R\ W) \rightarrow (\texttt{PVector}\ T' \ldots)$

当 $R = (\texttt{Vector}\ (T \rightarrow T') \ldots)$ 和 $W = (\texttt{Vector}\ (T' \rightarrow T) \ldots)$ 。接下来，将描述每个新的原语操作。

inject-vector : $(\texttt{Vector}\ T \ldots) \rightarrow (\texttt{PVector}\ T \ldots)$
将一个向量标记为 PVector 类型的值。

inject-proxy : $Proxy(T \ldots \Rightarrow T' \ldots) \rightarrow (\texttt{PVector}\ T' \ldots)$
将矢量代理标记为 PVector 类型的值。

proxy? : $(\texttt{PVector}\ T \ldots) \rightarrow \texttt{Boolean}$
如果值是向量代理则返回 true，如果是实向量则返回 false。

project-vector : $(\texttt{PVector}\ T \ldots) \rightarrow (\texttt{Vector}\ T \ldots)$
假设输入是一个向量 (而不是一个代理)，将返回向量。

proxy-vector-length : $(\texttt{PVector}\ T \ldots) \rightarrow \texttt{Boolean}$
给定一个向量代理，将返回基础向量的长度。

proxy-vector-ref : $(\texttt{PVector}\ T \ldots) \rightarrow (i : \texttt{Integer}) \rightarrow T_i$
给定一个向量代理，将返回基础向量的第 i 个元素。

proxy-vector-set! : $(\texttt{PVector}\ T \ldots) \rightarrow (i : \texttt{Integer}) \rightarrow T_i \rightarrow \texttt{Void}$
给定一个向量代理，将一个值写入基础向量的第 i 个元素。

现在来讨论区分向量和代理的转换。首先，程序中的每个类型注释都必须 (递归地) 转换为用 PVector 替换 Vector 。接下来，必须在适当的位置插入 PVector 操作的用法。例如，用一个 inject-vector 来包装每个向量的创建。

```
(vector e_1 ... e_n)
⇒
(inject-vector (vector e'_1 ... e'_n))
```

在前一节中介绍的 `raw-vector` 操作符没有被注入。

```
(raw-vector e1 ... en)
⇒
(vector e1' ... en')
```

`vector-proxy` 原语的翻译如下。

```
(vector-proxy e1 e2 e3)
⇒
(inject-proxy (vector e1' e2' e3'))
```

将向量操作转换为条件表达式，检查值是否为代理，然后将其分派到适当的代理向量操作或常规向量操作。例如，下面是 `vector-ref` 的转换。

```
(vector-ref e1 i)
⇒
(let ([v e1])
  (if (proxy? v)
    (proxy-vector-ref v i)
    (vector-ref (project-vector v) i)
```

注意在实向量的情况下，必须在 `vector-ref` 之前应用 `project-vector` 。

10.5 真实成本

回想一下，`reveal-casts` 通道 (第 8.4) 节) 负责将 `Inject` 和 `Project` 降低到较低级别的操作中。特别是，`Project` 变成一个检查标记并检索基础值的条件表达式。这里，需要增加 `Project` 的翻译，以处理目标类型为 `PVector` 的情况。需要使用 `proxy-vector-length` 而不是 `vector-length` 。

```
(project e (PVector Any1 ... Anyn))
⇒
(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 关闭转换

关闭转换通过只需要一个小的调整。需要更新转换类型注释的辅助函数来处理 `PVector` 类型。

10.7 说明控制

更新 `explicate-control` 通道来处理 `PVector` 类型上的新基元操作。

10.8 选择指令

回想一下，`select-instructions` 通道负责将原始操作降低为 x86 指令。所以需要将新的 `PVector` 操作转换为 x86。为此，需要回答的第一个问题是如何区分可以驻留在 `PVector` 中的两种值 (向量和代理)。只需要一个位来完成这一点，并在每个向量的前面使用 64 位标记的 57 位 (见图 5.9) 。到目前为止，这一位已经被设置为 0 ，所以对于 `inject-vector` ，我们保持这种方式。

(Assign lhs (Prim 'inject-vector (**list** e_1)))
⇒
movq e_1', lhs'

另一方面，`inject-proxy` 将 57 位设为 1 。

(Assign lhs (Prim 'inject-proxy (**list** e_1)))
⇒
movq e_1', %r11
movq $(1 << 57)$, %rax
orq 0(%r11), %rax
movq %rax, 0(%r11)
movq %r11, lhs'

`proxy?` 操作将使用 `inject-vector` 和 `inject-proxy` 小心翼翼地隐藏起来的信息。它隔离第 57 位，以判断该值是实向量还是代理。

```
(Assign lhs (Prim 'proxy? (list e)))
⇒
movq e'1, %r11
movq 0(%r11), %rax
sarq $57, %rax
andq $1, %rax
movq %rax, lhs'
```

`project-vector` 操作翻译起来很简单，所以把它留给读者。

关于 `proxy-vector` 操作，运行时提供实现它们的过程 (它们是递归函数!)，所以这里只需要将这些向量操作转换为适当的函数调用。例如，下面是 `proxy-vector-ref` 的翻译。

```
(Assign lhs (Prim 'proxy-vector-ref (list e1 e2)))
⇒
movq e'1, %rdi
movq e'2, %rsi
callq proxy_vector_ref
movq %rax, lhs'
```

还有另一批向量操作要处理，那些针对 `Any` 类型的。回想一下，当 `Any` 类型的东西上有 `vector-ref` 时，$R_?$ 类型检查器会生成 `any-vector-ref`，`any-vector-set!` 和 `any-vector-length` 的情况也类似 (图 10.8) 。在第 8.7 节中，基于底层值是一个实向量的思想为这些操作选择指令。但是在当前设置中，基础值的类型是 `PVector` 。所以 `any-vector-ref` 可以被翻译成如下的伪 x86。首先从标记值中投影底层值，然后在运行时调用 `proxy_vector_ref` 过程。

```
(Assign lhs (Prim 'any-vector-ref (list e1 e2)))
movq ¬111, %rdi
andq e'1, %rdi
movq e'2, %rsi
callq proxy_vector_ref
movq %rax, lhs'
```

`any-vector-set!` 和 `any-vector-length` 运算符可以用类似的方式进行转换。

Exercise 38. 通过为 $R_{\texttt{While}}$ 扩展和调整编译器，实现一个渐进类型的 $R_?$ 语言的编译器。创建 10 个新的部分类型测试程序。除了测试这些新程序之外，还要在所有 $R_{\texttt{While}}$ 测试和 $R_{\texttt{Dyn}}$ 测试上测试编译器。有时在 $R_{\texttt{Dyn}}$ 程序中可能会出现类型检查错误，但是您可以通过在每个子表达式周围的 `Any` 类型中插入强制转换来调整它们，从而导致类型错误。虽然 $R_{\texttt{Dyn}}$ 没有显式强制转换，但可以通过调用一个未注释的身份函数来包装子表达式 `e` 来诱导一次强制转换，例如: `((lambda (x) x) e)` 。

图 10.19 提供编译 $R_?$ 所需的所有通道的概览。

10.9 延伸阅读

这一章只是渐进式打字的皮毛。这里描述的基本方法缺少渐进类型实现中需要的两个关键要素: 指责跟踪 [109, 111] 和节省空间的类型转换 [60, 61] 。指责跟踪解决的问题是，当对高阶值进行强制转换失败时，它通常会在程序中远离原始强制转换的某个点上这样做。过失跟踪是一种通过强制转换和代理传播额外信息的技术，以便在强制转换失败时，错误消息可以指向源程序中强制转换的原始位置。

空间效率高的强制转换所解决的问题也与高阶强制转换有关。事实证明，在部分类型程序中，函数或向量可以在运行时进行很多类型转换。使用本章描述的方法，每次强制转换都会添加另一个 `lambda` 包装器或向量代理。这不仅会占用相当大的空间，而且会使函数调用和向量操作变慢。例如，在最坏的情况下，一个部分类型的快速排序版本可以围绕向量建立一个长度为 $O(n)$ 的代理链，将算法的总时间复杂度从 $O(n^2)$ 改变为 $O(n^3)$ ！ Herman et al. [60] 提出了一个解决这个问题的方法，即使用 Henglein [59] 的强制演算来表示类型强制转换，这种方法通过将它们压缩成简洁的标准形式来防止创建长代理链。Siek et al. [103] 给出了压缩强压的算法，Kuhlenschmidt et al. [76] 演示如何在 Grift 编译器中实现这些想法。

`https://github.com/Gradual-Typing/Grift`

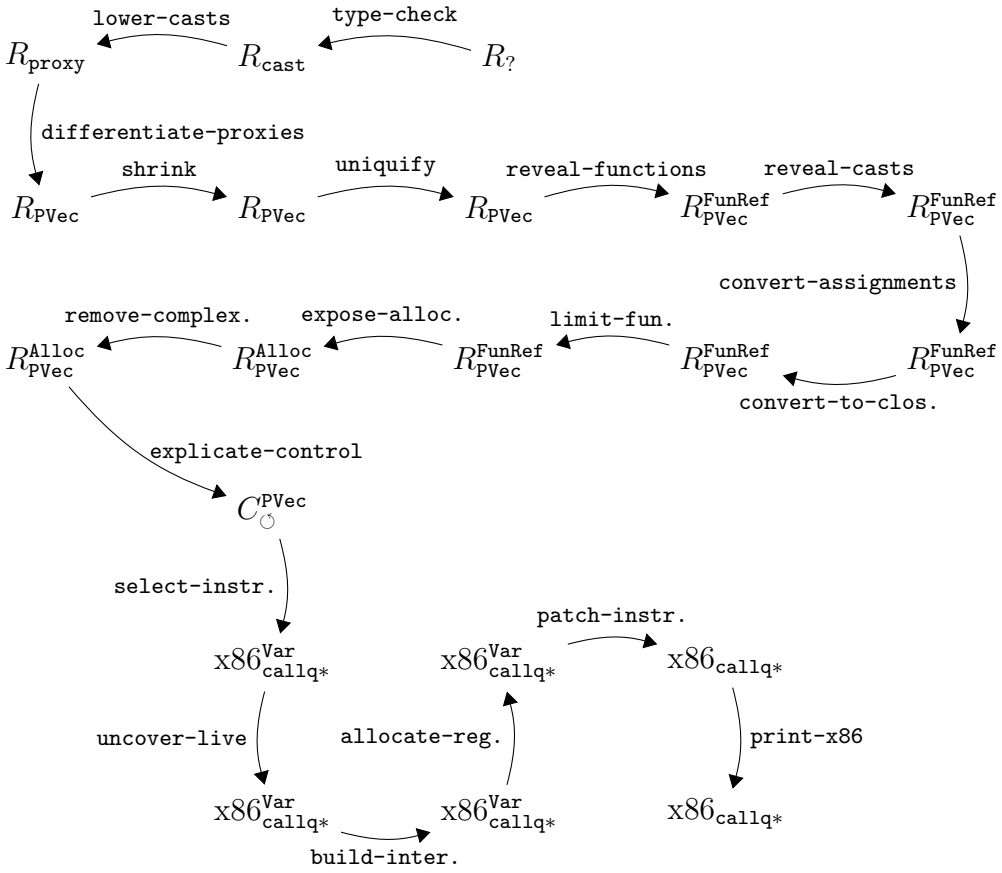


图 10.19: $R_{?}$ (渐进打字) 的通道图。

在渐进类型和其他语言特性 (如参数多态、信息流类型和类型推断等) 之间还有一些有趣的交互。我们向读者推荐在线渐进式打字参考书目:

http://samth.github.io/gradual-typing-bib/

11

参数多态性

本章研究类型化 Racket 的子集 $R_{\texttt{Poly}}$ 中参数多态性（也称为泛型）的编译。参数化多态通过参数化函数和数据结构所操作的类型来改进代码重用。例如，图 11.1 重新访问 `map-vec` 示例，但这次给出一个更适合的类型。这个 `map-vec` 函数是根据向量的元素类型参数化的。`map-vec` 的类型是以下由 `All` 和类型参数 `a` 指定的多态类型。

```
(All (a) ((a -> a) (Vector a a) -> (Vector a a)))
```

这个想法是，`map-vec` 可以用于参数 `a` 的类型的 *all* 选择。在图 11.1 中，将 `map-vec` 应用于一个整数向量，选择 `Integer` 表示 `a`，但也可以将 `map-vec` 应用于一个布尔向量 (和一个布尔函数)。

```
(: 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)
```

图 11.1: 使用参数多态性的 `map-vec` 示例。

```
type     ::= ... | (All (var ...) type) | var
def      ::= (define (var [var:type] ...) : type exp)
           | (: var type)
             (define (var var ...) exp)
R_Poly   ::= def ... exp
```

图 11.2: $R_{\texttt{Poly}}$ 的具体语法，扩展 $R_{\texttt{While}}$ (图 9.1)。

```
type     ::= ... | (All (var ...) type) | var
def      ::= (Def var ([var:type] ...) type '() exp)
           | (Decl var type)
             (Def var (var ...) 'Any '() exp)
R_Poly   ::= (ProgramDefsExp '() (def ...) exp)
```

图 11.3: $R_{\texttt{Poly}}$ 抽象语法，扩展 $R_{\texttt{While}}$ (图 9.2).

图 11.2 定义 $R_{\texttt{Poly}}$ 的具体语法，图 11.3 定义抽象语法。为函数定义添加第二种形式，其中类型声明出现在 `define` 之前。在抽象语法中，`Def` 中的返回类型是 `Any`，但是为了在类型声明中使用返回类型，应该忽略它。(`Any` 来自于使用与第 8 章相同的解析器。) 类型声明的存在使函数可以使用 `All` 类型，从而使其具有多态性。对类型语法进行扩展，以包括多态类型和类型变量。

通过在 *type* 非终结符中包含多态类型，选择使它们成为头等类型，这对编译器有有趣的影响。许多具有多态性的语言，比如 C++ [107] 和 Standard ML [87] ，只支持二级多态性，所以看看一级多态性的例子是很有用的。在图 11.4 中，定义一个函数 `apply-twice` ，它的参数是一个多态函数。在函数类型下面出现多态类型是通过 *type* 语法的普通递归结构和 `All` 类型分类为 *type* 来实现的。`apply-twice` 的函数体将多态函数应用于布尔值和整数。

图 11.7 中 $R_{\texttt{Poly}}$ 的类型检查器有三个新职责 (与 $R_{\texttt{While}}$ 相比)。扩展函数应用程序的类型检查，以处理操作符表达式是多态函数的情况。在这种情况下，通过将形参的类型与实参的类型匹配来推导类型实参。`match-types` 辅助函数通过递归降序通过形参类型 `pt` 和相应的实参类型 `at` 来执行这一

```
(: 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)
```

图 11.4: 说明一级多态性的示例。

推断，确保它们是相等的，除非左边有类型形参 (在形参类型中)。如果是第一次遇到类型形参，那么算法将推断出类型形参与右边对应的类型的关联 (在实参类型中)。如果不是第一次遇到类型参数，算法将查找其推导出的类型，并确保它与右边的类型相等。一旦推导出类型参数，操作符表达式就被包装在一个 `Inst` AST 节点中 (用于实例化)，该节点记录操作符的类型，但更重要的是，记录推导出的类型参数。应用程序的返回类型是多态函数的返回类型，但是使用 `subst-type` 函数将类型参数替换为推导的类型参数。

类型检查器的第二个职责是扩展 `type-equal?` 函数来处理 `All` 类型。对于其他类型 (如函数和向量类型)，这就不那么简单，因为尽管两个多态类型是等价类型，但它们在语法上可能是不同的。例如，`(All (a) (a -> a))` 等价于 `(All (b) (b -> b))`。如果两个多态类型仅在类型参数名称的选择上不同，则应该认为它们是相等的。图 11.8 中的 `type-equal?` 函数重命名第一个类型的类型形参，以匹配第二个类型的类型形参。

类型检查器的第三个职责是确保只有已定义的类型变量才出现在类型注释中。图 11.9 中定义的 `check-well-formed` 函数递归地检查一个类型，确保每个类型变量都已定义。

类型检查器的输出语言是 $R_{\texttt{Inst}}$，在图 11.5 中定义。类型检查器使用 `Poly` 形式将类型声明和多态函数组合成一个单独的定义，以使多态函数在编译器的下一次传递中更方便地处理。

图 11.6 中列出多态 `map-vec` 示例上的类型检查器的输出

type	::=	... | (All (*var* ...) *type*) | *var*
exp	::=	... | (Inst *exp type* (*type* ...))
def	::=	(Def *var* ([*var*:*type*] ...) *type* '() *exp*)
	|	(Poly (*var* ...) (Def *var* ([*var*:*type*] ...) *type* '() *exp*))
$R_{\texttt{Inst}}$	::=	(ProgramDefsExp '() (*def* ...) *exp*)

图 11.5: $R_{\texttt{Inst}}$ 的抽象语法扩展 $R_{\texttt{While}}$ (图 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)
```

图 11.6: `map-vec` 示例上的类型检查器的输出。

```
(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^)]))
))
```

图 11.7: $R_{\texttt{Poly}}$ 语言的类型检查器。

```
(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^))]))
```

图 11.8: 用于类型检查 $R_{\texttt{Poly}}$ 的辅助功能。

```
(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)]))
```

图 11.9: 格式良好的类型。

11.1 编译多态性

广义地说，有四种编译参数多态性的方法，在下面描述它们。

单形式化 为使用它的每一组类型参数生成多态函数的不同版本，产生类型专门化的代码。这种方法可以产生最有效的代码，但需要整个程序编译 (不需要单独编译)，并增加代码大小。就目前的目的而言，单一化是行不通的，因为在一级多态性中，有时无法确定在编译期间使用哪些泛型函数与哪些类型参数。(它可以在运行时通过实时编译完成。) 该方法用于在 NESL [14] 和 ML [112] 中编译 C++ 模板 [107] 和多态函数。

统一表示 为每个多态函数生成一个版本，但要求所有值都具有公共的 "装箱" 格式，例如 $R_{\texttt{Any}}$ 中的 `Any` 类型标记值。非多态代码 (即单态代码) 的编译方式类似于动态类型语言 (如 $R_{\texttt{Dyn}}$) 中的代码，在这种语言中，原语操作符要求它们的参数从 `Any` 中投射出来，并将它们的结果注入 `Any` 中。(在面向对象的语言中，投影是通过虚拟方法调度来实现的。) 统一表示方法与单独编译和一流多态性兼容。但是，它产生的代码效率最低，因为它在整个程序中引入开销，包括非多态代码。这种方法在 CLU [80, 79] 、ML [21, 7] 和 Java [15] 的实现中使用。

混合表示 使用类型变量的装箱表示生成每个多态函数的一个版本。单态代码通常被编译 (如在 $R_{\texttt{While}}$ 中)，在单态和多态之间的边界执行转换 (例如，当一个多态函数被实例化和调用时)。这种方法与单独编译和一级多态性兼容，并保持了单态代码的效率。在单态代码和多态代码之间的边界增加了开销。这种方法在 ML [77] 和 Java 的实现中使用，从 Java 5 开始，添加自动装箱功能。

类型传递 在单态和多态代码中都使用未装箱表示。每个多态函数被编译为一个带有描述类型参数的额外参数的单一函数。生成的代码使用类型信息来解如何在运行时访问未装箱的值。在 Napier88 语言 [90] 和 ML [57] 的实现中使用了这种方法。类型传递与单独编译和一级多态

```
(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)
```

图 11.10: 类型擦除后的多态 `map-vec` 示例。

性兼容，并保持了单态代码的效率。在多态代码中，由于对类型信息进行调度，会产生运行时开销。

在本章中，使用混合表示方法，部分原因是它具有良好的属性，部分原因是它可以直接使用已经构建的支持渐进类型的工具来实现。为了编译多态函数，只添加一个新通道，`erase-types` ，将 $R_{\texttt{Inst}}$ 编译为 $R_{\texttt{cast}}$ 。

11.2 擦除类型

使用第 8 章中的 `Any` 类型来表示类型变量。例如，图 11.10 显示 `erase-types` 通道到多态 `map-vec` 的输出 (图 11.1) 。类型参数 `a` 的出现被 `Any` 代替，多态的 `All` 类型从 `map-vec` 类型中删除。

这种类型擦除过程在实例化点上产生挑战。例如，考虑图 11.6 中 `map-vec` 的实例化。`map-vec` 的类型是

```
(All (a) ((a -> a) (Vector a a) -> (Vector a a)))
```

它被实例化为

```
((Integer -> Integer) (Vector Integer Integer)
  -> (Vector Integer Integer))
```

擦除后，map-vec 的类型是

```
((Any -> Any) (Vector Any Any) -> (Vector Any Any))
```

但是需要将其转换为实例化的类型。这在目标语言 $R_{\texttt{cast}}$ 中很容易做到，只使用一个 cast 。在图 11.10 中，map-vec 的实例化已经被编译为 map-vec 的类型到实例化类型的 cast 。强制转换的源类型和目标类型必须一致（图 10.4），确实如此，因为源和目标都是从相同的 map-vec 的多态类型获得的，将类型参数替换为前者中的 Any 以及稍后推导的 type 参数。(回想一下，Any 类型与任何类型一致。)

为了实现 erase-types 通道，建议定义一个名为 erase-type 的递归辅助函数，该函数应用以下两个转换。它将类型变量替换为 Any

x
⇒
Any

并且删除多态的 All 类型。

(All xs T_1)
⇒
T_1'

对程序中的所有类型注释应用 erase-type 函数。

在表达的翻译方面，Inst 的情况是一个有趣的例子。将其转换为 Cast ，如下所示。子表达式 e 的类型是多态类型 (AllxsT) 。转换的源类型是 T 的擦除，即类型 T' 。目标类型 T'' 是将实参类型 ts 替换为 T 中的类型形参 xs ，然后进行类型擦除的结果。

(Inst e (All xs T) ts)
⇒
(Cast e' T' T'')

其中，T'' = (erase-type (subst-type s T)) ，s = (map cons xs ts) 。

最后，将每个多态函数翻译为一个常规函数，其中类型擦除已应用于所有类型注释和主体。

(Poly ts (Def f ([x_1 : T_1] ...) T_r *info* e))

⇒
`(Def` f `([`x_1 `:` T_1'`]` ...`)` T_r' *info* e'`)`

Exercise 39. 为多态语言 $R_{\texttt{Poly}}$ 实现一个编译器，通过扩展和适应你的编译器为 $R_?$ 。创建 6 个使用多态函数的新测试程序。其中一些应该利用一级多态性。

图 11.11 提供编译 $R_{\texttt{Poly}}$ 所需的所有通道的概览。

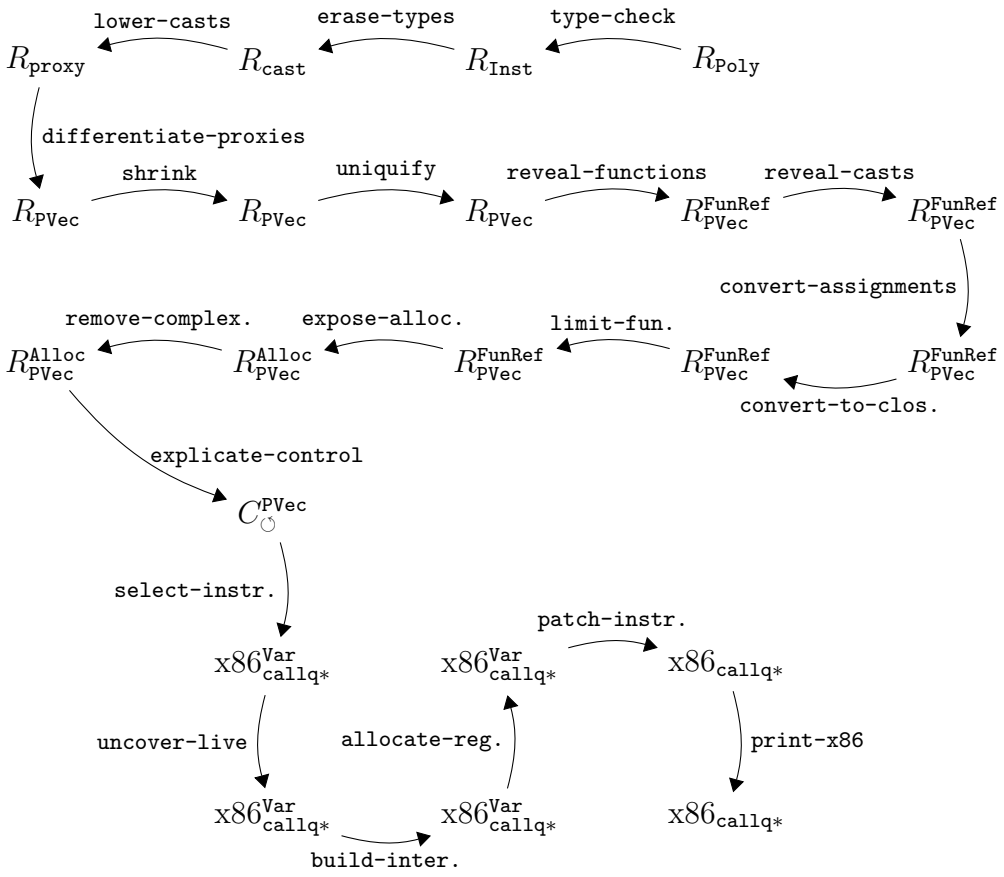


图 11.11: $R_{\texttt{Poly}}$ (参数多态) 通道图。

12

附录

12.1 解释器

在 interp-Rint.rkt,interp-Rvar.rkt 等文件中为每一种源语言 $R_{\texttt{Int}}$ 、$R_{\texttt{Var}}$ 、... 提供解释器。中间语言 $C_{\texttt{Var}}$ 和 $C_{\texttt{If}}$ 的解释器是 interp-Cvar.rkt 和 interp-C1.rkt 文件中。$C_{\texttt{Vec}}$ 、$C_{\texttt{Fun}}$ 、pseudo-x86 and x86 的解释器在 interp.rkt 文件中。

12.2 公用功能

本节中描述的实用程序函数位于支持代码的 utilities.rkt 文件中。

interp-tests interp-tests 函数对每个指定的测试运行编译器通过和解释器，以检查每次通过是否正确。interp-tests 功能有以下参数:

name (a string) 标识编译器的名称;

typechecker 只有一个参数的函数，在遇到类型错误时使用 error 函数引发错误，或在遇到类型错误时返回 #f 。如果没有类型错误，类型检查器将返回该程序。

passes 每次传递一个条目的列表。一个条目是一个包含四项内容的列表:

1. 一个给出通道名称的字符串;
2. 实现通道 (AST 到 AST 的转换程序) 的函数;
3. 为输出语言实现解释器 (从 AST 到结果值的函数) 的函数;
4. 输出语言的类型检查器。支持代码中提供 R 和 C 语言的类型检查器。例如, $R_{\texttt{Var}}$ 和 $C_{\texttt{Var}}$ 的类型检查器在 `type-check-Rvar.rkt` 和 `type-check-Cvar.rkt` 中。类型检查器条目是可选的。支持代码没有为 x86 语言提供类型检查器。

source-interp 源语言的解释器。附录 12.1 中的口译员做出很好的选择。

test-family (a string) 例如，`"r1"`、`"r2"` 等。

tests 指定要运行哪些测试的测试号列表。(见下文)

`interp-tests` 函数假定子目录 `tests` 有一个 Racket 程序集合，这些程序的名称都以家族名开头，后跟下划线和测试号，以文件扩展名 `.rkt` 结尾。此外，对于每个调用 `read` 一次或多次的测试程序，除了文件扩展名为 `.in` 外，还有一个同名的文件，该文件为 Racket 程序提供了输入。如果预期测试程序的类型检查失败, 那么应该有一个同名但扩展名为 `.tyerr` 的空文件。

`compiler-tests` 运行编译器传递生成 x86 (`.s` 文件), 然后运行 GNU C 编译器 (gcc) 生成机器码。它运行机器代码并检查输出是否为 42 。`compiler-tests` 函数的形参类似于 `interp-tests` 函数的形参，由以下几个部分组成

- 编译器名称 (字符串);
- 类型检查器;
- 通道的描述;
- 测试族名称;
- 测试编号的列表。

compile-file 需要编译器传递的描述 (见注释为 `interp-tests`) 并返回一个函数, 给出一个程序文件名称 (字符串以 `.rkt` , 适用于所有的经过, 并将输出写入到一个文件的名字是一样的程序文件名称, 但 `.rkt` 替换为 `.s` 。

read-program 获取一个文件路径，并将该文件 (它必须是一个 Racket 程序) 解析为一个抽象语法树。

parse-program 接受抽象语法树的 s 表达式表示，并将其转换为基于结构的表示。

assert 它接受两个参数，一个字符串 (`msg`) 和一个布尔值 (`bool`) ，如果布尔值为 `msg` ，则显示消息 `bool` 。

lookup 接受一个键和一个列表，并返回与给定键关联的第一个值 (如果有的话)。如果不是, 则会触发一个错误。列表可能同时包含不可变对 (用 `cons` 构建) 和可变对 (用 `mcons` 构建)。

12.3 x86 指令集快速参考

表 12.1 列出一些 x86 指令及其作用。用 $A \to B$ 表示 A 的值被写入位置 B 。地址偏移量是以字节为单位给出的。指令参数 A, B, C 可以是直接常量 (如 `$4`) 、寄存器 (如 `%rax`) 或内存引用 (如 `-4(%ebp)`) 。大多数 x86 指令最多只允许一条指令引用一个内存。其他操作数必须是直接数或寄存器。

指令	运算
`addq` A, B	$A + B \to B$
`negq` A	$-A \to A$
`subq` A, B	$B - A \to B$
`imulq` A, B	$A \times B \to B$
`callq` L	输入返回地址并跳转到标签 L
`callq` $*A$	在地址 A 处调用函数。
`retq`	弹出返回地址，然后跳转到它
`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 必须是寄存器)
`cmpq` A, B	比较 A 和 B ，并设置标志寄存器 (B 不能是即时寄存器)
`je` L `jl` L `jle` L `jg` L `jge` L	如果标志寄存器与指令的条件代码匹配，则跳转到标签 L ，否则转到下一个指令。条件代码是 `e` 表示“等于”，`l` 表示“小于”，`le` 表示“小于或等于”，`g` 表示“大于”，`ge` 表示“大于或等于”。
`jmp` L	跳转到标签 L
`movq` A, B	$A \to B$
`movzbq` A, B	$A \to B$ ，其中 A 是一个单字节寄存器 (例如，`al` 或 `cl`) ，B 是一个 8 字节寄存器，并且 B 的额外字节被设为零。
`notq` A	$\sim A \to A$ (按位补码)
`orq` A, B	$A \vert B \to B$ (按位或)
`andq` A, B	$A \& B \to B$ (按位与)
`salq` A, B	B << $A \to B$ (向左移动，其中 A 是一个常数)
`sarq` A, B	B >> $A \to B$ (向右移动，其中 A 是一个常数)
`sete` A `setl` A `setle` A `setg` A `setge` A	如果标志匹配条件代码，则为 $1 \to A$ ，否则为 $0 \to A$ 。关于条件代码的描述，请参阅上面的 `je` 。A 必须是单字节寄存器 (如 `al` 或 `cl`) 。

表 12.1: 本书中使用的 x86 指令的快速参考。

$$
\begin{array}{lcl}
\mathit{type} & ::= & \texttt{Integer} \mid \texttt{Boolean} \mid \texttt{(Vector } \mathit{type} \ldots \texttt{)} \mid \texttt{Void} \\
& \mid & \texttt{(}\mathit{type} \ldots \texttt{ -> } \mathit{type}\texttt{)} \mid \texttt{Any} \\
\mathit{ftype} & ::= & \texttt{Integer} \mid \texttt{Boolean} \mid \texttt{Void} \mid \texttt{(Vector Any} \ldots \texttt{)} \\
& \mid & \texttt{(Any} \ldots \texttt{ -> Any)} \\
\mathit{exp} & ::= & \ldots \texttt{(inject } \mathit{exp}\ \mathit{ftype}\texttt{))} \mid \texttt{(project } \mathit{exp}\ \mathit{ftype}\texttt{)} \\
& \mid & \texttt{(any-vector-length } \mathit{exp}\texttt{)} \mid \texttt{(any-vector-ref } \mathit{exp}\ \mathit{exp}\texttt{)} \\
& \mid & \texttt{(any-vector-set! } \mathit{exp}\ \mathit{exp}\ \mathit{exp}\texttt{)} \\
& \mid & \texttt{(boolean? } \mathit{exp}\texttt{)} \mid \texttt{(integer? } \mathit{exp}\texttt{)} \mid \texttt{(void? } \mathit{exp}\texttt{)} \\
& \mid & \texttt{(vector? } \mathit{exp}\texttt{)} \mid \texttt{(procedure? } \mathit{exp}\texttt{)} \\
\mathit{def} & ::= & \texttt{(define (}\mathit{var}\ \texttt{[}\mathit{var}\texttt{:}\mathit{type}\texttt{]} \ldots \texttt{) :}\ \mathit{type}\ \mathit{exp}\texttt{)} \\
R_{\texttt{Any}} & ::= & \mathit{def} \ldots\ \mathit{exp}
\end{array}
$$

图 12.1: $R_{\texttt{Any}}$ 的具体语法，扩展 R_λ (图 7.4) 。

$$
\begin{array}{lcl}
\mathit{atm} & ::= & \mathit{int} \mid \mathit{var} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{(read)} \mid \texttt{(-}\ \mathit{atm}\texttt{)} \mid \texttt{(+}\ \mathit{atm}\ \mathit{atm}\texttt{)} \\
\mathit{stmt} & ::= & \mathit{var}\ \texttt{=}\ \mathit{exp}\texttt{;} \\
\mathit{tail} & ::= & \texttt{return}\ \mathit{exp}\texttt{;} \mid \mathit{stmt}\ \mathit{tail} \\
C_{\texttt{Var}} & ::= & (\mathit{label}\texttt{:}\ \mathit{tail}) \ldots
\end{array}
$$

图 12.2: $C_{\texttt{Var}}$ 中间语言的具体语法。

12.4 中间语言的具体语法

$R_{\texttt{Any}}$ 的具体语法定义在图 12.1 中。

$C_{\texttt{Var}}$ 、$C_{\texttt{If}}$ 、$C_{\texttt{Vec}}$ 和 $C_{\texttt{Fun}}$ 的具体语法分别定义在图 12.2 、12.3 、12.4 和 12.5 中。

$$
\begin{array}{lcl}
\mathit{atm} & ::= & \mathit{int} \mid \mathit{var} \mid \mathit{bool} \\
\mathit{cmp} & ::= & \texttt{eq?} \mid \texttt{<} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{(read)} \mid \texttt{(-}\ \mathit{atm}\texttt{)} \mid \texttt{(+}\ \mathit{atm}\ \mathit{atm}\texttt{)} \\
 & \mid & \texttt{(not}\ \mathit{atm}\texttt{)} \mid \texttt{(}\mathit{cmp}\ \mathit{atm}\ \mathit{atm}\texttt{)} \\
\mathit{stmt} & ::= & \mathit{var}\ \texttt{=}\ \mathit{exp}\texttt{;} \\
\mathit{tail} & ::= & \texttt{return}\ \mathit{exp}\texttt{;} \mid \mathit{stmt}\ \mathit{tail} \mid \texttt{goto}\ \mathit{label}\texttt{;} \\
 & \mid & \texttt{if (}\mathit{cmp}\ \mathit{atm}\ \mathit{atm}\texttt{) goto}\ \mathit{label}\texttt{; else goto}\ \mathit{label}\texttt{;} \\
C_{\texttt{If}} & ::= & (\mathit{label}\texttt{:}\ \mathit{tail})\ldots
\end{array}
$$

图 12.3: $C_{\texttt{If}}$ 中间语言的具体语法。

$$
\begin{array}{lcl}
\mathit{atm} & ::= & \mathit{int} \mid \mathit{var} \mid \mathit{bool} \\
\mathit{cmp} & ::= & \texttt{eq?} \mid \texttt{<} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{(read)} \mid \texttt{(-}\ \mathit{atm}\texttt{)} \mid \texttt{(+}\ \mathit{atm}\ \mathit{atm}\texttt{)} \\
 & \mid & \texttt{(not}\ \mathit{atm}\texttt{)} \mid \texttt{(}\mathit{cmp}\ \mathit{atm}\ \mathit{atm}\texttt{)} \\
 & \mid & \texttt{(allocate}\ \mathit{int}\ \mathit{type}\texttt{)} \\
 & \mid & \texttt{(vector-ref}\ \mathit{atm}\ \mathit{int}\texttt{)} \mid \texttt{(vector-set!}\ \mathit{atm}\ \mathit{int}\ \mathit{atm}\texttt{)} \\
 & \mid & \texttt{(global-value}\ \mathit{var}\texttt{)} \mid \texttt{(void)} \\
\mathit{stmt} & ::= & \mathit{var}\ \texttt{=}\ \mathit{exp}\texttt{;} \mid \texttt{(collect}\ \mathit{int}\texttt{)} \\
\mathit{tail} & ::= & \texttt{return}\ \mathit{exp}\texttt{;} \mid \mathit{stmt}\ \mathit{tail} \mid \texttt{goto}\ \mathit{label}\texttt{;} \\
 & \mid & \texttt{if (}\mathit{cmp}\ \mathit{atm}\ \mathit{atm}\texttt{) goto}\ \mathit{label}\texttt{; else goto}\ \mathit{label}\texttt{;} \\
C_{\texttt{Vec}} & ::= & (\mathit{label}\texttt{:}\ \mathit{tail})\ldots
\end{array}
$$

图 12.4: $C_{\texttt{Vec}}$ 中间语言的具体语法。

$$
\begin{array}{lcl}
\mathit{atm} & ::= & \mathit{int} \mid \mathit{var} \mid \texttt{\#t} \mid \texttt{\#f} \\
\mathit{cmp} & ::= & \texttt{eq?} \mid \texttt{<} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{(read)} \mid \texttt{(-}\ \mathit{atm}\texttt{)} \mid \texttt{(+}\ \mathit{atm}\ \mathit{atm}\texttt{)} \mid \texttt{(not}\ \mathit{atm}\texttt{)} \mid \texttt{(}\mathit{cmp}\ \mathit{atm}\ \mathit{atm}\texttt{)} \\
 & \mid & \texttt{(allocate}\ \mathit{int}\ \mathit{type}\texttt{)} \mid \texttt{(vector-ref}\ \mathit{atm}\ \mathit{int}\texttt{)} \\
 & \mid & \texttt{(vector-set!}\ \mathit{atm}\ \mathit{int}\ \mathit{atm}\texttt{)} \mid \texttt{(global-value}\ \mathit{name}\texttt{)} \mid \texttt{(void)} \\
 & \mid & \texttt{(fun-ref}\ \mathit{label}\texttt{)} \mid \texttt{(call}\ \mathit{atm}\ \mathit{atm}\ldots\texttt{)} \\
\mathit{stmt} & ::= & \texttt{(Assign}\ \mathit{var}\ \mathit{exp}\texttt{)} \mid \texttt{(Return}\ \mathit{exp}\texttt{)} \mid \texttt{(collect}\ \mathit{int}\texttt{)} \\
\mathit{tail} & ::= & \texttt{(Return}\ \mathit{exp}\texttt{)} \mid \texttt{(seq}\ \mathit{stmt}\ \mathit{tail}\texttt{)} \\
 & \mid & \texttt{(goto}\ \mathit{label}\texttt{)} \mid \texttt{(If (}\mathit{cmp}\ \mathit{atm}\ \mathit{atm}\texttt{) (goto}\ \mathit{label}\texttt{) (goto}\ \mathit{label}\texttt{))} \\
 & \mid & \texttt{(tail-call}\ \mathit{atm}\ \mathit{atm}\ldots\texttt{)} \\
\mathit{def} & ::= & \texttt{(define (}\mathit{label}\ [\mathit{var}\texttt{:}\mathit{type}]\ldots\texttt{)}\texttt{:}\mathit{type}\ \texttt{((}\mathit{label}\ \texttt{.}\ \mathit{tail}\texttt{)}\ldots\texttt{))} \\
C_{\texttt{Fun}} & ::= & \mathit{def}\ldots
\end{array}
$$

图 12.5: $C_{\texttt{Fun}}$ 语言，用函数扩展 $C_{\texttt{Vec}}$ (图 12.4) 。

参考文献

[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] John Backus. *The History of Fortran I, II, and III*, pages 25–74. Association for Computing Machinery, New York, NY, USA, 1978. ISBN 0127450408. URL `https://doi.org/10.1145/800025.1198345`.

[11] 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`.

[12] V. K. Balakrishnan. *Introductory Discrete Mathematics*. Dover Publications, Incorporated, 1996. ISBN 0486691152.

[13] 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`.

[14] 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.

[15] 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.

[16] Daniel Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, 1979. ISSN 0001-0782.

[17] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994. ISSN 0164-0925.

[18] Randal E. Bryant and David R. O'Hallaron. *x86-64 Machine-Level Programming*. Carnegie Mellon University, September 2005.

[19] 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.

[20] Luca Cardelli. The functional abstract machine. Technical Report TR-107, AT&T Bell Laboratories, 1983.

[21] Luca Cardelli. Compiling a functional language. In *ACM Symposium on LISP and Functional Programming*, LFP '84, pages 208–217. ACM, 1984.

[22] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985. ISSN 0360-0300.

[23] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 98–105. ACM Press, 1982. ISBN 0-89791-074-5.

[24] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[25] C. J. Cheney. A nonrecursive list compacting algoirthm. *Communications of the ACM*, 13(11), 1970.

[26] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 222–232. ACM Press, 1984. ISBN 0-89791-139-3.

[27] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):pp. 346–366, 1932. ISSN 0003486X. URL `http://www.jstor.org/stable/1968337`.

[28] 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`.

[29] Keith Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2nd edition, 2011.

[30] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *Compiler Construction*, 1998.

[31] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001. ISBN 0070131511.

[32] 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`.

[33] Olivier Danvy. Three steps for the CPS transformation. Technical Report CIS-92-02, Kansas State University, December 1991.

[34] 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`.

[35] E. W. Dijkstra. Why numbering should start at zero. Technical Report EWD831, University of Texas at Austin, 1982.

[36] 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`.

[37] R. Kent Dybvig. *The SCHEME Programming Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987. ISBN 0-13-791864-X.

[38] R. Kent Dybvig. *Three Implementation Models for Scheme*. PhD thesis, University of North Carolina at Chapel Hill, 1987.

[39] 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`.

[40] R. Kent Dybvig and Andrew Keep. P523 compiler assignments. Technical report, Indiana University, 2010.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] Matthew Flatt and PLT. The Racket reference 6.0. Technical report, PLT Inc., 2014. `http://docs.racket-lang.org/reference/index.html`.

[46] Matthew Flatt, Robert Bruce Findler, and PLT. The racket guide. Technical Report 6.0, PLT Inc., 2014.

[47] Daniel P. Friedman and Matthias Felleisen. *The Little Schemer (4th Ed.)*. MIT Press, Cambridge, MA, USA, 1996. ISBN 0-262-56099-2.

[48] Daniel P. Friedman and David S. Wise. Cons should not evaluate its arguments. Technical Report TR44, Indiana University, 1976.

[49] 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.

[50] Assefaw Hadish Gebremedhin. *Parallel Graph Coloring*. PhD thesis, University of Bergen, 1999.

[51] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, May 1996. ISSN 0164-0925. doi: 10.1145/229542.229546. URL https://doi.org/10.1145/229542.229546.

[52] Abdulaziz Ghuloum. An incremental approach to compiler construction. In *Scheme and Functional Programming Workshop*, 2006.

[53] 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.

[54] 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.

[55] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in lcf. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 119–130, New York, NY, USA, 1978. Association for Computing Machinery. ISBN 9781450373487. doi: 10.1145/512760.512773. URL https://doi.org/10.1145/512760.512773.

[56] 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.

[57] 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.

[58] 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.

[59] Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

[60] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient

gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.

[61] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167–189, 2010.

[62] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index register allocation. *J. ACM*, 13(1):43–61, January 1966. ISSN 0004-5411. doi: 10.1145/321312.321317. URL `https://doi.org/10.1145/321312.321317`.

[63] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, 3C and 3D*, December 2015.

[64] 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`.

[65] 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.

[66] 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.

[67] 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.

[68] Andrew W. Keep. *A Nanopass Framework for Commercial Compiler Development*. PhD thesis, Indiana University, December 2012.

[69] 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.

[70] 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.

[71] A. B. Kempe. On the geographical problem of the four colours. *American Journal of Mathematics*, 2(3):193–200, 1879. ISSN 00029327, 10806377. URL `http://www.jstor.org/stable/2369235`.

[72] 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.

[73] 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.

[74] S. Kleene. Introduction to Metamathematics, 1952.

[75] 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`.

[76] 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.

[77] 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.

[78] 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`.

[79] 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.

[80] 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.

[81] 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.

[82] David W. Matula, George Marble, and Joel D. Isaacson. Graph coloring algorithms††this research was supported in part by the advanced research projects agency of the department of defense under contract sd-302 and by the national science foundation under contract gj-446. In RONALD C. READ, editor, *Graph Theory and Computing*, pages 109 – 122. Academic Press, 1972. ISBN 978-1-4832-3187-7. doi: https://doi.org/10.1016/B978-1-4832-3187-7.50015-5. URL `http://www.sciencedirect.com/science/article/pii/B9781483231877500155`.

[83] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. *Sys-*

tem V Application Binary Interface, AMD64 Architecture Processor Supplement, October 2013.

[84] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960. ISSN 0001-0782.

[85] Microsoft. x64 architecture. `https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture`, March 2018.

[86] Microsoft. x64 calling convention. `https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention`, July 2020.

[87] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, 1990. ISBN 0-262-63132-6.

[88] 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`.

[89] E.F. Moore. The shortest path through a maze. In *Proceedings of an International Symposium on the Theory of Switching*, April 1959.

[90] 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`.

[91] Erik Österlund and Welf Löwe. Block-free concurrent gc: Stack scanning and copying. In *Proceedings of the 2016 ACM SIGPLAN In-*

ternational 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`.

[92] Jens Palsberg. Register allocation via coloring of chordal graphs. In *CATS '07: Proceedings of the thirteenth Australasian symposium on Theory of computing*, pages 3–3, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc. ISBN 1-920-68246-5.

[93] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[94] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999. ISSN 0164-0925.

[95] 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.

[96] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 2002. ISBN 0072474777.

[97] 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.

[98] 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.

[99] 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`.

[100] 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.

[101] 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`.

[102] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

[103] 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.

[104] 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.

[105] Guy L. Steele. Rabbit: A compiler for Scheme. Technical report, Cambridge, MA, USA, 1978.

[106] Guy L. Steele, Jr. Data representations in pdp-10 maclisp. AI Memo 420, MIT Artificial Intelligence Lab, September 1977.

[107] Bjarne Stroustrup. Parameterized types for C++. In *USENIX C++ Conference*, October 1988.

[108] 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.

[109] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium*, 2006.

[110] 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.

[111] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, ESOP, pages 1–16, March 2009.

[112] 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.

[113] 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.