Source Code Analysis and Type Inference for R

By

NICHOLAS ULLE

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Statistics

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Duncan Temple Lang, Chair

Norman Matloff

Ethan Anderes

Committee in Charge

2021

To Louis, who never got the chance.

Contents

Abstract

R is a dynamic, interpreted programming language designed for statistical computing. In contrast to languages more traditionally used in software development and engineering, code analysis and tools for code analysis are not common in the R community, with some notable exceptions. Nevertheless, a general framework that facilitates the development of novel code analyses for R is valuable. This dissertation presents a collection of strategies and software for static analysis of R code. Two of the three parts focus on type inference, a specific kind of static analysis which attempts to determine the type of data produced by each expression in the code.

The first part describes a framework for creating static analyses and transformations of R code based on contemporary techniques and research. The framework provides tools to search code for specific syntactic patterns, extract information about different ways in which code can be evaluated depending on run-time conditions, and examine how data propagate from definitions of variables to expressions which use those variables.

The second part presents a static type inference strategy for R code. The strategy leverages the static analysis framework developed in the first chapter. In contrast to languages like C and Java, R code is generally not annotated with types and there is no built-in syntax to add type annotations. Thus type inference is necessary in order to get information about types. Information about types is useful for transforming and translating code, checking code for errors, and reasoning about code.

The third part presents strategies for collecting type information from foreign routines written in C and called from R. The type inference strategy for R code can more accurately infer types if it has type signatures for these routines. Even in C code, the R type of an R object does not have to be specified in the code, so type inference is non-trivial.

Acknowledgements

Completing this dissertation would not have been possible without the support and feedback of many people.

First and foremost, deepest thanks to my advisor, Duncan Temple Lang. Your bottomless knowledge of R, endless enthusiasm for statistical computing, and unquestioning support for my interest in teaching have all shaped who I am now. Your patience and neverending confidence that I'd eventually finish writing kept me going even when I had none myself. I've truly enjoyed our discussions over the years and look forward to more.

I'm also very grateful to the other members of my committee, past and present. Norm Matloff, thank you for always being friendly and willing to chat, promoting me and my work to others, and providing general advice. Ethan Anderes, your enthusiasm for statistical computing and the Julia language is delightful; I hope we can chat about Julia again soon. James Sharpnack, teaching with you was a great experience; it rekindled my interest in Python and SciPy. Wolfgang Polonik, your guidance while I was in the RTG program was invaluable, and your humor and wit always lift my spirits. Zhendong Su, you played an important role in the early stages of my research by providing opportunities to learn more about computer science.

Thanks to Clark Fitzgerald for many interesting and productive discussions about research and teaching in statistics, data science, and computer science. Your feedback over the years has been invaluable. Thanks for the bike as well; it kept me sane during the pandemic. I look forward to more discussions and collaborations in the future.

Thanks to Deb Nolan for being a role model as a data scientist, educator, and academic. In spite of the pandemic, you made me feel welcome at UC Berkeley during my year there. Now that my dissertation is complete, I'm excited to focus on our planned collaboration.

Thanks to Carl Stahmer and the rest of the UC Davis DataLab for encouraging me during my final year, providing life advice, and ensuring that my work at DataLab didn't interfere with the completion of my dissertation.

Thanks to the staff of the UC Davis Department of Statistics, particularly Pete Scully and Nehad Ismail, for helping me and other students find funding, solve technology problems, and navigate the university bureaucracy.

Throughout graduate school, my friends both at Davis and farther afield provided much-needed advice, distraction, commiseration, and escape. Thanks Cecilia, Chris A., Chris C., Dmitriy, Gary, Hugo, Jamshid, Justin, Kavi, Lingfei, Luna, Nicholas A., Nick B., Rick, Seva, Shuyang, Tiffany H., Tiffany W., and Zengqun. Thanks as well to my friends in Wong Hall: Raymond, Matt, Olivia, Clark, Taeyen, Hoseung, and Po.

Last but not least, thanks to my family—Karl, Barbara, and Hayley—for supporting my decisions, believing in me even when I don't, and putting up with all of the board games I want to play every time I come home.

Introduction

Source code analysis is the process of programmatically extracting information from source code in order to summarize, suggest improvements to, or transform the code. *Static* source code analysis does this without evaluating the code. Static source code analysis is a valuable tool for understanding and improving code in contemporary statistical computing.

In contrast to languages which originated in the computer science community—such as those more traditionally used in software development and engineering—code analysis and tools for code analysis are not common in the R community, with some notable exceptions. Most existing code analysis tools for R do not leverage accumulated programming languages research about code analysis, nor provide reusable infrastructure on which to build new tools. They are typically built for a specific purpose and based on R's built-in metaprogramming features. Nevertheless, a general framework that facilitates the development of novel code analyses is valuable.

This dissertation presents a collection of strategies and software for static analysis of code written in R. Two of the three chapters focus on *type inference*, a specific kind of static analysis which attempts to determine the type of data produced by each expression in the code. The dissertation is organized as follows:

- Chapter 1 describes a framework for creating static analyses and transformations of R code, based on contemporary techniques and research. The framework provides tools to search code for specific syntactic patterns, extract information about different ways in which code can be evaluated depending on run-time conditions, and examine how data propagate from variable definitions to expressions which use those variables. The chapter discusses the contexts in which each of these approaches to code analysis is useful, and also introduces the package **rstatic**, which implements the framework. The code analysis framework in this chapter is the foundation for the analyses presented in Chapter 2.
- Chapter 2 presents a static type inference strategy for R code. Information about types is useful for transforming and translating code, checking code for errors, and reasoning

about code. In contrast to languages like C and Java, R code is generally not annotated with types and there is no built-in syntax to add type annotations. Thus type inference is necessary in order to get information about types. The type inference strategy is based on a strategy originally developed by Damas and Milner (1982) for the ML programming language. In addition to describing the type inference strategy for R code, the chapter also documents features of R which make type inference challenging, and introduces the packages **typesys** and **RTypeInference**, which together are a prototype implementation of the strategy.

- Chapter 3 presents strategies for collecting type information from foreign routines written in C or Fortran and called from R. The type inference strategy from Chapter 2 can more accurately infer types in R code if it has type signatures for these routines. Routines called with R's `.Call`, `.External`, and `.External2` interfaces use the R Internals programming interface in order to compute directly on R objects. Even in C code, the R type of an R object does not have to be specified in the code, so type inference is non-trivial. The chapter describes a strategy to infer the type signatures of C routines which compute on R objects.

Chapter 1

A Framework for Static Analysis of R Code

1.1 Introduction

This chapter describes a framework for static analysis and transformation of R code. The purpose of the framework is to bring contemporary strategies and data structures for static analysis to R in a way that takes advantage of R's functional and object-oriented programming features. The framework is organized around extracting information about three different characteristics of code:

- The *syntax* and structure of the code, meaning the form and order in which expressions are written. Representing and navigating this structure is a necessary first step to programmatically extract other information from the code.
- The *control flow*, meaning the order in which expressions in the code will be evaluated at run-time. Control structures such as if-expressions and for-loops mean that code will not necessarily be evaluated in the order it is written, and that some sections of code might not be evaluated at all.
- The *data flow*, meaning—for each variable—the association between expressions which assign or modify the value of the variable and expressions which use the value of the variable. We refer to these two kinds of expressions respectively as the *definitions* and *uses* of the variable. Data flow is important to consider when analyzing R code because variables can be redefined at any point.

A specific static analysis can depend on information about any or all of these characteristics. For example, locating all calls to functions which load data (such as `read.csv`) is an analysis

that only depends on syntax. On the other hand, a global variables analysis—described in Example 1—depends on data flow and can also incorporate control flow.

Example 1. The first step to cross validate a statistical model on a given data set is to split the data into k subsets, called *folds*. Each fold should contain approximately the same number of observations as the others, randomly selected without replacement. The function `split_eq` is an implementation of this step. The function has a parameter `shuffle` to control whether or not the data set is randomly shuffled before it is split. The function also has a subtle bug: the variable `x` is only defined when `shuffle` is `TRUE`. Here's the code for the function:

```
split_eq = function(data, k, shuffle = TRUE)
{
  n = nrow(data)
  if (shuffle)
    x = data[sample(n, n), ]
  groups = rep(1:k, length = n)
  split(x, groups)
}
```

Listing 1.1: The `split_eq` function splits a data frame into `k` groups of approximately equal size. The function doesn't work as intended when `shuffle` is `FALSE`.

Since `TRUE` is the default argument for `shuffle`, the bug might go unnoticed for a while. When `shuffle` is `FALSE`, the function looks for a *global variable* `x`, a variable defined somewhere outside of the function. If the user of the function has a variable `x` in their workspace, it may seem like the function works correctly, or the function may return an unexpected result rather than emitting an error. When bugs cause unexpected results rather than errors, they are more difficult to detect and diagnose by hand, especially in functions that are longer or more complex than `split_eq`.

A tool to find expressions which use global variables would help with detecting this kind of bug. Creating such a tool is feasible because the code contains information about where variables are defined in the function, and R has well-defined rules for scoping.

R packages such as **codetools** (Tierney 2020) and **globals** (Bengtsson 2018) provide functions to find expressions which use global variables. The approach these packages take is syntax-based: they find variables which are not assigned at any earlier point in the code. This approach

is simple, inexpensive to compute, and often produces correct results. However, it does not produce the correct result for the `split_eq` function, because there is an assignment to `x` before the expression `split(x, groups)`. The problem is that the assignment is conditional—that is, inside of an if-statement—and these packages do not take control flow and data flow into account.

Data flow information gets to the heart of the matter. When the variable `x` is used in `split(x, groups)`, the value of `x` can originate from two different places—the definition `x = data[sample(n, n), ]` or a definition outside of the body of the function. With that information, the programmer can see that the code doesn't work as intended and make a correction. Control flow information adds that the first definition corresponds to when `shuffle` is `TRUE` and the second corresponds to when `shuffle` is `FALSE`. This additional information makes it easier to pinpoint the source of the bug. □

It's possible to extend or fork the packages mentioned in Example 1 so that they correctly detect the global variable in the example function. That said, our goal is broader—to provide a framework which is convenient to use to implement any kind of static analysis and transformation of R code. This framework is also the foundation for the type inference strategy presented in Chapter 2.

The package **rstatic** is an implementation of the framework for static analysis and transformation of R code. The package provides data structures to represent code which are based on contemporary code analysis research and facilitate extracting and using information about syntax, control flow, and data flow. These data structures are implemented using R's object-oriented programming systems so that package users can extend them as needed. The package also provides functions for common analysis and transformation steps. The framework and package are informed and inspired by the prior work of Temple Lang, Peng, et al. on the **CodeDepends** package (2018).

R provides its own functions to convert code into a data structure which can be programmatically analyzed and transformed. R represents code as *language objects*. Each language object is an *abstract syntax tree* (AST), meaning code is represented as a tree of subexpressions. For instance, the expression `mean(x + 1)` is the parent of the subexpressions `mean` and `x + 1`. The expression `x + 1` is in turn the parent of the subexpressions `x` and `1`. Figure 1.1 shows the AST for this expression. The AST is primarily useful in analyses where the focus is syntax rather than control or data flow.

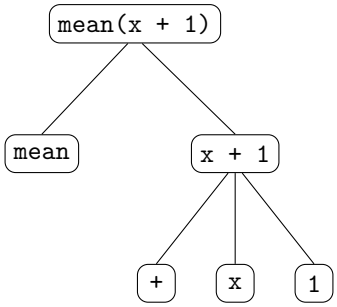

Figure 1.1: The abstract syntax tree for the expression `mean(x + 1)`.

The **rstatic** package also provides an abstract syntax tree data structure (described in Section 1.2) as one of three ways to represent code. We will always refer to the **rstatic** representation as an abstract syntax tree, and the built-in representation as a language object or *parse tree*. These terms follow the conventions of the R documentation.

This chapter presents the static analysis framework and demonstrates the major features of **rstatic** through examples. Here's how the chapter is organized:

- Section 1.2 introduces the abstract syntax trees provided by **rstatic**, which are primarily useful in analyses that search for specific syntax patterns. Additionally, understanding **rstatic**'s ASTs is foundational to using its other representations for code. This section also addresses why **rstatic** provides its own classes for ASTs rather than using R's language objects.

- Section 1.3 focuses on control flow. The section begins by demonstrating why ASTs are unwieldy for static analyses which depend on control flow information. The section then introduces *control flow graphs* (CFGs). A CFG is a graph where nodes represent expressions and edges represent control flow. If-statements create branches in the graph and loops create cycles. CFGs are primarily useful for analyses sensitive to the order in which code will be evaluated at run-time. For instance, an analysis to determine the point(s) where a variable is no longer needed must find the last expression where the variable is evaluated.

- Section 1.4 focuses on data flow. The section first describes the *iterative data flow analysis algorithm*, a well-known and general algorithm for solving a variety of static analysis problems. The section then introduces *static single assignment* (SSA) form, a form of CFG where each variable definition has a unique identifier, so that when a variable is used in an expression, its definition can be located immediately. SSA form is essential to the type

inference strategy described in Chapter 2, and also useful for other analyses sensitive to the values assigned to variables at run-time.

1.2 Analyzing the Syntax and Structure of Code

The starting point for static code analysis is the code's syntax and structure. Using knowledge about the semantics of the language, we can manually or programmatically identify structural patterns to gain insight into what the code does and how it works.

An abstract syntax tree retains most of the structure of the code from which it is derived. This means ASTs are well-suited to analyses which search code for specific structural patterns. It also means ASTs closely resemble code, so there is minimal cognitive overhead when reasoning about them. Information about control flow and data flow can be obtained from an AST through analysis, so ASTs are a starting point for building richer analyses as well.

This section introduces how to use ASTs to analyze and transform code. The focus is specifically on the ASTs provided by the **rstatic** package. These are used throughout the package, so understanding them is crucial to using the package. They have several advantages over R's language objects, which are discussed throughout the section. We begin with an example.

Example 2. Consider the problem of analyzing an R script to extract the paths of data files loaded by the script. This kind of analysis is useful for summarizing the relationships between files in a project. Packages such as **CodeDepends** (Temple Lang, Peng, et al. 2018) already implement this analysis, so the purpose of this example is not the analysis itself. Instead, the purpose is to demonstrate that **rstatic** provides a convenient high-level interface for implementing this kind of analysis, and also to introduce how to use the package.

The first step in any static analysis is to convert the code into an appropriate representation. In this case, we'll search for syntax patterns such as calls to `read.csv` and other functions which load data. Since the focus of the analysis—at least at this stage—is syntax, an AST is an appropriate representation. The **rstatic** functions `quote_ast` and `to_ast` convert code and R language objects, respectively, into **rstatic** ASTs.

After converting the code to an AST, the next step is to visit each expression in the AST and test whether the expression is a call to a function which reads data. In the AST, calls to functions are represented by the class `Call`. So `is(x, "Call")` tests whether an expression `x` is a call. The analysis can ignore expressions which are not calls.

As an aside, the **rstatic** classes—including `Call`—which represent expressions in an AST all inherit from the `ASTNode` class. Intermediate classes group semantically similar expressions. For instance, there are intermediate classes to group control flow statements (`ControlFlow`) and literal values (`Literal`). Classes at the lowest level of the hierarchy represent concrete expressions, such as if-statements (`If`), function calls (`Call`), and literal integers (`Integer`). Figure 1.2 shows the hierarchy of classes. The hierarchy is designed to be convenient for distinguishing different kinds of expressions and for creating methods which handle whole groups of semantically similar expressions.

For calls, the analysis must also test whether the name of the called function is `read.csv` or `readRDS`. The **rstatic** AST classes use named fields for subexpressions, and for the `Call` class the field `fn` contains the name of the called function. Names are represented in the AST by the `Symbol` class. The **rstatic** function `ast_name` returns the name represented by a `Symbol` as a string. So for a call `x`, the code to test whether the called function is named `read.csv` or `readRDS` is:

```
1 readers = c("read.csv", "readRDS")
2 ast_name(x$fn) %in% readers
```

This code can be extended to check for other functions by adding their names to the `readers` vector. Note that in a call to an anonymous function, the `fn` field contains the AST for the function rather than a name, but the above code still works correctly, because the `ast_name` function returns `NA` if its argument is not a `Symbol`.

The **rstatic** function `ast_find_all` returns all expressions in an AST which satisfy some condition. The condition must be provided to `ast_find_all` as a test function which accepts an expression and returns a logical value. The `ast_find_all` function calls the test function on each expression in the AST and returns a list of all expressions where the test function returned `TRUE`. Here's how to use `ast_find_all` and the conditions described in the preceding paragraphs to get a list of all calls to `read.csv` and `readRDS` in an AST `ast`:

```
1 is_reader_call = function(x, readers = c("read.csv", "readRDS"))
2 {
3   is(x, "Call") && ast_name(x$fn) %in% readers
4 }
5 exps = ast_find_all(ast, is_reader_call)
```

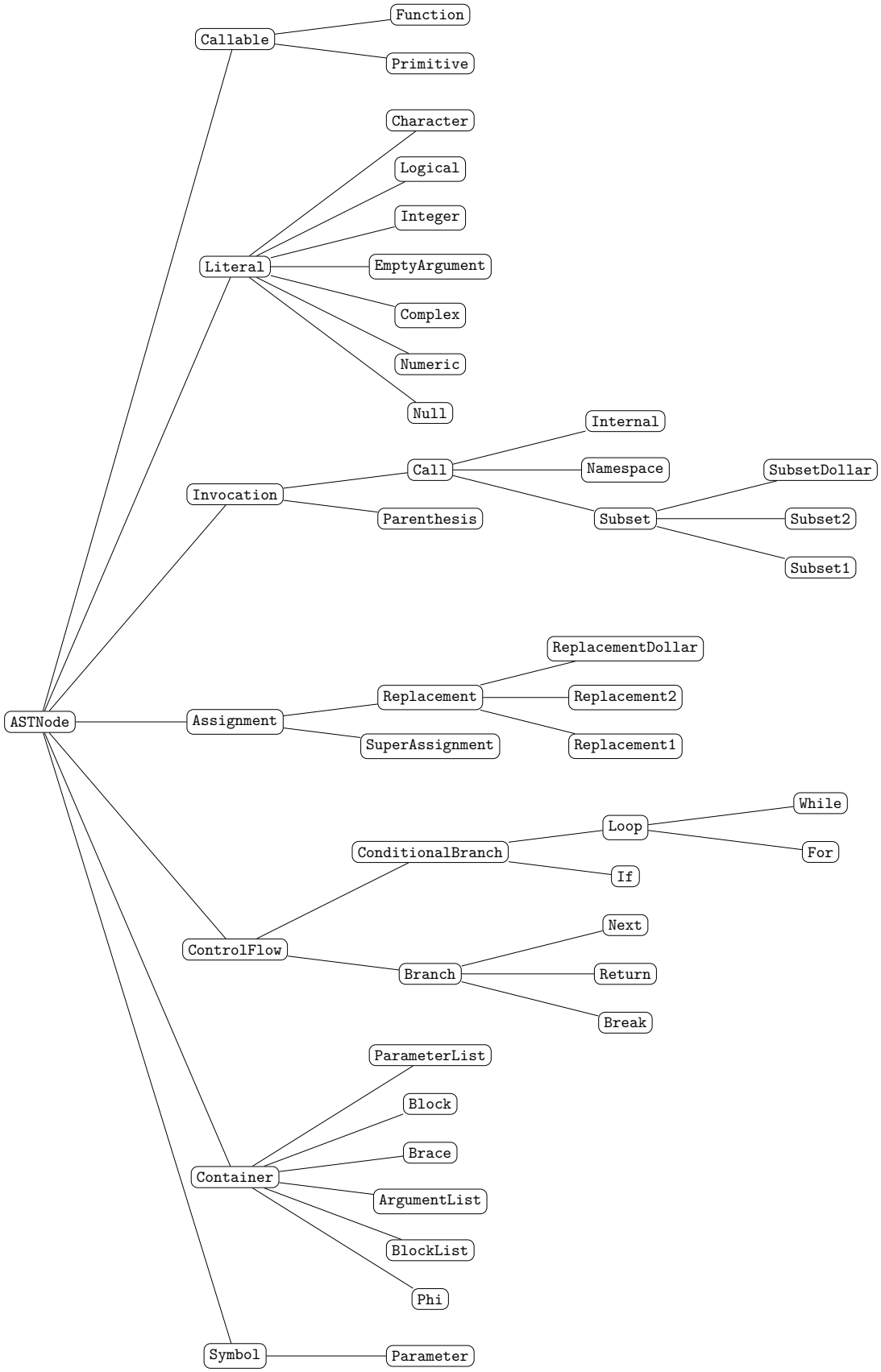

Figure 1.2: The hierarchy of classes provided by the **rstatic** package to represent expressions.

Listing 1.2: The code for a static analysis which uses **rstatic** to find all calls to `read.csv` or `readRDS` in the AST `ast`.

To make the example and analysis concrete, consider the following excerpt from an R script which merges earthquake data with population density data:

```
quakes = read.csv("/home/gus/data/quakes.csv")
quakes = subset(quakes, 2012 <= year & year <= 2020)
pd = readRDS("/home/gus/data/popden.rds")[c("county", "popden")]
quakes_pd = merge(quakes, pd, by = "county")
```

Listing 1.3: An excerpt of an R script for analyzing earthquake data. This code loads earthquake data, loads population density data, and merges the two.

This code will be the test case for the analysis. The result from the analysis should be the paths to the `quakes.csv` and `popden.rds` files. Converting the code in Listing 1.3 to an AST and then running the analysis in Listing 1.2 produces this list:

```
[[1]]
<Call> $args $fn $parent
read.csv("/home/gus/data/quakes.csv")
[[2]]
<Call> $args $fn $parent
readRDS("/home/gus/data/popden.rds")
```

Listing 1.4: The list `exps`, result of applying the reader calls analysis in Listing 1.2 to the earthquakes analysis code in Listing 1.3.

The final step of the analysis is to extract the paths from the list of calls to the `read.csv` and `readRDS`. The list of arguments to a call can be accessed directly through the `Call` class' `args` field. For each of the two calls in the list `exps` shown in Listing 1.4, the path—which is the first argument—is a character literal. Character literals are represented by the class `Character`, which inherits from the class `Literal`. The `Literal` class' field `value` provides access its value. Thus the code to get the value of the first argument of each call in the list `exps` is:

```
lapply(exps, function(x) x$args[[1]]$value)
```

In calls to `read.csv` and `readRDS`, the file path does not necessarily have to be the first argument. The analysis can address this by normalizing the calls, so that the arguments are

explicitly assigned to and in the same order as the parameters of the called function. The **rstatic** function `match_call` carries out this normalization step. After normalization, for `read.csv` and `readRDS`, the path will always be the first argument and will always be assigned to the parameter `file`. Other functions which load data may have different parameters. The analysis can be extended to handle these by using a table which pairs reader functions with the name of their file path parameter.

Finally, the argument for the file path will not be a character literal in every call to `read.csv` and `readRDS`. The argument can also be some other expression, such as a variable name or a call to another function. For variable names, if the variable is *constant*—meaning it is assigned a literal value before the call and does not change—the value can be recovered with static analysis. We recommend using data flow information and the strategies of Section 1.4 to handle the variable name case. For other expressions, it's possible to recover the value with static analysis if the operands are literals or constants. To do so, the analysis has to emulate evaluation, which may be undesirable or computationally infeasible depending on the expression. □

The goal of the static analysis in Example 2 is to extract the paths of data files loaded by the code. Since the analysis can be described in a simple way in terms of syntax—finding calls to specific functions and extracting specific arguments from those calls—the AST is a convenient and suitable data structure on which to carry it out. The example illustrates a general strategy for creating syntax-based static analyses: visit each expression in the AST, collecting information from the expressions which are relevant to the analysis at hand.

Example 2 also highlights several features of the **rstatic** package's AST and functions:

- Expressions are represented by classes with only one semantic interpretation. For example, `Call` objects represent function calls and only function calls. This is in contrast to R's language objects, whose `call` class is a catch-all for any expression with call syntax, including return expressions and parentheses.

- Expressions are represented by classes arranged in a meaningful hierarchy. For example, the classes `While` and `For` are both subclasses of `Loop`. Thus an analysis can provide a method that handles loops generally, or specific methods for each kind of loop. R's language object classes are not part of an inherent class hierarchy.

- Expressions have named fields. As a result, code to manipulate ASTs is explicit about which components of the AST it accesses. In contrast, code to manipulate R's language

objects uses integer indexes, so understanding the code depends on knowing exactly what kind of language object is being manipulated and the positions of its components.

- The `ast_find_all` function automates traversing the AST, so that analysis developers can focus on their analysis question rather than on traversal. The function returns a list of expressions which can be used with R's built-in functions for working with lists or in additional calls to `ast_find_all`.

The classes **rstatic** provides to represent expressions are R6 classes. The R6 class system is provided by the package **R6** (Chang 2021). A distinguishing feature of R6 classes is that their instances have *reference semantics*, similar to R environments or reference classes. If one assigns an R6 object to multiple variables, changing any one of them changes all of them—no copies are made. In other words, R6 objects are mutable and assigning one to a variable creates a reference rather than a copy. This differs from the copy-on-write semantics of most R built-in objects. This feature is convenient when working with ASTs, since it means changing an expression changes the AST, and that expressions can store a reference to their parent expression to simplify bidirectional traversal. The next example shows one way this feature makes it easy to transform code.

Example 3. Example 2 showed how to use **rstatic** to locate calls in code which read files from disk. The paths to files in that example were all *absolute paths*, beginning from the root directory of the filesystem. The code would be more portable if it used paths relative to some project directory. The goal of this example is to use **rstatic** to replace all of the absolute paths in the code with relative paths.

Suppose the project directory is `/home/gus` and that the `make_relative_path` function converts a relative path into an absolute path, given the path to the project directory. So for example, this code returns the relative path `data/popden.rds`:

```
1 make_relative_path("/home/gus/data/popden.rds", initial = "/home/gus")
```

In Example 2, the result from using `ast_find_all` to find calls to reader functions is a list called `exps`. Listing 1.4 shows the list. The expressions in the list are `ASTNode` objects (which use R6) and have reference semantics, so changing them will change the AST. Thus to change the paths in the code, an analysis can change the paths in the result from `ast_find_all`. In contrast, for ordinary R language objects, it would be necessary to build the transformation into the traversal which finds the reader calls, or to do a second traversal just for the transformation.

Using **rstatic** and the list of calls `exps` from Example 2, the code to transform all paths in the code is:

```
lapply(exps, function(x) {
  old_path = x$args[[1]]$value
  new_path = make_relative_path(old_path, initial = "/home/gus")
  x$args[[1]]$value = new_path
})
```

This transformation can be extended to handle a wide variety of different reader functions as described at the end of Example 2.

After applying the transformation, the AST object `ast` reflects the changes. The **rstatic** function `to_r` converts an AST back into an R language object, which can then be evaluated or saved to a file. □

Example 3 shows that because the classes which represent expressions in **rstatic** have reference semantics, for simple transformations it is not always necessary to traverse the AST. Instead, one can reuse or build on the results of prior, related analyses, the way the example does with the results from the analysis of calls to reader functions.

The other advantage of having reference semantics is that each expression in the AST stores a reference to its parent expression. This makes it possible to get the context of expressions found with `ast_find_all` or by any other means, and even to replace whole expressions. The next example demonstrates a simple application of parent references.

Example 4. Once again consider the calls to reader functions found by the analysis in Example 2. Suppose we want to use that analysis as part of a larger analysis to determine which expressions in the code depend on loaded data sets. The first step, and the goal of this example, is to get the names of the variables to which loaded data sets are assigned.

An analysis can get the names of these variables using the list `exps` of calls to reader functions (from Example 2) and the `parent` field on every AST expression. In the original code in Listing 1.3, the parent expression of each call to a reader function is an assignment. After getting the `Assignment` expression from the `parent` field, the analysis can get the `Symbol` for the assigned variable from the expression's `write` field:

```
lapply(exps, function(x) x$parent$write)
```

The result is a list which contains the two `Symbol`s `quakes` and `pd`.

The preceding analysis assumes that calls to reader functions will always be the right-hand subexpression of an assignment, which will not always be true. There are several different ways to address this. One is to check the class of `x$parent`, and if it is not an `Assignment` object, take some additional action. For instance, the analysis could traverse further up the tree (by getting the parent's parent) in search of an assignment expression, or could return a value (such as `NULL`) which indicates that the result of the call is not directly assigned to a variable. Another approach is to write a completely new analysis which uses `ast_find_all` to find all assignment expressions which contain calls to reader functions. □

The point of Example 4 is that because the expressions in **rstatic** ASTs have parent references, there is a simple way to create an analysis that begins from anywhere in the AST and traverses upwards towards the root, the expression which is the ancestor of all other expressions in the AST. This would be difficult to do with R's built-in language objects, since they do not feature any way to get their parent.

The **rstatic** package provides the function `ast_transform_all` in addition and as a complement to `ast_find_all`. The `ast_transform_all` function calls a transformation function on each expression in the AST and replaces the expression with whatever the transformation function returns. For transformations where the information extracted in the course of the transformation is not needed elsewhere, the `ast_transform_all` function is a more appropriate choice than using `ast_find_all` and then transforming the results.

The `ast_find_all` and `ast_transform_all` functions are convenient for finding patterns within a single expression. For patterns composed of multiple expressions, it's helpful to have finer control over how the AST is traversed. The **rstatic** package's `children` function is well-suited to this case. The function returns the list of immediate children for a given expression. Both the `ast_find_all` and `ast_transform_all` all functions call the `children` function in order to determine which expressions to visit next during traversal.

1.2.1 Using and Extending the AST Classes

The AST class hierarchy is one of the main features of **rstatic**. Every class in the hierarchy inherits from the `ASTNode` class. The `ASTNode` class provides basic features such as the `parent` field, a `copy` method, and a `.data` field.

The `.data` field is intended for storing arbitrary metadata. Analyses can use the `.data` field

Class	Description	Inherits From
`Brace`	Braces `{ }`	`Container`
`ArgumentList`	List of arguments in a call	`Container`
`ParameterList`	List of parameters in a function definition	`Container`
`Next`	`next` expression	`Branch`
`Break`	`break` expression	`Branch`
`Return`	`return()` expression	`Branch`
`If`	`if` expression	`ConditionalBranch`
`For`	`for` expression	`Loop`
`While`	`while` expression	`Loop`
`Call`	Function call	`Invocation`
`Parenthesis`	Parentheses `( )`	`Invocation`
`Namespace`	Namespace operator `::` or `:::`	`Call`
`Subset`	Subset operator `[`, `[[`, or `$`	`Call`
`Assignment`	Variable assignment operator	`ASTNode`
`SuperAssignment`	Super assignment operator `<<-`	`Assignment`
`Replacement`	Replacement assignment, e.g., `length(x) = 5`	`Assignment`
`Symbol`	Variable name	`ASTNode`
`Parameter`	Parameter name in a function definition	`Symbol`
`Function`	Function definition (unevaluated)	`Callable`
`Primitive`	Primitive function (internally implemented)	`Callable`
`EmptyArgument`	Empty or "missing" function argument	`Literal`
`Null`	`NULL` value	`Literal`
`Logical`	Literal logical value	`Literal`
`Integer`	Literal integer	`Literal`
`Numeric`	Literal decimal number	`Literal`
`Complex`	Literal complex number	`Literal`
`Character`	Literal string	`Literal`

Table 1.1: Classes in **rstatic** that represent concrete components of the R language.

to store results when the results are associated with individual expressions or locations in the code. This circumvents the bookkeeping that would be necessary to match results to expressions if the results were returned separately. Language objects cannot store additional information this way. While it is possible to add R attributes to some classes of language object, not all of them support attributes. For instance, R explicitly forbids setting attributes on `symbol` objects.

A major advantage of **rstatic**'s AST class hierarchy over R's built-in language objects is that the AST class hierarchy can be extended to represent new syntactic structures or to specialize analyses for specific kinds of calls. Here are a few examples of where it would be productive to extend **rstatic** with new classes:

- To represent R comments. To date, R does not provide a way to represent comments as language objects. However, R's built-in `parse` function does record information about comments in a `srcref` attribute on parsed code. When **rstatic** generates an AST, it could

use this information and a custom `Comment` class to include the comments in the AST, without the need to develop an entirely new parser. Comments can contain information that is relevant to code analysis or be of interest in their own right. For instance, the **roxygen2** package uses specially-formatted comments to extend the R language (Wickham, Danenberg, et al. 2021). The package currently uses regular expressions to find these comments.

- To represent the **magrittr** package's `%>%` pipe operator (Bache and Wickham 2020). The pipe operator calls the expression in its second operand using the result of its first operand as the first argument. That is, `42 %>% f` is the same as `f(42)`. The pipe operator uses non-standard evaluation to transform the syntax of the surrounding code, so its semantics differ from all of R's built-in functions. A custom `Pipe` class to represent the pipe operator in ASTs would facilitate transforming code into and out of pipe operator form (for instance, as a normalization step before applying analyses and transformations which were not designed with the pipe operator in mind).

- To represent the **ggplot2** package's `+` plus operator for combining layers of plots (Wickham 2016). A custom class would facilitate analyses and transformations which extract information about or rearrange/change the layers in plots. For instance, since plots often have more than two layers, the custom class store a list of all layers in the plot. The implementor of the custom class could also provide helper functions to convert the custom class to and from the `Call` class ordinarily used for the plus operator.

After creating a custom AST class, additional work is necessary to get the custom class into generated ASTs. There are two ways to go about this:

1. By transforming the AST after it's been constructed, for instance with `ast_transform_all`
2. By changing how the `to_ast` function constructs the AST

The `to_ast` function provides a parameter `call_handlers` to override how expressions represented by the `call` class in R's parse trees are converted into **rstatic** AST objects. The `call_handlers` argument should be a named list of handler functions. Each handler function overrides AST construction for calls to the named function. The handler function must accept three arguments: the R `call` object to convert, the name of the called function (a character vector), and further arguments passed down from the `to_ast` function.

Changing AST construction to get custom classes into an AST is possible but more difficult when the classes correspond to syntax which is not represented by a call. In that case, the `to_ast` function does not provide a parameter to override construction, so instead the `to_ast` function must be modified. This is not a serious limitation because R users typically introduce new semantics to the language through the call syntax, and because transforming the AST after it's been constructed is still viable.

1.2.2 Related Work

The **codetools** package (Tierney 2020) was the first code analysis package for R. The package provides documented functions to detect global variables, to check function definitions for possible problems with how variables are used, and to print language objects as a tree. The package also provides several undocumented functions to support these, including a function to collect information while traversing a language object. Updates to the package are infrequent, and it was labelled as potentially unstable until 2018. Lack of detailed documentation makes the package relatively difficult to use for new analyses, although the function it provides for finding global variables is used or extended by several other code analysis packages.

The **lintr** package (Hester 2017) provides functions to check for stylistic problems in R code, a process called *linting*. The package uses regular expression pattern matching on the string representation of code in order to carry out its analyses. This approach is adequate for the problems the package is designed to detect, since they all have to do with how the code is written (syntax) rather than what the code means or does (semantics). The package is not designed nor intended to facilitate development of new analyses.

The **covr** package (Hester 2018) analyzes R code to determine which parts of the code is tested by the code's associated test cases. Unlike **rstatic** and all of the other packages mentioned in this section, **covr** uses *dynamic code analysis*. In a dynamic code analysis, the code is instrumented to collect information at run-time and then run. Dynamic code analysis can collect information that is unknowable before run-time, but requires running the code. The **covr** package is not designed nor intended to facilitate new analyses.

1.3 Representing the Control Flow of Code

The goal of many static analyses is to determine characteristics the code will have at run-time. For instance, we may want to analyze code to determine the points at which variables can be freed because they are no longer needed, to replace constant variables with their literal values, or to infer the types of values returned by expressions (the topic of Chapter 2). Something these analyses have in common is that they depend on control flow—the order in which code will be evaluated—because they typically emulate part of the evaluation model.

Control flow expressions (if-expressions and loops) create regions of code which are only evaluated if certain conditions are satisfied at run-time. As a result, running the same code with different inputs can cause different sequences of expressions to be evaluated. We call these sequences of expressions *evaluation paths*, since each traces a path through the code from evaluated expression to evaluated expression. It's usually not possible to statically determine which evaluation path will be selected at run-time, so static analyses which depend on control flow must analyze and account for all possible evaluation paths.

While it's possible to implement any analysis or transformation with only abstract syntax trees, ASTs model the code's syntax rather than its control flow. This means a simple top-down or bottom-up traversal of an AST generally won't visit expressions in order of evaluation (for any evaluation path). Moreover, expressions which are consecutive in an evaluation path can be relatively distant from each other in the AST. For example, in the body of a loop, a `break` expression redirects evaluation to the first expression after the loop, which will be at a different level of the AST. As a consequence, to use the AST, a static analysis or transformation which depends on control flow must implement custom traversal logic in addition to the logic of the analysis.

This section begins with an example (Example 5) which highlights the difficulties of using the AST to implement an analysis that depends on control flow. After the example, the section introduces control flow graphs (CFGs), a representation for code which models the code's control flow, so that traversing evaluation paths is simple. The next section, Section 1.4, will use this representation as a fundamental building block for data flow analyses.

Example 5. Suppose we want to create an analysis which detects redundant calls, where the arguments have the same values as a prior call to the same function. Listing 1.5, which is an excerpt of real code to simulate water flow at Shasta Dam in California (Adams 2018), provides

an example of a redundant call. The second call to `Vdiscretize` will produce the same result as the first call:

```
1 spillRc = Vc
2 spillRw = Vw
3 spillR = spillRc + spillRw
4 spillRdisc = Vdiscretize(spillRc, spillRw)
5 spillRopts = t(Vdiscretize(spillRc, spillRw)[1:2, ])
```

Listing 1.5: A modified excerpt from a simulation of water flow at Shasta Dam (Adams 2018).

Redundant calls waste time and can also waste memory. The goal of this example is identify redundant calls; a later example, Example 9, shows how to transform the code to eliminate redundant calls.

Detecting redundant calls is complicated by the fact that calls which are syntactically identical are not always redundant, because the values of variables used as arguments can change between call sites. For instance, if `spillRc` is reassigned just before the last line of Listing 1.5, the second call to `Vdiscretize` is no longer redundant:

```
1 spillRdisc = Vdiscretize(spillRc, spillRw)
2 spillRc = spillRc + 1
3 spillRopts = t(Vdiscretize(spillRc, spillRw)[1:2, ])
```

Listing 1.6: The last two lines of Listing 1.5 with an assignment to `spillRc` inserted between them. The second call to `Vdiscretize` is no longer redundant.

As a consequence, the analysis must keep track of which variables are reassigned between calls, using information about the order in which the code will be evaluated.

We'll use the AST to implement the redundant calls analysis in order to show why a simple top-down or bottom-up traversal is not adequate for analyzing code in full generality. The analysis will initially use the `ast_find_all` function (from Section 1.2) to traverse the AST. Let `is_redundant_call` be the test function passed to `ast_find_all` to implement the logic of the analysis. The `is_redundant_call` function is actually a closure, since it will maintain a persistent list of calls, called the *available calls*, during traversal. The available calls are the calls which have already been made in the code (and therefore their result already "available" to use). The action of `is_redundant_call` depends on the class of the expression it is passed:

- `Call` expressions. Check whether the call is in the list of available calls. If it is, return `TRUE`. If it's not, append it to the list and return `FALSE`.
- `Assignment` expressions. Remove all calls which contain the assigned variable from the available calls. Return `FALSE`.
- All other expressions. Return `FALSE`.

Then the code to run this analysis on an AST `ast` is

```
ast_find_all(ast, is_redundant_call)
```

This simple analysis returns the correct list of redundant calls for the water flow simulation code in Listing 1.5 and Listing 1.6.

The results from the analysis are not always correct for code which contains control flow expressions (`if`, `for`, `while`, and `repeat`). For instance, suppose the second call to `Vdiscretize` in Listing 1.5 is in the body of a for-loop:

```
spillRdisc = Vdiscretize(spillRc, spillRw)
for (i in seq_along(spillRopts)) {
    spillRopts[[i]] = Vdiscretize(spillRc, spillRw)[i, ]
    spillRc = spillRc + 1
}
```

Listing 1.7: The last two lines of Listing 1.5 changed so that the second call to `Vdiscretize` is in the body of a for-loop. Now the second call is not redundant.

Since the assignment to `spillRc` is at the end of the loop, the analysis doesn't visit the assignment until after visiting the second call to `Vdiscretize`. Thus the analysis marks the second call to `Vdiscretize` as redundant, even though it is only redundant on the first iteration of the loop.

We can make the analysis handle the for-loop in Listing 1.7 correctly by changing how the analysis traverses the AST. Instead of using the `ast_find_all` function, we can create a custom top-down traversal (for instance, using the `children` function from Section 1.2). Then when the analysis reaches a for-loop, it can:

1. Collect the names of all variables assigned in the loop, including the *induction variable*—the counter variable defined by the loop.

2. Search the loop for redundant calls, ignoring any which have variables collected in step 1 as arguments.

The custom traversal can still use the `is_redundant_call` function to handle other expressions, provided it collects a list of all of the calls for which `is_redundant_call` returns `TRUE` (meaning the call is redundant).

Although the modified analysis handles the for-loop in Listing 1.7 correctly, it will still produce incorrect results for other control flow expressions, including control flow expressions which interact with for-loops, such as `break` and `next`. For example, consider this version of Listing 1.7 with a break expression added:

```
1 spillRdisc = Vdiscretize(spillRc, spillRw)
2 for (i in seq_along(spillRopts)) {
3     if (break_condition) {
4         spillRc = spillRc + 1
5         break
6     }
7     # ...
8     spillRopts[[i]] = Vdiscretize(spillRc, spillRw)[i, ]
9 }
```

Listing 1.8: The code from Listing 1.7 changed to include a break expression. Now the second call to `Vdiscretize` is redundant. This kind of redundancy can be difficult to spot in loops which contain a lot of code. The additional code, including code to compute the variable `break_condition`, is omitted here but indicated by the `...` comment.

Since the assignment to `spillRc` is followed by `break`, the assignment will never affect the call to `Vdiscretize` on the last line of the loop. However, the assignment will cause the analysis to fail to detect that the second call to `Vdiscretize` is redundant.

We can fix the analysis by adding another special case to handle break expressions in if-expressions, but there are many more cases the analysis must handle in order to produce correct results for general code. For instance, the current version of the analysis will not handle while-loops, repeat-loops, the `next` expression, the `return` expression, and all interactions between these expressions correctly. We could refine the analysis handle all of these cases correctly, but since the semantics of control flow expressions are the same from one analysis to the next, it's more efficient and convenient to use a code representation specifically designed for traversal

along evaluation paths. Then analysis implementors can focus on the logic of the analysis rather than how to traverse the code. □

The conclusion of Example 5 is that the AST inadequate as a basis for analyses which depend on control flow and need to correctly handle a variety of code. In spite of that, before implementing an analysis, one should weigh the simplicity of the AST against the need for correctness. As the example showed, it only takes a few lines of code to create an analysis of the AST that is correct for code with simple control flow.

The meaning of each control flow expression is the same regardless of the analysis, so it's possible to handle control flow separately from any specific analysis. This approach is convenient because then we can reuse the same representation for code and traversal tools rather than implementing something new for each analysis. It also minimizes the number of places where bugs can be introduced due to missing or incorrectly handled control flow cases.

A *control flow graph* (CFG) is a graphical representation for code which models the code's control flow. Each node in the graph is a *basic block*, an ordered sequence of expressions. Evaluation of a basic block is all-or-nothing: the entire sequence of expressions in the block is evaluated in order, or none of them are. This means basic blocks cannot contain control flow expressions. Instead, control flow expressions are represented by directed edges from the end of one basic block to the beginning of another. An if-expression creates a branch in the graph and a loop creates a cycle. The next example demonstrates a CFG.

Example 6. Consider again the code from Listing 1.8 in Example 5, which is based on code to simulate water flow at Shasta Dam (Adams 2018). The code includes a loop, if-expression, and break expression. This example presents the control flow graph for the code, in order to make the details of the CFG representation clear.

Figure 1.3 shows the control flow graph for the code in Listing 1.8. The graph has eight basic blocks, which are shown as text boxes in the figure.

The entry point to the code is basic block 1, which contains the call to the `Vdiscretize` function on line 1 of Listing 1.8. Control always flows from block 1 to block 2, as indicated by the directed edge between the blocks in the CFG.

Basic block 2 is the entry to the for-loop in Listing 1.8. In a CFG, for-loops, while-loops, and repeat-loops correspond to multiple basic blocks, some of which will be arranged in a cycle. For the loop in this code, the cycle is $3 \to 4 \to 5 \to 6 \to 3$. At run-time, each iteration of the

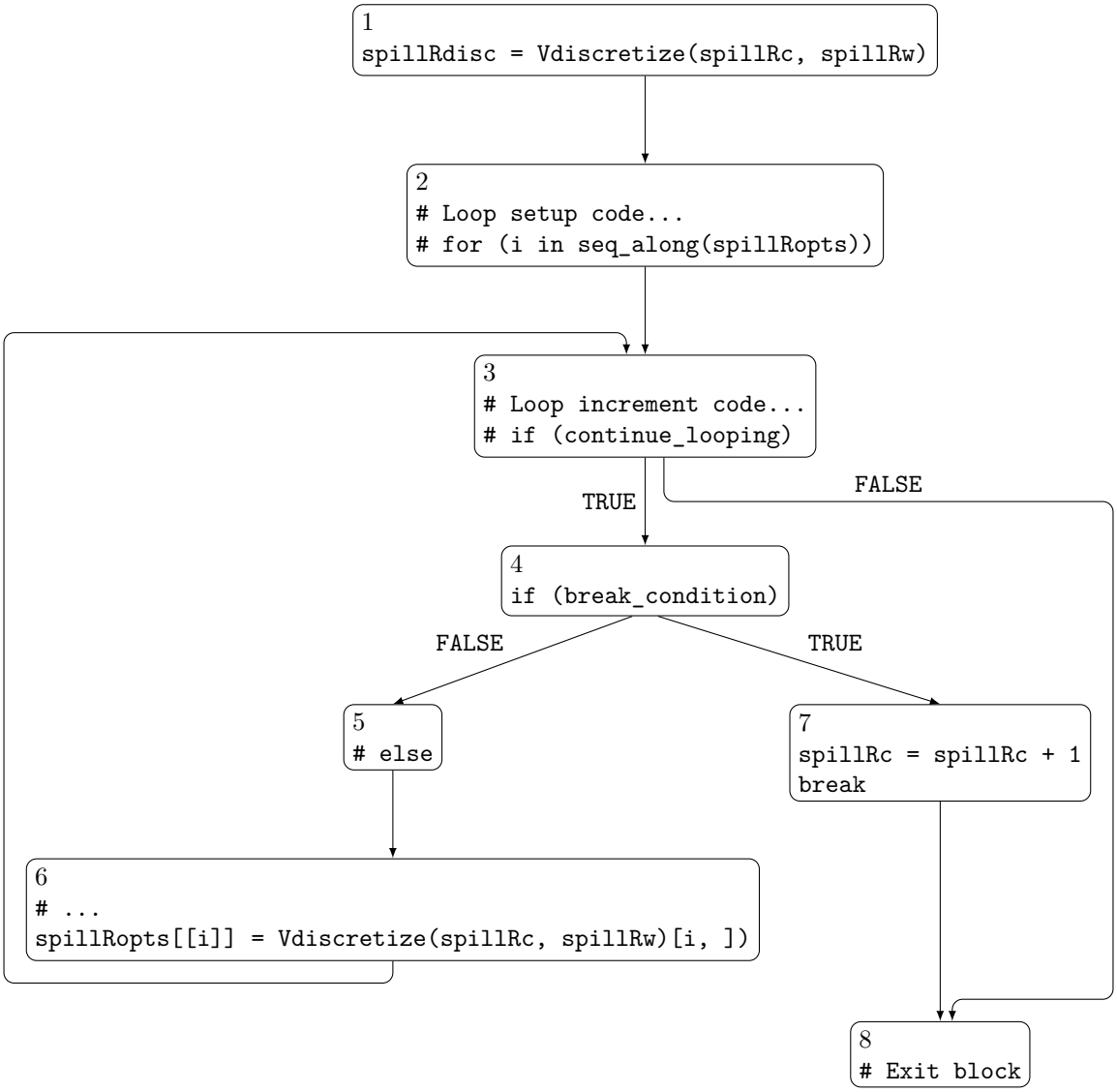

Figure 1.3: The control flow graph for the code in Listing 1.8. Code to set up and increment the loop induction variable is omitted.

loop where `break_condition` is `FALSE` corresponds to one traversal of this cycle. Besides the blocks in the cycle, the loop also includes block 2, which evaluates the vector over which the loop iterates and sets up any counters needed by the loop, and block 7, which corresponds to the body of the if-statement in the body of the loop. From block 2, control flows to block 3.

Basic block 3 checks whether there are more elements for the loop to iterate over and updates the induction variable `i` accordingly. If there are more elements to iterate over, control flows to block 4. If there are no more elements to iterate over, control flows to block 8, which is the end point for the code.

The basic block 4 is the entry point to the if-expression in Listing 1.8. If-expressions (and other tests of conditions) correspond to *branches* in the CFG, where there are two different edges out of a basic block. Since block 3 tests whether there are more elements over which the for-loop must iterate, it also has two out edges. Block 4 tests the condition `break_condition`. If the condition is `TRUE`, control flows to block 7, and if the condition is `FALSE`, control flows to block 5, so these two block correspond to the two branches of the if-expression.

Basic block 7 corresponds to the body of the if-expression in Listing 1.8. It contains the expression which increments the variable `spillRc`, and also the `break` expression. The `break` expression redirects control flow to the first expression after the for-loop. Since there aren't any expressions after the for-loop, control flows from the `break` expression directly to the end point for the code, block 8.

Basic block 5 contains the code which is evaluated when the if-expression is `FALSE`. If the if-expression in the code had an else-expression, block 5 would be the beginning of the body of the else-expression. Since the if-expression does not have an else-expression, block 5 does not contain any computations. This means that technically block 5 could be omitted from the graph, with the edges into and out of block 5 replaced by an edge from block 4 to block 6. We included block 5 in Figure 1.3 in order to make explicit both branches of the if-expression.

The basic block 6 contains the last line of the for-loop, which calls the `Vdiscretize` function to compute an element of `spillRopts`. At the end of this block, control flows back to block 3, which checks whether the loop should run for another iteration. The edge from block 6 to block 3 creates the cycle which defines the loop. This edge is called a *backedge*, since it leads back to a basic block (block 3) that's always evaluated before this block (block 6).

Finally, block 8 is the *exit block*, the end point for the code.

The **rstatic** packages provides the function `to_cfg` to compute the control flow graph for an

rstatic AST or R language object. The CFG is returned as a list of `BasicBlock` objects. Each basic block contains the expressions as `ASTNode` objects (see Section 1.2). The CFGs returned by the `to_cfg` function always have a single entry block and single exit block. □

Example 6 presented the basic concepts of control flow graphs. We'll use CFGs in Section 1.4 as the basis for analyses which extract information from code about where variables are defined and used. By using CFGs, the analyses generally do not have to handle control flow expressions as special cases during traversal. The trade-off, made apparent by the example, is that a CFG has more structural differences from the original code than an AST. This means there is a higher upfront cost analysis implementors in terms of learning how to use the CFG.

1.3.1 Related Work

The package **cyclocomp** (Csardi 2016) computes CFGs for R code in order to compute the code's *cyclomatic complexity*. Cyclomatic complexity is a measure of computational complexity based on the number of evaluation paths in the CFG which have at least one edge not shared with any other path. The CFGs the **cyclocomp** package computes do not preserve the actual expressions in the code, so they are not well-suited to code analysis which goes beyond examining the graph structure.

1.4 Analyzing How Data Flows Through Code

As code is evaluated, data "flows" or propagates from expressions which define variables to expressions which use those variables. A *data flow analysis* is a static code analysis that depends on and collects information about what and how data flows between expressions at run-time.

For instance, a simple data flow analysis question is: given a variable `x` which is reassigned several times and an expression `mean(x)`, which value assigned to `x` will reach the expression at run-time? This kind of question is non-trivial for most code because most code contains control flow expressions which change the flow of data depending on run-time conditions. This means that in order to produce correct results, a data flow analysis must consider all of the possible evaluation paths for the code. As a consequence, data flow analyses are often formulated as analyses of control flow graphs.

Two widely-used techniques to implement data flow analyses are *iterative data flow analysis* and *static single assignment* (SSA) form (Cooper and Torczon 2012). The **rstatic** package

provides tools which facilitate using either technique. As of writing, **rstatic** is the only package we're aware of which provides this functionality for R (see Section 1.4.3 for related work). The type inference strategy presented in Chapter 2 uses the SSA form generated by **rstatic**.

In the iterative approach, the data flow analysis problem is framed as computing a set of values, expressions, or properties for each basic block in a CFG. Each set is defined by an equation which relates it to the sets of adjacent basic blocks. The sets are computed by assigning them approximate initial values and then iteratively updating each set using the equations until the sets converge. Section 1.4.1 describes iterative data flow analysis in more detail.

The static single assignment form is a form of the CFG in which each variable name uniquely refers to a single assignment. This means that for any expression which uses a variable, no additional analysis is necessary to locate the corresponding assignment. Section 1.4.2 presents the details of the SSA form.

1.4.1 Iterative Data Flow Analyses

Many data flow analyses can be framed as computing a set of information about each basic block in the control flow graph or each expression in the code. For instance, the redudant calls analysis of Example 5 can be framed as trying to compute, at each expression, the set of all calls which are already available (that is, have already been computed). Then a call is redundant if it appears in its own set of available calls. Iterative data flow analysis is a general strategy to compute such sets, whether they contain values, expressions, or other information.

The main benefit of iterative data flow analysis is that one can use it to solve multiple data flow analysis problems without having to develop, implement, and maintain several different algorithms, each with their own idiosyncrasies. Some examples are:

- Reaching definitions analysis, which addresses the question posed in the second paragraph of Section 1.4: given an expression that uses a variable `x`, which values assigned to `x` can `x` have at that expression? This analysis is necessary to construct *use-definition chains*, a mapping from each use of a variable to all of its possible definitions, and to construct the static single assignment form described in Section 1.4.2. The results from this analysis have many other uses as well, such as determining which variables depend on each other and determining the possible dimensions and shapes for a variable in a given expression.

- Live variables analysis, which, at each expression, determines which variables are *live*, or

necessary to compute subsequent expressions. Once a variable is no longer live, there's no reason to keep its value in memory. One can use the results from this analysis to transform code so that memory is freed as soon as variables are no longer needed.

Cooper and Torczon (2012) and Nielson et al. (2010) describe additional analyses which can be implemented with iterative data flow analysis.

Iterative data flow analysis can operate at the level of basic blocks or individual expressions. We will explain the strategy at the level of basic blocks. In order to use the strategy at the level of expressions, one can construct a control flow graph where each expression is placed in its own basic block; the edges in such a CFG have the same interpretation as in an ordinary CFG. It is also possible to use the strategy at the level of basic blocks and then propagate the result sets to the level of expressions (Nielson et al. 2010).

The result set for a basic block characterizes run-time properties which hold immediately before or immediately after the block is evaluated, depending on the specific analysis. The core of the iterative data flow analysis strategy is the update or *transfer* equation, which relates the result set for one block to the result sets of adjacent blocks in the CFG. The transfer equation for basic block b with result set $\textsc{Result}(b)$ has a general form for many analyses:

$$\textsc{Result}(b) = \bigsqcup_{a \in A_b} \textsc{Gen}(a) \cup (\textsc{Result}(a) \setminus \textsc{Kill}(a)) \tag{1.1}$$

Depending on the analysis, the operation $\sqcup$ can be a union or intersection, and the set of basic blocks A_b can contain all predecessors or all successors of b. The sets $\textsc{Gen}(a)$ and $\textsc{Kill}(a)$ are computed from the block a alone and also depend on the analysis. The next example demonstrates the result sets and specific transfer equation for a reaching definitions analysis.

Example 7. Consider the following code to convert temperatures to Kelvin:

```
1 to_kelvin = function(temperature, unit) {
2     if (unit == "fahrenheit")
3         temperature = (temperature - 32) * 5 / 9
4     temperature + 273.15
5 }
```

Listing 1.9: A function to convert Fahrenheit or Celsius temperatures to Kelvin.

Figure 1.4 shows the CFG for the code. For each basic block b, the result set we want to compute is ReachesIn(b), the set of definitions which can apply to, or *reach*, each variable at the beginning of the block. So for this analysis, the result set for a block characterizes properties which hold immediately before the block is evaluated.

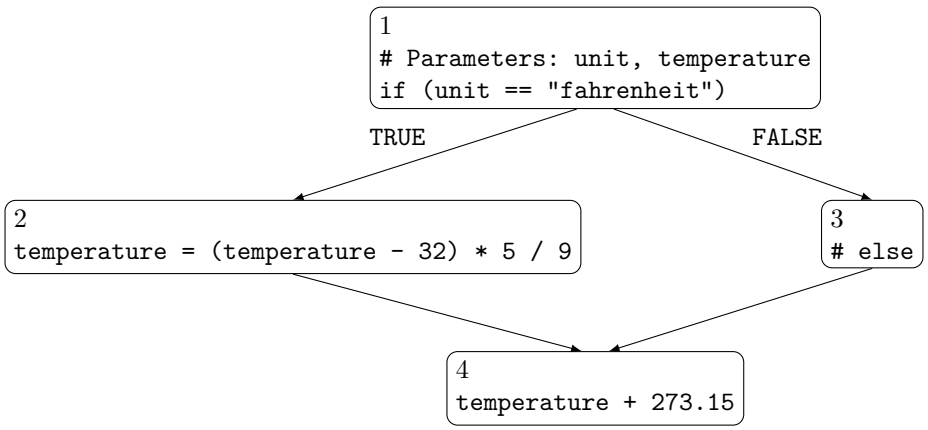

Figure 1.4: The control flow graph for the code in Listing 1.9.

Consider basic block 1. The parameters `temperature` and `unit` are implicitly defined when the function is called, so their definitions should be in the result set for this block. We'll generally record definitions as assignment expressions, but for these two implicit definitions, we'll simply record the parameter names. No other parameters or variables are already defined at the beginning of block 1. Thus we can conclude:

$$\text{ReachesIn}(1) = \{\texttt{temperature}, \texttt{unit}\}$$

Now consider the sets ReachesOut(b) of definitions which can reach each variable at the end of a block b rather than the beginning. We'll need these sets to compute the result sets. There are two different ways a definition can reach a variable at the end of a block:

1. The definition is the last definition of that variable in the block. Let the set LastDef(b) contain the last definition of each variable in the block b.

2. The definition reaches the variable at the beginning of the block, and the variable is not redefined in the block. Let the set Replaced(b) contain definitions which are replaced by new definitions in the block.

For instance, since there are no definitions within block 1,

$$\textsc{LastDef}(1) = \emptyset$$

$$\textsc{Replaced}(1) = \emptyset.$$

On the other hand, `temperature` is redefined in block 2, so

$$\textsc{LastDef}(2) = \{\texttt{temperature = (temperature - 32) * 5 / 8}\}$$

$$\textsc{Replaced}(2) = \{\texttt{temperature}\},$$

where `temperature` in the second set refers to the parameter's implicit definition.

We can translate the discussion of the preceding paragraphs into an equation:

$$\textsc{ReachesOut}(b) = \textsc{LastDef}(b) \cup (\textsc{ReachesIn}(b) \setminus \textsc{Replaced}(b))$$

From this equation, it follows that:

$$\textsc{ReachesOut}(1) = \{\texttt{temperature}, \texttt{unit}\}$$

That is, the implicit parameter definitions from the beginning of block 1 reach the end of block 1, because block 1 does not redefine either parameter.

The definitions which reach a variable at the beginning of a block b are exactly the definitions which reach that variable at the end of all of that block's immediate predecessors. This observation leads to the transfer equation for the reaching definitions analysis:

$$\begin{aligned}\textsc{ReachesIn}(b) &= \bigcup_{a \in \text{pred}(b)} \textsc{ReachesOut}(a) \\ &= \bigcup_{a \in \text{pred}(b)} \textsc{LastDef}(a) \cup (\textsc{ReachesIn}(a) \setminus \textsc{Replaced}(a))\end{aligned}$$

Compare this to Equation 1.1. For a reaching definitions analysis:

$$\begin{aligned}
\sqcup &:= \cup \\
A_b &:= \text{pred}(b), \text{ the immediate predecessors of block } b \\
\textsc{Gen}(b) &:= \textsc{LastDef}(b) \\
\textsc{Kill}(b) &:= \textsc{Replaced}(b)
\end{aligned}$$

These will differ for other analyses.

Now we can compute the results set for block 2:

$$\begin{aligned}
\textsc{ReachesIn}(2) &= \bigcup_{a \in \{1\}} \textsc{ReachesOut}(a) \\
&= \textsc{ReachesOut}(1) \\
&= \{\texttt{temperature}, \texttt{unit}\}
\end{aligned}$$

This is also the result set for block 3, since blocks 2 and 3 have the same predecessors in the CFG. Since `temperature` is redefined in block 2, it follows that:

$$\begin{aligned}
\textsc{ReachesOut}(2) &= \textsc{LastDef}(2) \cup (\textsc{ReachesIn}(2) \setminus \textsc{Replaced}(2)) \\
&= \{\texttt{temperature = (temperature - 32) * 5 / 8}, \texttt{unit}\}.
\end{aligned}$$

There are no definitions in block 3, so:

$$\begin{aligned}
\textsc{ReachesOut}(3) &= \textsc{LastDef}(3) \cup (\textsc{ReachesIn}(3) \setminus \textsc{Replaced}(3)) \\
&= \emptyset \cup (\textsc{ReachesIn}(3) \setminus \emptyset) \\
&= \{\texttt{temperature}, \texttt{unit}\}.
\end{aligned}$$

Finally, we can compute the result set for block 4:

$$\begin{aligned}
\textsc{ReachesIn}(4) &= \bigcup_{a \in \{2,3\}} \textsc{ReachesOut}(a) \\
&= \textsc{ReachesOut}(2) \cup \textsc{ReachesOut}(3) \\
&= \{\texttt{temperature = (temperature - 32) * 5 / 8}, \texttt{temperature}, \texttt{unit}\}
\end{aligned}$$

In words, at the beginning of block 4, the value of `temperature` can come from either the implicit parameter definition or the definition in block 2. The value of `unit` will come from the implicit parameter definition. This completes the example. □

Now that we've seen an example of result sets and a transfer equation, we turn to the iterative algorithm for computing the result sets. Before using the algorithm, it's necessary to determine several details specific to the analysis:

1. The Result sets:
   - What properties the elements represent.
   - Whether they correspond to the beginning or end of each block. If they correspond to the beginning, then the analysis is called a *forward analysis* and the transfer equation uses immediate predecessors, $A_b = \text{pred}(b)$. If they correspond to the end, then the analysis is called a *backward analysis* and the transfer equation uses immediate successors, $A_b = \text{succ}(b)$.
2. The Gen sets. For a block b, the set $\textsc{Gen}(b)$ contains properties which come into effect in that block.
3. The Kill sets. For a block b, the set $\textsc{Kill}(b)$ contains properties which cease to be in effect in that block.
4. The $\sqcup$ operation. If a property is in effect for block b when it's in effect for any adjacent block in A_b, then $\sqcup = \cup$. If a property is in effect for block b only when it's in effect for all adjacent blocks in A_b, then $\sqcup = \cap$.

Listing 1.10 shows the iterative data flow analysis algorithm. The initial values for the result sets depend on the analysis, but are generally the empty set $\emptyset$ or the set Ω of all possible result set elements. For analyses where $\sqcup = \cap$, the sizes of the result sets monotonically decrease with each iteration, so the algorithm always terminates. For analyses where $\sqcup = \cup$, the sizes of the result sets monotonically increase with each iteration, so the algorithm always terminates provided that Ω is finite. Nielson et al. (2010) prove these properties of the algorithm.

1. Compute the GEN and KILL sets for each block.

2. Set initial values for the result sets.

3. While `changed`:

    a) `changed = FALSE`

    b) For each block b in the CFG:

        i. Use the transfer equation to compute the result set for block b.

        ii. If the new result set differs from the old result set, `changed = TRUE`.

Listing 1.10: The iterative data flow analysis algorithm (Cooper and Torczon 2012).

In the reaching definitions analysis in Example 7, we computed the result sets in a single iteration. This is because for each result set we computed, we first computed all of the result sets on which it depends. For code which does not contain any loops, such as the code from that example (Listing 1.9), the CFG will not contain any cycles, so it's possible to compute the result sets in this order. If the result sets are computed in a different order or the code does contain loops, then more iterations are usually necessary.

The next example provides an overview of how to use the **rstatic** package and iterative data flow analysis to implement a static analysis.

Example 8. The redundant calls analysis from Example 5 is a version of a classic data flow analysis called *available expressions analysis*. An available expressions analysis finds the set of available expressions—expressions computed by the preceding code—at the beginning of each basic block. The goal of this example is to use the iterative data flow analysis algorithm to implement an available calls analysis (in R, most expressions which compute a result are calls). By using the algorithm, the analysis will handle all forms of control flow correctly and only require a few lines of custom code to implement.

The transfer equation for the available calls analysis is:

$$\textsc{AvailIn}(b) = \bigcap_{a \in \text{pred}(b)} \textsc{AvailGen}(a) \cup (\textsc{AvailIn}(a) \setminus \textsc{AvailKill}(a))$$

We want to use the results from the analysis to determine which calls are redundant, so we need to know which calls are available at the beginning of each block. Thus this is a forward analysis

and $A_b = \text{pred}(b)$. In order for a call to be available at the beginning of a block, it must be available by the end of each of the block's immediate predecessors, so $\sqcup = \cap$.

For a block b, AvailKill(b) is the set of all calls which are no longer available at the end of the block. A call is no longer available if some of its arguments are variables, and those variables are defined in the block. One way to compute this set is to start from the set Ω of all calls in the code and then remove all calls which do not contain variables defined in the block. The code to compute Ω for a CFG named `cfg` is:

```
all_calls = ast_find_all(cfg, is, "Call")
```

Next, we need to remove the calls which do not contain variables defined in the block. The **rstatic** helper function `ast_defs` returns a list of all variables defined in an AST, CFG, block, or expression. The **rstatic** helper function `contains` checks whether each expression in one list contains (possibly as subexpressions) any of the expressions in another. Thus the code to compute AvailKill(b) for one block `block` is:

```
defs = ast_defs(block)
avail_kill = all_calls[contains(all_calls, defs)]
```

Listing 1.11: The code to compute the AvailKill set for one block in the CFG.

We can apply this code to each block in the CFG in order to compute all of the AvailKill sets.

For a block b, AvailGen(b) is the set of all calls made in the block which are still available at the end of the block. One way to compute the set is expression by expression. When an expression redefines a variable, any calls which use that variable are removed. When an expression makes a call, that call is added to the set. The code to compute AvailGen(b) for one block `block` is:

```
avail_gen = list()
for (expr in block$contents) {
    # Remove any calls killed by this expression.
    defs = ast_defs(expr)
    avail_gen = avail_gen[!contains(avail_gen, defs)]
    # Add all calls made by this expression.
    calls = ast_find_all(expr, is, "Call")
    avail_gen = append(avail_gen, calls)
}
```

Listing 1.12: The code to compute the AvailGen set for one block in the CFG.

We can apply this code to each block in the CFG in order to compute all of the AvailGen sets.

The initial values for the AvailIn sets depend on the block. For the entry block, no calls are available before the block is evaluated because it's the entry block. Thus the initial (and final) result set for the entry block is the empty set $\emptyset$. For all other blocks, we can use Ω, the set of all calls, as the initial result set. Since $\sqcup = \cap$, the iterative data flow analysis algorithm will shrink these result sets to their actual values.

The **rstatic** function `dfa_solve` is an implementation of the iterative data flow analysis algorithm. Suppose `cfg` is the CFG to analyze, `initial` is the list of initial result sets, `avail_kill_list` is the list of kill sets, and `avail_gen_list` is the list of gen sets. Then the code to compute the result sets with the iterative data flow analysis algorithm is:

```
1 avail_in = dfa_solve(cfg,
2     initial = initial, universe = all_calls,
3     kill = avail_kill_list, gen = avail_gen_list,
4     forward = TRUE, meet = "intersect")
```

Listing 1.13: The code to compute the AvailIn result sets for the available calls analysis.

The parameter `forward` controls whether the sets A_b in the transfer equation contain predecessors or successors. The parameter `meet` controls whether the operation $\sqcup$ in the transfer equation is union or intersection.

We can test the analysis with the Shasta dam water flow simulation code from Listing 1.8 in Example 5. Figure 1.5 shows the control flow graph for this code. Table 1.2 shows the AvailKill and AvailGen sets for each block. Table 1.3 shows the AvailIn result sets computed by the iterative data flow analysis algorithm. This version of the analysis returns correct results regardless of which control flow expressions the code contains and how they are arranged, unlike the redundant calls analysis in Example 5. While it was necessary to understand CFGs and the iterative data flow algorithm in order to implement this version of the analysis, the actual code to implement the analysis with **rstatic** is short and straightforward. □

Note that the results of an iterative data flow analysis at the level of basic blocks can be propagated to the level of expressions. Suppose block b contains n_b expressions indexed $e = 1, \ldots, n_b$. Compute Kill and Gen sets for each expression. Call these sets $\text{KillExp}_b(e)$

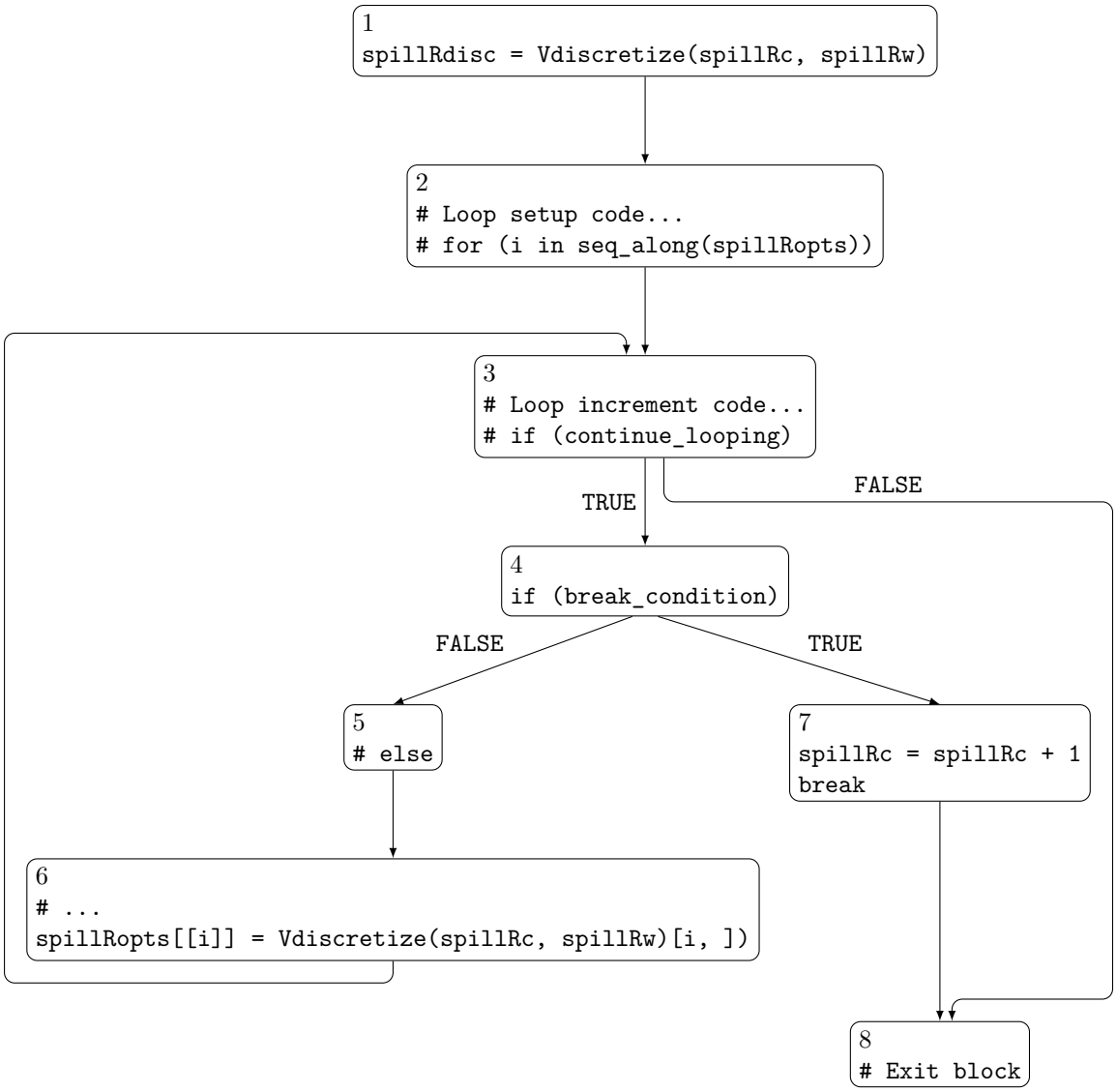

Figure 1.5: The control flow graph for the code in Listing 1.8 from Example 5. This figure is the same as Figure 1.3.

b	AvailKill(b)	AvailGen(b)
1	–	`Vdiscretize(spillRc, spillRw)`
2	`Vdiscretize(spillRc, spillRw)[i, ]`	`seq_along(spillRopts)`
3	`Vdiscretize(spillRc, spillRw)[i, ]`	–
4	–	–
5	–	–
6	`seq_along(spillRopts)`	`Vdiscretize(spillRc, spillRw)` `Vdiscretize(spillRc, spillRw)[i, ]`

7	`Vdiscretize(spillRc, spillRw)`
	`Vdiscretize(spillRc, spillRw)[i, ]`
	`spillRc + 1`
		`spillRc + 1`

8	–	–

Table 1.2: The AvailKill and AvailGen sets for the available calls analysis on the CFG in Figure 1.5. The call to `seq_along` is evaluated in block 2 as part of the loop setup code.

b	AvailIn(b)
1	–
2	`Vdiscretize(spillRc, spillRw)`
3	`Vdiscretize(spillRc, spillRw)`
	`Vdiscretize(spillRc, spillRw)[i, ]`
4	`Vdiscretize(spillRc, spillRw)`
5	`Vdiscretize(spillRc, spillRw)`
6	`Vdiscretize(spillRc, spillRw)`
7	`Vdiscretize(spillRc, spillRw)`
8	`spillRc + 1`

Table 1.3: The AvailIn sets for the available calls analysis on the CFG in Figure 1.5.

and $\text{GenExp}_b(e)$, respectively. Then use the block-level result set and the transfer equation to compute:

$$\begin{aligned} \text{ResExp}_b(0) &= \text{Result}(b) \\ \text{ResExp}_b(e) &= \text{GenExp}_b(e-1) \cup (\text{ResExp}_b(e-1) \setminus \text{KillExp}_b(e-1)) \end{aligned} \tag{1.2}$$

The ResExp_b sets are the expression-level result sets. This process can be repeated for each block in the CFG.

The next example shows one way to use the results from an iterative data flow analysis to transform code.

Example 9. The motivation for the available calls analyses in Example 8 (and originally Example 5) is to identify redundant calls and transform the code to eliminate them. The strategy to transform the code is to create a variable the first time a call is made, and then replace all subsequent, redundant calls with that variable. The goal of this example is to give an overview of how to implement such a transformation for the code reperesented by the CFG in Figure 1.5, using **rstatic** and the AvailIn sets from Example 8.

For the code in Figure 1.5, a call is redundant if it is in the AvailIn set for its block. For instance, the first call to `Vdiscretize` is in block 1. The AvailIn(1) set is empty (see Table 1.3), so the call is not redundant. On the other hand, the second call to `Vdiscretize` is in block 6 and is redundant, because the call is in the AvailIn(6) set.

There are two different cases for how to transform calls:

1. If the call is not redundant, create a new variable and insert an assignment expression to store the result of the call in the new variable. Replace the call with the new variable.

2. If the call is redundant, replace the call with the variable created in case 1.

Creating a new variable in case 1 rather than reusing existing variables when possible ensures that the variable will not be reassigned at some point later in the code. It's possible that the new variable will never be used as a replacement anywhere, but because of R's copy-on-write semantics, the run-time and memory cost of creating unnecessary variables is minimal. It's also possible to have a subsequent analysis and transformation eliminate unnecessary variables.

The transformation is simpler to implement if we create names for the new variables beforehand. Let `all_calls` be the set of all calls in the code, which can be computed with the `find_nodes` function. Then compute a vector `new_vars` of new variable names—one for each call—making sure they do not conflict with any variable names in the code.

We can use the **rstatic** function `block_transform` to apply the transformation. The `block_transform` function applies a transformation function to each top-level expression in each block in the CFG. The transformation function must have four parameters:

- `expr`, the expression to transform.
- `index`, the index of the expression within the block.
- `block_index`, the index of the block within the CFG.
- `...`, for passing additional arguments

The transformation function must return an expression or list of expressions to replace `expr` in the block.

Let `replace_redundant` be the transformation function. The set `all_calls` of all calls in the code, the vector `new_vars` of new variable names, and the `avail_in` sets from Example 8 are additional arguments to the function. So the code to remove redundant calls from a CFG (after carrying out an available calls analysis) is:

```
1 block_transform(cfg, replace_redundant,
2   all_calls = all_calls,
3   avail_in = avail_in,
4   new_vars = new_vars)
```

The first thing the `replace_redundant` function does is create a list of results and extract all of the calls in the expression `expr`:

```
result = list(expr)
calls = find_nodes(expr, is, "Call")
```

Next, the function loops over each call. For each call, it computes the position of the call in the `all_calls` set. It does this using the **rstatic** helper function `match_object`, an implementation of R's `match` function for lists of objects:

```
for (call in calls) {
  m = match_object(call, all_calls, 0L, match_fun = `==`)
```

Next, the function creates a new Symbol object for the variable and replaces the call with the new object. This uses the **rstatic** function `replace_in`, which replaces a subexpression in an expression:

```
  new_var = Symbol$new(new_vars[m])
  replace_in(call$parent, call, new_var)
```

Finally, the function checks whether the call is in the `avail_in` set for the block. If it's not, the call is not redundant, so the function creates an assignment expression to assign the result of the call to the new variable:

```
  avail_in_b = avail_in[[block_index]]
  m = match_object(call, avail_in_b, 0L, match_fun = `==`)
  if (m != 0) {
    assignment = Assignment$new(copy(new_var), copy(call))
    result = append(result, assignment, 0)
  }
}
```

The function returns `result`, a list of top-level expressions. The `block_transform` function replaces `expr` in the block with these expressions. This completes the transformation. □

The available calls analysis of Example 8 and transformation of Example 9 show one way to remove redundant calls which works correctly regardless of the control flow expressions in the code being analyzed. Achieving this goal was made simpler by separating the analysis from the transformation and by using the iterative data flow analysis algorithm to compute the analysis results. The **rstatic** package provides an implementation of the algorithm and also many helper functions which facilitate implementing analyses and transformations such as this one.

1.4.2 The Static Single Assignment Form

Static single assignment (SSA) form is a form of the control flow graph where variables are renamed so that they satisfy two properties (Cooper and Torczon 2012):

1. Each definition of a variable has a unique name (no redefinitions).
2. Each use of a variable refers to a single, specific definition.

The purpose of these properties is to eliminate ambiguity about the flow of data in code. The SSA form makes explicit which definitions of a variable reach each use, and conversely, which uses of a variable depend on each definition. These relationships, called use-definition chains, characterize which expressions in the code depend on each other and how. This information is relevant for:

- Transformations which rearrange or parallelize expressions.
- Inferences about values of variables, such as constant propagation, type inference, and dimension inference.
- Making sense of unfamiliar code, as a human reader.

The SSA form is an essential component of the type inference strategy described in Chapter 2.

As we explained in Section 1.4, a variable can have multiple definitions. The definition that reaches expressions which use the variable depends on which evaluation path in the CFG is taken at run-time. However, the SSA form requires that each use refers to a single definition. The SSA form addresses this by introducing an artificial function called φ (denoted by `PHI` in code). The φ-function selects the appropriate version of a variable based on the evaluation path taken at run-time. φ-functions are always placed at the beginning of a block, so that there is no ambiguity about variables used within the block. The order of the φ-functions at the beginning of a block doesn't matter, since each only interacts with one variable. Example 10 shows how the SSA form uses φ-functions in practice.

The SSA form is strictly a tool for analyses and transformations. We will never run code in SSA form, so an implementation of the φ-function is not necessary. After analyses and transformations that use SSA form are complete, we can transform the code back into an ordinary CFG, AST, or parse tree.

This subsection presents three examples. The first example introduces the SSA form, while the other two show applications.

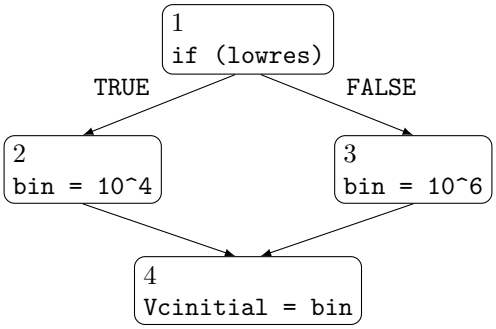

Figure 1.6: The control flow graph for the code in Listing 1.14.

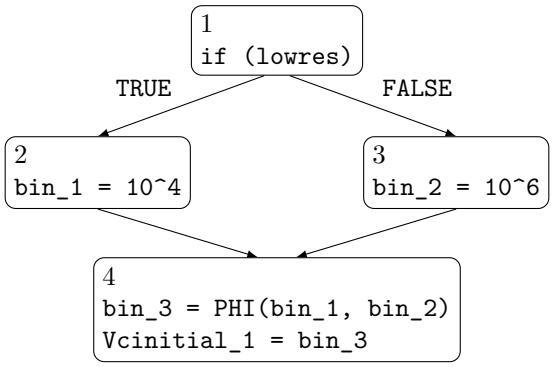

Figure 1.7: The SSA form of the control flow graph for the code in Listing 1.14.

Example 10. This example demonstrates the SSA form and the role of φ-functions. Consider this initialization code from the Shasta Dam water flow simulation (Adams 2018):

```
1 if (lowres)
2   bin = 10^4
3 else
4   bin = 10^6
5 Vcinitial = bin
```

Listing 1.14: An excerpt of the initialization code from the Shasta Dam water flow simulation (Adams 2018).

Figure 1.6 shows the CFG for the code, and Figure 1.7 shows the SSA form of the CFG for the code. The **rstatic** function `to_ssa` converts a CFG into SSA form.

In the CFG, block 2 and block 3 both contain definitions of the `bin` variable. The SSA form attaches a subscript to each of these definitions so that they can be uniquely identified. In code, we denote the subscript by appending `_n` to the end of the variable name, where `n` is replaced by the subscript. Then the assignment in block 2 defines `bin_1`, and the assignment in block 3 defines `bin_2`.

The expression in block 4 uses the `bin` variable, but the definition of `bin` in block 4 depends

on which evaluation path the code takes at run-time. If the condition in the if-statement is `TRUE`, then the definition is `bin_1`. If the condition is `FALSE`, then the definition is `bin_2`.

The SSA form uses a φ-function to indicate that the definition depends on which evaluation path is taken. The φ-function will return `bin_1` if block 2 is evaluated before block 4 at run-time, and will return `bin_2` if block 3 is evaluated before block 4 at run-time. The SSA form creates a new subscripted variable `bin_3` for the result of the φ-function. This variable replaces `bin` in block 4.

For the remainder of this chapter, we display SSA forms of code as code listings rather than CFGs. For example, the SSA form as a code listing for Listing 1.14 is:

```
1 if (lowres)
2   bin_1 = 10^4
3 else
4   bin_2 = 10^6
5 bin_3 = PHI(bin_1, bin_2)
6 Vcinitial_1 = bin_3
```

Listing 1.15: The SSA form of the code from Listing 1.14.

We do this to make the chapter text easier to read, with the understanding that static code analyses which use the SSA form use the SSA form of the CFG. □

The φ-function embodies a point in the code where there is more than one possible definition for a variable. The evaluation path taken at run-time determines which definition reaches subsequent uses. It is only necessary to add φ-functions to basic blocks in a CFG with more than one incoming edge. For instance, block 4 in Example 10 has two two incoming edges because the if-expression in the code.

The next example shows an application of the SSA form.

Example 11. An *alias* is a variable assigned the value of another variable. The purpose of this example is to show that the SSA form facilities creating an analysis which detects aliases. Detecting aliases is useful because static analyses can share information between aliases if they are known to be aliases.

Aliases are created by assignments where both the left and right-hand side are variables. Thus one way to find all assignment expressions which create aliases in a CFG `cfg` is:

```
is_alias = function(node) {
    is(node, "Assignment") && is(node$write, "Symbol") &&
        is(node$read, "Symbol")
}
aliases = ast_find_all(cfg, is_alias)
```

Listing 1.16: An analysis to find assignment expressions which create aliases.

If two variables are aliases for one another and one is reassigned, they are no longer aliases in the subsequent code. For instance, in this code based on the Shasta Dam water simulation (Adams 2018), the variables `k` and `Vcinitial` are aliases for one another until line 5:

```
k = 4.5 * 10^6
Vcinitial = k
Vinitial = Vcinitial + Vwinitial
# ...
k = 0.2 * 10^3
MaxV = 1.5 * (Vcinitial + Vwinitial)
```

Listing 1.17: A modified excerpt of the initialization code from the Shasta Dam water flow simulation (Adams 2018). The `...` comment indicates code was omitted.

For the code in Listing 1.17, the alias analysis in Listing 1.16 returns the assignment expression `Vcinitial = k`, meaning that `Vcinitial` and `k` are aliases. The analysis does not detect that they stop being aliases for one another after line 5 of Listing 1.17. We could address this by changing the analysis to provide more precise information about regions of code where aliases are active, but by instead using the SSA form we can get this information without any extra implementation effort.

The SSA form of the code in Listing 1.17 is:

```
k_1 = 4.5 * 10^6
Vcinitial_1 = k_1
Vinitial_1 = Vcinitial_1 + Vwinitial_1
# ...
k_2 = 0.2 * 10^3
MaxV_1 = 1.5 * (Vcinitial_1 + Vwinitial_1)
```

Listing 1.18: The SSA form of the code in Listing 1.17. The `...` comment indicates code was omitted.

For this code, the result of the alias analysis in Listing 1.16 is the assignment expression `Vcinitial_1 = k_1`, meaning `Vcinitial_1` and `k_1` are aliases. Since the SSA form does not contain redefinitions, these variables are aliased for the entirety of the code. In other words, using the SSA form ensures that the results from the alias analysis are applicable to all of the code, thus making them simpler for subsequent analyses to use. □

The main benefit of the SSA form for the alias analysis in Example 11 is that it gives a unique identifier to every definition and does not allow redefinitions. This property is useful for most analyses which collect information about values of variables, because they can use the identifiers to unambiguously associate information with the values of variables at specific points in the code.

The next example demonstrates another application of the SSA form, one which foreshadows how we use the SSA form for type inference Chapter 2.

Example 12. A variable is *constant* if its value is the same every time the code is evaluated. If a variable is constant and we can determine its value with static analysis, then we can transform the code, replacing all instances of the variable with its value. This transformation is called *constant propagation*.

Constant propagation is beneficial because literal values are more informative than variables—their type, dimensions, and value-dependent properties (whether the value is the missing value, for example) are all exposed. Subsequent static analyses can use this information, as can human readers trying to understand the code. This example discusses two cases where the SSA form facilitates constant propagation.

As the first case, consider this version of the initialization code for the Shasta Dam water flow simulation (Adams 2018):

```
1 bin = 10^6
2 Vcinitial = bin
3 # ...
4 if (Vcinitial >= bin)
5   Vwinitial = 0
6 else
```

```
7     Vwinitial = 10^4
8   Vinitial = Vcinitial + Vwinitial
```

Listing 1.19: An excerpt of the initialziation code for the Shasta Dam water flow simulation (Adams 2018), with modifications. The `...` comment indicates omitted code.

The variable `Vwinitial` is assigned different values depending on the condition in the if-expression. Since condition is written in terms of constants, we can see that the condition will always be `TRUE`. The code in the body of the `else` expression is *dead code*, code that never runs. We want the constant propagation analysis to detect that the `else` expression is dead code, so `Vwinitial` is constant.

As the second case, consider a different version of the code where the if-expression depends on a setting loaded from a file:

```
1   settings = readRDS("default_settings.rds")
2   bin = 10^6
3   # ...
4   if (settings[["lowres"]])
5     Vwinitial = 10^6 - bin
6   else
7     Vwinitial = 0
8   Vinitial = Vcinitial + Vwinitial
```

Listing 1.20: A version of the code from Listing 1.19 where the condition is not constant, but `Vwinitial` is still constant.

In this case, a static analysis can't determine whether the if-expression's condition will be `TRUE` or `FALSE` at run-time. Nevertheless, since `bin` is the constant `10^6`, we can see that `Vwinitial` is assigned the same value on both branches of the if-statement. This is another scenario we'd like our constant propagation analysis to be able to handle. The analysis should compare the potential assignments for each variable. If they are all the same, then the variable is a constant, as `Vwinitial` is here.

Both cases are addressed by the *sparse conditional constant propagation* (SCCP) algorithm (Wegman and Zadeck 1991). The algorithm uses the SSA form of the CFG to identify blocks of dead code and to determine where different assignments of a variable have the same value. Without the SSA form for R code, we would not be able to use this algorithm. We won't describe the entire algorithm here, but the remainder of this example discusses how the algorithm uses

the SSA form to handle the two cases.

Consider the SSA form of the code from Listing 1.19:

```
1 bin_1 = 10^6
2 Vcinitial_1 = bin_1
3 # ...
4 if (Vcinitial_1 >= bin_1)
5   Vwinitial_1 = 0
6 else
7   Vwinitial_2 = 10^4
8 Vwinitial_3 = PHI(Vwinitial_1, Vwinitial_2)
9 Vinitial_1 = Vcinitial_1 + Vwinitial_3
```

Listing 1.21: The SSA form of the code from Listing 1.19.

The SCCP algorithm propagates constants from defintions to uses. The φ-function on line 8 specifies that the value of `Vwinitial_3` can be either `Vwinitial_1` or `Vwinitial_2`, depending on which evaluation path is selected at run-time. Both `Vwinitial_1` and `Vwinitial_2` are constants, so the algorithm can propagate their values into the φ-function:

```
7 Vwinitial_3 = PHI(0, 10^4)
```

Similarly, since `bin_1` and `Vcinitial_1` are constants, the SCCP algorithm can propagate their values into the condition for the if-expression:

```
3 if (10^6 >= 10^6)
```

Then the algorithm can deduce that the condition is `TRUE`, so the code in the body of the if-expression will always be evaluated rather than the code in the body of the else-expression. Thus the φ-function will always return `0`, and the analysis determines that `Vwinitial_3` is constant.

Now consider the second case, in Listing 1.20. The SSA form of the code is:

```
1 settings_1 = readRDS("default_settings.rds")
2 bin_1 = 10^6
3 # ...
4 if (settings_1[["lowres"]])
5   Vwinitial_1 = 10^6 - bin_1
```

```
6 else
7   Vwinitial_2 = 0
8 Vwinitial_3 = PHI(Vwinitial_1, Vwinitial_2)
9 Vinitial_1 = Vcinitial_1 + Vwinitial_3
```

Listing 1.22: The SSA form of the code from Listing 1.20.

Again the SCCP algorithm propagates constants from definitions to uses. Since `bin_1` is a constant, the assignment on line 4 becomes:

```
4 Vwinitial_1 = 0   # 10^6 - 10^6
```

Thus `Vwinitial_1` is a constant, and propagating its value to the φ-function on line 7 produces:

```
7 Vwinitial_3 = PHI(0, 0)
```

Since both of the values in the φ-function are the same, the value of `Vwinitial_3` is the same no matter which evaluation path is selected at run-time. Thus the analysis can conclude that `Vwinitial_3` is constant.

By using the SSA form, the SCCP algorithm is simple and handles control flow expressions correctly, even when they are nested or contain multiple assignment expressions. Also note that when the code is in SSA form, we can think of constant propagation as an analysis which maps as many SSA names as possible to literal values. The resulting mapping can be used to transform the code by substitution, but can also be used without transforming the code. As in the alias analysis in Example 11, the fact that the SSA form provides names for specific assignments rather than for variables is crucial to avoid ambiguity. □

The SSA form makes the results of a reaching definitions analysis explicit in the CFG, so that subsequent analyses can use this information. In particular, analyses can check which definitions are associated with an expression which uses a variable, or which expressions depend on a given definition. Since definitions have unique names in the SSA form, the form also makes it simple to associate information with the value of a variable over a particular region of code, even if that variable is reassigned in the code.

1.4.3 Related Work

The **CodeDepends** package (Temple Lang, Peng, et al. 2018) provides tools for analyzing dependencies between blocks of R code. The package was the primary inspiration for **rstatic**,

and there is some overlap in functionality. The centerpiece of the **CodeDepends** package is the function `getInputs`, which analyzes expressions to determine their inputs and outputs (in terms of variables, files, and side effects). While **rstatic** provides functions to find variables that are defined by or used in expressions, they are more narrowly focused than the **getInputs** function. The other functions in the **CodeDepends** package are thorough implementations of specific code analyses. Many of these can be built from the functions in **rstatic** (and some are examples in this chapter), but the focus of **rstatic** is on providing reusable operations for building code analyses, rather than providing complete analyses.

The R Optimizations with Static Analysis (**ROSA**) project (Sen et al. 2017) is a research project that investigates applying textbook compiler optimizations to R code. Many of these optimizations rely on information collected through data flow analyses, similar to the data flow analyses provided in **rstatic**. Unlike **rstatic**, **ROSA** is not an R package, is not developed in R, and is not intended to facilitate experimentation with and development of new analyses by the R community. The **ROSA** software can also compile a subset of R code to C++ code that manipulates R objects.

1.5 Conclusion

This chapter presented a framework for static analysis and transformation of R code. The framework is based on modern code analysis techniques and implemented in the package **rstatic**. The framework provides strategies for handling three different characteristics of code:

- For analyzing the code's syntax and structure, the abstract syntax tree is the simplest and most appropriate representation to use. The AST is also a building block for all other analysis techniques. The **rstatic** package provides its own hierarchy of classes to represent ASTs. These classes make it possible to access the parent of any expression in the AST and to attach arbitrary metadata to expressions. The classes are organized into a hierarchy based on the meanings of expressions, so that analysis implementors can use generic functions and specialized methods to handle specific kinds of expressions. The **rstatic** package also provides several functions, such as `ast_find_all`, for searching and transforming ASTs.

- For analyzing properties of the code which depend on control flow, the control flow graph is a more appropriate representation than the AST. Each node in the graph is a basic

block, a sequence of expressions which do not affect control flow. Each edge in the graph represents control flow from one basic block to others. Consequently, the edges of the CFG are an explicit representation of the different ways in which the code can be evaluated at run-time. This is convenient for traversing the expressions in the code in order of evaluation. Most analyses which depend on control flow also depend on data flow, so we generally do not use the CFG on its own, but instead as a foundation for data flow analysis techniques. The **rstatic** package provides data structures and functions to convert R code to and from CFG form.

- For analyzing how data flows between expressions, we presented two different approaches, both based on the CFG. The iterative data flow analysis algorithm is a general algorithm for computing sets of information about the basic blocks or expressions in a CFG. The **rstatic** function `dfa_solve` implements the algorithm for R code. The static single assignment form is a form of the CFG which explicitly indicates which definitions of variables can reach uses of those variables. The key feature of the SSA form is that it gives a unique name to each variable definition, so that there is no ambiguity in expressions which use variables. The SSA form is useful for analyses which collect information related to the values of variables or how expressions depend on one another. The **rstatic** package provides functions to convert R code to SSA form.

This framework is useful for a wide variety of different analyses, such as detecting and removing redundant expressions, detecting aliases, propagating constants, and determining the lifetimes of variables. It is also the foundation for the type inference strategy described in Chapter 2.

Chapter 2

Type Inference for R Code

2.1 Introduction

A *type* is a category for data values in a programming language. Values of the same type have properties in common, such as their functionality and representation in memory (Cooper and Torczon 2012). For instance, in R the values `TRUE` and `FALSE` have type `logical`, and quoted strings have type `character`.

This chapter presents a *type inference* strategy for R code. Type inference is a static analysis which examines code in order to infer a type for every expression. R does not provide a mechanism to declare or annotate the types of expressions in code, so type inference is necessary if one wants to recover the types without running the code.

The applications for types which motivate this chapter are:

- **Reasoning about and summarizing code**. A value's type characterizes the operations with which it is compatible, so type information facilitates reasoning about code. For instance, a function's *type signature* specifies the types of arguments the function accepts and the type of result the function returns. Type signatures are helpful for identifying the appropriate function to solve a problem, and for using functions and their results correctly.

- **Checking code for errors**. Code can be checked programatically for *type errors*—errors caused by applying an operation to an incompatible type of value. R checks for type errors at run-time, but inferring the types and checking for errors before run-time prevents wasting time and compute resources on erroneous code.

- **Transforming or translating code**. A value's type characterizes its size and representation (for example, integer, floating point, or double-precision floating point) in memory.

This information is necessary in order to translate code into hardware instructions, because most hardware instructions assume specific sizes and representations for their operands. Translating R code into hardware instructions is desirable because doing so can reduce run-times by orders of magnitude and makes it possible to run R code on new platforms and hardware, such as graphical processing units (Temple Lang 2014).

The first example demonstrates the basic idea of type inference.

Example 13. Suppose we want to infer the type of the variable `y` in this code to generate observations from a linear model:

```
1 b0 = 10.2
2 b1 = 4
3 n = 100
4 x = runif(n)
5 e = rnorm(n)
6 y = b0 + b1 * x + e
```

Listing 2.1: Code to generate 100 observations from a linear model.

The value of `y` is the result of the expression `b0 + b1 * x + e`. The type of this value depends on the functions the expression calls (`+` and `*`) and their arguments (`b0`, `b1`, `x`, and `e`).

Following R's order of operations, first consider the arguments in the multiplication `b1 * x`. The variable `b1` is assigned the literal value `4`. In R, the literal value `4` is a numeric vector, which corresponds to type `double` (R also has an `integer` type, but literal numbers default to the `double` type even if they are integer-valued). The variable `x`—the other argument in the multiplication—is assigned the result of the call `runif(n)`. The `runif` function always returns a numeric vector. Thus `b1 * x` returns a numeric vector, since the multiplication operator `*` returns a numeric vector if both of its operands are numeric vectors.

Next, consider the expression `b0 + b1 * x`. The variable `b0` is assigned the literal value `10.2`, which is a numeric vector. Thus `b0 + b1 * x` returns a numeric vector, since the addition operator `+` also returns a numeric vector if both of its operands are numeric vectors.

Finally, consider again the entire expression `b0 + b1 * x + e`. The variable `e` is assigned the result of the call `rnorm(n)`. Like the `runif` function, the `rnorm` function always returns a numeric vector. It follows that the entire expression `b0 + b1 * x + e` returns a numeric vector. Therefore the inferred type for `y` is `double`.

In addition to inferring the type of `y`, the analysis can also infer the dimensions. The sequence of steps is the same, but each step depends on the restrictions expressions place on dimensions rather than types. For instance, the variable `b1` is a scalar since it is assigned the scalar `4`. The variable `x` is a vector with `n` elements, because the length of the value returned by `runif` is equal to the value of its first argument. By R's rules for vectorized operations, it follows that the result of `b1 * x` is a vector with length `n`. Continuing this analysis eventually leads to the conclusion that `y` is a vector with length `n`.

Furthermore, the variable `n` is assigned the literal value `100`, so the analysis can infer than `y` has length `100`. If `n` was not assigned a literal value, it would not be possible to determine the exact length of `y`, but the symbolic length `n` is nevertheless an insight into how `y` is related to other variables such as `x` and `e` (which also have length `n`). □

The design of the type inference strategy described in this chapter is guided by several observations—the first three of which were alluded to in Example 13:

- Type inference depends on knowledge about the restrictions functions place on types—especially R's built-in functions. A baseline of knowledge must be built into the type inference strategy. Additional knowledge can be collected programatically by analyzing function definitions. The type inference strategy described in this chapter can only analyze functions written in R, but Chapter 3 presents a type inference strategy for calls from R to routines written in C.

- Combining information across the entirety of code can reveal properties that are not evident from analyzing expressions in isolation. The type inference strategy described in this chapter combines information by using a constraint system based on the investigations of Heeren et al. (2002) into type inference for ML code.

- The steps to infer dimensions are similar to the steps to infer types, to the extent that the two can be done simultaneously as components of one analysis. The type inference strategy described in this chapter only infers dimensions for expressions where the dimensions depend on a single value or variable, but can be extended to handle more complex cases.

- The type and classes of an R object are both necessary for the applications described at the beginning of this chapter, because these provide a richer characterization of an object than the type alone. In particular, R makes extensive use of *generic functions* which dispatch

specialized methods based on the classes of their arguments. The type inference strategy described in this chapter is designed to infer classes in addition to types where possible.

Here's how the remainder of this chapter is organized:

- Section 2.2 describes conventions for how code is represented in the chapter and during type inference. In particular, the type inference strategy depends on the static single assignment (SSA) form for R code introduced in Chapter 1.
- Section 2.3 describes the Damas-Milner type inference strategy (Damas and Milner 1982), which is the foundation for our own type inference strategy.
- Section 2.4, our primary contribution, describes cases where the original Damas-Milner type inference strategy is insufficient to infer types for R code, identifies whether static type inference is still possible, and where it is, suggests how to adapt the strategy.
- Section 2.5 is a brief overview of two R packages we created for type inference on R code. Subsection 2.5.1 describes the **typesys** package, which provides data structures to represent types and functions to compute on them. Subsection 2.5.2 describes the **RTypeInference** package, a prototype implementation of the type inference strategy described in this chapter.
- Section 2.6 describes related work by other researchers.

2.2 Background on How We Represent Code

The type inference strategy described in this chapter is designed to operate on the static single assignment (SSA) form of R code. The SSA form was introduced in Chapter 1; see Section 1.4.2 for an overview of the SSA form and its trade-offs compared to other representations for code.

The **rstatic** package, which was also introduced in Chapter 1, provides data structures and functions to convert R code into SSA form. The package also provides a function to convert from SSA form back into ordinary R code given that no transformations were applied to the SSA form. Simple algorithms to convert from SSA form back into ordinary code even if transformations were applied exist (Cooper and Torczon 2012), and one of these could be implemented for the **rstatic** package if necessary.

The motivation for using SSA form is that it explicitly indicates which definitions can reach a variable at any given point in the code. To see why this matters, consider that a variable can be redefined at any point in R code. If the code also contains if-statements and loops, which definition will reach the variable can depend on run-time conditions. For example, in the code below, the definition of the variable `problems` depends on the value of `is_converged`:

```
1 problems = NULL
2 if (!is_converged) {
3   is_problem = sapply(fit$Rhat, max, na.rm = TRUE) > 1.1
4   problems = names(fit$Rhat)[is_problem]
5 }
6 result = list(activity = activity, problems = problems)
```

This example is simple, so it's apparent that either definition (line 1 or line 4) could reach `problems` at line 6. In general, a non-trivial static analysis is necessary to programatically determine which definitions can reach each variable at each point in code. This analysis is one of the analyses carried out when code is converted into SSA form, and the results are explicitly indicated in the form.

In other words, in SSA form, each variable corresponds to a single definition. Where the R code redefines a variable, the SSA form gives the variable a new name. Where two or more definitions can reach a variable in the R code, the SSA form inserts a new variable defined by a call to the φ-function. The call to the φ-function indicates all definitions which can reach that variable. As a result, the type of a variable in SSA form is the type of the definition. If the definition is a φ-function, the type is a combination of the types of the arguments (this is defined more precisely in Section 2.4).

The SSA form also provides a simple way to handle one of R's unique features: *replacement functions*. A replacement function is a function which returns a modified copy of its first argument and has a name ending with `<-`. For instance, the `length<-` function returns a copy of its first argument with the length changed (based on other arguments). R provides syntactic sugar for calling replacement functions, so the code to call `length<-` on a vector `x` and replace `x` with a copy that has length 8 is:

```
1 length(x) = 8
```

Listing 2.2: Syntactic sugar to call the `length<-` function. The value of `x` is replaced with the result of the call, a copy of `x` which has length 8.

According to the *R Language Definition* (R Core Team 2019a), the expression is equivalent to:

```
1 `*tmp*` = x
2 x = "length<-"(`*tmp*`, value = 8)
3 rm(`*tmp*`)
```

Listing 2.3: The meaning of the code in Listing 2.2.

In SSA form, there's no need for the temporary variable `*tmp*`. The replacement expression can be written as:

```
1 x_2 = "length<-"(x_1, value = 8)
```

Listing 2.4: The SSA form of the code from Listing 2.2.

The SSA form makes it explicit that the value of the variable `x` is changed by the replacement expression. This is important for type inference, since replacement expressions can change the type, class, or dimensions of an object.

Where relevant, the examples in this chapter indicate whether or not the code is in SSA form.

2.3 The Damas-Milner Type Inference Strategy

The type inference strategy for R is based on Damas-Milner (DM) type inference, a type inference strategy invented by Damas and Milner for the ML programming language (1982). We chose DM type inference as a foundation because ML and R are both functional programming languages and have several characteristics in common. In both, most objects do not have reference semantics (where modifying the object affects all variables which refer to the object) and most functions do not have side effects. This section describes the DM type inference strategy, in preparation for the modifications presented in Section 2.4.

Heeren et al. (2002) recommend dividing the DM type inference strategy into two steps: a *constraint generation* step where information is collected from individual expressions, and a *constraint resolution* step where the collected information is combined in order to draw conclusions about the types in the code. The main benefits of this division are that the type inference algorithm can provide more precise information about which expressions cause type errors and that it simplifies the implementation by separating concerns. Subsection 2.3.1 describes the constraint generation step, and Subsection 2.3.2 describes the constraint resolution step.

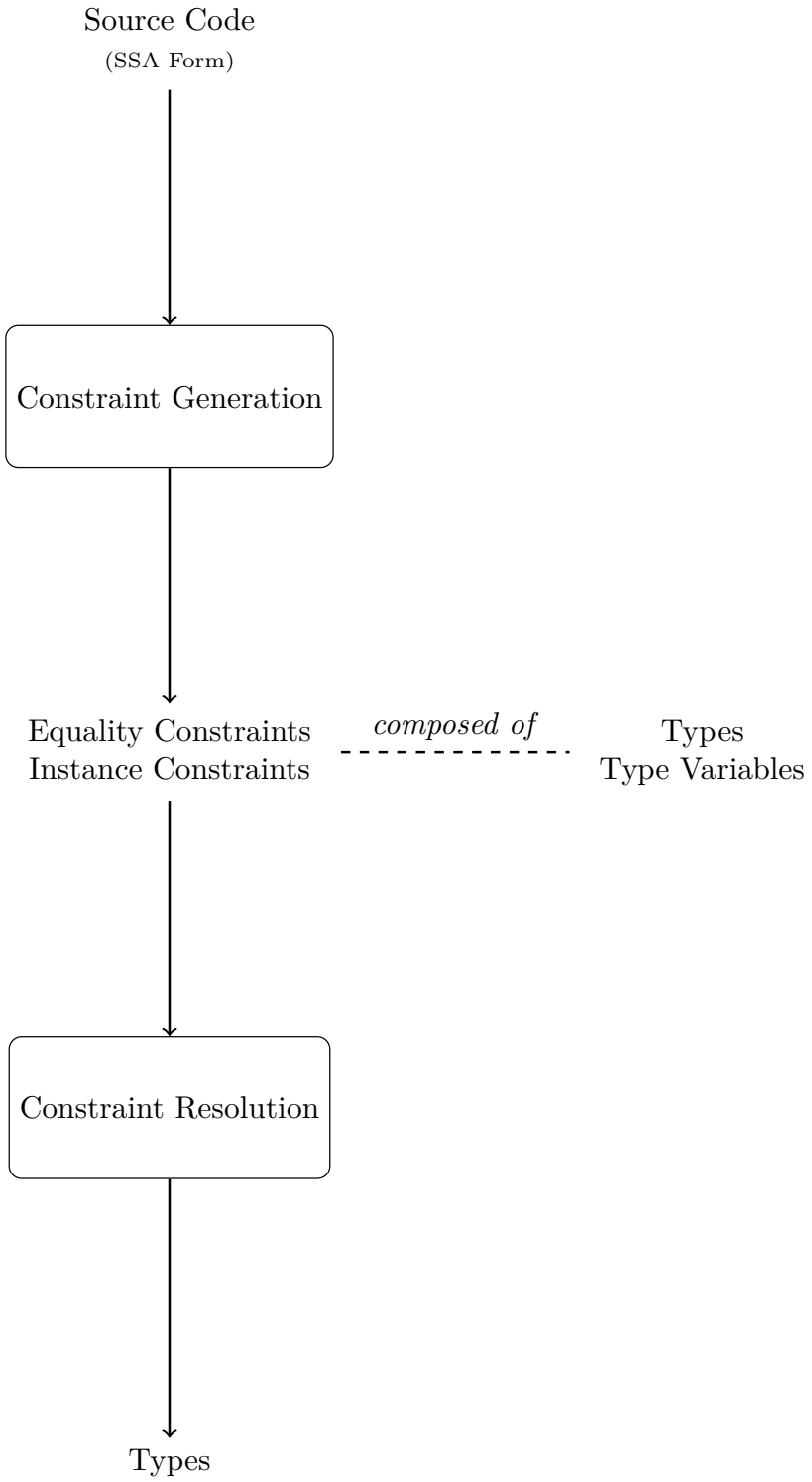

Figure 2.1: The type inference strategy is composed of two steps: constraint generation and constraint resolution. The constraint generation step takes source code in SSA form as input, generates a type variable for each non-literal expression in the code, and generates constraints on those type variables based on how the results of expressions are used in subsequent code. The constraints are written in terms of types and type variables. The constraint resolution step unifies each constraint to find a substitution of type variables so that the constraint is satified. The end result is a type for the result of each expression.

2.3.1 Constraint Generation

The goal of type inference is to determine the type of the result of every expression (including subexpressions) in the code. At the beginning of inference, these types are unknown for all expressions except literal values. As a placeholder for this unknown information, the type inference strategy generates a *type variable* for each expression that isn't a literal value.

Type variables are analogous to mathematical variables—just as one solves for the values of variables in a mathematics problem, the type inference strategy solves for the values (which are types) of type variables. In this chapter, type variables are denoted by lowercase Greek letters, and the notation `e` : τ means the result of the expression `e` has type τ. An *assumption set* records which expression in the code corresponds to each type variable.

The constraint generation step of type inference analyzes each expression in the code in order to determine constraints on the types of results the expressions can return without causing an error. For instance, the condition in an if-expression must return a logical scalar or else the code will raise an error. Constraints are recorded in terms of the type variables which correspond to the expressions.

This section is divided into two subsections. The first explains the different kinds of constraints produced by the constraint generation step, while the second explains the rules for generating the constraints from expressions.

The Kinds of Constraints

The type inference strategy uses two different kinds of constraints: equality constraints and instance constraints. This subsection explains these two kinds of constraints.

An *equality constraint*, denoted $\alpha \equiv \beta$, means that the type α must equal the type β. The next example demonstrates an equality constraint.

Example 14. Consider the expression `if (y > 4) y else y^2` and suppose the type inference strategy generated the type variable τ as a placeholder for the type of the result of `y > 4`. We denote this by `y > 4` : τ. The condition in an if-expression must be a logical scalar, so based on the if-expression, the strategy can generate a constraint $\tau \equiv$ `integer`.

Note that $\tau \equiv$ `integer` is a constraint on τ, not a definition of τ. If the code contains type errors, then τ can also appear in other, contradictory constraints. □

Equality constraints are too restrictive for constraining the type signatures of functions. The

type inference strategy can infer a type signature for a function based on its definition and can also infer a type signature for a function based on a call. The strategy can combine information from these two sources by putting a constraint between them. However, an equality constraint would mean that the function only accepts the types of arguments in the call.

There are many functions which can accept different argument types at different call sites. These functions are called *polymorphic* functions. For example, R's `length` function is polymorphic. A function can be polymorphic in one or multiple arguments.

In addition to the use already described, the type inference strategy uses type variables in the type signatures of functions to represent arguments that can have any type. This is a natural result of how the type inference strategy works. If a function's definition doesn't put any restrictions on the type of an argument, then applying the type inference strategy to the definition will return a type variable for that argument. In other words, the type inference strategy will not be able infer a type for the argument because there are no restrictions on its type. The next example shows how the strategy uses type variables to represent the type signatures of polymorphic functions.

Example 15. The `length` function is polymorphic and always returns an integer, so it can be typed as $\alpha \to$ `integer`. The arrow $\to$ denotes a *function type*, which maps a list of argument types (here just the single argument type α) to a return type (`integer`). □

Instead of equality constraints, the type inference strategy uses *instance constraints* to combine information from calls and function definitions. An instance constraint means that one type is an instance of another—a one-way relationship. During constraint resolution, information in an instance constraint is propagated from the type signature for the definition to the type signature for the call, but not vice-versa. This way the strategy can accommodate polymorphic functions that have different type signatures at different call sites.

A constraint that type α is an *instance* of a type β is denoted $\alpha \leq \beta$. During constraint resolution, information is passed one way from β to α by replacing the type variables in β with new type variables and then treating the constraint as an equality constraint. Section 2.3.2 describes the constraint resolution step in detail.

When the code being analyzed contains nested function defintions, handling instance constraints correctly requires extra information. For example, suppose we apply type inference to this code, which is based on an example from Milner (1978):

```
1 tag_two = function(a) {
2   tag = function(x) list(x, a)
3   list(tag(1), tag("hi"))
4 }
```

During the constraint generation step, the strategy creates a type variable α for the parameter `a`. It also creates a type variable ξ for `x`. The `tag` function returns a list which contains `a`, so the type signature inferred from the definition of `tag` contains the type variable α:

$$\xi \to \texttt{list}(\xi,\ \alpha)$$

The strategy creates instance constraints for each of the two calls to `tag` on line 3. When the instance constraints are resolved during constraint resolution, it is correct to replace ξ with a new type variable for each call, since `tag` is polymorphic in `x` (so ξ can be replaced by different types at different call sites). On the other hand, it is not correct to replace α with a new type variable for each call—α is the same type for both calls. More generally, if an instance constraint contains type variables associated with parameters of an enclosing function, they should not be replaced with new type variables during constraint resolution. Thus instance constraints are usually accompanied by a set of *monomorphic* type variables—type variables which should not be replaced during constraint resolution.

We denote the set of monomorphic type variables for an instance constraint by M or by showing the actual set. Moreover, we denote a constraint that α is an instance of β accompanied by the monomorphic set M by $\alpha \leq_M \beta$. The monomorphic sets can be computed during the constraint generation step using a simple analysis that collects the parameters of all enclosing functions for each expression (Heeren et al. 2002).

The next example reiterates why the DM type inference strategy uses instance constraints.

Example 16. Suppose two call expressions, `length(TRUE)` and `length(3i)`, are in the code being analyzed. Assign a type variable to each expression and to each `length` subexpression:

$$\begin{aligned} \texttt{length(TRUE)} : \beta_1 \quad \text{and} \quad \texttt{length} : \tau_1 \\ \texttt{length(3i)} : \beta_2 \quad \text{and} \quad \texttt{length} : \tau_2 \end{aligned} \tag{2.1}$$

Based on the first call, `length` is a function which accepts a single logical argument. Based on the second call, `length` is a function which accept a single complex argument. We can represent

this information with two equality constraints:

$$\tau_1 \equiv \texttt{Logical} \to \beta_1 \quad \text{and} \quad \tau_2 \equiv \texttt{Complex} \to \beta_2 \tag{2.2}$$

Suppose we also know that `length` is a polymorphic function with type $\alpha \to$ `integer`, where the type variable α indicates that the function accepts any type—so α is not an unknown for which to solve, but will be replaced with a specific type at each call site. We could (incorrectly) try to use this information by generating two more equality constraints:

$$\tau_1 \equiv \alpha \to \texttt{integer} \quad \text{and} \quad \tau_2 \equiv \alpha \to \texttt{integer} \tag{2.3}$$

The problem is that (2.2) and (2.3) together are contradictory, since they imply $\alpha =$ `Logical` but also $\alpha =$ `Complex`. Instance constraints provide enough flexibility to avoid a contradiction:

$$\tau_1 \leq_\emptyset \alpha \to \texttt{integer} \quad \text{and} \quad \tau_2 \leq_\emptyset \alpha \to \texttt{integer} \tag{2.4}$$

During constraint resolution, each instance constraint is rewritten as an equality constraint, and the type variables on the right-hand side are replaced with new type variables. So the two constraints in (2.4) become:

$$\tau_1 \equiv \alpha_1 \to \texttt{integer} \quad \text{and} \quad \tau_2 \equiv \alpha_2 \to \texttt{integer} \tag{2.5}$$

Then $\alpha_1 =$ `Logical` and $\alpha_2 =$ `Complex`, so there is no contradiction. The type inference strategy can't generate the constraints in (2.5) from the beginning, because it's not always possible to determine which types correspond to polymorphic functions before constraint resolution. □

Example 16 assumes the code being analyzed is not part of a function definition, so the set $M = \emptyset$ for both constraints in Statement 2.4. Example 18 demonstrates code where constraint generation produces instance constraints with a non-empty set M.

The Constraint Generation Rules

During constraint generation, expressions are analyzed from last to first. Heeren et al. (2002) describe constraint generation rules for five different kinds of expressions. Here are the five rules, adapted to R's syntax but otherwise left as-is:

1. **Literal Values**. Do not generate any new type variables or constraints. The type of the result of the expression is the type of the literal value.

2. **Symbols** (including variable names, parameters, and function names). Generate a new type variable τ for the expression and add it to the assumption set. The type of the result of the expression is τ.

3. **Calls**. Recursively apply the constraint generation rules to the subexpressions. Suppose τ is the type variable generated for the called function and $\alpha_1, \ldots, \alpha_n$ are the type variables generated for the arguments. Generate a new type variable β for the call, and add it to the assumption set. Generate a new equality constraint:

$$\tau \equiv (\alpha_1, \ldots, \alpha_n) \to \beta \tag{2.6}$$

   In words, the type of the called object must be a function type which accepts the argument types and returns the return type. The type of the result of the expression is β.

4. **Assignments**. Recursively apply the constraint generation rules to the subexpression on the right-hand side. Suppose τ is the type variable generated for the expression on the right-hand side. For each type variable τ_e in the assumption set which corresponds to an expression e of the symbol on the left-hand side, generate a new instance constraint:

$$\tau_e \leq_{M_e} \tau \tag{2.7}$$

   Where M_e contains all type variables which correspond to parameters at expression e in the code. The type of the result of the assignment expression is τ.

5. **Function Definitions**. Recursively apply the constraint generation rules to the body of the function. Suppose n is the number of parameters and β is the type variable generated for the result of the function. For each parameter from $p = 1, \ldots, n$:

   a) Generate a new type variable α_p and add it to the assumption set.

   b) For each type variable τ_e in the assumption set which corresponds to an expression e of the parameter p, generate a new equality constraint:

$$\tau_e \equiv \alpha_p \tag{2.8}$$

Note that this step is similar to the rule for assignments but generates equality constraints instead of instance constraints (so the type of a parameter is not an instance of another type, so parameters cannot be polymorphic functions).

The type of the result of the function definition is $(\alpha_1, \ldots, \alpha_n) \to \beta$.

The next example shows how the type inference strategy applies rules 1–4 to R code. Example 18 shows how the type inference strategy applies rule 5 to R code.

Example 17. Recall the code from Listing 2.1 in Example 13, which generates observations from a linear model:

```
1 b0 = 10.2
2 b1 = 4
3 n = 100
4 x = runif(n)
5 e = rnorm(n)
6 y = b0 + b1 * x + e
```

Listing 2.5: Code to generate 100 observations from a linear model, originally shown in Listing 2.1.

The constraint generation analysis begins at the expression `y = b0 + b1 * x + e`. In the order of operations, `b1 * x` is the first call evaluated. The analysis applies the rule for symbols to the subexpressions `b1`, `x`, and `*`, generating a new type variable for each:

$$\texttt{b1} : \tau_1 \qquad \texttt{x} : \tau_2 \qquad \texttt{*} : \tau_3 \tag{2.9}$$

Next, the analysis applies the rule for calls—which has two parts—to `b1 * x`. First, the analysis generates a new type variable β_1 for the result of the call:

$$\texttt{b1 * x} : \beta_1 \tag{2.10}$$

Second, the analysis generates an equality constraint on the type (τ_3) of the called function ($*$), based on the types of the arguments (τ_1, τ_2) and return type (β_1) for the call:

$$\tau_3 \equiv (\tau_1, \tau_2) \to \beta_1 \tag{2.11}$$

The analysis proceeds to the call `b0 + (b1 * x)`; we use parentheses to indicate that the analysis already visited `b1 * x`. The analysis generates new type variables for the symbols `b0` and `+`, as well as for the result of the call:

$$\texttt{b0} : \tau_4 \qquad \texttt{+} : \tau_5 \qquad \texttt{b0 + b1 * x} : \beta_2 \tag{2.12}$$

The analysis also generates an equality constraint on the type of the called function `+`:

$$\tau_5 \equiv (\tau_4, \beta_1) \rightarrow \beta_2 \tag{2.13}$$

Next, the analysis proceeds to the call `(b0 + b1 * x) + e`. The analysis creates new type variables for `+` and `e`, as well as for the result of the call:

$$\texttt{e} : \tau_6 \qquad \texttt{+} : \tau_7 \qquad \texttt{b0 + b1 * x + e} : \beta_3 \tag{2.14}$$

Note that `+` : τ_7 does not conflict with Statement 2.12, since type variables correspond to expressions, not to R variables. The analysis also generates an equality constraint on the called function `+`:

$$\tau_7 \equiv (\beta_2, \tau_6) \rightarrow \beta_3 \tag{2.15}$$

The analysis proceeds to the overall expression `y = b0 + b1 * x + e` and applies the rule for assignments. No new type variables are generated. There are no type variables in the assumption set bound to a symbol that matches `y`, so no constraints are generated either.

Next, the analysis moves to the expression `e = rnorm(n)`. For the right-hand side of the expression, the analyses creates new type variables for the symbols and the result of the call:

$$\texttt{n} : \tau_8 \qquad \texttt{rnorm} : \tau_9 \qquad \texttt{rnorm(n)} : \beta_4 \tag{2.16}$$

The analysis also generates a constraint for the call:

$$\tau_9 \equiv \tau_8 \rightarrow \beta_4 \tag{2.17}$$

The overall expression `e = rnorm(n)` is an assignment, and in this case, `e` : τ_6 is in the assumption set (Statement 2.14), so the analysis removes τ_6 from the assumption set and generates this

instance constraint:

$$\tau_6 \leq_\emptyset \beta_4 \tag{2.18}$$

That is, the type (τ_6) of the expression `e` on line 6 of Listing 2.5 must be an instance of the type (β_4) of the expression `rnorm(n)` on line 5. The instance constraint is subject to the empty set $\emptyset$ since this code is not part of a function definition.

For the next expression, `x = runif(n)`, the analysis again applies the rule for symbols, the rule for calls, and the rule for assignments. The analysis generates these three type variables:

$$\texttt{n} : \tau_{10} \qquad \texttt{runif} : \tau_{11} \qquad \texttt{runif(n)} : \beta_5 \tag{2.19}$$

The analysis also generates these two constraints:

$$\tau_{11} \equiv \tau_{10} \to \beta_5 \qquad \tau_2 \leq_\emptyset \beta_5 \tag{2.20}$$

Next, the analysis proceeds to the expression `n = 100`. The analysis applies the rule for literal values to `100`, which has type `double`, and does not generate any new type variables or constraints. Then the analysis applies the rule for assignments to the overall expression. This leads to two instance constraints, since there are two type variables which correspond to expressions of `n`:

$$\tau_8 \leq_\emptyset \texttt{double} \qquad \tau_{10} \leq_\emptyset \texttt{double} \tag{2.21}$$

The expressions `b1 = 4` and `b0 = 10.2` follow the same pattern of analysis as `n = 100`. For each, the analysis applies the rule for literal values and the rule for assignments. This leads to two constraints:

$$\tau_1 \leq_\emptyset \texttt{double} \qquad \tau_4 \leq_\emptyset \texttt{double} \tag{2.22}$$

The constraint generation analysis is complete. Figure 2.2 shows a summary of the type variables and constraints generated. Example 19 presents the constraint resolution step for these constraints. □

Example 17 showed the application of the DM type inference rules 1–4. The last rule, rule 5, is for function definitions. The next example shows a case where the type inference strategy applies rule 5.

Example 18. Suppose code from Listing 2.1 is rewritten as a function with parameters for the

$$
\begin{aligned}
\texttt{y = b0 + b1 * x + e} &\quad \begin{cases} \texttt{b1} : \tau_1, \quad \texttt{x} : \tau_2, \quad \texttt{*} : \tau_3, \quad \texttt{b0} : \tau_4, \quad \texttt{+} : \tau_5, \quad \texttt{e} : \tau_6, \quad \texttt{+} : \tau_7 \\ \texttt{b1 * x} : \beta_1, \quad \texttt{b0 + b1 * x} : \beta_2, \quad \texttt{b0 + b1 * x + e} : \beta_3 \\ \tau_3 \equiv (\tau_1, \tau_2) \to \beta_1 \\ \tau_5 \equiv (\tau_4, \beta_1) \to \beta_2 \\ \tau_7 \equiv (\beta_2, \tau_6) \to \beta_3 \end{cases} \\
\texttt{e = rnorm(n)} &\quad \begin{cases} \texttt{n} : \tau_8, \quad \texttt{rnorm} : \tau_9 \\ \texttt{rnorm(n)} : \beta_4 \\ \tau_9 \equiv \tau_8 \to \beta_4 \\ \tau_6 \leq_\emptyset \beta_4 \end{cases} \\
\texttt{x = runif(n)} &\quad \begin{cases} \texttt{n} : \tau_{10}, \quad \texttt{runif} : \tau_{11} \\ \texttt{runif(n)} : \beta_5 \\ \tau_{11} \equiv \tau_{10} \to \beta_5 \\ \tau_2 \leq_\emptyset \beta_5 \end{cases} \\
\texttt{n = 100} &\quad \begin{cases} \tau_8 \leq_\emptyset \texttt{double} \\ \tau_{10} \leq_\emptyset \texttt{double} \end{cases} \\
\texttt{b1 = 4} &\quad \begin{cases} \tau_1 \leq_\emptyset \texttt{double} \end{cases} \\
\texttt{b0 = 10.2} &\quad \begin{cases} \tau_4 \leq_\emptyset \texttt{double} \end{cases}
\end{aligned}
$$

Figure 2.2: The type variables and constraints generated by the analysis for each expression in Listing 2.5.

covariate and the function which generates the error term. Here's the code after rewriting:

```
1 function(x, errfun) {
2   b0 = 10.2
3   b1 = 4
4   n = length(x)
5   e = errfun(n)
6   b0 + b1 * x + e
7 }
```

Listing 2.6: The code from Listing 2.5 to generate 100 observations from a linear model, rewritten as a function.

This example examines how this change in the code changes the constraints the type inference strategy generates. Since constraint generation rules 1–4 were described in detail in Example 17, we only point out differences and the application of rule 5. Figure 2.3 shows all of the generated type variables and constraints, with the differences indicated.

Before applying the rule for function definitions, the type inference strategy analyzes the code in the body of the function. As usual, the analysis proceeds from last to first expression and analyzes each expression in the order of operations. Thus constraint generation begins with the expression `b0 + b1 * x + e` (and its subexpressions), which produces the same constraints as in Example 17.

The first difference is the expression `e = errfun(n)` on line 5. The constraint generation rules for symbols, calls, and assignments all apply as seen before. The generated type variables are:

$$\texttt{n} : \tau_8 \qquad \texttt{errfun} : \tau_9 \qquad \texttt{errfun(n)} : \beta_4 \tag{2.23}$$

The original had `rnorm` : τ_9 and `rnorm(n)` : β_4. The generated constraints are:

$$\tau_9 \equiv \tau_8 \to \beta_4 \qquad \tau_6 \leq_{M_1} \beta_4 \tag{2.24}$$

Since this expression is inside of a function definition, the instance constraint on τ_6 is subject to a nonempty set M_1. The type inference strategy will determine the contents of M_1 when it analyzes the function definition.

The next difference is the expression `n = length(x)` on line 4. The type inference strategy

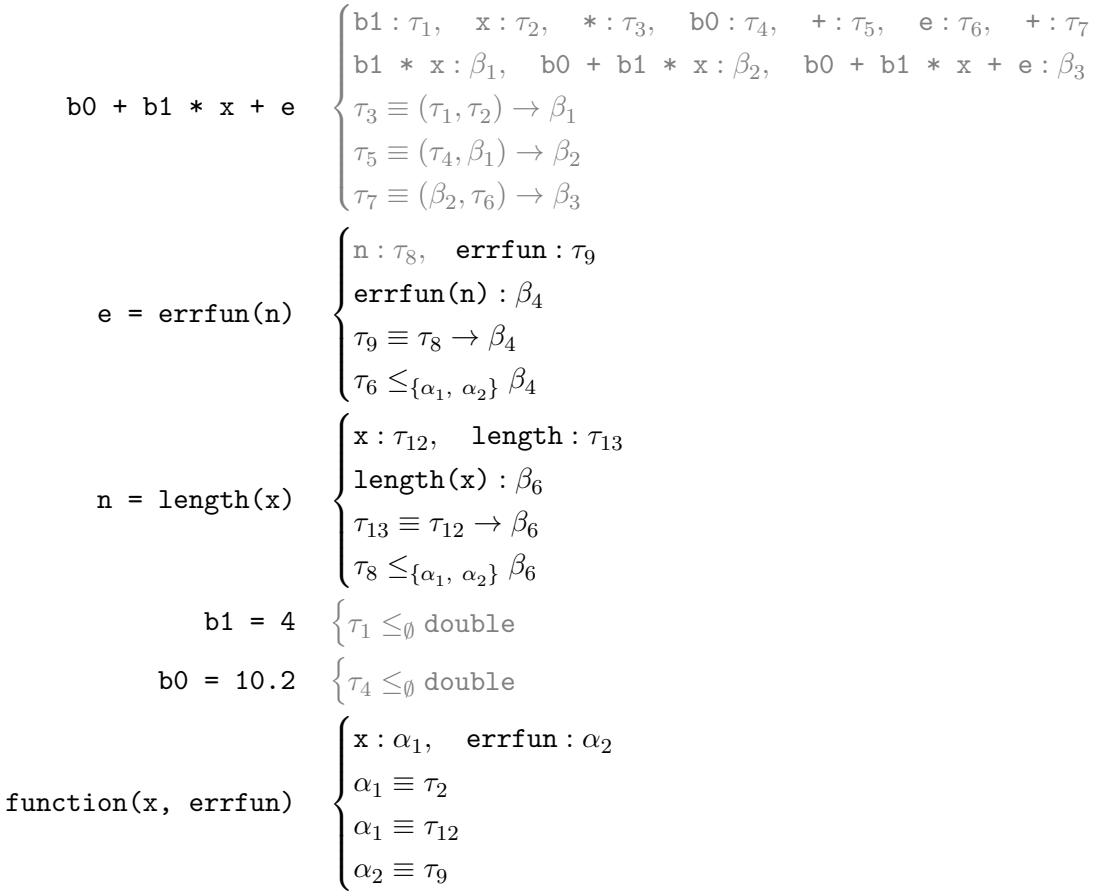

Figure 2.3: The type variables and constraints generated by the analysis for each expression in Listing 2.6. Differences in the type variables and constraints compared to Figure 2.2 are shown in black. The type variables and constraints on τ_{10}, τ_{11}, and β_5 (from a call to `rnorm`) are no longer generated.

generates three new type variables:

$$\texttt{x} : \tau_{12} \qquad \texttt{length} : \tau_{13} \qquad \texttt{length(n)} : \beta_6 \tag{2.25}$$

These type variables are named τ_{12}, τ_{13}, and β_6 to avoid confusion with τ_{10}, τ_{11}, and β_5, which appeared in Example 17 but do not appear in this example. The type inference strategy also generates these constraints:

$$\tau_{13} \equiv \tau_{12} \to \beta_6 \qquad \tau_8 \leq_{M_2} \beta_6 \tag{2.26}$$

Again, the instance constraint is subject to a set (M_2) which will be determined when the type inference strategy analyzes the function definition.

The remaining expressions in the body of the function are the same as in the original, so we do not repeat the details here. Finally, the type inference strategy analyzes the function definition `function(x, errfun)` and applies the corresponding rule. First, the strategy generates a type variable for each of the parameters:

$$\texttt{x} : \alpha_1 \qquad \texttt{errfun} : \alpha_2 \tag{2.27}$$

The type inference strategy also adds these type variables to the sets M_1 and M_2 for the two instance constraints generated for expressions in the body of the function. This ensures that α_1 and α_2 cannot be replaced with fresh type variables in the instance constraints, so they can each only have one type throughout the body of the function. Then the strategy generates an equality constraint for each type variable that corresponds to an expression of `x` (τ_2, τ_{12}) and an expression of `errfun` (τ_9):

$$\alpha_1 \equiv \tau_2 \qquad \alpha_1 \equiv \tau_{12} \qquad \alpha_2 \equiv \tau_9 \tag{2.28}$$

With that done, the constraint generation step is complete. □

2.3.2 Constraint Resolution

The constraint resolution step uses the generated constraints to *solve* the type inference problem by replacing as many type variables as possible with more specific types. A crucial part of constraint resolution is *unification*, which finds a substitution that maintains the equality of two terms containing variables while eliminating as many variables from each as possible (Robinson 1965). We denote a substitution that substitutes β for α by $\alpha := \beta$.

Constraint resolution consists of three substeps, which repeat until the constraint set is empty:

1. Unify the two terms of an eligible constraint to find a substitution.

   Equality constraints are always eligible.

   Eligibility for instance constraints is more complicated. For an instance constraint $\tau \leq_M \beta$, the type variables in β but not M are said to be *free*. Just before unification, free type variables are replaced with new type variables and the constraint is converted to an equality constraint, so that τ is unified with an instance of β rather than β itself. An instance constraint is only eligible after all other constraints on the free type variables have been unified.

   Formally, an instance constraint $\tau \leq_M \beta$ is eligible if the free variables are not *active* in any other constraints. For an equality constraint, all type variables in the constraint are active. For an instance constraint $\tau \leq_M \beta$, all type variables in τ and all type variables in both β and M (that is, the non-free type variables) are active.

2. Remove the unified constraint from the set of constraints.

3. Apply the substitution from step 1 to the remaining constraints and to the existing substitution from previous iterations.

For constraints generated by the rules listed in Section 2.3.1, this process always terminates (Heeren et al. 2002).

The substitution from constraint resolution can be applied to the assumption sets from constraint generation to see the type inferred for each expression in the code being analyzed. For expressions where the constraints don't provide enough information to determine a specific type, the inferred type is a type variable. For parameters in function definitions, leftover type variables can also indicate that the function is polymorphic in that parameter.

The next example demonstrates the constraint resolution process.

Example 19. Example 17 described the constraint generation step for code to generate

observations from a linear process (Listing 2.5). Recall that the constraints generated were:

$$\begin{aligned}
\tau_3 &\equiv (\tau_1, \tau_2) \to \beta_1 & \tau_2 &\leq_\emptyset \beta_5 \\
\tau_5 &\equiv (\tau_4, \beta_1) \to \beta_2 & \tau_6 &\leq_\emptyset \beta_4 \\
\tau_7 &\equiv (\beta_2, \tau_6) \to \beta_3 & \tau_8 &\leq_\emptyset \texttt{double} \\
\tau_9 &\equiv \tau_8 \to \beta_4 & \tau_{10} &\leq_\emptyset \texttt{double} \\
\tau_{11} &\equiv \tau_{10} \to \beta_5 & \tau_1 &\leq_\emptyset \texttt{double} \\
& & \tau_4 &\leq_\emptyset \texttt{double}
\end{aligned}$$

The goal of this example is to resolve these constraints. Figure 2.2 shows how the type variables in the constraints correspond to expressions in the code.

The code being analyzed calls the functions `*` (associated with type τ_3), `+` (τ_5 and τ_7), `rnorm` (τ_9), and `runif` (τ_{11}). The type inference strategy needs information about the type signatures of these functions in order to resolve the constraints correctly. For R's built-in functions, this information can be recorded in a table built into the type inference system. The table can be generated manually, or generated programmatically using the techniques described in Chapter 3. For functions which are not built in, this information can be collected by running type inference on their definitions. Here are the types for the four functions:

$$\begin{aligned}
\texttt{*} &: (\texttt{double}, \texttt{double}) \to \texttt{double} \\
\texttt{+} &: (\texttt{double}, \texttt{double}) \to \texttt{double} \\
\texttt{rnorm} &: \texttt{double} \to \texttt{double} \\
\texttt{runif} &: \texttt{double} \to \texttt{double}
\end{aligned}$$

Technically, all of these functions are polymorphic for specific sets of types. We address this in Section 2.4, but for now we use monomorphic type signatures to keep the focus on how constraint resolution works. For the same reason, we disregard the parameters of `rnorm` and `runif` which have default arguments.

The type inference strategy treats the additional information about the called functions as though it comes from assignments. Thus the strategy adds several more instance constraints to

the constraint set:

$$\tau_3 \leq_\emptyset (\texttt{double}, \texttt{double}) \to \texttt{double}$$

$$\tau_5 \leq_\emptyset (\texttt{double}, \texttt{double}) \to \texttt{double}$$

$$\tau_7 \leq_\emptyset (\texttt{double}, \texttt{double}) \to \texttt{double}$$

$$\tau_9 \leq_\emptyset \texttt{double} \to \texttt{double}$$

$$\tau_{11} \leq_\emptyset \texttt{double} \to \texttt{double}$$

Since equality constraints are always eligible for unification, they are a good starting point for constraint resolution. The constraint $\tau_3 \equiv (\tau_1, \tau_2) \to \beta$ leads to the substitution:

$$\tau_3 \coloneqq (\tau_1, \tau_2) \to \beta_1$$

This means replace τ_3 with $(\tau_1, \tau_2) \to \beta_1$ everywhere τ_3 appears. Now the constraint can be removed from the constraint set. After applying the substitution, the constraint set is:

$$\begin{array}{lll}
 & \tau_2 \leq_\emptyset \beta_5 & (\tau_1, \tau_2) \to \beta_1 \leq_\emptyset (\texttt{double}, \texttt{double}) \to \texttt{double} \\
\tau_5 \equiv (\tau_4, \beta_1) \to \beta_2 & \tau_6 \leq_\emptyset \beta_4 & \tau_5 \leq_\emptyset (\texttt{double}, \texttt{double}) \to \texttt{double} \\
\tau_7 \equiv (\beta_2, \tau_6) \to \beta_3 & \tau_8 \leq_\emptyset \texttt{double} & \tau_7 \leq_\emptyset (\texttt{double}, \texttt{double}) \to \texttt{double} \\
\tau_9 \equiv \tau_8 \to \beta_4 & \tau_{10} \leq_\emptyset \texttt{double} & \tau_9 \leq_\emptyset \texttt{double} \to \texttt{double} \\
\tau_{11} \equiv \tau_{10} \to \beta_5 & \tau_1 \leq_\emptyset \texttt{double} & \tau_{11} \leq_\emptyset \texttt{double} \to \texttt{double} \\
 & \tau_4 \leq_\emptyset \texttt{double} &
\end{array}$$

Constraint resolution can move on to the next constraint.

The remaining four equality constraints can be processed in exactly the same way, so that the substitution becomes:

$$\tau_3 \coloneqq (\tau_1, \tau_2) \to \beta_1$$

$$\tau_5 \coloneqq (\tau_4, \beta_1) \to \beta_2$$

$$\tau_7 \coloneqq (\beta_2, \tau_6) \to \beta_3$$

$$\tau_9 \coloneqq \tau_8 \to \beta_4$$

$$\tau_{11} \coloneqq \tau_{10} \to \beta_5$$

After these substitutions, the constraint set becomes:

$$\tau_2 \leq_\emptyset \beta_5 \qquad (\tau_1, \tau_2) \to \beta_1 \leq_\emptyset (\texttt{double}, \texttt{double}) \to \texttt{double}$$

$$\tau_6 \leq_\emptyset \beta_4 \qquad (\tau_4, \beta_1) \to \beta_2 \leq_\emptyset (\texttt{double}, \texttt{double}) \to \texttt{double}$$

$$\tau_8 \leq_\emptyset \texttt{double} \qquad (\beta_2, \tau_6) \to \beta_3 \leq_\emptyset (\texttt{double}, \texttt{double}) \to \texttt{double}$$

$$\tau_{10} \leq_\emptyset \texttt{double} \qquad \tau_8 \to \beta_4 \leq_\emptyset \texttt{double} \to \texttt{double}$$

$$\tau_1 \leq_\emptyset \texttt{double} \qquad \tau_{10} \to \beta_5 \leq_\emptyset \texttt{double} \to \texttt{double}$$

$$\tau_4 \leq_\emptyset \texttt{double}$$

Except for the first two, none of the instance constraints have free type variables, so they are eligible for unification. Consider $\tau_8 \leq_\emptyset \texttt{double}$. The type inference strategy converts this constraint to the equality constraint $\tau_8 \equiv \texttt{double}$ and then unifies the terms. The result is the substitution:

$$\tau_8 \coloneqq \texttt{double}$$

The type inference strategy applies this substitution to the existing substitution, replacing τ_8 where it appears. So the substitution becomes:

$$\tau_3 \coloneqq (\tau_1, \tau_2) \to \beta_1 \qquad \tau_8 \coloneqq \texttt{double}$$

$$\tau_5 \coloneqq (\tau_4, \beta_1) \to \beta_2$$

$$\tau_7 \coloneqq (\beta_2, \tau_6) \to \beta_3$$

$$\tau_9 \coloneqq \texttt{double} \to \beta_4$$

$$\tau_{11} \coloneqq \tau_{10} \to \beta_5$$

After repeating this process for the other three instance constraints with `double` as the second term, the substitution becomes:

$$\tau_3 \coloneqq (\texttt{double}, \tau_2) \to \beta_1 \qquad \tau_8 \coloneqq \texttt{double}$$

$$\tau_5 \coloneqq (\texttt{double}, \beta_1) \to \beta_2 \qquad \tau_{10} \coloneqq \texttt{double}$$

$$\tau_7 \coloneqq (\beta_2, \tau_6) \to \beta_3 \qquad \tau_1 \coloneqq \texttt{double}$$

$$\tau_9 \coloneqq \texttt{double} \to \beta_4 \qquad \tau_4 \coloneqq \texttt{double}$$

$$\tau_{11} \coloneqq \texttt{double} \to \beta_5$$

The remaining constraints are:

$$\tau_2 \leq_\emptyset \beta_5 \qquad (\texttt{double}, \tau_2) \to \beta_1 \leq_\emptyset (\texttt{double}, \texttt{double}) \to \texttt{double}$$

$$\tau_6 \leq_\emptyset \beta_4 \qquad (\texttt{double}, \beta_1) \to \beta_2 \leq_\emptyset (\texttt{double}, \texttt{double}) \to \texttt{double}$$

$$(\beta_2, \tau_6) \to \beta_3 \leq_\emptyset (\texttt{double}, \texttt{double}) \to \texttt{double}$$

$$\texttt{double} \to \beta_4 \leq_\emptyset \texttt{double} \to \texttt{double}$$

$$\texttt{double} \to \beta_5 \leq_\emptyset \texttt{double} \to \texttt{double}$$

Consider the first instance constraint on the left, $\tau_2 \leq_\emptyset \beta_5$. This constraint is not eligible for unification, because the type variable β_5 also appears in another constraint. The same is true for the second instance constraint on the left, $\tau_6 \leq_\emptyset \beta_4$.

On the other hand, all of the instance constraints on the right are eligible for unification, since they do not have any free type variables. Consider the first one:

$$(\texttt{double}, \tau_2) \to \beta_1 \leq_\emptyset (\texttt{double}, \texttt{double}) \to \texttt{double}$$

The type inference strategy converts this to an equality constraint, and then unifies it, matching each term on the left to one on the right. The resulting substitution is:

$$\tau_2 \coloneqq \texttt{double} \qquad \beta_1 \coloneqq \texttt{double}$$

This substitution is applied to the remaining constraints and the previous substitution. The type inference strategy repeats this process for all of the instance constranits on the right, and the final substitution becomes:

$$\tau_3 \coloneqq (\texttt{double}, \texttt{double}) \to \texttt{double} \qquad \tau_8 \coloneqq \texttt{double} \qquad \beta_1 \coloneqq \texttt{double}$$

$$\tau_5 \coloneqq (\texttt{double}, \texttt{double}) \to \texttt{double} \qquad \tau_{10} \coloneqq \texttt{double} \qquad \beta_2 \coloneqq \texttt{double}$$

$$\tau_7 \coloneqq (\texttt{double}, \texttt{double}) \to \texttt{double} \qquad \tau_1 \coloneqq \texttt{double} \qquad \beta_3 \coloneqq \texttt{double}$$

$$\tau_9 \coloneqq \texttt{double} \to \texttt{double} \qquad \tau_4 \coloneqq \texttt{double} \qquad \beta_4 \coloneqq \texttt{double}$$

$$\tau_{11} \coloneqq \texttt{double} \to \texttt{double} \qquad \tau_2 \coloneqq \texttt{double} \qquad \beta_5 \coloneqq \texttt{double}$$

The remaining constraints are:

$$\texttt{double} \leq_{\emptyset} \texttt{double} \qquad \texttt{double} \leq_{\emptyset} \texttt{double}$$

These two constraints are both satisified (that is, type inference can unify them), but do not provide any additional information. Applying the final substitution to the assumption sets in Figure 2.2 yields the inferred type for each expression in the code. □

2.4 Adapting the Type Inference Strategy to R

Damas and Milner designed the DM type inference strategy for a simplified version of the ML programming language. While ML and R have some common features (as explained in Section 2.3), they also have many differences. For instance, vectors are a primitive data structure in R.

This section discusses features of R which are not addressed by the original Damas-Milner type inference strategy or for which the strategy infers incorrect types. We propose modifications to the strategy for features which can be handled with static analysis, and point out features which cannot. The section is divided into subsections which address individual features.

2.4.1 The Grammar of Types

The type inference strategy uses a grammar of types to represent the information it collects. The grammar must be able to represent all of the types used in R. In fact, the grammar must be more expressive than R's built-in type system, because it must:

- Distinguish between scalars and vectors. Many code optimizations are specialized only for scalars or only for vectors, and can provide significant improvement in the performance of R code (Temple Lang 2014). These optimizations are a major motivation for type inference.

- Represent incomplete information. For example, if the type inference strategy can determine that an object is a vector, but not the type of its elements, that's still useful information and needs to be represented somehow.

- Represent type signatures of functions, including their argument types and return type. Section 2.3 provided an example of how the type inference strategy uses type signatures

during the constraint resolution step. Inferred type signatures are also a useful aid for code comprehension.

This section provides a brief overview of the type inference strategy's grammar of types for R. The grammar is informed by Damas and Milner's grammar of types for ML and our own observations about R and its built-in type system. Table 2.1 shows examples of the types in the grammar. Names of types in the grammar are always capitalized so that they can be distinguished from types in R's built-in type system; in cases where there's still ambiguity, we'll specify which is meant.

	Type	Associated R Type
Scalars	`Logical`	`logical`
	`Integer`	`integer`
	`Numeric`	`double`
	`Complex`	`complex`
	`String`	`character`
	`Raw`	`raw`
	`Null`	`NULL`
Data Structures	`Vector`$[\tau]$	Depends on τ
	`Matrix`$[\tau]$	Depends on τ
	`Array`$[\tau]$	Depends on τ
	`List`$[\tau_1, \ldots, \tau_n]$	`list`
	`DataFrame`$[\tau_1, \ldots, \tau_n]$	`list`
	`Environment`$[\tau_1, \ldots, \tau_n]$	`environment`
Functions	$(\alpha_1, \ldots, \alpha_n) \to \beta$	`closure`, `special`, `builtin`

Table 2.1: Examples of the types in the type inference strategy's grammar of types, and the types they correspond to in R's built-in type system. Greek letters denote parameters which can be replaced by types.

The grammar provides a set of scalar types to represent scalar versions of R's atomic types. For instance, the `Logical` type represents the type of a length-1 `logical` vector. The scalar types generally have the same name as their R counterpart, with the exception of `Numeric` (which corresponds to `double`) and `String` (which corresponds to `character`).

The grammar provides type variables as placeholders for types. A type variable is valid in any term where a type is valid. The grammar also provides *bounded type variables*, type variables for which the value is restricted to a specific set of types. Bounded type variables are necessary to represent the type signatures of many R functions. For instance, the `rnorm` function requires that its first argument is logical, integer, or numeric. The type inference strategy can also use bounded type variables to provide information about an object even if it cannot determine the exact type. As in Section 2.3, we use Greek letters to denote type variables.

The grammar represents types of data structures as *composite types*—types which contain other types as components. For instance:

- The `Vector` type represents a generic vector and has the element type as a component. Thus `Vector[Integer]` specifically represents an integer vector. If the type inference strategy determines a `Vector` has length 1, the strategy converts the `Vector` to its element type during constraint resolution.
- The `List` type represents a generic list and can have multiple components, since each element of a list can have a different type.

The composite types provided by the grammar include some data structures, such as matrices and data frames, which are characterized by their S3 class rather than their type in R's built-in type system. Section 2.4.2 explains why these are treated as types in the grammar.

The type inference strategy can represent incomplete information by using type variables as components of composite types. For example, the `Vector[`τ`]` type represents a vector with element type τ.

The grammar also represents function types as composite types, where the argument types and return type are components. For example, the `nchar` function, which counts the number of characters in each element of a character vector is:

$$(\texttt{Vector[String]},\ \texttt{String},\ \texttt{Logical},\ \texttt{Logical}) \to \texttt{Vector[Integer]}$$

The four argument types correspond to the four parameters of the function: `x`, `type`, `allowNA`, and `keepNA`.

The subsequent sections provide additional examples of how the type inference strategy uses the various types in the grammar, and introduces new terms to handle specific features of R.

2.4.2 The Relationship Between Types and S3 Classes

The *S3 class system* is the most prevalent of several object-oriented programming systems built into R. Every R object has one or more S3 classes. R uses S3 classes for *method dispatch*: when a *generic function* is called, R checks the class of one argument—by default, the first—and dispatches the call to the *method* which corresponds to that class. In other words, how a call to a generic function will be evaluated at run-time depends on the S3 class of one of its arguments.

In order to effectively analyze calls to generic functions, the type inference strategy must infer S3 classes in addition to types. This is because even though the methods of a generic function all have the same parameters, the expected argument types for the parameters can differ, as can the return types. As a result, the strategy will be able to infer more specific types if it can statically determine which method will be called at each call to a generic function.

With information about S3 classes, one can also transform code to use *static dispatch*, where calls to generic functions are replaced by direct calls to the appropriate method. This transformation reduces or eliminates the overhead of method dispatch at run-time, and is also useful for static analyses which depend on knowing the specific functions called by the code.

The type inference strategy uses two different approaches to handle S3 classes:

1. The type inference strategy treats many classes built into R as distinct types in the grammar of types. For instance, the `Matrix` and `DataFrame` types represent matrices and data frames, respectively.

   The strategy must still accept these types anywhere their underlying R type (as returned by `typeof`) would be accepted. For example, a data frame is a valid argument to a function which requires a list. The strategy can handle this by having each type store a list of types with which they can be unified.

   The advantage of this approach is that it leverages the existing capabilities of the strategy to infer types. One drawback is that different instances of an R class can have different types, and this approach will generally only infer the class, not the class and type. For classes known to have multiple types, such as the `matrix` class, this disadvantage can be mitigated by making the class a composite type in the grammar of types (see Section 2.4.1 for more about composite types). Another drawback is that each class must be individually implemented in the grammar of types.

2. The type inference strategy stores a class vector as a component of each type. When the strategy unifies two types during constraint resolution, it sets the class vector of both to the union of their class vectors. Taking the union ensures that class information is not lost during constraint resolution.

   As an example, suppose τ_1 and τ_2 are two type variables, and τ_1 has the class `queue`. If the type inference strategy makes the substitution $\tau_1 \coloneqq \tau_2$, then the class `queue` must be added to τ_2 or information is lost.

The advantage of this approach is that the type inference strategy can provide basic support for any class, even those which the strategy does not know about ahead of time. The main drawback of this approach is that by taking unions of class vectors, the strategy will not necessarily infer classes in the correct order, which matters for method dispatch. An additional drawback is that for classes inferred with this approach, the strategy does not check that the code uses the classes correctly.

During constraint generation, the type inference strategy can determine classes from literal values and constructor functions (see Section 2.4.3 for more about constructor functions) in the same way that types can be determined. The strategy must also handle cases where the code changes the class of an object.

We introduce a new form of constraint, a *class constraint*, denoted $\tau_1 \equiv_{\text{class}} \tau_2$, to represent that τ_1 and τ_2 have the same class vector. We also allow either side of the constraint to be an actual class vector. For example,

$$\tau_1 \equiv_{\text{class}} \texttt{(data.frame, list)}$$

means that the class vector of τ_1 is the vector `("data.frame", "list")`. The next example demonstrates how the strategy can use class constraints to handle code which uses the `class` function to change the class of an object.

Example 20. The `class` function gets or sets the class attribute of an R object. Consider this code (in SSA form) to create an object with the custom S3 class `counter`:

```
1 count_1 = c(a = 3, b = 2)
2 # class(count) = "counter"
3 count_2 = "class<-"(count_1, value = "counter")
```

Listing 2.7: Code to set the class of an object to `counter`. See Section 2.2 for more about how replacement expressions are represented in SSA form.

We can handle this by changing how the type inference strategy handles calls to the `class<-` function during constraint generation. Suppose the type inference strategy generates the type variable β_1 such that

$$\texttt{"class<-"(count\_1, value = "counter")} : \beta_1.$$

Then we can modify the strategy to retrieve the class attribute from the `value` parameter in the call and store it with β_1.

Now consider this code, based on Listing 2.7, to set the class of an R object to `counter` but also keep all classes already on the object:

```
1 count_1 = c(a = 3, b = 2)
2 count_2 = "class<-"(count_1, value = c("counter", class(count_1)))
```

Listing 2.8: The code from Listing 2.7 modified to set the class to `counter` but keep existing classes on the object.

The class of `count_2` depends on the class of `count_1`.

During constraint generation, the type inference strategy will generate type variables for the expressions in line 2. Suppose τ_1 corresponds to `count_1` in the subexpression `class(count_1)`. Suppose also that β_1 corresponds to the result of the call to the `class<-` function. The strategy can generate a class constraint to record that the class of β_1 depends on the class of τ_1:

$$\beta_1 \equiv_{\text{class}} (\texttt{counter},\ \tau_1)$$

The strategy will determine the type and class of τ_1 when it generates constraints for line 1.

During constraint resolution, the strategy can resolve class constraints immediately after it resolves equality constraints. This propagates the information about the class of τ_1 to β_1.

More generally, the argument to the `value` parameter in a call to `class<-` can be an expression which is not solely composed of literals and calls to `c` or `class`. For this we rely on constant propagation—a code transformation described in Chapter 1 which replaces variables and some calls with their literal values, provided those values can be determined statically. If the value of the argument cannot be determined statically, then the type inference strategy is unable to determine the class.

Finally, note that the `class` function is not the only way to set the class attribute on an object. The `structure` function and the `attr<-` function provide two additional ways to set the class attribute:

```
1 count_1 = c(a = 3, b = 2)
2 # With structure
3 count_2 = structure(count_1, class = "counter")
4 # With attr
```

```
5 count_2 = "attr<-"(count_1, "class", value = "counter")
```

Listing 2.9: Two additional ways to set the class attribute on an R object, for comparison to using the `class` function, as in Listing 2.7.

These are syntactically similar to a call to `class<-`, so the same strategy can be applied. In the case of the `structure` function, the type inference strategy must check whether the `class` parameter is set. In the case of the `attr<-` function, the strategy must check that argument to the second parameter, `which`, is `"class"`. Applying constant propagation to the code before type inference helps to ensure that this check is possible. □

The primary purpose of inferring S3 classes is to enable the type inference strategy to infer more specific types for calls to generic functions. There's a call to R's built-in `UseMethod` function in the body of every generic function, so we can use a simple static analysis which searches for calls to `UseMethod` to identify generic functions before type inference. The call to `UseMethod` also specifies which argument the function checks the class of for method dispatch.

The methods of a generic function are required to follow a specific naming convention: the name of the generic, a dot, and then the name of the class of the argument. For instance, the `mean` function is a generic function and `mean.Date` is its method for objects with class `Date`. Thus it's possible to statically identify the methods for a generic and the classes to which they correspond. We can generate a type signature for each method by applying the type inference procedure to each method. For R's built-in methods, we can also define the type signatures manually in order to provide greater accuracy.

We include a `Generic` type in the grammar of types to represent the type of a generic function. The `Generic` type is composite, and its components are the type signatures of the methods of the associated generic function. For instance, one component of the generic type signature for the `mean` function is the type signature of the `mean.Date` function. The `Generic` type also stores the name of the parameter for dispatch and the class associated with each method.

The type inference strategy generates an instance constraint for each call to a defined function, due to the assignment rule described in Section 2.3. The type signature on the right-hand side of the instance constraint comes from the called function's definition. For a call to a generic function, the right-hand side of the instance constraint will be a `Generic` type. The left-hand side of the instance constraint comes from the call site. The type inference strategy delays resolution of instance constraints which contain the `Generic` type until the class of the argument used for dispatch has been inferred (that is, until there is a concrete class for the argument on

the left-hand side of the constraint).

In cases where the class of an argument used for dispatch cannot be determined, the type inference strategy can instead replace the `Generic` type with a function type $\rightarrow$, since the methods must all have the same parameters. The return type for this function type is a bounded type variable, where the bounding set contains each method's return type. Likewise, each argument type is a bounded type variable, where the bounding set contains each method's type for that argument. This ensures that the type inference strategy can still resolve other constraints.

2.4.3 Constructor Functions

R provides a variety of functions for constructing vectors, as well as other kinds of objects. The return type for many of these functions is fixed or depends only on the type of one argument. For instance, the `logical` function always returns a `Vector` with `Logical` elements, and the `matrix` function returns a `Matrix` where the element type is the same as the element type of the argument to the `data` parameter. Table 2.2 shows a selection of these functions and their return types. This section presents a few examples of the constructor functions where the relationship between the arguments and the return type is more complicated, and discusses how the type inference strategy can handle these.

Example 21. The `vector` function constructs a vector where the element type and length are specified by arguments to the parameters `mode` and `length`, respectively. The argument to `mode` is the name of the element type as a string. For instance, this is the code to create a numeric vector with length `p`:

```
1 vector(mode = "numeric", p)
```

During constraint generation, the type inference strategy can detect calls to `vector` and check the value of the `mode` argument to determine the element type of the result. Applying constant propagation to the code before type inference facilitates this by ensuring that the value of the `mode` argument will be literal if it can be determined statically. If the value of the mode argument cannot be determined statically, the strategy can still conclude that the result is a `Vector` and attempt to infer the element type from expressions which use the result. □

The `:` operator and the `seq` family of functions construct vectors by generating an `Integer` or `Numeric` sequence. The `seq` function provides a great deal of flexibility in terms of how

	Return Type		
Function	Object	Elements	Dimensions
`logical`	`Vector`	`Logical`	`length`
`integer`		`Integer`	
`single`		`Numeric`	
`double`		`Numeric`	
`numeric`	⋮	`Numeric`	⋮
`complex`		`Complex`	
`character`		`Character`	
`raw`		`Raw`	
`vector`	`Vector`	Depends on `mode`	`length`
α_1`:`α_2	`Vector`	α_1	
`seq`		Depends on arguments	
`seq_len`	⋮	`Integer`	`length.out`
`seq_along(along =` α`)`		`Integer`	length(α)
`seq.int`	`Vector`	`Integer`	`length.out`
`rep(x =` α`)`	α		
`rep.int(x =` α`)`	α		length(α) $\cdot$ `times`
`rep_len(x =` α`)`	α		`length.out`
`matrix(data =` α`)`	`Matrix`	α	`nrow` $\times$ `ncol`
`array(data =` α`)`	`Array`	α	`dim`
`list(`$\alpha_1, \ldots, \alpha_n$`)`	`List`	$\alpha_1, \ldots, \alpha_n$	n
`c(`$\alpha_1, \ldots, \alpha_n$`)`	`Vector` or `List`	$\vee_{i=1}^{n} \alpha_i$	$\Sigma_{i=1}^{n}$ length(α_i)

Table 2.2: Return types for some of R's built-in functions for constructing vectors, matrices, arrays, and lists. Greek letters denote the type of an argument, so for example in the expression α_1`:`α_2, the α_1 and α_2 denote the type of the first and second argument, respectively. Section 2.4.4 discusses the return type of the `c` function in greater detail. The operator $\vee$ selects the greatest type among its operands according to R's order of implicit coercion (Figure 2.4). Names in monospace in the Dimensions column refer to parameters of the function.

the sequence is generated, so the type of the result depends on multiple arguments. The next example documents how the arguments to `seq` affect the type of its result.

Example 22. The `seq` function has five named parameters which fall into three categories:

- `from` and `to` control the first and last value in the sequence, respectively.
- `by` controls the step size of the sequence.
- `length.out` and `along.with` control the length of the sequence. The `along.with` parameter sets the length of the sequence to the length of its argument.

The function accepts arguments to any combination of these parameters which characterizes a sequence. For instance, a call to the function with arguments to `to`, `by`, and `length.out` is valid. Adding a fourth argument to `along.with` is not valid, since then the sequence is overdetermined—the length is provided in two different ways. The function provides default arguments for all of the parameters, so calling the function with less than three parameters is also valid.

The function returns an `Integer` vector in four different cases:

1. The length of the sequence is 0.

   Example: `seq(1.8, 3.5, length.out = 0)`.

2. Only `from` or `from` and `to` have arguments, and `from` is integer-valued (not necessarily type `Integer`).

   Examples: `seq(10)` and `seq(1, 3.1)`

3. All of `from`, `to`, `by`, and `length.out` have no argument or an `Integer` argument. In addition, if `length.out` has an argument, it must be small enough that the sequence can have an integer-valued step of 1 or more. Likewise, if `along.with` has an argument, the length of the argument must be small enough that the sequence can have an integer-valued step of 1 or more.

   Examples: `seq(1L, 5L, 1L)` and`seq(1L, 5L, length.out = 2L)`.

4. Only `length.out` or `along.with` has an argument.

   Example: `seq(length.out = 3.1)` or `seq(along.with = letters)`

The function returns a `Numeric` vector in all other cases.

Cases 1 and 2 depend on the values of the arguments, so the type inference strategy can only infer the type when the arguments are constant. There's an exception to this for case 2: if the argument to `from` has type `Integer`, then the type inference strategy can infer that the function will return an `Integer` vector even if the argument is not constant.

Case 3 depends only on the types of the arguments if `length.out` and `along.with` do not have arguments. Then the type inference strategy can infer that the function will return an `Integer` vector. Otherwise, the argument to `length.out` must be constant or it must be possible to determine the length of the argument to `along.with` statically in order to determine whether the function will return an `Integer` vector.

Case 4 does not depend on the values or types of the arguments, so the type inference strategy can infer that the function will return an `Integer` vector.

For cases where the return type depends on the values of the arguments and the arguments are not constant, the type inference strategy can only conclude that the result can be either an `Integer` or `Numeric` vector. □

2.4.4 Implicit and Explicit Coercions

R automatically and silently coerces objects from less general types to more general types in order to satisfy the requirements of functions. This kind of coercion is called *implicit coercion*. As an example, R computes the expression `3 + TRUE` by implicitly coercing the logical value `TRUE` to the numeric value `1`. R also provides a variety of functions to explicitly coerce objects to specific types and classes. This section describes R's implicit coercion rules, documents the functions R provides for explicit coercion, and discusses how the type inference strategy can handle these.

$$\texttt{logical} \prec \texttt{integer} \prec \texttt{double} \prec \texttt{complex} \prec \texttt{character}$$

Figure 2.4: The order of implicit coercion in R. Read the notation $\prec$ as, "is less general than" or "can be implicitly coerced to." For example, a logical value can be implicitly coerced to an integer value.

Figure 2.4 shows R's vector types in order from least to most general. This order determines which types can be coerced to other types. For example, as the least general type, `logical`, can be implicitly coerced to any other vector type.

One way the type inference strategy can accommodate implicit coercion is by allowing less

general types on the right-hand side of an instance constraint to be unified with more general types on the left-hand side. For example, suppose the function `f` has type `Integer` $\to$ `Complex`, and the code contains the call `f(TRUE)`. First consider what happens during constraint generation. For the call, the strategy will generate two type variables and an equality constraint:

$$f : \tau, \qquad \texttt{f(TRUE)} : \beta, \qquad \tau \equiv \texttt{Logical} \to \beta$$

For the function definition, the strategy will generate an instance constraint:

$$\tau \leq_M \texttt{Integer} \to \texttt{Complex}, \quad \text{where } M \text{ depends on the context.}$$

The left-hand side of the instance constraint corresponds information from the call, while the right-hand side corresponds to information from the function definition. Now consider what happens during constraint resolution. The function type `Logical` $\to \beta$ will be substituted for the type variable τ because of the equality constraint. As a result, the instance constraint will become:

$$\texttt{Logical} \to \beta \leq_M \texttt{Integer} \to \texttt{Complex}$$

The DM type inference strategy reports a type error for this constraint, because `Logical` and `Integer` are different types. The modified type inference strategy instead unifies this constraint without error, because the type on the left (`Logical`) can be implicitly coerced to the type on the right (`Integer`).

Specific functions can also perform implicit coercion which differs from the order in Figure 2.4. For instance, the first argument to the `runif` function controls the number of randomly sampled observations returned by the function, so it should be an integer value, but the function will also accept a numeric value. The type inference strategy can handle these functions by using bounded type variables to represent the argument types. For each argument, the bounding set is the set of all accepted types.

The next example discusses the `c` function, for which the return type depends on how the arguments can be implicitly coerced.

Example 23. The `c` function attempts to construct a vector by coercing all of its arguments to a common type and then combining them. If it's not possible to coerce the arguments to a common type, then the function combines its arguments into a list instead of a vector. The `c`

function provides a simple way to construct vectors from literal values, but we discuss it here rather than in the section about constructor functions (Section 2.4.3) because it coerces its arguments.

This example explores how the type inference strategy can handle the `c` function. Consider this expression to create a vector of university course codes:

```
1 course_numbers = c(141, "141A", c("141B", "141C"), other_stats)
```

The result from the inner call to `c` is a character vector, since both of the arguments are strings. For the outer call, the first three arguments are all scalars or vectors, and can be implicitly coerced to a character vector based on Figure 2.4. It follows that the outer call returns character vector or a list depending on the type of the fourth argument, `other_stats`.

If `other_stats` is a vector or `NULL`, then the result of the outer call is a character vector. The element type of `other_stats` doesn't matter, because the other arguments require the result to be a character vector if it is a vector.

If `other_stats` is not a vector or `NULL`, then the result of the call is a list. The types of the list elements come from the types of the arguments, so the first element, for instance, is numeric.

In general, the type of the result from `c` is the most general type (in the sense of Figure 2.4) among the arguments if all of the arguments are vectors or `NULL`, and a list if any of the arguments are not.

A limitation of the DM type inference strategy is that it can only infer the relationship between the argument types and return type of a function when the return type only depends on one of the argument types. Nevertheless, we can adapt the strategy to handle the `c` function by including a `ListJoin` operator in the grammar of types. The `ListJoin` operator selects the most general type among its operands, so for instance `ListJoin(Integer, Numeric)` is equal to `Numeric`. If any of the operands are not scalar or vector types, then the result is instead a `List` where the components are the operands. Then the return type of the `c` function can be described as `ListJoin`$(\alpha_1, \ldots, \alpha_n)$, where $\alpha_1, \ldots, \alpha_n$ are the argument types. The strategy can resolve `ListJoin`s as early as possible during the constraint resolution step. The drawback of this approach is that it depends on manually specifying the type signature for the `c` function. □

R objects can also be coerced explicitly, by calling a coercion functions. Table 2.3 shows the return types for some of R's built-in coercion functions. As the table shows, most of these functions have a fixed return type or a return type with a simple relationship to one of the

argument types. The exception is the `as` function.

	Return Type	
Function	Object	Elements
`as.logical`	`Vector`	`Logical`
`as.integer`		`Integer`
`as.single`		`Numeric`
`as.double`		`Numeric`
`as.numeric`	⋮	`Numeric`
`as.complex`		`Complex`
`as.character`		`Character`
`as.raw`		`Raw`
`as.vector`	`Vector`	Depends on `mode`
`as.matrix(x =` α`)`	`Matrix`	Element type of α
`as.array(x =` α`)`	`Array`	Element type of α
`as.list(x =` α`)`	`List`	Element type of α
`as.null`	`Null`	
`as`	Depends on `Class`	

Table 2.3: Return types for some of R's built-in functions for coercing objects to specific types or classes. Greek letters denote the type of an argument.

The `as` function coerces an R object to a type or class specified in a string argument to the function's `Class` parameter. Thus determining the return type for a call to `as` depends on the argument being constant, similar to what we saw for the `vector` function in Example 21. When the value of the argument to the `Class` parameter cannot be determined statically, a call to `as` does not provide the type inference strategy with any information about its return type.

2.4.5 Indexing

R provides a variety of functions for indexing vectors, matrices, arrays, lists, data frames, and other data structures. Indexing functions can return objects with a different type or class than the object being indexed. This section presents examples of cases where indexing can potentially cause problems for the type inference strategy, and where possible, describes ways the strategy can be modified to handle them.

One way to use indexing is to assign a new value to an element of a vector, matrix, or array. If the new value's type is less general than the original element type, then R implicitly coerces the new value to the element type. If the new value's type is more general, then R implicitly coerces the element type to the new value's type. Thus setting an element of a vector, matrix, or array can change the type of all the elements. The first example examines how the type inference

strategy can handle this feature.

Example 24. Consider the following code, which constructs a numeric vector `x` and then assigns a string to the third element. The code is in SSA form:

```
1 x_1 = c(10, 20, 21)
2 # x[3] = "30"
3 x_2 = "[<-"(x_1, 3, value = "30")
```

Listing 2.10: The SSA form distinguishes between `x` before the element is changed (`x_1`) and after the element is changed (`x_2`). The value of `x_2` is the result of coercing `x_1` from a numeric vector to a character vector and changing the third element. The comment on line 2 shows the expression on line 3 before the code was converted to SSA form. See Section 2.2 for more about the SSA form.

The type of `"30"` (`character`) is more general than the type of `x_1` (`numeric`), so R coerces `x_1` into a character vector in the call to `[<-`. Thus `x_2` is a character vector.

The problem for the type inference strategy is to correctly infer the type of the result returned by the `[<-` function. Suppose the new value is a vector, matrix, or array. This is the case shown in Listing 2.10. The overall type of the result is the same as the indexed object: a vector if it's vector, a matrix if it's a matrix, and an array if it's an array. The element type is whichever is more general between the element type of the indexed object and the element type of the new value.

Suppose instead that the new value is a list. Then the result returned by `[<-` is a list. The elements of the indexed object are not coerced, nor are the elements of the list.

The type inference strategy must handle calls to `[<-` on a case-by-case basis since the behavior of the function changes depending on the types of its arguments. We include a `TypeDependent` type in the grammar of types in order to represent the type signature of this function and other functions where the behavior varies depending on the argument types.

The `TypeDependent` type is a composite type that follows the same design as the `Generic` type we introduced to handle generic functions in Section 2.4.2. Each component is a type signature associated with one possible behavior of the function corresponding to a specific set of argument types. During constraint resolution, the type inference strategy delays resolution of instance constraints which contain the `TypeDependent` type until sufficient information has been inferred about the types of the arguments at the call site. With this approach, the type inference strategy can represent the type of the `[<-` function.

This example highlights that the SSA form is important for the type inference strategy because it allows the strategy to associate different types with the same variable at different points in the code. It would not be correct to infer a single type for the original variable `x` across all of the code. Quiroga and Ortin (2017) provide additional details about using the SSA form of code to improve the precision of type inference for dynamically typed languages. □

A general rule is that indexing an object with R's `[` indexing operator returns an object with the same overall type. An exception to this rule occurs when indexing multidimensional objects such as matrices, arrays, and data frames. When the `[` indexing operator's parameter `drop` is `TRUE` (the default), then if the result from indexing a multidimensional object is one-dimensional, it's converted to a vector.

Whether the result of indexing is one-dimensional depends on the value and type of the indices. Each dimension of an object can be indexed with the `[` operator in four different ways:

1. By position. This case corresponds to an integer or numeric vector index argument. A dimension is dropped for each scalar index argument. For instance, suppose `x` is a three-dimensional array. Then `x[1, , 10]` drops the first and third dimension. If the index argument is negative, the elements at the positions in the argument are dropped instead of kept; if all except one element of a dimension are dropped, then that dimension is dropped.

2. By name. This case corresponds to a character vector index argument. As with the by position case, a dimension is dropped for each scalar index argument.

3. By condition. This case corresponds to a logical vector index argument. A dimension is dropped if the index argument only contains one `TRUE` value. For example, if `x` is a 3×2 matrix, then `x[c(T, F, F), ]` drops the first dimension.

4. With an empty index. In this case the entire dimension is returned, so there is no change in type.

Since the cases correspond to types, the type inference strategy can generally determine which case each index argument corresponds to by inferring the type of the argument.

In cases 1 and 2, the type inference strategy needs to know whether or not the index is a scalar in order to determine whether dimensions will be dropped. In other words, the type inference strategy must also infer dimensions, or at least scalar versus non-scalar, for each expression. Our

type inference strategy provides limited support for inferring dimensions; see Section 2.4.9 for details.

For a negative index in case 1, the type inference strategy needs to know the value of the index. Constant propagation replaces variables with values where it's possible to do so with static analysis. If the value of an index cannot be determined statically, it is not possible to infer the exact type of the result from the indexing expression alone.

Case 3 is similar to a negative index in case 1: the type inference strategy needs to know the value of the index in order to determine whether dimensions will be dropped. Once again, if constant propagation cannot determine the index's value, then the indexing expression alone does not provide enough information to determine the exact type.

For both negative indexes and logical indexes (case 3), an indexing expression provides some information about type of its result even if the values of the indexes cannot be determined statically. For instance, indexing a numeric matrix will always return a numeric matrix or a numeric vector. Thus the type inference strategy can always put a bound on the type variable associated with the result of an indexing operation. Depending on the code, the type inference strategy may also be able to infer the type of the result from other expressions which use the result. For instance, if a subsequent expression indexes the result, the number of indexes can reveal the number of dimensions and thereby the type. On the other hand, it is not necessarily possible to infer the type of the result from other calls in the code, since most R functions will implicitly coerce matrices and arrays to vectors or will accept either. For instance, all of R's basic mathematical functions accept vectors, matrices, or arrays.

The next example discusses the different ways to index a matrix by position.

Example 25. Consider this code to extract one row from a matrix:

```
1 x = diag(5)
2 x[1, ]
```

Listing 2.11: Code to extract the first row of a 5×5 matrix. The result is simplified to a vector.

Since only one row is selected here, a dimension is dropped and the value returned by the `[` indexing operator is simplified to a vector. When multiple rows are selected, no dimensions are dropped and the result is a matrix. This same reasoning applies to columns, and for higher-dimensional arrays, to all dimensions.

Now consider a different way to extract the same row from the matrix:

```
x[-(2:5), ]
```

Listing 2.12: Another way to extract the first row of a 5×5 matrix.

Again the result is simplified to a vector. For a matrix with n rows, the negative index must contain exactly $n - 1$ distinct values in order for the first dimension to be dropped. Since the values must be distinct, knowing the length of the index is not sufficient to determine whether a dimension will be dropped. For instance, consider this code, which is yet another way to extract the first row of the matrix:

```
x[-c(2, 3, 4, 5, 2), ]
```

Listing 2.13: Yet another way to extract the first row of a 5×5 matrix. Notice that `2` is repeated in the first index.

Here the first index contains five values, but because the value `2` is repeated, it has the same effect as the length-4 index in Listing 2.12, and the result is a vector. Similarly, for any matrix or array we can construct an index with length $n - 1$ that behaves like a shorter index because of repeated elements, and returns a matrix rather than a vector. □

When `drop = FALSE`, the `[` indexing operator always returns an object of the same type as the object being indexed, so this case does not pose a problem for type inference.

Packages which extend matrices, arrays, and data frames don't always follow the default `drop = TRUE`. For instance, the **tibble** package (Müller and Wickham 2019) breaks compatibility with ordinary data frames by defaulting to `drop = FALSE`. As a result, we expect the type inference strategy can provide more specific results for code which data structures from uses these packages.

Finally, it is important that the type inference system tracks the names of elements, even if their position is not known. For lists, where each element can have a different type, it's sometimes possible to infer the type for a named element whose position is not clear from the code. The next example discusses this situation.

Example 26. Consider this assignment to an element of the list `progress`:

```
progress["a"] = 1
```

Listing 2.14: Code to assign an element by name. It may not be possible to infer the position of the changed element.

From this code, we can infer that the element `"a"` of the list is numeric. Even with a larger sample of code, it will not necessarily be possible to infer the position of the element `"a"` in the list.

If subsetting in the code always uses names rather than positions, the position of the elements in the list is irrelevant for type inference and for code transformations. For instance, the code in Listing 2.20 of Example 30 reads a data frame from disk and then accesses columns from the data frame by name only. Lack of information about positions did not pose any difficulty for type inference in that example.

In code that uses a mix of names and positions for subsetting, it may not be possible to match names to positions. Typical cases where it is possible is when the named list is created with the `list` function or renamed in the code. When it is not possible to match names and positions, the type inference system should instead infer types for names and positions independently, using whatever information is available. □

2.4.6 Characteristics of Lists and Data Frames

In order to infer complete type information for a list, the type inference strategy must infer the type of each element, because the elements can have heterogeneous types. This is also true for data frames, since data frames are lists where each element is a vector and corresponds to one column. This section begins with some examples of functions which create lists and data frames, then discusses the `$` operator as a way to identify lists and data frames.

Table 2.4 lists some of the functions which create lists and data frames in R. Type inference for calls to the `list` and `data.frame` constructor functions is relatively straightforward. For both functions, the arguments become the elements of the resulting list or data frame. Thus if type inference strategy can infer the types of the arguments, it can infer complete type information for the result of the call.

Return Type	Function
`List`	`list`
	`lapply`
`DataFrame`	`data.frame`
	`read.table`
	`read.csv`
	`read.fwf`
	`read.delim`

Table 2.4: Examples of functions which create lists and data frames.

Another entry point for data frames in code is through functions such as `read.csv`, which read tabular data from a file. Most of R's built-in functions for reading tabular data have a

`colClasses` parameter so that the user can specify the S3 classes of the columns. When the argument to `colClasses` is constant, the type inference strategy can use the information it provides about the classes of the columns to infer their types as well. This is because each of R's vector types corresponds to a different class, usually with the same name (except type `double`, which corresponds to class `numeric`). The next example demonstrates how the type inference strategy can use the argument to `colClasses` in a call to a reader function.

Example 27. Consider this code to read a data set from a comma-separated values (CSV) file:

```
1 cols = c("character", "factor", "numeric", "integer")
2 dogs = read.csv("dogs.csv", colClasses = cols)
```

Listing 2.15: Code to read data from a CSV file, with the S3 classes of the first four columns specified in the argument to `colClasses`.

Based on the argument to `colClasses`, the type inference strategy can conclude that the first four columns of the data frame returned by the call to `read.csv` are a character vector, a factor, a numeric vector, and an integer vector.

In the argument to `colClasses`, the string `"NULL"` indicates that a column should be skipped when the file is read. Thus the type inference strategy should ignore `"NULL"` elements when using the argument to infer column types.

If there are fewer elements in the argument to `colClasses` than there are columns in the data set, R recycles the elements in the argument. In practice, the argument often contains an element for each column if it's provided at all. As an optional heuristic, we can make the type inference strategy assume that the argument always contains a class for every column, and thereby determine the number of columns. □

When no argument is set for the `colClasses` parameter in a call to a reader function, it is usually not possible to statically infer the types of the columns from the call alone. Depending on the code, the type inference strategy may still be able to infer the types of the columns from how the data frame is used in other expressions. The next example demonstrates this.

Example 28. The following code (not in SSA form) is an excerpt from an R script that collects online classified advertisements and organizes them into a data frame. The code coerces multiple columns of the data frame `posts` to specific types:

```
1 cols = c("latitude", "longitude", "price", "sqft")
2 posts[cols] = lapply(posts[cols], as.numeric)
```

```
3 cols = c("city", "pets", "laundry", "parking")
4 posts[cols] = lapply(posts[cols], factor)
5 posts$house_date = as.Date(posts$house_date)
```

Listing 2.16: Code to coerce the columns of the data frame `posts`.

Lines 1-2 and 3-4 follow the same pattern: `cols` is assigned a set of column names, and then each of those columns is replaced with the result of applying a function to the column.

For instance, line 2 applies the `as.numeric` function to the `latitude`, `longitude`, `price`, and `sqft` columns. Since the `as.numeric` function always returns a numeric vector, the type inference strategy can infer that these four columns are numeric vectors.

Similarly, the type inference strategy can infer from line 3 and 4 that the columns `city`, `pets`, `laundry`, and `parking` are factors. The type inference strategy treats factor as a type, even though it is technically an S3 class in R (see Section 2.4.2 for why we treat some S3 classes as types).

Finally, the call to `as.Date` on line 7 returns a `Date` object, so the type inference strategy can infer the type of the `house_date` column. □

Another approach to determine column types is to create a separate analysis which reads the first few rows of each tabular data loaded in the code and checks the types of the columns. This is a dynamic approach, since it depends on running (an approximation of) the code. The data files may not be available before run-time, but if they are, this approach always produces the correct types. The information can then be passed on to type inference (for instance, by inserting the `colClasses` in the code).

R's functions for reading tabular data also have other parameters that can be informative about the returned data frame. For example, the `col.names` parameter controls the names of the columns in the data frame. When this parameter has a constant argument, the type inference strategy can use the argument to determine the names of the columns in the data frame.

The examples so far have focused on data frames; now we consider lists. One way lists can arise in code is from calls to the apply-family of functions, particularly `lapply` and `sapply`. The `lapply` function always returns a list, but the type inference strategy must also infer the types of the elements in the list. If the return type of the applied function is always the same, this is straightforward. If the return type of the applied function depends on its arguments, it is not always possible to statically infer the element types from a call to `lapply`, although it

is possible to infer a set of potential element types. The next example demonstrates how the applied function affects what the type inference strategy can infer from a call to `lapply`.

Example 29. Consider this code to compute lengths of two samples:

```
1 lengths = lapply(samples, length)
```

Listing 2.17: Code to compute the lengths of the elements of a list.

The `length` function always returns a scalar integer, the each element of the list returned by the call to `lapply` will be a scalar integer.

Now suppose that instead of computing lengths, the code computes medians:

```
1 n = 100
2 samples = list(x = rnorm(n), y = rnorm(n) + rnorm(n) * 1i)
3 medians = lapply(samples, median)
```

Listing 2.18: Code to compute the medians of the elements of a list.

In this case, the types of the elements in the list returned by `lapply` depend on the types of the elements in the list `samples`. This is because the `median` function returns a numeric result when given a numeric argument (or an argument which can be implicitly coerced to a numeric vector), and returns a complex result when given a complex argument. Since the list `samples` is defined by a call to `list` in the code, the type inference strategy can determine the number and types of its elements. As a result, the strategy can also determine the complete type of the result from the call to `lapply`.

When the type inference strategy cannot infer complete type information for the first argument (for the `X` parameter) in a call to `lapply`, then the strategy can still bound the type variables for the elements of the returned list. For instance, suppose it's not possible to determine the types of the elements of `samples` in Listing 2.18. The strategy can still infer that the types of the elements in the result from the call to `lapply` must be numeric or complex scalars, since the `median` function only returns numeric or complex scalars.

Finally, consider this code to load a list of data sets from files:

```
1 files = list.files(pattern = "[.]rds$")
2 data = lapply(files, readRDS)
```

Listing 2.19: In the worst case, it is not possible to determine the element types.

In this case, it is not possible to statically determine the types of the elements in the list returned by `lapply` from the call alone. The problem is that the `readRDS` function can return any type. In general, the information the type inference strategy can infer from a call to `lapply` is only as good as the type information the strategy has about the arguments. □

How an object is used can also reveal that the object is a list. For instance, the typical use for the `$` operator is to extract named list elements (including data frame columns). The operator is not valid for vectors, even if the elements are named. As a result, the type inference strategy can use calls to the `$` operator to conclude that the target object is a list or data frame. The next example demonstrates this.

Example 30. Consider this sample of code from an analysis of data about universities in the United States:

```
1 colleges = readRDS("college_scorecard_2013.rds")
2 table(colleges$ownership)
3 ug_avg = mean(colleges$undergrad_pop, na.rm = TRUE)
4 plot(density(colleges$undergrad_pop, na.rm = TRUE))
5 populous = colleges[colleges$state %in% c("CA", "TX", "FL", "NY"), ]
6 populous$state = droplevels(populous$state)
```

Listing 2.20: A sample of code from a data analysis.

As discussed previously, the call to `readRDS` on line 1 is not informative for type inference. However, for this code it's possible to infer the type of the result from the call based on how the `colleges` variable is used in other expressions.

The `$` operator's first argument must be a list, data frame, or environment, so the type inference strategy can infer that the value of the `colleges` variable must have one of these types. Moreover, the type inference strategy can infer from the `[` indexing operation on line 5 that `colleges` is multidimensional. Lists and environments are one-dimensional, so `colleges` must be a data frame.

The code also provides information about the names and types of the columns in the data frame. For instance, the type inference strategy can infer from the expression `colleges$ownership` on line 2 that the data frame has a column whose name begins with `ownership`. From other calls to `$`, the strategy can also infer columns `undergrad_pop` and `state`. The `$` operator does

partial matching of element names, so there is no guarantee that these are the full names of the columns.

Partial matching is a potential source of bugs, so it's good practice to use full column names in non-interactive code. As an optional heuristic, the type inference strategy can assume that code being analyzed will not use partial matching.

The type inference strategy can infer a set of potential types for the `undergrad_pop` column from line 4. The `density` function coerces its first argument to a numeric vector, so the `undergrad_pop` column must be a logical, integer, or numeric vector.

The type inference strategy can infer the type of the `state` column from line 6. The first argument to the `droplevels` function must be a factor, so the `state` column must be a factor.

Thus by examining how a list or data frame is used in expressions, the type inference strategy can infer a substantial amount about the element types. This is especially important for data frames, because data frames are often loaded from files rather than constructed in the code. □

2.4.7 Assertions

An idiom to prevent bugs in code is to test the type and class of an object before using it. Table 2.4.7 shows some of the functions R provides to test the type or class of an object. If the tested object has a type or class incompatible with subsequent code, then the code can call the `stop` function to raise an error. R also provides a `stopifnot` function as a shortcut for testing a condition and calling `stop` if the condition does not hold. These kinds of tests are called *assertions*. Assertions are a source of type and class information. This section discusses how the type inference strategy can use assertions for inference.

Assertions can be detected in code with a static analysis pass which runs before type inference. The typical form of an assertion is an if-expression where the condition checks the type or class of an object and the body contains a call to `stop`. Once each assertion in the code has been identified, the call to the type or class test function can be annotated with the type or class the object must have for the assertion to succeed (so no error is raised). The type inference strategy can then use the annotations during constraint generation in order to generate additional constraints. The next example shows the typical structure of an assertion and further discusses this approach.

Example 31. The body of the `toTitleCase` function in R's built-in **tools** package contains an assertion:

	Tests For...	
Function	Object	Elements
`is.logical`	`Vector`	`Logical`
`is.integer`		`Integer`
`is.single`		`Numeric`
`is.double`		`Numeric`
`is.numeric`	⋮	`Numeric`
`is.complex`		`Complex`
`is.character`		`Character`
`is.raw`	`Vector`	`Raw`
`is.atomic`	`Vector` or `Null`	
`is.vector`	`Vector` or `List`	
`is.matrix`	`Matrix`	
`is.array`	`Array`	
`is.list`	`List`	
`is.data.frame`	`DataFrame`	
`is.null`	`Null`	
`is.factor`	`Factor`	
`is`	Depends on `class2`	
`inherits`	Depends on `what`	
`typeof`		
`class`		

Table 2.5: Some of R's built-in functions for testing the type or class of an object. The `typeof` and `class` functions return the type and class of an object, respectively, so they are often used in tests of type and class.

```
1 if(typeof(text) != "character")
2   stop("'text' must be a character vector")
```

Listing 2.21: Code from the `toTitleCase` function to ensure that `text` is a character vector.

The assertion checks that the function's parameter `text` is a character vector.

We can use the `ast_find_all` function from the **rstatic** package to create an analysis which finds most assertions in code. The test function should return `TRUE` for if-expressions which contain a call to a function for testing type or class in the condition and contain a call to `stop` in the body. Then for each assertion, we can extract the object the assertion targets and the class or type the object must have. Finally, we can store this information in the `.data` field of the call to the type or class test function, for the type inference strategy to use during constraint generation. For the code in the listing, the analysis records that `text` must have type `Vector[String]` in the `.data` field of the call to `typeof`.

The type inference strategy must check the `.data` field on calls for information about assertions.

During constraint resolution, the strategy will generate an additional constraint that the type variable for `text` in the call to `typeof` must have type `Vector[String]`.

Thus we have a general strategy for using type and class information from assertions. The analysis to find assertions described in this example will not find all assertions, but can be improved. For instance, the analysis will only recover the top-level condition if the call to `stop` is inside of nested if-expressions, and the analysis does not detect calls to `stopifnot`. □

Instead of using assertions to raise an error if an object doesn't have a specific type or class, code can use if-expressions in order to carry out different computations depending on the type or class of the object. In this case, the code typically indicates the type of the object for each branch of the if-expression. Using this information effectively is a problem for the DM type inference strategy, since the information corresponds to specific regions of the code (the different branches of the if-expression), but the strategy shares information across all instances of a variable.

For example, consider this code pattern:

```
1 if (is.numeric(x)) {
2     # ...
3 } else if (is.character(x)) {
4     # ...
5 } else {
6     stop("Invalid type for x!")
7 }
```

The pattern suggests that `x` can be either a numeric vector or a character vector. During constraint resolution, any information collected about `x` in one branch of the if-expression will be propagated to all type variables associated with `x`. Thus if we make the type inference strategy assume that `x` is a numeric vector in the `TRUE` branch, this type information will be propagated to other parts of the code (such as the `FALSE` branch) where it is not correct to assume.

One potential solution to this problem is to use the *static single information* (SSI) form of the code instead of the SSA form. The SSI form is an extension of SSA form where variables are provided with unique names for each branch in control flow in the code (Ananian 2001). By using the SSI form, the type inference strategy would be able to infer different types for a variable on different branches even if the variable is not redefined.

2.4.8 Scope and Environments

A variable's *scope* is the region of code where the variable is defined and therefore can be used. R provides a system of scoping rules so that variable names can be reused for different purposes in different contexts. For instance, the code in the body of a function can define a variable with the same name as one defined elsewhere, and modifying one will not affect the other. This section provides a brief overview of how variable scope and lookup works in R, and then discusses why handling scope poses a problem for the type inference strategy.

The data structure R uses to keep track of variables is called an *environment*. Each environment consists of a *frame*, which is a list of name-value pairs, and an *enclosure*, which is a reference to another environment (R Core Team 2019a). Assigning a value to a variable inserts or updates the variable's name and value in the frame of the current environment. For example, evaluating the expression `x = 1` inserts the value `1` under the name `x` in the frame of the current environment.

When R looks up the value of a variable, it first checks the frame of the current environment. If there is an entry for the variable's name, R returns the paired value. If there is not, then R repeats this process for the enclosure of the current environment. If R is unable to find the variable in any enclosing environment, it emits an error stating that the variable is not defined.

The *global environment* is where R evaluates top-level expressions entered at the R prompt or in scripts. Each call to a function implicitly creates a new environment in which to evaluate the code in the body of the function. R uses *lexical scoping*, so the enclosure for the new environment is the environment where the function was defined (rather than where it was called). This means code in the body of a function can use *non-local variables*, variables from the environment where the function was defined or its enclosing environments.

For instance, this is valid R code and the result of the last line is `15`:

```
1 x = 10
2 add_x = function(y) x + y
3 add_x(5)
```

The code defines both the variable `x` and the function `add_x` in the global environment. When the code calls `add_x`, R creates a new environment for which the enclosure is the global environment. When R looks up the variable `x` in order to evaluate `x + y`, it searches the environment for the call first. There's no variable `x` defined in that environment, so R searches the enclosing environment and finds that `x` is `10`.

An important nuance is that when a function uses a non-local variable, the definition (and therefore value) of that variable can change between calls. For instance, consider this code, where the result of each call to `add_x` is shown as a comment:

```
x = 10
add_x(5) # 15
x = 20
add_x(5) # 25
```

The point is that if the code in the body of a function uses a non-local variable, that variable can refer to a different definition each time the function is called. As a result, we say that R's variable lookup is *dynamic*. The value of a non-local variable depends on the state of its environment at run-time.

Dynamic variable lookup poses a problem for the type inference strategy, since the strategy only analyzes code in the body of a function once—not once per call to the function—and the type of a non-local variable can be different at different call sites. Note that the SSA form does not solve this problem, since the SSA form is computed separately for each function and does not provide a representation of data flow for non-local variables.

One potential solution to this problem is to use *interprocedural static single assignment* (ISSA) form. This is an extension of SSA form which transforms functions so that non-local variables are parameters, and the appropriate variable is inserted as an argument for the parameter at each call site (Staiger et al. 2007). The type inference strategy can already handle parameters, so it is likely that no other modifications would be necessary. However, we have not implemented the ISSA form or tested this approach (yet). It is also unclear how to combine the ISSA form with other SSA forms that address problems for the type inference strategy, such as the static single information form mentioned in Section 2.4.7. Without the ISSA form, the type inference strategy assumes by default that non-local variables will not be redefined between calls.

2.4.9 Dimensions, Recycling, and Loops

The DM type inference strategy does not attempt to infer the dimensions of objects in the code. Nonetheless, for many R expressions it is possible to infer the dimensions of the result. Information about dimensions has several applications:

- To apply optimizing transformations for scalars and vectors to the code.

- To detect errors before run-time, by checking that indexes are not out of bounds and that the operands for matrix multiplication or other linear algebra operations are conformable.
- As an aid to readers trying to understand code, particularly scientific computations (in which linear algebra is common).

This section provides examples of cases where it is possible to infer the dimensions of objects, either as exact values or as symbolic relationships to other objects.

Dimension inference can be carried out simultaneously with type inference, or as a separate analysis. The advantage of combining the two analyses is that they can share information. For instance, the type inference strategy already infers a lower bound on the number of elements in each list, since it infers a type for each element it detects in the code (see Section 2.4.6), so a separate dimension inference analysis is redundant for lists. On the other hand, separating the two analyses provides greater flexibility in terms of the strategy that can be used. If the two analyses are simultaneous, then the dimension inference analysis must traverse the code in the same way as the type inference analysis.

It's possible to infer the exact dimensions of literal values, and it's often possible to infer the dimensions of objects returned by constructor functions as well. Table 2.2, from the section about constructor functions, shows the dimensions of the result for many of R's built-in constructor functions. Where the dimensions depend on an argument, it's possible to infer an exact value if the argument is constant. For example, `logical(3)` returns a vector with 3 elements. If the argument is not constant, it is still possible to keep track of the dimensions symbolically. For instance, from a call `y = character(n)` we can infer that the result has length `n`. If the definition of `n` is `n = length(x)`, then we can further infer that `x` and `y` have the same length.

Besides constructor functions, functions for reading tabular data sets from files, such as those in Table 2.4, are another entry point for multidimensional objects. These functions provide parameters such as `nrows`, `row.names`, `col.names`, and `colClasses` whose arguments can indicate the dimensions of the loaded object. The next example demonstrates this.

Example 32. Consider this code to read part of a tabular data set:

```
1 dogs100 = read.csv("dogs.csv", nrows = 100,
2     col.names = c("breed", "size", "avg_lifespan", "avg_cost"))
```

Listing 2.22: Code to read the first 100 rows from a CSV file.

Based on the argument to the `nrows` parameter, we can conclude that the data frame resulting from the call will have exactly 100 rows. Based on the argument to the `col.names` parameter, we can conclude that the data frame will have 4 columns, since the argument must have the same number of elements as there are columns in the data set.

Arguments to the `row.names` parameter can be used to infer the number of rows in a data frame by checking the number of names (similar to `col.names` for columns). Arguments to the `colClasses` parameter are not required to contain an element for every column. However, the number of elements in the argument is a lower bound on the number of columns. Furthermore, if subsequent code only accesses the columns named in the argument to `colClasses`, then as a heuristic, we can assume that this lower bound is the exact number of columns. □

R *recycles* vector elements, which means that shorter vectors are extended by repeating the elements in order to match the length of longer vectors in vectorized (elementwise) computations. For instance, to evaluate the expression `c(1, 2) + c(3, 4, 5, 6)`, R repeats the elements `1` and `2` of the first operand and then computes the elementwise sum of `c(1, 2, 1, 2)` and `c(3, 4, 5, 6)`. As a result of recycling, it is not correct for dimension inference to assume that two operands to a binary operator must have the same length.

Loops and other forms of iteration are often necessary in code that deals with multidimensional objects. Dimension inference can extract two kinds of information from loops:

1. The number of iterations in the loop and how this number corresponds to objects in the loop. This is important because it can reveal the dimensions of objects, either symbolically or exactly. For example, if a loop accesses `x[i]` in every iteration, where `i` is a loop variable that increases by 1 in every iteration, then the number of iterations is a lower bound on the length of `x`.

   In a for-loop, the length of the object being iterated over is an upper bound on the number of iterations. The length is the exact number of iterations if the body of the loop does not contain a `break` statement or other expression which exits the loop early. Dimension inference can determine the relationship between the dimensions of the object being iterated over and other objects used within the loop if the indexes used are constant or follow a simple pattern (such as counting up by 1 from 1 to the number of iterations).

   In a while-loop, the number of iterations will not necessarily correspond to an object created before the loop. However, dimension inference can still create a symbolic placeholder for

the number of iterations, as a lower bound for the sizes of objects indexed in the loop.

2. Which variables are used as indexes and how they are updated in each iteration. These variables are important because they are the link between the number of iterations and the dimensions of objects used or modified in the loop. Dimension inference can attempt to identify ranges and patterns in index variables. In for-loops, the loop variable is often an index, and the object being iterated over is often created by calling a sequence function (see Section 2.4.3). Dimension inference can also check for common non-automatic index updates, such as an increment by one (`a = a + 1`).

The DM type inference strategy does not provide a way to handle loops, so the type inference strategy must also be adapted for loops. In for-loops, the loop variable is modified at every iteration, and its type is determined by the element type(s) of the object the loop iterates over. In the case of a list with heterogeneous element types, the type of the loop variable changes across the iterations. In this case, it may not be possible to determine the exact type of the loop variable (or any objects based on it) at the end of the loop, unless the exact number of iterations is known.

The next example discusses dimension and type inference for a while-loop. While-loops are less structured than for-loops, so they provide less information for static analyses to use.

Example 33. Consider this code for a rejection sampler which generates `n` samples from a truncated normal distribution:

```
1  samp = numeric(n)
2  accepted = 0
3  while (accepted < n) {
4    x = runif(1, -2, 1)
5    y = runif(1, 0, dtrunc(0) + 0.1)
6    if (y < dtrunc(x)) {
7      accepted = accepted + 1
8      samp[accepted] = x
9    }
10 }
```

Listing 2.23: Code for a univariate rejection sampler.

The `samp` vector is initially a vector of `n` zeros. We can infer that both before and after the loop runs, `samp` will have `n` elements—this is important to check because R automatically extends vectors if elements are assigned to positions beyond the length. We can make this conclusion because `samp` is indexed by `accepted`, and:

- Initially, `accepted` is 0.
- Each iteration increases `accepted` by 0 or 1.
- The loop terminates when `accepted >= n`.

As a result, `accepted` will take all integer values from 0 to `n` over the course of the loop.

Dimension inference for the loop must collect and combine these pieces of information in order to determine the final size of `samp`. The variable `accepted` must be identified as an index. Then a lower bound, upper bound, and step size for `samp` must be established. If any of these pieces of information cannot be inferred, it may not be possible to infer the size of `samp` after the loop. This process generalizes to other loops, and the same information is necessary to determine the final dimensions of objects modified by the loops.

Based on our earlier discussion of constructor functions (Section 2.4.3), the type inference strategy can infer that `samp` is a numeric vector, since it is constructed with `numeric` and since the values assigned to `samp` in the loop are also numeric. □

2.4.10 Value-based Types

An R variable can be assigned an object of any type, so it's possible for the type of a variable to depend on a value computed at run-time. For instance, the following code is valid:

```
1 val = rnorm(1)
2 x = if (val >= 0) val^2 else "negative"
```

In this case, it is not possible for a static analysis to determine the exact type of the result from the if-expression. However, it is apparent from the code that the result will be either a numeric value or a string. The type inference strategy handles types which depend on values, such as the result here, by bounding the associated type variable with the set of possible types. So for this code, the type inference strategy bounds the type variable associated with the if-expression with `Numeric` and `String`. This way the type inference strategy provides as much information about the type as is apparent in the code. A potential extension to the strategy is to collect the

conditions for each possible type from the code. Then if more information about the run-time values becomes available, the type can be refined.

2.4.11 Non-standard Evaluation

R provides functions to intercept and modify expressions passed to functions as arguments before they are evaluated, a feature known as *non-standard evaluation*. The `library` function is an example of a function which uses non-standard evaluation. In a call to `library`, the name of the package to load does not have to be a quoted string. For instance, `library(lattice)` is a valid expression to load the **lattice** package. The function interprets the argument `lattice` as a string, even though it would ordinarily be interpreted as a variable. The popular **tidyverse** packages (Wickham, Averick, et al. 2019) make extensive use of non-standard evaluation.

The type inference strategy assumes expressions follow standard evaluation rules. The strategy can be modified to handle non-standard evaluation for specific functions by changing how constraints are generated for calls to those functions. In some cases it is also be possible to use an analysis and transformation before type inference in order to rewrite code which uses non-standard evaluation as code which uses standard evaluation. For example, the `%>%` pipe operator provided by the **magrittr** package (Bache and Wickham 2020) inserts the expression in its first operand as the first argument of the expression in its second operand. That is, `x %>% f()` is equivalent to `f(x)`. Since this is effectively a syntactic transformation, we can create a transformation pass which rewrites code that uses the pipe operator as ordinary calls.

R also provides functions to parse or construct and to evaluate expressions at run-time, a process called *metaprogramming*. For example, the following code constructs the call `sum(1, 2)` and evaluates it:

```
1 ex = call("sum", 1, 2)
2 eval(ex)
```

As it does for any call, the type inference strategy will attempt to infer the type of the result from a call to `eval` based on how it's used in subsequent code. However, the strategy does not use any information from the arguments to the call—it does not attempt inference on code which is constructed at run-time.

2.5 The Type Inference Packages

We implemented a prototype of the type inference strategy as two R packages. The **typesys** package provides data structures to represent terms in the grammar of types (Section 2.4.1) and functions to unify types (unification is explained in Section 2.3). The **RTypeInference** package provides functions to generate constraints from R code and to resolve constraints (Section 2.3). The **RTypeInference** package depends on both **typesys** and **rstatic** (see Chapter 1). This modular design is intended to make it convenient to use **rstatic** and **typesys** in other applications.

This section contains two subsections. Section 2.5.1 introduces the **typesys** package, while Section 2.5.2 introduces the basics of the **RTypeInference** package.

2.5.1 The typesys Package

The **typesys** package provides S4 classes to represent terms in the grammar of types. It also provides functions to unify terms. This section briefly describes the classes in the package, and then provides a few examples of creating and unifying terms.

Using S4 classes to represent terms has safety, extensibility, and organizational benefits. S4 ensures that constructed objects have valid components, and users can create their own terms in the grammar by extending the classes provided by the package.

The classes are organized into a hierarchy with the `Term` class at the root. The `Term` class is a virtual class which represents a general term in the grammar. It has slots `class` and `dim` for storing information collected during type inference about class and dimensions, respectively. Classes which correspond to concrete R types have names that begin with `R`. Important subclasses of `Term` are:

- The `Scalar` class, which represents atomic scalar types in the grammar. The package provides a subclass of `Scalar` for each of R's atomic types. For instance, its subclass `RInteger` represents the `Integer` type.
- The `Composite` class, which represents terms that contain other terms as components. The class has a slot `components` for storing components. Important subclasses of `Composite` are:
  - The `RVector` class, which represents an R vector. As a convention, the element type of the vector is the first and only component.

  - The `RList` class, which represents an R list. Since list elements can have heterogeneous types, the class uses a component for each element. The class has a slot `indexes` for the inferred position of each element. Example 35 examines this class in more detail.
  - The `RFunction` class, which represents an R function. As a convention, the return type is the first component. Subsequent components are the argument types.

- The `Variable` class represents a type variable. Since this class inherits `Term`, type variables can be used in any expression of the grammar where types can be used. The `Variable` class has a slot `bound` for a list of types the variable is restricted to, and a slot `name` for the unique name of the type variable. The package provides functions to generate unique names for type variables.

Creating Terms

Now we turn to some examples of creating terms with **typesys**.

Example 34. The code to create the type `Vector[Numeric]` is:

```
ty = RVector(RNumeric)
```

The element type of an `RVector` can be accessed with the `[[` indexing operator. So `ty[[1]]` returns the element type for the vector. The element type can also be accessed directly through the `components` slot. □

Example 35. Consider this list:

```
list(1.1, TRUE, list("hi", "hello"))
```

The code to construct the type of this list is:

```
RList(RNumeric, RLogical, RList(RString, RString))
```

The positions of the element types in the `components` slot do not correspond to the positions of the elements in the list. Instead, the `index` slot stores a vector with the position to which each element type corresponds. This is necessary for type inference because the positions of the elements cannot always be determined statically. In that case, the element type is still recorded in the `components`, but its entry in the `index` slot is `NA`.

In a named list, element types are associated with names rather than or in addition to positions. The `RList` class can represent named lists. For example, the code to create a list type with an integer element named "a" and string element named "b" is:

```
RList(a = RInteger, b = RString)
```

Types for elements at unknown positions can also optionally be named. The code to create a list type with an unnamed integer element and a string element named "hi", both at unknown positions, is:

```
RList(.na_index = list(RInteger, hi = RString))
```

Listing 2.24: Use the `.na_index` parameter to add element types at unspecified positions in the list.

The `index` for this type is `NA NA`, since the positions are unknown. □

Example 36. Consider the type signature of the `sin` function. The function has exactly one parameter, `x`. The argument to `x` can be numeric, complex, or any type that can be coerced to numeric. The return type is numeric if the argument is numeric, and complex if it's complex.

We can represent the type signature by creating a bounded type variable. The variable is bounded by the `Vector[RNumeric]` and `Vector[RComplex]` types, meaning it cannot assume any other types. The code to create the bounded type variable is:

```
bound = list(RVector(RNumeric), RVector(RComplex))
t1 = Variable("t1", bound = bound)
```

Then the code to construct the type signature for the function is:

```
sig = RFunction(t1, t1)
```

The `RVector` type is valid for length-1 vectors, so this type signature is appropriate for the `sin` function even if the argument might be a scalar.

Since R's implicit coercion rule is built into how **typesys** unifies terms, it is not necessary to represent types that can be coerced to numeric in the type signature. □

Unification

The **typesys** package's `unify` function unifies two terms, returning a substitution which makes them equal. This subsection provides a brief example of the `unify` function.

Example 37. Suppose we know that the type `RVector(RComplex)` is equal to the type variable `t1` based on an expression in the source code being analyzed. Since `RVector(RComplex)` does not contain any type variables and is (trivially) equivalent to itself, we can conclude that all

instances of `t1` can be replaced by `RVector(RComplex)`. That is, the substitution to make is `t1` ≔ `RVector(RComplex)`.

The **typesys** code to find the substitution in this example is:

```
1 ty = RVector(RComplex)
2 tvar = Variable("t1")
3 result = unify(ty, tvar)
```

Listing 2.25: The entry point for unification in **typesys** is the generic `unify` function.

The result from the `unify` function is an S4 `Substitution` object, which represents a substitution of zero or more type variables.

Generally, we want a substitution so that we can apply it to terms in the type language. We can apply a `Substitution` to a term by calling the object on the term (like a function). For instance, the code to apply the substitution to `tvar` is `result(tvar)`, and this call returns the type `RVector(RComplex)`. Applying the substitution to a term that does not contain the substituted type variable `t1` returns the term unchanged. Substitutions can also be composed by applying one substitution to another. □

2.5.2 The RTypeInference Package

The **RTypeInference** package is a prototype implementation of the type inference strategy described in Sections 2.3 and 2.4. This section provides a brief overview of how to use **RTypeInference**.

The two steps in the type inference strategy, constraint generation and constraint resolution, correspond to two functions provided by the package:

1. The `generate_constraints` function performs the constraint generation step. The input to the function is source code—as an **rstatic** control flow graph in SSA form—and optionally a named list of type signatures for functions called in the code.

2. The `resolve_constraints` function performs the constraint resolution step. The input to the function is the output from the `generate_constraints` function—a `Constraints` object.

The next example shows the details of using these two functions.

Example 38. The goal of this example is to infer types for the code to generate observations from a linear model (Example 13 at the beginning of the chapter). The code is:

```
1 b0 = 10.2
2 b1 = 4
3 n = 100
4 x = runif(n)
5 e = rnorm(n)
6 y = b0 + b1 * x + e
```

Listing 2.26: The code from Example 13, to generate 100 observations from a linear model.

We can use the **rstatic** function `quote_cfg` to convert this code into a CFG in SSA form. Then we can call the **RTypeInference** function `generate_constraints` on the CFG in order to generate type variables and constraints. As described in Section 2.3, the constraint generation step creates an assumption set to keep track of the type variables associated with expressions in the code and a list of constraints on those type variables.

The `generate_constraints` function returns the assumption set and list of constraints as a `Constraints` object. The S4 class `Constraints` has a slot `map` for the assumption set and a slot `constraints` for the list of constraints.

The assumption set is represented by the S4 class `SymbolMap`, which is a list-like data structure. The structure of this class differs from the assumption sets described in Section 2.3: each element corresponds to one SSA name from the code instead of one expression. This is a design decision intended to make it easier to look up types for specific variables. Each element stores information about all type variables associated with that SSA name, and also the type of the definition associated with the SSA name (if the definition is included the target code).

The assumption set for the code in Listing 2.26 is:

```
1 <RTypeInference::SymbolMap>
2 `b0_1` defined as RNumeric
3   used as t9
4 `b1_1` defined as RNumeric
5   used as t11
6 `n_1` defined as RNumeric
7   used as t2; t5
```

```
`runif` no definition
  used as t1
`x_1` defined as t3
  used as t12
```

Listing 2.27: The assumption set from calling `generate_constraints` on the code in Listing 2.26.

As an example of how to interpet the assumption set, this assumption set indicates `b0_1` is defined as a numeric scalar. Since only one type variable (`t9`) is associated with `b0_1`, we can conclude that `b0_1` is only used once in the code.

The list of constraints is an ordinary R list. Each element is one constraint, represented by the S4 classes `Equivalence` (for equality constraints) and `ImplicitInstance` (for instance constraints). The constraints constrain the type variables in the assumption set. Note that the `generate_constraints` function automatically converts instance constraints to equality constraints if the right-hand side is a concrete type (that is, does not contain any type variables).

The list of constraints for the code in Listing 2.26 is:

```
...
[[4]]
t4 == (t5) → t6
  from rnorm(n_1)
[[5]]
t9 == RNumeric
  from b0_1
[[6]]
t11 == RNumeric
  from b1_1
[[7]]
t12 instance of t3
  from x_1
[[8]]
t10 == (t11, t12) → t13
  from b1_1 * x_1
...
```

Equivalence constraints are denoted by `==`, while instance constraints are denoted by `instance of`. Each constraint includes a reference (in the `src` slot) to the expression in the code that caused the constraint to be generated.

For example, the last constraint shown means that the type variable `t10` is equal to a function with argument types `t11` and `t12` and return type `t13`. This constraint is generated from the expression `b1_1 * x_1`. In the assuption set, `t11` is a type variable for `b1_1`, and `t12` is a type variable for `x_1`. So this constraint is on the type of the multiplication operator `*` at this call site.

The `generate_constraints` function does not make any assumptions about symbols that are not defined in the target code, and does not attempt to look up their definitions externally. For the code in Listing 2.26, the symbols `runif`, `rnorm`, `+`, and `*` are defined elsewhere (they are built-in R functions). Notice that these are listed as having no definition in the assuption set in Listing 2.27. This lack of information does not prevent type inference, but the results will be more precise if we provide the `generate_constraints` function with an initial assumption set which contains the type signatures for these functions.

We can call `resolve_constraints` on the `Constraints` object in order to resolve the constraints. The result is a substitution (a **typesys** `Substitution` object) that solves the system of constraints, so that as many type variables as possible are eliminated or mapped to concrete types. The substitution can be applied to the assumption set to determine which types were found for each SSA name in the code. The assumption set for the code in Listing 2.26 after applying the substitution is:

```
<RTypeInference::SymbolMap>
`b0_1` defined as RNumeric
  used as RNumeric
`b1_1` defined as RNumeric
  used as RNumeric
`n_1` defined as RNumeric
  used as RNumeric; RNumeric
`runif` no definition
  used as (RNumeric) → t3
`x_1` defined as t3
  used as t19
```

```
12 `rnorm` no definition
13   used as (RNumeric) → t6
14 `e_1` defined as t6
15   used as t20
16 `+` no definition
17   used as (t14, t20) → t16; (RNumeric, t13) → t14
18 `*` no definition
19   used as (RNumeric, t19) → t13
20 `y_1` defined as t16
21   used as t21
```

The type inference strategy inferred the exact type for the constants `b0`, `b1`, and `n`, and also inferred that `runif`, `rnorm`, `+`, and `*` are functions. Since we did not provide type signatures for these functions, the types for the other SSA names are mostly still type variables, which means they are unknown. For instance, the type inference strategy was unable to determine the type of `y`. □

The type inference functions will produce better results if we provide better initial information. In particular, we need to provide type signatures for the built-in functions. We can define the type signatures manually using **typesys** and pass them to the `generate_constraints` function.

Adding a database of type signatures for R's built-in functions is one way the **RTypeInference** package could be improved in the future. The database would likely have to be created manually because many of R's built-in functions are not written in R code. It may be possible to apply the type inference strategy for C code from Chapter 3 to create the database. However, unlike the foreign routines discussed in that chapter, R's built-in routines call private routines that are not part of the R Internals programming interface, and the strategy would have to be adapted to handle these.

Example 39. The results from Example 38 improve if we specifying type signatures for the built-in functions called by the code in Listing 2.26. We can pass type signatures into the `generate_constraints` function by defining an initial assumption set which contains an entry for each of the functions. Here's the code to create an empty assumption set and add the type signature (called `runif_sig`) for `runif`:

```
1 map = SymbolMap()
```

```
map$runif = runif_sig
```

The `generate_constraints` function accepts an initial assumption set as its second argument. The rest of the type inference process remains the same as described in Example 38.

With the type signatures, the type inference functions are able to compute correct types for more expressions:

```
<RTypeInference::SymbolMap>
`b0_1` defined as RNumeric
  used as RNumeric
`b1_1` defined as RNumeric
  used as RNumeric
`n_1` defined as RNumeric
  used as RNumeric; RNumeric
`runif` defined as (RInteger, RNumeric, RNumeric) → RNumeric
  used as (RNumeric, RNumeric, RNumeric) → RNumeric
`x_1` defined as RNumeric
  used as RNumeric
`rnorm` defined as (RInteger, RNumeric, RNumeric) → RNumeric
  used as (RNumeric, RNumeric, RNumeric) → RNumeric
`e_1` defined as RNumeric
  used as RNumeric
`+` defined as (RNumeric, RNumeric) → RNumeric
  used as (RNumeric, RNumeric) → RNumeric; (RNumeric, RNumeric) → RNumeric
`*` defined as (RNumeric, RNumeric) → RNumeric
  used as (RNumeric, RNumeric) → RNumeric
`y_1` defined as RNumeric
  used as RNumeric
```

Definitions in the assumption set automatically override definitions in the code, so we can also use the assumption set to override the type associated with a specific SSA name. This is especially important for accommodating type annotations.

As of writing there is no official standard for annotating types in R code, but several annotation syntaxes have been proposed as extensions (rather than additions) to the language.

Type annotations can be handled by a separate analysis pass that populates an assumption before constraint generation. This way the type inference strategy can support several different kinds of annotations, provided someone develops an analysis pass to generate the assupmtion set for each. □

2.6 Related Work

Temple Lang (2014) was the first to propose compiling R code using the LLVM Compiler Infrastructure. He identified type inference as a key requirement for compiling R code, and took initial steps to explore type inference for R. His work is the main motivation for our own.

Sen et al. (2017) investigated using static analysis and type inference to improve the performance of R code. They developed a software system, ROSA, that can analyze R code, apply optimizations, and translate the result into C++ code that uses the **Rcpp** package. The system focuses on a specific set of optimizations the authors identified as likely to improve performance, and only translates a subset of the language. ROSA provides improvements in run-times and memory usage over the R interpreter in a variety of benchmark examples. ROSA is implemented in C++, which makes it more difficult to understand, extend, and adapt. The type inference rules for R functions are hard-coded, so adding new rules to the system (for example, for other functions) is relatively difficult. Their type inference system is not designed for translating R objects into native C types, which is part of why they rely on **Rcpp**.

Damas-Milner type inference is one of the two major branches of type inference research. The original Damas-Milner type inference strategy was designed for functional languages with immutable data structures. These characteristics are why we choose the Damas-Milner strategy as a basis for our type inference strategy. One of the key innovations of the Damas-Milner strategy at the time it was published is the way it represents polymorphism with type variables that can be filled in with any possible type.

The strategies from other branch of type inference research generally do not use type variables; instead, they traverse the code and collect a set of possible types for each expression. An early example of this approach is the Cartesian Product Algorithm (CPA), which was proposed by Agesen (1995). Concrete types are especially relevant to object-oriented languages, since method dispatch depends on argument types. Moreover, a compiler can use concrete types to select specialized methods during compilation. R is relatively unique in how it blends functional and

object-oriented programming: idiomatic R code uses map operations (apply functions) and closures, but also uses (S3) classes and method dispatch. It is unclear whether CPA-based type inference would be more effective than Damas-Milner type inference for R, but there is a clear need for concrete type information.

Recent type inference research focuses on dynamically typed programming languages. *Soft typing* (Aiken et al. 1994) extends the Damas-Milner type system to include types that are conditional on control flow in the code. This makes it possible to eliminate type checks selectively, so that checks are only made for objects whose type is truly dynamic. Since this is an extension of the Damas-Milner type system, it is feasible to modify **RTypeInference** to use this approach. However, the solver is relatively complicated compared to unification, and we are not aware of any type inference software based on soft typing in widespread use.

Even more recently, Siek and Taha (2006) proposed *gradual typing*, where a dynamically typed language has optional type annotations, but type checks are seamlessly and transparently eliminated anywhere annotations and inference provide sufficient information. In other words, gradual typing puts control over whether code is dynamically or statically typed in the hands of the programmer. This approach is notable because it was implemented for Python as Reticulated Python (Vitousek et al. 2014), which in turn led to the officially-supported mypy type checker package (Lehtosalo et al. 2015). Gradual typing is an interesting option for R, since it is designed for dynamically typed languages, accommodates old code without type annotations, and allows for static type checking and performance improvements for new code. Turcotte and Vitek (2019) have signaled they are investigating gradual typing for R and may use it in forthcoming work.

2.7 Conclusion

Type inference is a static analysis which attempts to determine the type of the result of each expression in the target code. This type information is essential for code translation or compilation, and can also be used for checking for errors before run-time and summarizing code. Type inference is necessary to get type information for R code because R code does not contain type annotations.

The type inference strategy presented in this chapter is based on the Damas-Milner type inference strategy, which was originally designed for the ML programming language. We chose to use DM type inference as the basis for our strategy because both ML and R are functional

programming languages where functions typically do not have side effects most objects are immutable.

The strategy is divided into two steps: constraint generation and constraint resolution. During the constraint generation step, the strategy creates a type variable for each expression in the code, and then generates constraints on the type variables based on the kind of expression and how the result of the expression is used in other expressions. Constraint resolution solves for the values of the type variables and thereby produces a type for each expression in the code.

Many changes are necessary to make the Damas-Milner type inference strategy work on R code. Our strategy uses R code in static single assignment form, which was described in Chapter 1. The SSA form is important because it enables the type inference strategy to associate different types with different definitions of the same variable. Our strategy also uses a grammar of types designed specifically for R. The grammar includes all of R's built-in types, and also includes many of R's built-in S3 classes and special terms to handle specific language features. The type inference strategy attempts to infer S3 classes in addition to types, because information about classes is necessary to determine which methods will be called by S3 generic functions at run-time. The type inference strategy also builds R's implicit coercion rules into how it handles one kind of constraint. Because many R functions can accept many different types as arguments or return many different types, it is not always possible to infer a single, specific type for every expression. When it is not, the type inference strategy still attempts to produce a bounding set of potential types for each expresssion.

We created two R packages, **typesys** and **RTypeInference**, as a prototype implementation of the type inference strategy. The **typesys** package provides data structures to represent types in the grammar of types and to represent constraints. The package also provides functions to unify types by finding a substitution of type variables which makes them equal. The **RTypeInference** package uses **rstatic** and **typesys** to implement the constraint generation and constraint resolution steps.

Chapter 3

Type Inference for the R API

3.1 Introduction

The type inference system presented in Chapter 2 is designed to analyze R code, but R code can make calls to *foreign routines* written in C, C++, and Fortran. The specific parameter types and return types of these foreign routines can provide information about types in the calling R code (to be clear, by *parameter types* we mean the assumptions a routine makes about the types of its arguments, in contrast to *argument types*, the actual types of the arguments at a specific call site). We'd like to collect the parameter types and return types for foreign routines in order to incorporate them into the type inference system. This chapter describes one strategy for doing so, but first we'll see why doing so is non-trivial.

R provides five different interfaces with which to call foreign routines: `.C`, `.Fortran`, `.Call`, `.External`, and `.External2`. The latter three are designed specifically for calling routines that accept R objects as arguments and return R objects as results. In these routines (and in the R interperter itself), each R object is represented by a `SEXP`, a pointer to a `SEXPREC` data structure that contains the object's value and metadata. As a consequence, even though C and C++ require explicit type annotations, the annotation for an R object will be `SEXP`, regardless of the object's underlying R type. Thus for routines called with `.Call`, `.External`, or `.External2`, recovering the type information requires type inference on the routine's code. The next example demonstrates this with a C routine suitable for the `.Call` interface.

Example 40. The `out` routine, which computes the outer product of two vectors, is a canonical example of a computation worth writing in C for efficiency, taken from the Writing R Extensions manual (R Core Team 2019b). Here's the code for the routine:

```
#include <R.h>
#include <Rinternals.h>
SEXP out(SEXP x, SEXP y)
{
    int nx = length(x), ny = length(y);
    SEXP ans = PROTECT(allocMatrix(REALSXP, nx, ny));
    double *rx = REAL(x), *ry = REAL(y), *rans = REAL(ans);
    for(int i = 0; i < nx; i++) {
        double tmp = rx[i];
        for(int j = 0; j < ny; j++)
            rans[i + nx*j] = tmp * ry[j];
    }
    UNPROTECT(1);
    return ans;
}
```

Listing 3.1: The `out` routine computes the outer product of two numeric vectors (R Core Team 2019b). The calling R code is responsible for ensuring the arguments are numeric vectors.

The routine's two parameters and return value all have C type `SEXP`, meaning they are R objects. Their R type (or `SEXPTYPE`), the result of calling `typeof` on them in R, is not explicitly declared in the C code. Until we examine more of the code, we can't tell whether they are numeric vectors, character vectors, lists, or other types of R objects. That is, if we want the R types, we must infer them.

As we'll see throughout this chapter, the key to type inference for R objects in foreign code is inspecting the call where the object is defined and all calls where the object is used as an argument. We'll refer to these as *definitions* and *uses*, respectively.

The R Internals programming interface, included in R and declared by the header file `Rinternals.h`, is the primary mechanism for creating and manipulating R objects in C-compatible code. Many of its routines require that their arguments have specific R types, so when they are called on an object, they refine the set of possible types for the object. In addition, many of the routines return a result with a specific R type, so they also provide type information when they are used in a definition.

As a demonstration, consider the variable `ans`. We can see that `ans` is defined by a call to

the `allocMatrix` routine (we ignore the `PROTECT` macro, since it does not affect R types). This routine is provided by the R Internals interface. It returns a matrix, but the element type depends on the first argument. In this case, the first argument is the constant `REALSXP`, which is part of an enumeration in the `Rinternals.h` header file. This constant is the `SEXPTYPE` that corresponds to numeric vectors. Thus we can infer that `ans` is a numeric matrix, and therefore the `out` routine returns a numeric matrix.

We can also use type inference to deduce the parameter types for the parameters `x` and `y`. Let's take `x` as an example; the process for `y` is identical up to the parameter name. Since `x` is a parameter, there is no definition to provide type information, so we focus exclusively on how `x` is used. The first use of `x` is in a call to the `length` routine. The `length` routine accepts any R object as its argument, similar to the `length` function in R, so this use doesn't help us infer the type. The next use of `x` is in a call to the `REAL` routine. This is much more useful—the `REAL` routine expects its argument to be an R object with type `numeric`. So we can infer that `out` expects the argument for the parameter `x` to be numeric. Likewise, the argument for `y` is also expected to be numeric. □

The example shows that type inference is necessary for routines called with the `.Call`, `.External`, and `.External2` interfaces, because the R types are not explicitly declared in the code. It also shows that type inference is, in fact, feasible for such routines. In order to automate this process, we need a robust type inference algorithm.

Calls to foreign routines play an important role in R programming, so augmenting the type inference system of Chapter 2 to handle them is important. Routines written in C, C++, and Fortran typically run orders of magnitude faster and consume less memory than equivalent functions written entirely in R (Morandat et al. 2012). Moreover, R code can call routines in preexisting software libraries, so that the functionality of these libraries does not have to be reimplemented specifically for R.

In 2020, we found that 23.5% (or 4004) of the 16,987 packages on the Comprehensive R Archive Network (CRAN) contained C, C++, or Fortran source code. Of those, about 54% contained C code and 62% contained C++ code (some contained both). Less than 1% (137 packages) contained Fortran code.

This chapter develops a strategy to infer the types of R objects in C code. We chose to focus on C code for this initial investigation because the syntax and semantics are relatively simpler than C++ code, and C code is substantially more popular in CRAN packages than Fortran

code. We also discuss the potential for extending the strategy to C++ and Fortran.

Type inference is not necessary for foreign routines called with the `.C` and `.Fortran` interfaces. These interfaces are designed for calling routines that do not operate directly on R objects. Instead, the interfaces unbox R objects into standard C types to pass to routines. Nevertheless, this chapter also documents how to collect the type information provided by calls made with these interfaces.

Here is how the chapter is organized:

- Section 3.2 provides additional necessary background information. In particular, this section discusses the relative merits of two different but related tools for analyzing C and C++ code: LibClang and the LLVM Compiler Infrastructure. It also provides a brief introduction to the LLVM intermediate representation for code.

- Section 3.3 describes a strategy to collect information from foreign routines called with the `.C` and `.Fortran` interfaces. Since routines designed for these interfaces do not directly manipulate R objects, type inference is not necessary, but they still provide type information we'd like to collect.

- Section 3.4, our primary contribution, describes an algorithm for type inference on C routines called with the `.Call`, `.External`, and `.External2` interfaces. The algorithm is developed gradually by way of examples that expose necessary features.

- Section 3.5 describes how to connect results from this chapter to the type inference for R code described in Chapter 2. The types we infer for R objects in foreign routines are the same as the types we infer for R objects in R code, so we can reuse the vocabulary of types and type variables developed in Chapter 2 and provided by the **typesys** package.

- Section 3.6 describes related work by other researchers.

3.2 Background

The type collection and inference strategies described in this chapter are implemented in R. The main reason is that this allows us to build on and integrate with existing packages, particularly the packages described in Chapter 2. A language like C or C++ would provide better run-time performance, but prevent us from using R packages and act as a barrier to entry for other

members of the R community interested in contributing. Tools implemented in R are anecdotally easier to debug, modify, and extend.

We considered two different R packages as candidates for analyzing C code. Subsection 3.2.1 provides an overview of both and explains which we chose to use and why. Following that, Subsection 3.2.2 briefly introduces the LLVM intermediate representation, the medium for code analysis in our strategy.

3.2.1 Packages for C Code Analysis

The first candidate for analyzing C code from R is the **RCIndex** package (Temple Lang 2011). The package is a binding to LibClang, the source code analysis library that powers the Clang C compiler. The LibClang library provides routines for parsing and navigating C and C++ code as abstract syntax trees (ASTs). This data structure was explained in Chapter 1 for R code; see Section 1.2 for an overview.

Abstract syntax trees emphasize the syntactic structure of source code, but for type inference we are more interested in the flow of control (branches and loops) and the flow of data (definitions and uses) in the code. We already saw the importance of the latter in Example 40. In an AST, control flow expressions and their contents appear in the same order as the original code, so additional analysis is necessary to determine whether and where variables are conditionally defined or redefined (including in loops). The LibClang library does not provide routines to do this analysis, nor does it provide routines to find the definitions and uses of variables. Thus we would need to develop our own, for instance by traversing the AST with a recursive function or a loop and stack.

On the other hand, ASTs retain the syntactic structure of the original source code, so they are relatively easy to understand and begin using, provided one has a thorough understanding of the represented source language.

The second candidate for analyzing C code from R is the **Rllvm** package (Temple Lang 2014). The package is a binding to the LLVM compiler infrastructure. LLVM is a complete set of tools for constructing a compiler, and is used in popular compilers for a variety of languages, including C, C++, and Fortran. LLVM provides routines for navigating and extracting information from the LLVM intermediate representation (IR). The LLVM IR is a low-level, language-agnostic format that represents code as a control flow graph (CFG) in static single assignment (SSA) form. CFGs and SSA form were described in Sections 1.3 and 1.4.2 for R code, but we also

explain more about the LLVM IR specifically in Subsection 3.2.2.

Control flow graphs emphasize the control flow in the source code, and SSA form adds explicit data flow information. In particular, in SSA form, each variable name corresponds to a single definition in the source code. If a variable is redefined at some point, a different name is assigned to the second definition. As a consequence, any use of a variable can be correctly matched to its specific definition by name. Additional analysis to check for redefinitions is not necessary. LLVM even provides routines to find the definition and all uses of a given variable. Moreover, instructions in the LLVM IR are less complex than expressions in an AST, since they cannot contain other instructions.

The drawbacks of the LLVM IR are that the structure and variable names often differ significantly from the original source code, and that it is relatively difficult to understand and begin using without prior experience with CFGs and SSA form.

Unlike LibClang, LLVM provides routines and shell tools to optimize code. These optimizations reduce the size and complexity of the IR without changing its effect, usually at the cost of making it more difficult to match instructions in the IR to expressions in the original source code. Some of these optimizations are helpful for type inference. For instance, constant folding replaces constant variables with their literal value, and literals have known types.

We believe the benefits of LLVM outweigh the upfront cost of learning to work with the IR, so we chose to use **Rllvm** to implement the type inference strategy.

3.2.2 The LLVM Intermediate Representation

This section shows how to generate the LLVM intermediate representation from source code and explains its overall structure.

Example 41. Let's revisit the `out` routine from Example 40. Suppose the code is stored in the file `out.c`. In order to use LLVM (and **Rllvm**) to analyze the code, we must first convert the code into LLVM IR. We can do this with the `clang` compiler. In a UNIX shell, the command is:

```
1 clang -fno-discard-value-names -S -emit-llvm out.c
```

Listing 3.2: The `clang` command to run in a UNIX shell in order to generate LLVM IR from a C source file.

The `-fno-discard-value-names` flag instructs `clang` to preserve the original variable names from the C code where possible. The `-S` flag instructs `clang` to emit the IR as human-readable

"assembly" code, which we'll learn more about later in this example (without the `-S` flag, `clang` emits the IR as bitcode, a binary format). Finally, the `-emit-llvm` flag instructs `clang` to generate IR rather than a compiled executable or library. The command produces the file `out.ll`, which contains the IR code.

In the LLVM IR, a *module* is a top-level container for all other code objects. A module can contain routines, global variables, data structures, and other information. For C and C++ code, a module corresponds to one or more *translation units*, the combined code after processing all preprocessor instructions in a source file. The `out.ll` file is a single module.

In a module, each routine's code is arranged into *blocks*. Each block contains a sequence of *instructions* that run uninterrupted, without branching. The last instruction in each block is always a *terminator instruction* that branches to another block or exits the routine. The IR's structure is very similar to the control flow graphs for R code described in Section 1.3.

Within a block, instructions can call routines, carry out arithmetic, make comparisons, access memory, and perform other primitive operations. Only terminator instructions can branch to other blocks.

For every instruction that produces a result, the LLVM IR automatically defines a variable to refer to that result. Since the IR is in static single assignment form, each variable name is unique, and cannot be redefined. Local variables are always prefixed with `%`, and global variables are always prefixed with `@`.

Let's look at a single instruction from the `out.ll` file:

```
1  %call1 = tail call i32 @Rf_length(%struct.SEXPREC* %y) #2
```

This instruction returns a result, which is assigned to a local variable, `%call1`. The instruction itself (on the right-hand side of the equal sign) begins with `tail call`, so it is a `call` instruction. Call instructions call another routine and return its result. The prefix `tail` indicates a possible optimization, but can be safely ignored for our purposes. The name of the instruction is followed by arguments. For a `call` instruction, the first argument is the type of result returned by the routine to be called. In this case, it's `i32`, a 32-bit integer. The second argument is the name of the routine to call and its arguments. This instruction calls the `@Rf_length` routine with the variable `%y` as the only argument. The `#2` at the end of the instruction indicates that this instruction is subject to attributes defined elsewhere in the module; we generally won't need these attributes for type inference. Other instructions follow the same pattern: a variable

definition, the instruction's name, and then the instruction's arguments. For instructions that do not return a result, the variable definition is omitted.

Now consider the type signature and entire first block of the `out` routine. The first block is called the `entry` block, since it's the entry point to the routine. In the IR, blocks always begin with their name and a colon:

```
1  define %struct.SEXPREC* @out(
2      %struct.SEXPREC* %x, %struct.SEXPREC* %y) local_unnamed_addr #0 {
3
4  entry:
5    %call = tail call i32 @Rf_length(%struct.SEXPREC* %x) #2
6    %call1 = tail call i32 @Rf_length(%struct.SEXPREC* %y) #2
7    %call2 = tail call %struct.SEXPREC* @Rf_allocMatrix(
8        i32 14, i32 %call, i32 %call1) #2
9    %call3 = tail call %struct.SEXPREC* @Rf_protect(
10       %struct.SEXPREC* %call2) #2
11   %call4 = tail call double* @REAL(%struct.SEXPREC* %x) #2
12   %call5 = tail call double* @REAL(%struct.SEXPREC* %y) #2
13   %call6 = tail call double* @REAL(%struct.SEXPREC* %call3) #2
14   %cmp42 = icmp sgt i32 %call, 0
15   br i1 %cmp42, label %for.body.lr.ph, label %for.cond.cleanup
```

Listing 3.3: The type signature and first block of the LLVM IR for the `out` routine from Example 40.

The entry block contains several `call` instructions, as well as an `icmp` instruction and a `br` instruction. The `br` instruction is an example of a terminator instruction that specifies which block to branch to next. The branch can depend on a condition. If-statements, loops, and other conditional control flow structures are translated into the IR as multiple blocks connected by conditional branch instructions. Here the `br` instruction depends on the value of the variable `%cmp42`. When the value is true, the program branches to the `for.body.lr.ph` block. When the value is false, the program branches to the `for.cond.cleanup` block.

By default, `clang` does not apply any optimizations to the emitted LLVM IR. We can enable optimizations by adding the `-O` flag to the command in Listing 3.2. The flag should be followed by a number that indicates the level of optimization from 0 (no optimization) to 4. Higher

levels of optimization tend to reduce the size of the generated IR, but also produce IR that is structurally less like the original source code. The effect of the code remains the same, but computations may be rearranged and unnecessary computations eliminated. We recommend using at least optimization level 1, because `clang` annotates the code with additional information collected during optimization. For instance, with optimizations on, `clang` attaches a `readonly` attribute to parameters in routines that are not modified by the code in the routine (we'll see why this information is important in Section 3.3). □

We will explain more details of the LLVM IR as needed in the examples in this chapter. For a longer introduction to the IR, see *LLVM Essentials* (Sarda and Pandey 2015), and for a complete reference, see the LLVM documentation (`https://llvm.org/docs/`).

3.3 The `.C` and `.Fortran` Interfaces

The `.C` and `.Fortran` interfaces are intended for calls to C and Fortran routines, respectively, that do not operate directly on R objects. The `.C` interface can also be used with other languages that are C-compatible, such as C++. This section describes how to collect type information from foreign routines called with the two interfaces and provides an example. Subsection 3.3.1 addresses the `.C` interface, and Subsection 3.3.2 addresses the `.Fortran` interface.

When one calls a foreign routine with either interface, the interface automatically unboxes the R objects passed as arguments into C and Fortran types. The mapping from R types to C and Fortran types is shown in Table 3.1. It is possible to pass other types of R objects, but is only supported for backwards compatibility with old code (R Core Team 2019b). Since R does not distinguish between scalar and vector types, the `.C` interface maps all R types to C pointer types. Additionally, complex numbers are mapped into the `RComplex` type, which is defined by a header file provided with R. Like R, Fortran does not distinguish between scalar and vector types.

The type inference strategy from Chapter 1 can usually infer the actual argument types for a call to a foreign routine from the surrounding R code. However, this approach assumes that the arguments are correctly specified, rather than checking the signature of the foreign routine for its parameter types. Since the `.C` and `.Fortran` interfaces do not check the parameter types at run-time either, it is considered good practice to explicitly check or coerce the arguments in the R code preceding the call. When checks or coercions are present, they reduce the likelihood of a

R Type	C Type	Fortran Type
logical	`int *`	`INTEGER`
integer	`int *`	`INTEGER`
double	`double *`	`DOUBLE PRECISION`
complex	`Rcomplex *`	`DOUBLE COMPLEX`
character	`char **`	`CHARACTER(255)`
raw	`unsigned char *`	none

Table 3.1: Mapping from R types to C types for the `.C` interface and Fortran types for the `.Fortran` interface. This table was originally published in *Writing R Extensions* (R Core Team 2019b).

type error and provide additional information for type inference.

Examining the type signature and source code for foreign routines is a simpler and more direct approach which avoids making any assumptions. After extracting the type signature for a foreign routine with **Rllvm**, we can statically type check arguments by comparing their inferred types to the parameter types. On the other hand, if the R code does not provide enough information to infer the actual argument types, we can use the parameter types as an approximation. Finally, the type signature is useful information even for someone that isn't interested in type inference. For example, a code transformation could use the parameter types to programmatically insert checks and coercions into the R code. This prevents type errors, which can cause bugs that are difficult to diagnose.

The `.C` and `.Fortran` interfaces require that foreign routines return results by modifying their arguments rather than by built-in return mechanisms. As a result, collecting the return type from these routines is not useful. However, it is useful to collect information about whether each argument is read from, written to, or both. This information allows us to determine which arguments contain return values, avoid copying arguments that will not be modified, and potentially to statically detect incorrect usage of a routine. Thus we will collect this information in addition to type signatures.

3.3.1 The `.C` Interface

This section documents how to collect the type signature and information about which arguments are modified from C routines called with the `.C` interface. The steps to do this with the **Rllvm** package are shown in Listing 3.4.

1. Convert the C code to LLVM IR with `clang`.

2. Parse the IR with the `parseIR` function and get the target routine within the module. Then get the parameters from the routine with the `getParameters` function.

3. For each parameter, get the C type with the `getType` function, translate the C type into an R type, and get the `readonly` flag.

Listing 3.4: The steps to collect the type signature and information about which arguments are modified from a C routine.

The next example illustrates these steps.

Example 42. The `convolve` routine, which convolves two numeric vectors, is another example of a computation one might want to write in C rather than R for efficiency reasons, again from the *Writing R Extensions* manual (R Core Team 2019b). Here's the code:

```
void convolve(double *a, int *na, double *b, int *nb, double *ab)
{
    int nab = *na + *nb - 1;
    for(int i = 0; i < nab; i++)
        ab[i] = 0.0;
    for(int i = 0; i < *na; i++)
        for(int j = 0; j < *nb; j++)
            ab[i + j] += a[i] * b[j];
}
```

Listing 3.5: The `convolve` routine convolves of two numeric vectors (R Core Team 2019b).

Before we can use **Rllvm** to analyze the code, we must compile the code into LLVM IR. Example 41 provides an example of how to do this. Suppose the resulting IR file is named `convolve.ll`.

Next, we use **Rllvm** to parse the IR in `convolve.ll`, get the routine within the parsed module, and then get the routine's parameters. Once we have the parameters, we get the C type of each one. Here's the code for these steps:

```
library(Rllvm)
m = parseIR("convolve.ll")
```

```
convolve = m$convolve
parameters = getParameters(convolve)
param_types = lapply(parameters, getType)
```

Listing 3.6: Code to get the type signature for the `convolve` routine.

The next step is to use the mapping in Table 3.1 in reverse in order to find the R types. To do this, we need some way to represent the R types (**Rllvm** provides a representation for the C types). The simplest approach is to use strings to represent the types, the same as in R itself. For type inference or other analyses that already use the **typesys** package, it is more convenient to use the representations provided by that package. The advantages of **typesys** are described in Section 2.5.1.

Once we've selected a representation, the actual process of translating the C types to R types is a straightforward lookup of the C type in Table 3.1. In the `convolve` routine, the parameters `a`, `b`, and `ab` have C type `double *`, so their equivalent R type is `double`. The parameters `na` and `nb` have C type `int *`, so their equivalent R type is `integer`. We always translate the C type `int *` into the R type `integer`, even though `int *` also corresponds to the R type `logical`. In R, `logical` vectors can be implicitly coerced into an `integer` vectors, so translating to type `integer` allows for both possibilities.

The final step is to collect information about which arguments are modified by the routine. If the code is compiled into LLVM IR with any level of optimization enabled (see Section 3.2.2), then `clang` annotates each parameter that is not modified by its parent routine with a `readonly` flag. We can check for this flag with **Rllvm**. Here's the code:

```
param_readonly = sapply(parameters, onlyReadsMemory)
```

For the `convolve` routine, all of the parameters except for `ab` are read-only. Based on this, we can conclude that the routine returns its result by modifying the argument to the `ab` parameter. The other parameters are only inputs. A limitation of this approach is that it does not tell us whether the routine reads from `ab` before writing to `ab`. □

By analyzing the LLVM IR instructions in a routine, it's possible to overcome the limitation of using the `readonly` flag described in Example 42. One strategy is to check the uses of the parameter in the routine for instructions that read from the parameter. The examples in Section 3.4.2 use a similar approach to infer R types for parameters with C type `SEXP`. We chose

not to investigate this approach further for the `.C` interface since we do not have any immediate applications for information about which parameters are read by a foreign routine.

3.3.2 The `.Fortran` Interface

At the time of writing, there are no stable Fortran compilers based on the LLVM compiler infrastructure, so it is not yet possible to compile Fortran code into LLVM IR. However, the Flang compiler (*Flang* 2021) and the LFortran compiler (*LFortran* 2021), both in the early stages of development, are based on LLVM. The LFortran compiler development team plans to release a viable compiler by Fall 2021.

Once there is software available to compile Fortran code into LLVM IR, we expect that routines called with the `.Fortran` interface can be analyzed with the strategy in Listing 3.4. One aspect that is not yet clear is how Fortran types will be represented in the LLVM IR. It may be necessary to modify the strategy to take this into account.

3.4 The `.Call`, `.External`, and `.External2` Interfaces

The `.Call`, `.External`, and `.External2` interfaces are intended for calls to foreign routines that use the R Internals programming interface. The routines can be written in C, C++, or any other C-compatible language. The routines must return an R object and accept R objects as arguments. The R types of these objects are not explicitly annotated in the code. Nevertheless, we can infer them from the code—as we saw in Example 40—by examining how parameters are used and how variables are defined and used.

Taking a divide-and-conquer approach, we split the type inference problem into two parts: the return type and the parameter types. The strategy for the return type is to examine how the returned value is defined and used prior to being returned. As we'll see, in some cases it's necessary to repeat this process recursively for parameters or variables used in the definition of the return value. On the other hand, the strategy for the parameter types is to search for uses of each parameter in calls to other routines that require specific types. Both strategies rely on LLVM to find definitions and uses.

Both strategies also rely on a database of type signatures for the C routines that compose the R Internals programming interface. Temple Lang (2021) found that CRAN packages use 395 of the provided routines. We manually inspected the source code for 235 of those routines in order

to create the database. Of the remaining 160 routines, none are used more than 6 times across all CRAN packages, and most do not return R objects nor accept R objects as arguments.

Subsection 3.4.1 describes inference for return types. Subsection 3.4.2 describes inference for parameter types. Both of those subsections focus on the `.Call` interface and C routines for clarity of exposition. Subsection 3.4.3 addresses differences that arise for routines designed to be called with the `.External` or `.External2` interfaces. Finally, Subsection 3.4.4 discusses difficulties that might arise in extending the strategy to C++ routines, especially those that use the **Rcpp** package.

3.4.1 Return Types

This subsection describes the strategy to infer the R return type of a routine. The core strategy is to work backwards from where the routine's result is returned, in order to infer its type. To do this, we use LLVM to get the definition and uses of variables. We refine the strategy up gradually in subsequent subsections. Each introduces examples of situations where the core strategy is unable to infer the correct type, and describes improvements to address those situations.

Example 43. This example demonstrates the core strategy. For clarity of exposition, the target routine for type inference is deliberately simple—it always returns the integer 42. Here's the C code for the routine:

```
1 SEXP return42()
2 {
3   SEXP ans = PROTECT(allocVector(INTSXP, 1));
4   INTEGER(ans)[0] = 42;
5   UNPROTECT(1);
6   return ans;
7 }
```

We'll carry out the actual analysis on the LLVM IR. Thus compiling the C code into LLVM IR is a prerequisite—see Example 41 for how to do so.

In LLVM IR, the `ret` instruction exits a routine and returns a value. LLVM arranges the control flow graph so that there is a single *exit* block that contains a `ret` instruction as the terminator instruction.

The `ret` instruction is always the starting point for inferring the return type. First we need to find the `ret` instruction. We can use the **Rllvm** functions `getBlocks` and `getTerminator` to get the list of basic blocks in the routine and get the terminator instruction for each block, respectively. Then we can search the terminators to find the `ret` instruction. The `ret` instruction for the `return42` routine is:

```
1 ret %struct.SEXPREC* %call1
```

The return instruction contains two pieces of information: the C return type (a `SEXPREC` struct), and the value to return (the variable `%call1`). When the C return type is `SEXP`, that doesn't tell us the R specific type, so we need to collect more information from the rest of the code. The most important information in the instruction is the variable `%call1`. In order to determine the return type of the routine, we must determine the R type of this variable.

The next step is to get the instruction where `%call1` is defined. To do this, we can use the **Rllvm** function `getOperand` to get the `%call1` from the `ret` instruction. It is the 1st operand. The **Rllvm** package represents variables as a reference to their definition, so getting a variable also gets the definition. Here's the definition for `%call1`:

```
1 %call1 = call %struct.SEXPREC* @Rf_protect(%struct.SEXPREC* %call) #2
```

This instruction is a call to the `Rf_protect` routine, which is provided by the R Internals programming interface. This routine is related to R's memory management, and returns its argument. So the definition for `%call1` is in terms of another variable, the argument `%call`.

At this point, we recursively apply the type inference procedure to the variable `%call`. That means the next step is to examine its definition, which is:

```
1 %call = call %struct.SEXPREC* @Rf_allocVector(i32 13, i64 1) #2
```

The `Rf_allocVector` routine is another routine provided by the R Internals programming interface. The routine creates a new R vector with the element type and length specified by the first and second arguments, respectively.

The first argument, 13, comes from an enumeration of `SEXPTYPE`s also defined by the R Internals programming interface. The enumeration is shown in Table 3.2. Looking up 13 in the table, we see that it means `INTSXP`, so this call creates an integer vector. In practice, we can do this lookup programmatically provided we first extract the enumeration from the `Rinternals.h` file—either manually or programmatically.

Enum	Type	Description
0	NIL	nil = NULL
1	SYM	symbols
2	LIST	lists of dotted pairs
3	CLO	closures
4	ENV	environments
5	PROM	promises: [un]evaluated closure arguments
6	LANG	language constructs (special lists)
7	SPECIAL	special forms
8	BUILTIN	builtin non-special forms
9	CHAR	scalar string type (internal only)
10	LGL	logical vectors
13	INT	integer vectors
14	REAL	real variables
15	CPLX	complex variables
16	STR	string vectors
17	DOT	dot-dot-dot object
18	ANY	make "any" args work
19	VEC	generic vectors
20	EXPR	expressions vectors
21	BCODE	byte code
22	EXTPTR	external pointer
23	WEAKREF	weak reference
24	RAW	raw bytes
25	S4	S4, non-vector
30	NEW	fresh node created in new page
31	FREE	node released by GC
99	FUN	closure or builtin or special

Table 3.2: All SEXPTYPEs enumerated in the Rinternals.h header file of the R source code. The descriptions here are also from the source code. All SEXPTYPEs are written without the suffix SXP here.

The second argument, 1, is the length of the new vector. Thus the instruction reveals that %call is a scalar integer. It follows that %call1 is a scalar integer, so the return42 routine always returns a scalar integer.

As is the case here, examining definitions is often sufficient to infer the return type of a routine. Nevertheless, the type inference strategy also inspects instructions that use the variables of interest for any additional information they can provide. We can use the **Rllvm** function getAllUsers to get these uses. The variable %call1 is only used once, as an argument in a call to the INTEGER routine:

```
1 %call2 = call i32* @INTEGER(%struct.SEXPREC* %call1) #2
```

Provided by the R Internals programming interface, the INTEGER routine is used to access an INTSXP object as an ordinary C int array. Consequently, this instruction confirms that the

variable `%call1` is an R integer vector, but doesn't provide any new information. □

Identify the return value—the only operand of the `ret` instruction—and apply this type inference procedure:

1. **Inspect the definition**. If the operand is not a parameter, inspect its definition for type information. For `SEXP`s, the definition will usually be a call to an R Internals routine. Look up the called routine in the database of routines. If the called routine's return type depends on the types of its arguments, repeat the type inference procedure for the arguments.

2. **Inspect the uses**. For each other instruction that uses the operand: if the instruction is a call to an R Internals routine, look up the called routine in the database of routines. The routine may put restrictions on the types of its arguments and thus the type of the operand.

Listing 3.7: The core strategy to infer a routine's return type. Subsequent sections of this chapter refine and expand upon the strategy.

The example demonstrates the core strategy to infer return types: start from the routine's result, inspect the definition and uses for more type information, and repeat as necessary if the type of the result depends on other variables. An outline of the strategy is shown in Listing 3.7. Note that routines called with the `.C`, `.External`, and `.External2` interfaces must return an R object, so for these routines the `ret` instruction will always have a variable or parameter as its operand.

Fundamental to this strategy is the fact that a routine can only get an R object through its arguments or by calling other routines—typically the ones provided by the R Internals programming interface. The majority of the routines provided by the programming interface to construct R objects only return specific types of objects; these routines are shown in Table 3.3. When these are used to define an R object, we can infer its type.

For routines in Table 3.3 that accept length or dimension arguments, such as `Rf_allocVector`, we can also infer the dimensions of the new R object, provided that these arguments are constants. Where they are not constants, it is still sometimes possible to relate the dimensions of the new object to other objects. For instance, if `b` is defined by a call to `Rf_allocVector` where the length argument is `LENGTH(a)`, then `a` and `b` are the same length. Furthermore, if `a` is parameter,

then we can conclude that `b` is the same length as the argument to `a`. It may then be possible to infer the length of `b` at a given call site based on the calling R code.

Routine	Argument(s)	Return	Return Dimensions
`Rf_ScalarLogical`	`int`	`LGL`	1
`Rf_ScalarInteger`	`int`	`INT`	1
`Rf_ScalarString`	`CHAR`	`STR`	1
`Rf_ScalarReal`	`double`	`REAL`	1
`Rf_ScalarComplex`	`Rcomplex`	`CPLX`	1
`Rf_ScalarRaw`	`Rbyte`	`RAW`	1
`Rf_allocList`	`int`	`LIST`	1
`Rf_install`	`char *`	`SYM`	1
`Rf_installTrChar`	`CHAR`	`SYM`	1
`Rf_mkString`	`char *`	`STR`	1
`Rf_allocSExp`	T	τ	1
`Rf_allocVector`	T, `R_len_t` n	τ	n
`Rf_allocMatrix`	T, `int` m, `int` n	τ	$m \times n$
`Rf_alloc3DArray`	T, `int` m, `int` n, `int` p	τ	$m \times n \times p$
`Rf_allocArray`	T, `INT` k	τ	$k_1 \times k_2 \times \ldots$
`Rf_mkNamed`	T, `char **`	τ	

Table 3.3: Routines provided by the R Internals programming interface to construct `SEXP`s of specific types. Some routines that are not called by any CRAN packages are omitted. T denotes any `SEXPTYPE`, while τ denotes a `SEXP` with `SEXPTYPE` T. All `SEXPTYPE`s are written without the suffix `SXP` here.

The strategy also examines uses of the result and any parameters or variables on which it depends. The uses serve as a secondary source of information should definitions prove insufficient to infer the type. It is especially informative when a parameter or variable is used as an argument in a call to a routine provided by the R Internals programming interface. Several frequently-used routines require that their arguments have specific `SEXPTYPE`s. These are shown in Table 3.4.

Now we turn our attention to situations where the core strategy cannot correctly infer the return type, and extensions to the strategy to handle these.

Containers, Conditionals, and Loops

Containers such as lists (`VECSXP`s) can have elements with heterogeneous types which must be inferred in addition to the overall type. This section describes how to modify the type inference strategy to infer the element types. The basic idea is to apply the type inference procedure recursively for each element, but variable indexing and elements which depend on a condition introduce subtleties. We begin with an example of the simplest case, to emphasize the basic idea.

Routine	Argument(s)
LOGICAL	LGL
LOGICAL0	LGL
LOGICAL_RO	LGL
LOGICAL_ELT	LGL
SET_LOGICAL_ELT	LGL
INTEGER	INT
INTEGER0	INT
INTEGER_RO	INT
INTEGER_ELT	INT
SET_INTEGER_ELT	INT
REAL	REAL
REAL0	REAL
REAL_RO	REAL
REAL_ELT	REAL
SET_REAL_ELT	REAL
COMPLEX	CPLX
COMPLEX0	CPLX
COMPLEX_RO	CPLX
COMPLEX_ELT	CPLX
RAW	RAW
RAW0	RAW
RAW_RO	RAW
RAW_ELT	RAW
SET_STRING_ELT	STR, R_xlen_t, CHAR
STRING_ELT	STR, R_xlen_t
STRING_PTR	STR

Table 3.4: Routines from the R Internals programming interface that require arguments with specific SEXPTYPEs. All SEXPTYPEs are written without the suffix SXP here.

Example 44. The **nloptr** package (Johnson and Ypma 2020) provides an interface to the nonlinear optimization library NLopt. In the package, the optimization function `nloptr` calls the routine `NLoptR_Optimize`. The routine returns a named list. This example uses the core strategy to infer the type of the result, and presents a modification of the strategy to infer its element types.

The C source code from `NLoptR_Optimize` that deals directly with the return value is:

```
1  int num_return_elements = 8;
2  SEXP R_result_list;
3  PROTECT(R_result_list = allocVector(VECSXP, num_return_elements));
4  // ...
5  SEXP names;
6  PROTECT(names = allocVector(STRSXP, num_return_elements));
7
8  SET_STRING_ELT(names, 0, mkChar("status"));
9  SET_STRING_ELT(names, 1, mkChar("message"));
10 // ... for all 8 elements
11 SET_STRING_ELT(names, 7, mkChar("version_bugfix"));
12 setAttrib(R_result_list, R_NamesSymbol, names);
13 // ...
14 SET_VECTOR_ELT(R_result_list, 0, R_status);
15 SET_VECTOR_ELT(R_result_list, 1, R_status_message);
16 // ... for all 8 elements
17 SET_VECTOR_ELT(R_result_list, 7, R_version_bugfix);
18
19 return(R_result_list);
```

Listing 3.8: The C code for `NLoptR_Optimize` related to the return value.

Based on the C code, the length and element names of the list are fixed. Of course, for programmatic analysis we will use the LLVM IR rather than the C code.

As in the previous example (Example 43), the first step for type inference is to inspect the definition of the variable returned by the routine's `ret` instruction. Here are the relevant instructions:

Routine	Argument(s)	Return
`Rf_copyMostAttrib`	α	α
`Rf_duplicate`	α	α
`Rf_protect`	α	α
`Rf_shallow_duplicate`	α	α
`LENGTH`	α	`int`
`LENGTH_EX`	α, `int`, `char *`	`int`
`Rf_length`	α	`R_len_t`
`Rf_xlength`	α	`R_len_t`
`SETLENGTH`	α	`int`
`XLENGTH`	α	`int`
`XLENGTH_EX`	α	`R_len_t`
`XTRUELENGTH`	α	`R_len_t`

Table 3.5: Routines provided by the R Internals programming interface that accept or return `SEXP`s of arbitrary types, but do not provide new information about types (some do provide information about dimensions). Routines that are not called by any CRAN packages are omitted. α denotes a `SEXP` with any `SEXPTYPE`.

```
1 %call127 = call %struct.SEXPREC* @Rf_allocVector(i32 19, i64 8) #4
2 ; ...
3 ret %struct.SEXPREC* %call127
```

The call to `Rf_allocVector` establishes that `%call127` is a list, because 19 corresponds to the list type `VECSXP` in the `SEXPTYPE` enumeration (Table 3.2). We can also see that this list has exactly eight elements, but the definition doesn't indicate the types or names of the elements. This example focuses on finding the element types and defers finding the element names to Example 50.

To find the element types, we need to analyze the instructions that use the returned variable `%call127`—especially calls that modify the variable or its elements. The first of these is a call to `Rf_protect`, which doesn't yield any useful type information because it doesn't modify its arguments and doesn't require arguments with specific `SEXPTYPE`s. Table 3.5 lists several routines provided by the R Internals programming interface which, like this one, do not yield new type information.

Next, `%call127` is used in a call to `Rf_setAttrib`, which sets an attribute on an R object. In this case, the attribute being set is the element names, so we'll skip over this call as well until Example 50.

Continuing on, the next instruction that uses `%call127` is a call to `SET_VECTOR_ELT`, the primary routine for setting list elements. The subsequent seven instructions that use `%call127`

are also calls to this routine. Here's the IR for a few of them:

```
%call183 = call %struct.SEXPREC* @SET_VECTOR_ELT(
    %struct.SEXPREC* %call127, i64 0, %struct.SEXPREC* %call141) #4
%call184 = call %struct.SEXPREC* @SET_VECTOR_ELT(
    %struct.SEXPREC* %call127, i64 1, %struct.SEXPREC* %call145) #4
; ... for all 8 elements
%call190 = call %struct.SEXPREC* @SET_VECTOR_ELT(
    %struct.SEXPREC* %call127, i64 7, %struct.SEXPREC* %call179) #4
```

Listing 3.9: Calls to `SET_VECTOR_ELT` to set elements of the list.

The first argument to `SET_VECTOR_ELT` is the list to modify, the second argument is the element position, and the third argument is the new value for the element. These eight instructions set all eight elements (positions 0–7) of `%call127`.

Calls to `SET_VECTOR_ELT` are where we modify the core type inference strategy. Since these calls set elements of the list `%call127`, the strategy should use the information in each call to infer the type of the respective element.

For instance, consider the first call:

```
%call183 = call %struct.SEXPREC* @SET_VECTOR_ELT(
    %struct.SEXPREC* %call127, i64 0, %struct.SEXPREC* %call141) #4
```

This call sets the first element (position 0) to the value of `%call141`. The core strategy would normally ignore `%call141`, but instead, it should infer its type. This can be done by applying type inference recursively, as with any other variable.

The variable `%call141` is defined by a call to the `Rf_allocVector` routine:

```
%call141 = call %struct.SEXPREC* @Rf_allocVector(i32 13, i64 1) #4
```

This definition is sufficient to conclude that `%call141` is an integer vector with one element (by similar reasoning as Example 43), and therefore so is the first element of the list `%call127`.

By following the same process for the other list elements, we can fully determine the element types for the list. Without the modification to the type inference strategy, the strategy would still find that the return type is list, but would not find the element types. □

Routine	Index
`SET_VECTOR_ELT`	Position
`Rf_defineVar`	Name
`R_do_slot_assign`	Name
`SETCAR`	1
`SETCADR`	2
`SETCADDR`	3
`SETCADDDR`	4
`SETCAD4R`	5
`SETCDR`	Tail elements

Table 3.6: Routines provided by the R Internals programming interface that modify the elements of a heterogeneous container.

The modified strategy correctly handles ordinary lists, as well as data frames and other classes that extend lists. Furthermore, we can generalize the modified strategy to handle other element-setting routines and heterogeneous container types. Table 3.6 lists the element-setting routines provided by the R Internals interface. These are discussed in the subsequent paragraphs, and Listing 3.10 shows the modified strategy.

Identify the return value—the only operand of the `ret` instruction—and apply this type inference procedure:

1. **Inspect the definition**. If the operand is not a parameter, inspect its definition for type information. For `SEXP`s, the definition will usually be a call to an R Internals routine. Look up the called routine in the database of routines. If the called routine's return type depends on the types of its arguments, repeat the type inference procedure for the arguments.

2. **Inspect the uses**. For each other instruction that uses the operand: if the instruction is a call to an R Internals routine, look up the called routine in the database of routines. The routine may put restrictions on the types of its arguments and thus the type of the operand. If the routine is an element-setting routine (Table 3.6), collect the index and type of the element by repeating the type inference procedure on the value to which the element is set.

Listing 3.10: The new strategy to infer a routine's return type. Modifications of Listing 3.7 to take into account list elements are shown in black.

The `Rf_defineVar` routine sets an element of an environment (`ENVSXP`). Likewise, the

`R_do_slot_assign` routine sets a slot of an S4 object (any `SEXP` with the S4 flag set). The only important difference between these routines and `SET_VECTOR_ELT` is that they select the element by name rather than position.

The `SETCAR` routine sets the first element of any `SEXP` structured as a linked list. For such objects, each element has two fields: `CAR` and `CDR`. The `CAR` field holds the value of the element, while the `CDR` field holds a reference to the next element. The pairlist (`LISTSXP`) is the canonical example of a `SEXP` strucured as a linked list (R Core Team 2019a). The related `SETCADR` routine sets the second element of a linked list object. Table 3.6 shows additional routines to set specific elements. The `SETCDR` routine differs from the others because it sets the `CDR` field, and thereby replaces the entire tail of the linked list. Despite these differences compared to `SET_VECTOR_ELT`, the modified strategy remains appropriate.

In Example 44, the calls to `SET_VECTOR_ELT` all had literal index arguments. Literal indices are a common pattern for setting the elements of a fixed-size container. Another common pattern is to use a counter variable, incrementing it each time a container element is set. If the counter's initial value is static and it is incremented unconditionally—the same way along all paths through the code—then it is effectively a sequence of constants. With any level of optimization enabled, LLVM carries out constant-folding, which replaces constants with their literal value. As a result, the modified type inference strategy also correctly infers element types when the index arguments are constant or an unconditional counter. The next example demonstrates the counter variable case.

Example 45. The **clv** package (Nieweglowski 2020) provides functions to assess the validity and stability of results from clustering algorithms. The `clusterScatterMeasures` routine is provided in the package's C code and returns a list of results. This list is constructed with calls to `SET_VECTOR_ELT`, but in this case the position of each element is specified by a counter variable. Here's the relevant C code from the routine:

```
1 int clv_ind = 11;
2 PROTECT( return_list = allocVector(VECSXP, clv_ind) );
3 pos = 0;
4 SET_VECTOR_ELT( return_list, pos++, max_intracluster_sxp );
5 SET_VECTOR_ELT( return_list, pos++, average_intracluster_sxp );
6 // ... for all 11 elements
```

```
7 SET_VECTOR_ELT( return_list,  pos++, cluster_size_sxp );
```

With optimization enabled, LLVM optimizes out the counter variable, so the calls to `SET_VECTOR_ELT` contain constant indices. Here's the IR at optimization level 2:

```
1 %call433 = tail call %struct.SEXPREC* @Rf_allocVector(i32 19, i64 11) #3
2 %call476 = tail call %struct.SEXPREC* @SET_VECTOR_ELT(
3     %struct.SEXPREC* %call433, i64 0, %struct.SEXPREC* %call18) #3
4 %call479 = tail call %struct.SEXPREC* @SET_VECTOR_ELT(
5     %struct.SEXPREC* %call433, i64 1, %struct.SEXPREC* %call22) #3
6 ; ... for all 11 elements
7 %call506 = tail call %struct.SEXPREC* @SET_VECTOR_ELT(
8     %struct.SEXPREC* %call433, i64 10, %struct.SEXPREC* %call68) #3
```

As a result of the optimization, the strategy from Example 44 can correctly infer the element types of the returned list. The constant `clv_ind` in the C code is also optimized out in the IR, so the strategy can also infer the length of the returned list. □

The counter variable pattern in the example (Example 45) also applies to other routines provided by the R Internals programming interface. One of these is the `SET_STRING_ELT` routine, which sets an element of a character vector. Since the elements of a vector must all have the same type, the `SET_STRING_ELT` routine doesn't provide new information about element types. Nonetheless, as we'll see later in Example 50, calls to this routine can provide information about element names. Regardless of the routine, LLVM can optimize out any unconditional counter variable. Moreover, if a counter variable is unconditional only up to some specific point in the code, then LLVM can optimize it out up to that point.

Sometimes containers vary in length and element type depending on run-time conditions. These containers may still be limited to a static set of potential lengths and element types. This situation does not appear to be common: in a programmatic analysis of all CRAN packages, Temple Lang (2021) found only 14 routines that return lists for which the length varies within a static set. Nonetheless, for containers that vary this way, the type inference strategy should identify the set of potential lengths and element types.

Example 46. This example describes how to adapt the type inference strategy to correctly handle containers for which length and element type vary within a static set. The idea generalizes

to any R object for which the type varies within a static set, such as the returned result of the example routine.

The package **cluster** (Maechler et al. 2021) provides functions for unsupervised learning methods. The package's `pam` function, an implementation of the k-mediods clustering algorithm, calls the routine `cl_Pam`. In normal usage, the `cl_Pam` routine returns a list which varies in length and element type depending on the arguments. If the clustering algorithm fails, the routine returns an integer error code instead of a list.

We begin by describing potential configurations for the returned list. The list can have 8 or 9 elements depending on whether the variable `keep_diss` is false or true, respectively. The variable `keep_diss` is the result of coercing the parameter `keep_diss_` to a logical value. The `cl_Pam` routine coerces many parameters this way, and in this example we will refer to the resulting variables as *coerced parameters*. As one would expect, the ninth element of the list is only set if `keep_diss` is true. The relevant C code from the routine for these steps is:

```
1 const Rboolean keep_diss = asLogical(keep_diss_);
2 ans = PROTECT(allocVector(VECSXP, keep_diss ? 9 : 8));
3 if(keep_diss) SET_VECTOR_ELT(ans, 8, dys_);
```

Listing 3.11: Code from `cl_Pam` to define the returned list and set its ninth element.

The dimensions of the fifth and sixth element of the list depend on the coerced parameter `all_stats`. For the sixth element, the class also depends on this parameter. Here's the relevant C code:

```
1 const Rboolean all_stats = asLogical(all_stats_);
2 SET_VECTOR_ELT(ans, 4, allocVector(INTSXP, all_stats ? kk : 1));
3 SET_VECTOR_ELT(ans, 5, all_stats ? allocMatrix(REALSXP, kk, 5)
4                                  : allocVector(REALSXP, 1));
```

Listing 3.12: Code from `cl_Pam` to set the fifth and sixth elements of the returned list.

The third and eighth element of the list are only set if the variable `do_syl` is true. List elements are initialized to the R `NULL` value, which has type `NILSXP`. So when the third and eighth element are not set, their type is `NILSXP`. The `do_syl` variable is computed from several coerced parameters, including `all_stats`. Here's the relevant C code:

```
const Rboolean do_syl = all_stats && (1 < kk && kk < n);
if(do_syl) {
    SET_VECTOR_ELT(ans, 2, allocMatrix(REALSXP, n, 4));
    SET_VECTOR_ELT(ans, 7, allocVector(REALSXP, 1));
}
```

Listing 3.13: Code from `cl_Pam` to set the third and eighth elements of the returned list. The code shown here is simplified from the original to remove details unrelated to type inference.

In summary, there are six possible configurations for the list depending on `keep_diss`, `all_stats`, and `do_syl`. Four in the case where `all_stats` is true, and two in the case where `all_stats` is false (since then `do_syl` must also be false). Type inference should collect each of the six possible configurations and the conditions on which they depend. Collecting the conditions is useful in case the arguments to the routine at a specific call site are constant, so the specific return type can be deduced. We can also use the potential lengths and element types as a bound in other analyses (for instance, to check for out-of-bounds errors).

Now we turn to the LLVM IR. As in previous examples, type inference begins at the return instruction:

```
cleanup241:                       ; preds = %cleanup, %if.end240
  %retval.1 = phi %struct.SEXPREC* [ %call56, %if.end240 ],
      [ %call146, %cleanup ]
  ; ...
  ret %struct.SEXPREC* %retval.1
```

Listing 3.14: The `cl_Pam` routine's return instruction and the `phi` instruction which defines the return value, both of which are in the `cleanup241` block.

The return value is in the variable `%retval.1`, which is defined by a `phi` instruction. A `phi` instruction indicates that a variable can take two different values depending on which path is taken through the code at run-time. Each value corresponds to a different immediate predecessor of the block that contains the `phi` instruction. These instructions are necessary because the LLVM IR is in static single assignment form, and they are analogous to the φ-functions described in Chapter 1.

In Listing 3.14, if the `cleanup241` block is reached via the `if.end240` block, the `phi` instruction defines `%retval.1` as `%call56`. If the block is reached via the `cleanup` block, the `phi` instruction

defines `%retval.1` as `%call146`. The variables `%call56` and `%call146` correspond to the list and the integer error code, respectively, that can be returned by the routine.

We can apply the type inference strategy recursively to each of the possible values `%call56` and `%call146` for `%retval.1`. The inferred type for `%retval.1`—and therefore the routine's result—is the union (in the sense of Section 2.5.1) of the types for the two values. Taking the union ensures that the strategy collects all possible return types. Moreover, this approach applies to any `phi` function encountered in the course of type inference.

The variable `%call146` corresponds to the integer case, which is simpler, so we consider it first. The type existing inference strategy in Listing 3.10 infers that this variable is a scalar integer, since it is defined by a call to `Rf_ScalarInteger` (see Table 3.3).

Now consider the variable `%call56`, which corresponds to the list case. The existing type inference strategy in Listing 3.10 infers that this variable is a list, since the variable is ultimately defined by a call to `Rf_allocVector` with `SEXPTYPE` argument 19, which corresponds to a list (see Table 3.2). Here are the instructions that define the variable:

```
1 %call55 = tail call %struct.SEXPREC* @Rf_allocVector(
2     i32 19, i64 %cond53) #10
3 %call56 = tail call %struct.SEXPREC* @Rf_protect(
4     %struct.SEXPREC* %call55) #10
```

The call to `Rf_protect` does not affect the type (see Table 3.5).

Unlike previous examples, the length argument in the call to `Rf_allocVector` is a variable rather than a literal value. When a dimension is set by a variable, the type inference strategy should attempt to infer potential values for that variable or trace its value back to a parameter. The key is to inspect the variable's definition.

The length argument `%cond53` is defined by a `select` instruction:

```
1 %cond53 = select i1 %tobool52.not, i64 8, i64 9
```

Listing 3.15

A `select` instruction selects a value based on its first operand, without branching. If the first operand is 1 (true), the second operand is selected; otherwise the third operand is selected. Thus the length of the list is 8 if `%tobool52.not` is 1, and is 9 otherwise.

The next step in the type inference strategy is to analyze the uses of the list `%call56` in order to determine the types of its elements. Example 44 already showed that the strategy in

Listing 3.10 correctly infers the types of elements for which the type and dimensions do not vary. Consequently, we will focus on the third, fifth, sixth, and eighth element, where the type and dimensions do vary. We will also consider the ninth element, which is only set when the list has 9 elements.

For the fifth element of the list, the type is static but the length depends on the coerced parameters `all_stats` and `kk` (see Listing 3.12 for the C code). The type inference strategy in Listing 3.10 finds that the element is an integer vector. The length of the element is defined by a `select` instruction, which we saw how to handle earlier in this example:

```
1 %cond98 = select i1 %tobool.not, i32 1, i32 %call
```

The variable `%call` is the coerced parameter `kk` in the C code. Thus the length of the fifth element is either 1 or the value of a parameter.

For the sixth element of the list, the class depends on the coerced parameters `all_stats` and `kk` (see Listing 3.12). Depending on `all_stats`, the element is constructed by a call to either `Rf_allocMatrix` or `Rf_allocVector`. As a result, in the IR the value of the element is defined by a `phi` instruction:

```
1 %cond109 = phi %struct.SEXPREC* [ %call105, %cond.true104 ],
2                                 [ %call107, %cond.false106 ]
3 %call110 = tail call %struct.SEXPREC* @SET_VECTOR_ELT(
4     %struct.SEXPREC* %call56, i64 5, %struct.SEXPREC* %cond109) #10
```

As with the `phi` instruction earlier in this example, we modify the type inference strategy to recursively infer the type for each of the instruction's operands. From there, the strategy in Listing 3.10 correctly infers that the sixth element is either a `kk` by 5 numeric matrix or a numeric scalar. The inferred type is the union of these.

For the third and eighth element of the list, the type depends on the variable `do_syl` (see Listing 3.13). If `do_syl` is false (not 1), then the elements retain their initial `NULL` values. If `do_syl` is true, then elements are set by a call to `SET_VECTOR_ELT`. Here's the block where the eighth element is set:

```
1 if.then117:                          ; preds = %cond.end108
2   %call118 = tail call %struct.SEXPREC* @Rf_allocVector(
3       i32 14, i64 1) #10
```

```
4    %call119 = tail call %struct.SEXPREC* @SET_VECTOR_ELT(
5        %struct.SEXPREC* %call56, i64 7, %struct.SEXPREC* %call118) #10
6    br label %if.end120
```

The type inference strategy in Listing 3.10 infers that the eighth element can be a numeric scalar, but not that it can be `NULL`. We need to modify the strategy to detect that the call to `SET_VECTOR_ELT` is not always executed after the list is defined.

If an instruction is not always executed, then there must be a path through the control flow graph that does not visit the block that contains the instruction. Figure 3.1 shows a subgraph of the control flow graph for the `cl_Pam` routine. The returned list is defined in the `if.end50` block, and the call to `SET_VECTOR_ELT` for the eighth element is in the `if.then117` block. There is clearly a path from `if.end50` to the exit which does not visit `if.then117`, so the call to `SET_VECTOR_ELT` is not always executed after the list is defined.

Formally, if block A in a control flow graph must be visited in order to reach the exit from block B, we say that A *post-dominates* B (Cooper and Torczon 2012). For example, in Figure 3.1, the block `if.end120` post-dominates the block `if.end50`, but the block `if.then117` does not.

The type inference strategy can determine whether calls that modify an R object are always executed by checking whether the block that contains the call post-dominates the block where the object was defined. LLVM provides routines to inspect dominance relations between blocks in the IR. In **Rllvm**, the `postDominates` function checks whether one block post-dominates another.

The type inference strategy should check post-dominance for every call that modifies an R object (such as `SET_VECTOR_ELT`). For the list returned by `cl_Pam`, the call to `SET_VECTOR_ELT` for the eighth element does not post-dominate the definition. Thus the eighth element is a numeric scalar or `NULL`. The inferred type is the union of these two types.

The third element of the list follows the same pattern as the eighth. The element is set to an `n` by 4 numeric matrix in a call to `SET_VECTOR_ELT`. The call does not post-dominate the definition of the list, so again, the element can also be `NULL`. As with the eighth element, the inferred type is the union of the two possibilities.

Finally, consider the ninth element of the list. This element is only set when the coerced parameter `keep_diss` is true, which coincides with when the list has 9 elements (see Listing 3.11). The call to `SET_VECTOR_ELT` for the ninth element does not post-dominate the definition of the list. However, because the condition for the call is the same as the condition for the list to have

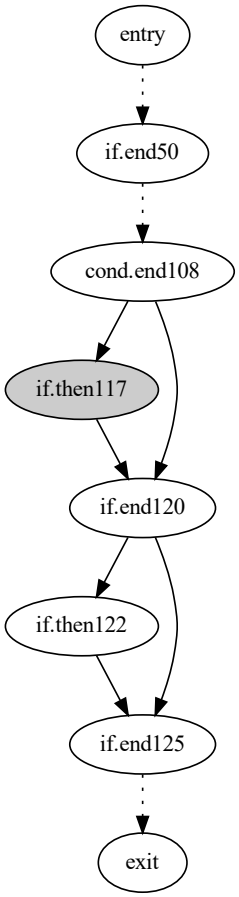

Figure 3.1: The control flow graph for the `cl_Pam` routine around the block `if.then117` (in gray). The list returned by the routine is defined in the block `if.end50`. Dotted edges indicate some intermediate blocks are not shown.

9 elements, we can tell that the call is always made when the list has 9 elements. In order for the type inference strategy to detect this, it must not only check post-dominance, but also collect the associated condition for calls that do not post-dominate the definition of the object they modify. We defer collecting conditions to Example 47. □

The example modifies the type inference strategy to handle R objects—especially lists and other containers—for which the type, class, or length vary within a static set. The example demonstrates that variation can occur at the definition of or at calls that modify an object. Here's a summary of these two cases:

1. Definitions. An object defined by a `phi` instruction, or defined by a call instruction where some of the arguments are defined by `select` or `phi` instructions.

2. Modifying calls. An object modified by a call that does not post-dominate the object's definition, meaning the call is not always executed.

These two cases correspond to changes in the definitions and uses cases, respectively, of the type inference strategy from Listing 3.10. The updated strategy is shown in Listing 3.16.

Identify the return value—the only operand of the `ret` instruction—and apply this type inference procedure:

1. **Inspect the definition**. If the operand is not a parameter, inspect its definition for type information. If the definition is:
   - A call to an R Internals routine. Look up the called routine in the database of routines. If the called routine's return type depends on the types of its arguments, repeat the type inference procedure for the arguments. If an argument that sets a dimension is defined by a `select` instruction, collect both operands as potential values for the dimension.
   - A `phi` instruction. Repeat the type inference procedure for each of the operands. The type is the union of the two types from the operands.
2. **Inspect the uses**. For each other instruction that uses the operand: if the instruction is a call to an R Internals routine, look up the called routine in the database of routines. The routine may put restrictions on the types of its arguments and thus the type of the operand. If the routine is an element-setting routine (Table 3.6):
   a) Collect the index and type of the element by repeating the type inference procedure on the value to which the element is set.
   b) Check whether the call post-dominates the definition of the object. If it does not, the type is the union of the initial type and the collected type.

Listing 3.16: The updated strategy to infer a routine's return type. Modifications of Listing 3.10 to take into account objects for which the type, class, or length vary are shown in black.

The updated type inference strategy is missing one important piece: how to collect the conditions on R objects that vary. The next example is a continuation of Example 46 to show how to do this.

Example 47. The list returned by the `cl_Pam` routine in Example 46 can have 8 or 9 elements. When the list has 9 elements, the ninth element is always set by a call to `SET_VECTOR_ELT`. However, the call does not post-dominate the definition of the list. In order to detect that the element is always set, the type inference strategy must collect the conditions for both the length of the list and the call to `SET_VECTOR_ELT`. Then the strategy can compare them to see that

they are the same. This example describes this process.

First consider the length of the list. In the IR, the length is set by the variable `%cond53`, which is defined by a `select` instruction:

```
%call7 = tail call i32 @Rf_asLogical(%struct.SEXPREC* %keep_diss_) #10
%tobool52.not = icmp eq i32 %call7, 0
%cond53 = select i1 %tobool52.not, i64 8, i64 9
```

The result of the `select` instruction depends on the variable `%tobool52.not`. This variable is defined by an `icmp` instruction, which compares two integers. The instruction includes a keyword that indicates the type of the comparison. In this case, the instruction checks whether `%call7` is equal (keyword `eq`) to 0. Since `%call7` is an integer that represents a logical value, this instruction amounts to a logical inversion.

In turn, the variable `%call7` is defined by a call to the R Internals routine `Rf_asLogical`. This routine coerces an R object into an integer representation where 1 means true and 0 means false. The argument to the routine is `%keep_diss_`, one of the parameters of the `cl_Pam` routine.

The type inference strategy should combine the information from the call to `Rf_asLogical` and the `icmp` instruction into a single condition for the `select` instruction. The length of the list is 8 when the parameter `%keep_diss_` is false, and 9 otherwise. LLVM provides several different instructions for elementary logic operations and comparisons, all of which must be handled. These instructions are listed in the LLVM documentation (https://llvm.org/docs/).

Next, consider the call to `SET_VECTOR_ELT` to set the ninth list element. The call is in the block `if.then122`, which does not post-dominate the block where the list is defined, `if.end50`. Figure 3.1 shows these blocks in the control flow graph for the routine. To find the condition for the call, the type inference strategy must search backwards through the control flow graph from `if.then122` to the nearest block which does post-dominate `if.end50`. The nearest post-dominator block is `if.end120`, which contains this branch instruction:

```
br i1 %tobool52.not, label %if.end125, label %if.then122
```

The branch instruction depends on `%tobool52.not`, just like the length of the list. By comparing the two conditions, the type inference strategy can infer that the the call to set the ninth element is always executed when the list has 9 elements.

In practice, the nearest post-dominator block will not necessarily be the immediate predecessor of the call block. Figure 3.2 shows one example. For the `call` block, `condition1` is the nearest

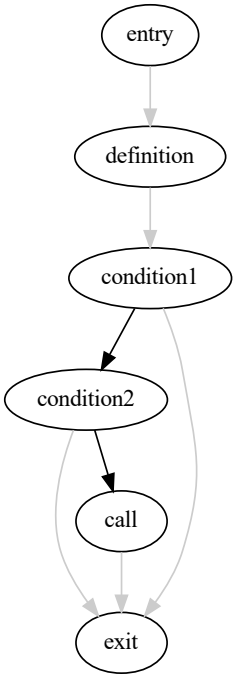

Figure 3.2: A control flow graph where execution of the block which contains the call depends on two conditions. Edges on the path between the `call` block and the nearest post-dominator of the `definition` block are shown in black. This CFG is based on structures in the actual CFG for the `cl_Pam` routine.

block which post-dominates the `definition` block. The type inference strategy should collect the branch condition from `condition1`. However, the intermediate block `condition2` is also important. Since the `condition2` block is not post-dominated by the `call` block, it must contain another condition necessary for the `call` block to execute. Listing 3.17 shows a general strategy to identify the conditions for a call based on these observations. □

1. Identify the nearest predecessor of the call block which also post-dominates the definition block. Collect the branch condition.

2. For each other block B along the path from the nearest predecessor to the `call` block: if B contains a conditional branch and B is not post-dominated by the call block, collect the branch condition.

Listing 3.17: A strategy to collect the conditions for a call which modifies an R object.

Another pattern for setting elements of a list or other container is to set them in a loop. The next example describes a modification of the type inference strategy to collect information about elements set in loops.

Example 48. Consider the package **stringdist**, which provides a variety of functions for approximate string matching. One of these, the `phonetic` function, calls the `R_soundex` routine in the package's C code. The routine returns either a character vector or a list, depending on its arguments. The character vector case is correctly inferred by the strategy in Listing 3.16. In contrast, the strategy requires modifications to infer the correct element types for the list, because the list elements are set in a loop. Here's the code from the routine to construct the list:

```
1 SEXP y = allocVector(VECSXP, n);
2 unsigned int nfail = 0;
3 int len_s, isna_s;
4 for (int i = 0; i < n; ++i) {
5   get_elem(x, i, bytes, 0L, &len_s, &isna_s, s);
6   if (isna_s) {
7     SEXP sndx = allocVector(INTSXP, 1);
8     INTEGER(sndx)[0] = NA_INTEGER;
9     SET_VECTOR_ELT(y, i, sndx);
```

```
10    } else {
11      SEXP sndx = allocVector(INTSXP, 4);
12      nfail += soundex(s, len_s, (unsigned int *)INTEGER(sndx));
13      SET_VECTOR_ELT(y, i, sndx);
14    }
15  }
16  return y;
```

Listing 3.18: An excerpt of the code from the `R_soundex` routine which constructs the returned list. Most statements unrelated to the element types have been removed.

The length of the list, `n`, is defined elsewhere in the routine as the length of the parameter `x`. All of the list elements are integer vectors (`INTSXP`), but the length varies. The length is 1 for some and 4 for others. The specific length for each element depends on run-time conditions and thus cannot be determined statically.

The loop calls sets every element of the list. This is apparent from examining the loop's induction variable. An *induction variable* is a variable that increases or decreases by a constant amount, called the *step*, in each iteration of a loop (Cooper and Torczon 2012). For this loop, the induction variable is `i`, its initial value is 0, the step is 1, and its final value is `n - 1`. Thus `i` will take every value from `0` to `n - 1` (inclusive) as the loop iterates. This variable `i` is also the index argument in the call to `SET_VECTOR_ELT`.

We need to modify the type inference strategy to detect that `SET_VECTOR_ELT` is called in a loop, and then check that it is called for every element of the list.

LLVM simplifies the two calls to `SET_VECTOR_ELT` in Listing 3.18 into one in the IR. Here's the instruction for the call, which is in the block `for.inc70`:

```
1  %call59 = call %struct.SEXPREC* @SET_VECTOR_ELT(
2      %struct.SEXPREC* %call39,
3      i64 %indvars.iv,
4      %struct.SEXPREC* %call54.sink) #9
```

Listing 3.19: The call to `SET_VECTOR_ELT` which sets the elements of the list returned by `R_soundex`.

In a control flow graph, loops are represented by cycles. Figure 3.3 shows that the block `for.inc70` is part of a cycle in the control flow graph for the `R_soundex` routine, and therefore part of a loop.

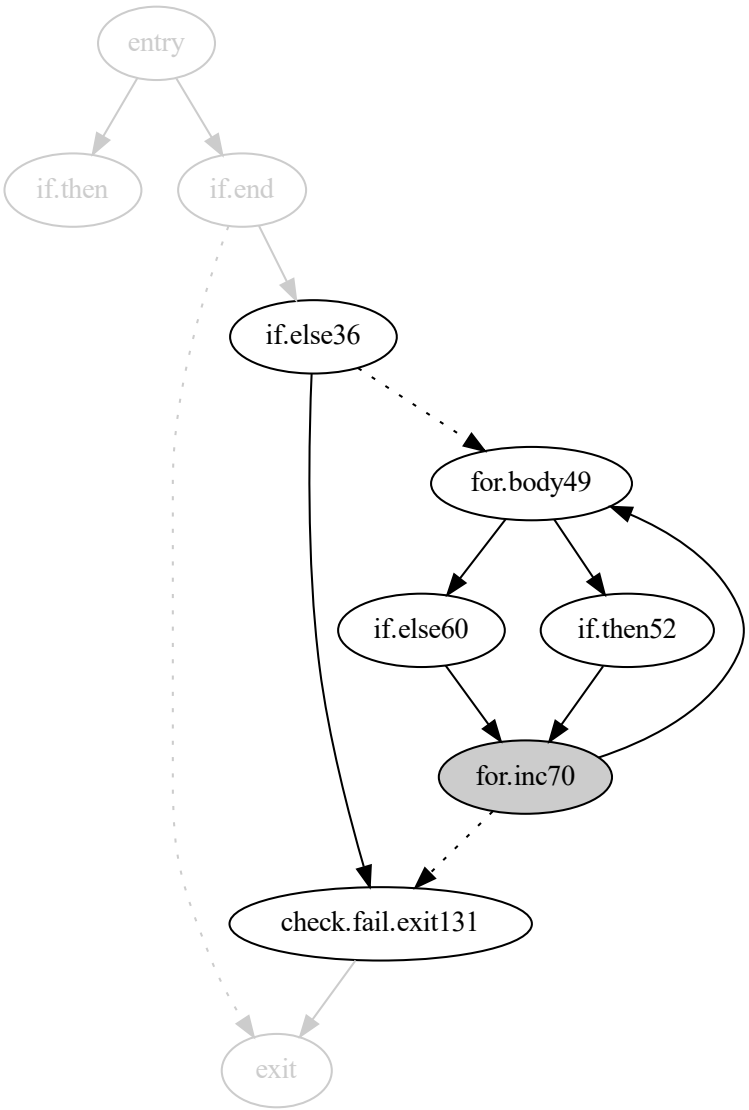

Figure 3.3: The control flow graph for the `R_soundex` routine around the loop which sets the elements of the returned list. The call to `SET_VECTOR_ELT` is in the block `for.inc70` (in gray). Dotted edges indicate some intermediate blocks are not shown. The `if.then` block (at top) has no descendants because it raises an error.

LLVM provides routines to detect and analyze loops. In **Rllvm**, this is a three step process. First, call `runLoopPass` on the module that contains the routine in order to transform loops in the IR into a canonical form. Second, call `loopAnalysis` on the routine in order to detect the loops in the routine. Finally, call `getLoops` on the routine to get a list of information about the loops. For each loop, the information includes the component blocks, induction variable, step, and bounds.

With the information from LLVM, the type inference strategy can check whether a block part of a loop by comparing it against the component blocks for each loop. If it is, then this process also identifies which loop, so the strategy can get the induction variable, step, and bounds.

The block `for.inc70` is part of a loop with induction variable `%indvars.iv`. The lower bound is 0 and the step is 1. The upper bound is the variable `%wide.trip.count`, which is the value of the variable `%call2`. The variable `%call2` is in turn defined by a call to the R Internals routine `Rf_length`:

```
1  %call2 = tail call i32 @Rf_length(%struct.SEXPREC* %x) #9
```

This call returns the length of the routine's parameter `x`. Thus the induction variable ranges from 0 to the length of `x` minus 1.

Once the strategy determines that a call to `SET_VECTOR_ELT` is in a block which is part of a loop, the next step is to determine the relationship between the index argument and the loop induction variable. For the call to `SET_VECTOR_ELT` in Listing 3.19, the index argument is `%indvars.iv`, which is the induction variable.

With the relationship between the index argument and loop induction variable established, the next step is to determine which elements of the list are set by the loop. To do this, the strategy must compare the range of values for the index argument to the length of the list. Since `%indvars.iv` is the index argument, the indices range from 0 to the length of `x` minus 1. The length of the list is in the list's definition:

```
1  %conv38 = sext i32 %call2 to i64
2  %call39 = tail call %struct.SEXPREC* @Rf_allocVector(
3       i32 19, i64 %conv38) #9
```

The `sext` instruction preserves the value of its operand, so the length of the list is equal to `%call2`, which is the length of the parameter `x`. It follows that every element of the list is set by the call to `SET_VECTOR_ELT`.

Finally, to infer the types of the list elements, we apply the type inference strategy recursively to the value argument in the call to `SET_VECTOR_ELT`. The strategy in Listing 3.16 infers that the value argument is an integer vector, and that the length can be either 1 or 4. □

Listing 3.20 shows the type inference strategy with all of the modifications for handling containers described in this section.

Identify the return value—the only operand of the `ret` instruction—and apply this type inference procedure:

1. **Inspect the definition**. If the operand is not a parameter, inspect its definition for type information. If the definition is:
   - A call to an R Internals routine. Look up the called routine in the database of routines. If the called routine's return type depends on the types of its arguments, repeat the type inference procedure for the arguments. If an argument that sets a dimension is defined by a `select` instruction, collect both operands as potential values for the dimension, and collect the condition from the instruction.
   - A `phi` instruction. Repeat the type inference procedure for each of the operands. The type is the union of the two types from the operands. Collect the condition from the nearest common predecessor block of the blocks mentioned in the `phi` instruction.
2. **Inspect the uses**. For each other instruction that uses the operand: if the instruction is a call to an R Internals routine, look up the called routine in the database of routines. The routine may put restrictions on the types of its arguments and thus the type of the operand. If the routine is an element-setting routine (Table 3.6):
   a) Check whether the call to the routine is in a loop. If it is:
      i. Collect the induction variable, step, and bounds.
      ii. Check whether the index is the induction variable. If it is, check whether the call is made in every iteration and every list element is set (compare the range of the induction variable to the list's length).

      Otherwise, collect the index of the element literally or symbolically.
   b) Collect the type of the element by repeating the type inference procedure on the value to which the element is set.
   c) Check whether the call post-dominates the definition of the object. If it does not, the type is the union of the initial type and the collected type. Collect the condition for the collected type by applying the procedure in Listing 3.17.

Listing 3.20: The type inference strategy from Listing 3.16 modified to collect conditions and handle loops. The modifications are shown in black.

Calls to Other Routines

The type inference strategy in Listing 3.20 only handles calls to routines provided by the R Internals programming interface. Since these routines are the only way to construct and modify R objects in foreign code, they make up the majority of calls on R objects. Nevertheless, R packages can include helper routines that manipulate R objects and are called from another routine—instead of or in addition to being called from R.

The type inference strategy cannot rely on a database of type signatures for these routines, since they vary across packages and package versions. Instead, we can apply the strategy recursively to infer their type signatures on demand. In addition to inferring types, the strategy should also determine if and how the arguments are modified (for example, with calls to `SET_VECTOR_ELT`), since R objects are passed to these routines by reference.

Example 49. The `NLoptR_Optimize` routine from the **nloptr** (Johnson and Ypma 2020), first presented in Example 44, contains a call to a helper routine. The routine returns a list, and one of its elements, `R_status_message` in Listing 3.8, is defined by a call to the helper routine `convertStatusToMessage`.

Suppose we apply the type inference strategy to the `convertStatusToMessage` routine, even though it's not called directly by `.Call`. The routine returns the value of the variable `R_status_message`:

```
1 PROTECT(R_status_message = allocVector(STRSXP, 1));
2 return R_status_message;
```

Based on the corresponding LLVM IR, the type inference strategy in Listing 3.20 infers that the routine returns a character vector (`STRSXP`) with length 1. The strategy can return this type information to use in the overall type inference for the list elements in Example 44. □

The point of the example is that the type inference strategy in Listing 3.20 works equally well for routines called with `.Call` and for helper routines. Types inferred for a helper routine can be used immediately in type inference for the calling routine.

The strategy in Listing 3.20 even infers the conditions for types that depend on a condition. These conditions will usually be in terms of the helper routine's parameters and local variables. Thus to use them as part of a larger type inference procedure, it is important for context to also record the helper routine's name. When a call to the helper routine has literal arguments,

these can be substituted into the condition to deduce a specific type—provided the condition is composed of elementary comparisons and logic operations.

The only necessary modification to the strategy in Listing 3.20 for helper routines is to check whether the arguments to the routine are modified. There are two different ways to do this.

One way is to use LLVM's built-in analysis of arguments that may be modified. We used this analysis for routines called with the `.C` interface in Example 42. This is convenient, but in testing we found that the analysis LLVM carries out is too conservative for R objects. Arguments were flagged as possibly modified any time they were passed to an R Internals routine, even if the routine did not actually modify the object.

The other way is to check the uses of the helper routine's parameters for any calls to routines which modify their arguments. These can be known R Internals routines (Table 3.6) or other helper routines, which must be checked recursively. This is approach leverages specific knowledge of the R Internals interface in order to get a more precise result, so it is preferable to relying on LLVM's built-in analysis.

Attributes

The attributes of an R object dictate its dimensions, class, and element names. Thus it is important for the type inference strategy to collect information from calls to routines that set attributes.

Routine	Attribute	Frequency
`Rf_setAttrib`	Any	2078
`SET_ATTRIB`	Any	31
`Rf_namesgets`	`R_NamesSymbol`	103
`Rf_lengthgets`		102
`Rf_classgets`	`R_ClassSymbol`	78
`Rf_dimnamesgets`	`R_DimNamesSymbol`	27
`Rf_dimgets`	`R_DimSymbol`	3

Table 3.7: Routines which set attributes on R objects, and how often they are called in CRAN packages (Temple Lang 2021). The `Rf_lengthgets` routine sets the length, which is not an attribute, but it is included here because length is closely related to dimensions.

The R Internals programming interface only provides seven routines to set attributes. The routine `Rf_setAttrib`, which sets a single attribute by name, makes up about 89% of calls to these routines (Temple Lang 2021). Routines which set specific attributes (see Table 3.7) make up a further 10%. The `SET_ATTRIB` routine, which can set multiple attributes on an object at

once, makes up the final 1%.

This section describes how to modify the type inference strategy of Listing 3.20 in order to collect information from routines which set attributes. We are only interested in R's built-in attributes, shown in Table 3.8. Other attributes have no bearing on the class, dimensions, and names of an object.

Example 50. Example 44 showed how to infer the element types in the list returned by the `NLoptR_Optimize` routine in the **nloptr** package (Johnson and Ypma 2020). This example shows how to modify the type inference strategy to infer the associated element names based on calls to `Rf_setAttrib`. Recall the C code for the routine, which was originally shown in Listing 3.8:

```
1  int num_return_elements = 8;
2  // ...
3  SEXP names;
4  PROTECT(names = allocVector(STRSXP, num_return_elements));
5
6  SET_STRING_ELT(names, 0, mkChar("status"));
7  SET_STRING_ELT(names, 1, mkChar("message"));
8  // ... for all 8 elements
9  SET_STRING_ELT(names, 7, mkChar("version_bugfix"));
10 setAttrib(R_result_list, R_NamesSymbol, names);
11 // ...
12 return(R_result_list);
```

Listing 3.21: The C code for `NLoptR_Optimize` related to the return value, abridged from what was shown in Listing 3.8 to focus on the names.

This code sets the `names` attribute on the returned list `R_result_list` to the character vector `names`. Each element of the vector becomes the name of the corresponding list element.

Here's the corresponding call to `Rf_setAttrib` in the LLVM IR:

```
1  %call140 = call %struct.SEXPREC* @Rf_setAttrib(
2      %struct.SEXPREC* %call127, struct.SEXPREC* %34,
3      %struct.SEXPREC* %call130) #4
```

The first argument is the target object, the second argument specifies the name of the attribute to set, and the third argument is the new value of the attribute. The variable `%34` is loaded from the global constant `@R_NamesSymbol`:

```
%34 = load %struct.SEXPREC*, %struct.SEXPREC** @R_NamesSymbol, align 8,
    !tbaa !4
```

This global constant refers to the names attribute. It is one of six global constants defined by the R Internals interface. Table 3.8 shows all of them. The type inference strategy can use the table to determine which attribute is set by a given call to `Rf_setAttrib` and handle the call appropriately. If call to `Rf_setAttrib` specifies the attribute name as a parameter or non-constant variable, then the attribute is run-time dependent. In the case of a parameter, it is still possible to recover the attribute name at specific call sites where the argument to the parameter is a literal or constant.

Attribute	Global Constant	Frequency
`names`	`R_NamesSymbol`	846
`class`	`R_ClassSymbol`	269
`dim`	`R_DimSymbol`	244
`dimnames`	`R_DimNamesSymbol`	138
`row.names`	`R_RowNamesSymbol`	52
`levels`	`R_LevelsSymbol`	29
Local Variable		403
Parameter of Parent Routine		16

Table 3.8: R's built-in attributes, organized by frequency of use in calls to `Rf_setAttrib` in CRAN packages (Temple Lang 2021). Italicized rows correspond to calls where the attribute argument was not a literal or a global constant.

To determine what the names have been set to, we need to inspect the `%call130` variable. As usual, we inspect its definition and uses. The variable is defined as a character vector (see Table 3.2) with 8 elements:

```
%call130 = call %struct.SEXPREC* @Rf_allocVector(i32 16, i64 8) #4
```

The `%call130` variable is then used in multiple calls to the `SET_STRING_ELT` routine. Here's an example of one, along with a related instruction:

```
%call132 = call %struct.SEXPREC* @Rf_mkChar(i8* getelementptr inbounds (
    [7 x i8], [7 x i8]* @.str.102, i64 0, i64 0)) #4
```

```
3  call void @SET_STRING_ELT(%struct.SEXPREC* %call130, i64 0,
4      %struct.SEXPREC* %call132) #4
```

The `SET_STRING_ELT` routine is similar to `SET_VECTOR_ELT`, but sets an element of a character vector. The first argument is the character vector, the second is the position, and the third is the new value, which must be a `CHARSXP`. Here the third argument, `%call132`, is defined by the preceding instruction, a call to `Rf_mkChar`, which converts a C character array into a `CHARSXP`.

In this case, the character array is the global constant `@.str.102`, defined as the literal `"status"` (in the LLVM IR, literal character arrays are always stored as global constants). The nested `getelementptr` instruction retrieves a pointer to (the first character of) this array. We can conclude that the combined result of these instructions is to set the first element of `%call130`—the name of the first element in the returned list—to `"status"`.

The same procedure correctly infers the names of the other seven list elements as well. This procedure is the basis for modifying the type inference strategy. A version of this procedure generalized to all of R's built-in attributes is shown in Listing 3.22. □

The example assumes that the value to which the attribute is set is constant. If the value comes from a parameter, it cannot be determined statically, but the parameter can be recorded symbolically as the source. This is noted in Listing 3.22. The case where inference doesn't work well is when the value comes from a local variable computed by one or more calls to other routines. Even in this case, it may be possible to infer the value if the routines follow a regular pattern, although we do not attempt to do so here.

The example and Listing 3.22 both assume that the call to `Rf_setAttrib` (or other attribute-setting routine) itself is not conditional. This is a reasonable assumption for attributes pertaining to names and classes, especially on returned objects, because these objects need to have a predictable structure in order for programmers to use them. However, to handle calls that do depend on a condition, we can further modify the strategy in Listing 3.22 to infer sets of potential values and conditions for attributes. The approach is the same as the one Examples 46 and 47 used for object-modifying calls that depend on a condition. That is, for each call to set the attribute, collect the potential value and its condition.

In about 20% of the calls to `Rf_setAttrib` in CRAN packages, the attribute name is passed via a variable rather than a global constant (Temple Lang 2021). In almost all of these cases, the variable is constant-valued, and type inference can inspect the value to determine which

attribute is set. There were only 16 cases where the attribute name depended on a parameter of the calling routine, making it impossible to infer from the C code alone, but still possible to infer if the parameter is assigned a constant argument in the calling R code.

1. Determine the attribute name.
   - For calls to the routines which set specific attributes, look up the attribute name in Table 3.7.
   - For calls to `Rf_setAttrib`, if the third argument (the attribute name) is a global constant, look it up in Table 3.8.
2. Find the value of the attribute. The exact strategy depends on the attribute:
   - For the `names`, `class`, and `row.names` attributes, the value of the attribute is a character vector. Collect the values of the elements from calls to `SET_STRING_ELT`.
   - For the `dimnames` attribute, the value of the attribute is a list of scalar strings. Collect the values of the elements from calls to `SET_VECTOR_ELT`.
   - For the `dim` attribute, the value of the attribute is an integer vector. Collect the values of the elements from assignments made after calling the `INTEGER` routine to get an `int` pointer to the values of the vector.

   Record the values literally when possible, or else symbolically.

Listing 3.22: A strategy to collect attributes, to supplement the type inference strategy in Listing 3.20.

S3 Classes

R's S3 class system is based on the `class` attribute. The strategy for collecting attributes in Listing 3.22 already describes how to collect the `class` attribute, but in this section we turn to an example to demonstrate that the stated strategy is effective.

Example 51. The **pingr** package (Csárdi 2020) provides tools to ping or otherwise check the status of a remote server. The package's function `nsl`, which performs a domain name server query, calls the C routine `r_nsl`. From a code analysis perspective, this routine is interesting because it constructs a data frame from scratch. Here's the relevant C code:

```
1 const char *resnames[] = { "answer", "flags", "" };
2 const char *recnames[] = { "name", "class", "type", "ttl", "data", "" };
```

```
SEXP result = PROTECT(mkNamed(VECSXP, resnames));
SEXP records = PROTECT(mkNamed(VECSXP, recnames));
SEXP row_names = PROTECT(Rf_allocVector(INTSXP, 2));
Rf_setAttrib(records, R_ClassSymbol, mkString("data.frame"));
SET_VECTOR_ELT(result, 0, records);
// ...
SET_VECTOR_ELT(records, 0, Rf_allocVector(STRSXP, cnt));
SET_VECTOR_ELT(records, 1, Rf_allocVector(INTSXP, cnt));
SET_VECTOR_ELT(records, 2, Rf_allocVector(INTSXP, cnt));
SET_VECTOR_ELT(records, 3, Rf_allocVector(INTSXP, cnt));
SET_VECTOR_ELT(records, 4, Rf_allocVector(VECSXP, cnt));
INTEGER(row_names)[0] = NA_INTEGER;
INTEGER(row_names)[1] = -cnt;
Rf_setAttrib(records, R_RowNamesSymbol, row_names);
// ...
return result;
```

Listing 3.23: Code related to the first element of the list returned by the `r_nsl` routine in the **pingr** package (Csárdi 2020). This element, called `records` in the code, is a data frame. Most other code from the routine is omitted.

The actual return value of the routine is a list. Type inference for the list follows the same strategy outlined in earlier examples, so here we'll focus exclusively on the first element of the list, which is called `records` in the C code. The `records` object is a data frame; we know this because the code sets the class attribute.

Let's look at the LLVM IR for the routine. The `records` variable in the C code corresponds to the `%call3` and `%call4` variables in the IR. Here's how the variables are defined:

```
%4 = getelementptr inbounds [6 x i8*], [6 x i8*]* %recnames,
    i64 0, i64 0
store i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.str.3,
    i64 0, i64 0), i8** %4, align 16
%call3 = call %struct.SEXPREC* @Rf_mkNamed(i32 19, i8** nonnull %4) #7
%call4 = call %struct.SEXPREC* @Rf_protect(%struct.SEXPREC* %call3) #7
```

The `Rf_mkNamed` routine constructs a named R object of a given type. In this case, the call constructs a list (see Table 3.3). The element names are set by the variable `%4`, which is a pointer to `%recnames`, a constant array of strings. Thus the type inference strategy can infer the type and the element names from the call to `Rf_mkNamed` and its arguments.

The next step is to check the uses of `%call4`, in order to determine its element types and attributes. The first use is a call to `Rf_setAttrib`:

```
%11 = load %struct.SEXPREC*, %struct.SEXPREC** @R_ClassSymbol,
    align 8, !tbaa !4
%call7 = call %struct.SEXPREC* @Rf_mkString(
    i8* getelementptr inbounds ([11 x i8], [11 x i8]* @.str.14,
    i64 0, i64 0)) #7
%call8 = call %struct.SEXPREC* @Rf_setAttrib(
    %struct.SEXPREC* %call4, %struct.SEXPREC* %11,
    %struct.SEXPREC* %call7) #7
```

As we saw in Example 50, the first argument is the object, the second is the name of the attribute to set, and the third is the new value. The variable `%11` corresponds to the global variable `@R_ClassSymbol`, so this instruction sets the `class` attribute. The value comes from the variable `%call7`, which is defined by a call to `Rf_mkString` with the global variable `@.str.14`. This global variable is a constant string with the value `"data.frame"`. Thus we have the everything we need to infer that the variable `%call4` is a data frame.

With the class determined, type inference can proceed to inspect the rest of the uses. Through calls to `SET_VECTOR_ELT`, they reveal the types of the columns in the data frame. Since this process is the same as for inferring the element types of a list, we omit the details.

One additional point of interest here is that the class is not the only attribute set on the `%call4` object. The code also sets the `row.names` attribute. This is apparent in the C code, and also in the IR. The attribute is deliberately set to an invalid vector (with two elements, where the first is `NA`), so that R will automatically generate a sequence of integer row names. Since this a specific pattern, we can detect it with type inference in order to infer the row names in addition to the class. □

S4 Classes

S4 classes are formal, which means the class, its slots (fields), and its slot types must all be defined explicitly (in R code) before the class can be used. As a consequence, the name of an S4 class is usually sufficient to get complete type information for any instance of the class.

Example 52. This example shows how to modify the type inference strategy from Listing 3.20 to identify and collect the class name of S4 objects constructed with the R Internals interface.

Consider the **float** package (Schmidt n.d.), which provides a 32-bit floating point number type for R. The type is represented by the S4 class `float32`. The package's routine `R_scale_spm` scales and centers a vector or matrix of floating point numbers, and returns the result in a new `float32` matrix.

Constructing an S4 object with the R Internals interface is a two step process, which is exhibited in C code for `R_scale_spm`. First, the code gets a prototype for the class by calling the `MAKE_CLASS` macro on the name of the class:

```
1 PROTECT(ret_s4_class = MAKE_CLASS("float32"));
```

The macro passes the name to the `R_do_MAKE_CLASS` routine, which returns the prototype.

Next, the code requests a new object by calling the `NEW_OBJECT` macro on the prototype:

```
1 PROTECT(ret_s4 = NEW_OBJECT(ret_s4_class));
```

The macro passes the prototype to the `R_do_new_object` routine, which returns the new object.

In the LLVM IR, both macros are translated into calls to the corresponding routines. Here are the calls:

```
1 %call44 = tail call %struct.SEXPREC* @R_do_MAKE_CLASS(
2     i8* getelementptr inbounds (
3         [8 x i8], [8 x i8]* @.str, i64 0, i64 0)) #5
4 %call46 = tail call %struct.SEXPREC* @R_do_new_object(
5     %struct.SEXPREC* %call44) #5
```

When an R object is defined by a call to `R_do_new_object`, as `%call44` is here, then the type inference strategy should find the preceding call to `R_do_MAKE_CLASS` and collect the name of the object's S4 class. In this case the argument to `R_do_MAKE_CLASS` is the the global constant `@.str`, which is the string `"float32"`. □

Once we have the class name for an S4 object, the types for the object's slots can usually be inferred from the class definition in the R code. However, S4 classes can be defined with slots that accept more than one type or accept any type. To handle these cases, we can have the type inference strategy in Listing 3.20 handle slots the same way other heterogeneous container elements are handled. The R Internals routine `R_do_slot_assign` sets a slot by name. This is noted in Table 3.6, which lists routines that set elements of heterogeneous containers.

3.4.2 Parameter Types

This subsection describes the strategy to infer the R parameter types of a routine. As with the the strategy to infer return types (Section 3.4.1), we use LLVM to get the uses of the parameters and analyze these to infer a type for each. Unlike that strategy, for parameters we cannot consider definitions because parameters do not have any. Once again, we refine the strategy gradually through examples, starting from a simple core strategy.

Example 53. This example demonstrates the core strategy to infer parameter types. Consider the **rpart** package (Therneau and Atkinson 2019), which fits classification and regression tree models. The primary model-fitting function in the package is the eponymous `rpart` function. This function, in turn, calls the C routine `rpart`, which is also provided by the package. In the C code, the type signature for the `rpart` routine is:

```
1 SEXP rpart(
2     SEXP ncat2, SEXP method2, SEXP opt2, SEXP parms2,
3     SEXP xvals2, SEXP xgrp2, SEXP ymat2, SEXP xmat2,
4     SEXP wt2, SEXP ny2, SEXP cost2)
```

We'll do the analysis on the LLVM IR rather the the C code, but we'll include snippets of C code in this example for reference.

Let's focus specifically on the first parameter `ncat2`. This parameter is used in two calls in the C code:

```
1 ncat = INTEGER(ncat2);
2 // ...
3 rp.numcat = INTEGER(ncat2);
```

In the IR, the `ncat2` parameter becomes `%ncat2`. We can use the **Rllvm** function `getAllUsers` to get the two instructions where the parameter is used. Here's the first:

```
%call = tail call i32* @INTEGER(%struct.SEXPREC* %ncat2) #6
```

The `INTEGER` routine requires an integer (`INTSXP`) argument (Table 3.4), so from this call the type inference strategy can conclude that the `ncat2` parameter is an integer vector. Since the second use of `ncat2` is another call to `INTEGER`, it doesn't provide any additional information.

The type inference strategy can infer that the parameter `xgrp2` is also an integer vector, since it is also used in a call to `INTEGER`. Likewise, the parameters `opt2`, `parms2`, `wt2`, and `cost2` are each used in a call to `REAL`, and therefore must be numeric vectors.

In addition to type, the uses of a parameter can indicate its class or dimensions. Thus for each parameter, the type inference strategy should check all of the uses. The `xmat2` parameter demonstrates this point. The combined uses of the parameter suggest it is a numeric matrix. Note the calls to `nrows`, `ncols`, and `REAL` in the C code:

```
rp.n = nrows(xmat2);
rp.nvar = ncols(xmat2);
dptr = REAL(xmat2);
```

Listing 3.24: C code from the `rpart` routine that uses the parameter `xmat2`.

In the IR, the three calls are translated into these three instructions:

```
%call36 = tail call i32 @Rf_nrows(%struct.SEXPREC* %xmat2) #6
%call37 = tail call i32 @Rf_ncols(%struct.SEXPREC* %xmat2) #6
%call40 = tail call double* @REAL(%struct.SEXPREC* %xmat2) #6
```

The `Rf_nrows` and `Rf_ncols` routines return the number of rows and columns, respectively, of a data frame, matrix, array, or other object with the `dim` attribute set. Thus `xmat2` has at least two dimensions. The call to `REAL` indicates that `xmat2` is also numeric and has homogeneous elements. Moreover, this call rules out the possibility that `xmat2` is a data frame, because data frames are lists, and lists are incompatible with the `REAL` routine. Thus the parameter `xmat2` is a matrix, array, or custom class; the code does not provide enough information to determine which.

Finally, consider the `ymat2` parameter. This parameter is only used in a call to `REAL`. However, the pointer `dptr` returned by the call is used in a loop which suggests `ymat2` is two-dimensional:

```
n = rp.n;
rp.num_y = asInteger(ny2);
rp.ydata = (double **) ALLOC(n, sizeof(double *));
dptr = REAL(ymat2);
for (i = 0; i < n; i++) {
    rp.ydata[i] = dptr;
    dptr += rp.num_y;
}
```

Listing 3.25: C code in the `rpart` routine that uses the parameter `ymat2`.

The loop iterates `n` times, the same as the number of rows in `xmat2`. In each iteration, the pointer is advanced by the value of `ny2`. This loop follows a C programming pattern for iterating over the first row of two-dimensional object stored in column-major order (R defaults to column-major order). The type inference strategy we propose does not use this information and only infers that `ymat2` is numeric. However, one could extend the strategy to search for loops that use the pointer returned by a call to `REAL` (or the other routines in Table 3.4) and match known patterns of iteration. This would make the strategy more complex, but also improve the accuracy of the inferred dimensions.

We defer type inference for the `method2`, `xvals2`, and `ny2` parameters to Example 54. □

The example demonstrates the core strategy to infer parameter types: for each parameter, analyze the uses for calls to R Internals routines that restrict the range of possible types. Calls to some routines—such as `Rf_nrows`—provide information about the class and dimensions of their arguments instead of or in addition to the type. As a consequence, it is important to inspect all of the uses of each parameter (rather than stopping at the first one to indicate the type). Listing 3.26 shows the core strategy.

For each parameter, get all uses. For each use, if the instruction is a call to:

- An R internals routine with restrictions on the types of its arguments. Look up the routine in the database of routines. Since the parameter is used as an argument, the restrictions also apply to the parameter.

If more than one type is found for a parameter, the inferred type is the union of the types. Collect the condition from the nearest common predecessor of the blocks which the contain uses with different types.

Listing 3.26: The core strategy to infer a routine's parameter types. Subsequent sections of this chapter refine and expand upon the strategy.

Examining all uses of each parameter is also important for another reason. Different paths of execution through a routine can assume different types for the parameters. For each parameter, we want the strategy to find all possible types. When a parameter can have more than one type, the inferred type should be the union, and the strategy should collect conditions on the potential types. To do this, the strategy can use the same technique we used for collecting conditions on return types in Listing 3.20. This step is included in the core strategy in Listing 3.26.

Coercions

The R Internals programming interface provides several routines to coerce R objects to specific types. These are listed in Table 3.9. Most of these routines return the missing value `NA` if coercion fails. When a parameter is coerced, we can think of the target type as the "natural" type for that parameter, in the sense that the parameter must be or be compatible with that type.

This section is about how to modify the type inference strategy for parameters in Listing 3.26 to collect type information from calls to coercion routines. The key idea is that the strategy should check how routines handle parameters for which coercion fails. If failure to coerce a parameter leads to an error, then that parameter must be coercible. If failure doesn't lead to an error, then the routine accepts other types of arguments for that parameter.

Example 54. Example 53 demonstrated type inference for all of the parameters of the `rpart` routine in the **rpart** package (Therneau and Atkinson 2019) except for `method2`, `xvals2`, and `ny2`. All three are coerced to `int`s in calls to the `Rf_asInteger` routine. We consider them in this example in order to examine how to modify the type inference strategy to handle coercions.

Routine	Argument(s)	Coercion-free Type
`Rf_asLogical`	τ	`LGL`
`Rf_asInteger`	τ	`INT`
`Rf_asReal`	τ	`REAL`
`Rf_asComplex`	τ	`CPLX`
`Rf_asChar`	τ	`STR`
`Rf_coerceVector`	τ, U	U
`Rf_asCharacterFactor`	`factor`	`factor`

Table 3.9: Routines from the R Internals interface to coerce objects to specific types. The coercion-free type (or class) is the natural way to represent the target type as a `SEXP`. U denotes any `SEXPTYPE` and τ denotes any atomic vector in the sense of the `isVectorAtomic` routine (see Table 3.10). The routines return an error if the first argument is not an atomic vector, and return a missing value if coercion fails.

We'll begin with the parameter `method2`. The first call to `Rf_asInteger` on `method2` is translated into this instruction in the LLVM IR:

```
1 %call5 = tail call i32 @Rf_asInteger(%struct.SEXPREC* %method2) #6
```

The `Rf_asInteger` routine accepts any `SEXP` and attempts to coerce the underlying data to the C type `int`. If the argument is an integer (`INTSXP`), no coercion actually occurs, and the effect is the same as calling the `INTEGER` routine. However, the argument could instead be a logical, numeric, complex, or character vector. In any of those cases, it's possible that the coercion will fail. On failure, `Rf_asInteger` returns a missing value rather than throwing an error.

The only other use of `method2` in the routine is another call to `Rf_asInteger`. For context, let's look at the C code that produces these two call instructions:

```
1 if (asInteger(method2) <= NUM_METHODS) {
2     i = asInteger(method2) - 1;
3     \\ ...
4 } else
5     error(_("Invalid value for 'method'"));
```

Listing 3.27: The C code that generates the two call instructions that use `method2`. The symbol `asInteger` is translated into `Rf_asInteger` by the C preprocessor.

The comparison on the first line checks that the result from the call to `asInteger` is less than or equal to the constant `NUM_METHODS`. Since R represents the missing value as a C `NaN` value with specific bits set, and ordering checks against `NaN` values are always false, this condition is

also false if the coercion failed. When the condition is false, the routine raises an error. Thus the routine expects that the argument for `method2` must be coercible to an integer, and raises an error if it is not.

In order to detect whether a routine raises an error if a coercion fails, the type inference strategy should follow the result of each call to a coercion routine. Assuming the result is a missing value, the strategy should check whether the code branches to a block that raises an error. There are two different ways to identify these blocks:

1. By the `unreachable` terminator instruction, which means execution does not jump to any other block after the block finishes executing. Since errors cause routines to exit early without returning anything, code that raises an error always includes this instruction.

2. By calls to the error routines provided by the R Internals interface. The main routine for raising errors is `Rf_error`, but there are also other routines that interact with R's error system. This approach has the advantage that the type inference system can recover the error message if it is constant and can detect warnings in addition to errors.

Either approach works for this particular routine. The relevant instructions are:

```
1    %call5 = tail call i32 @Rf_asInteger(%struct.SEXPREC* %method2) #6
2    %cmp = icmp slt i32 %call5, 5
3    br i1 %cmp, label %if.then, label %if.else
4
5  if.else:                                          ; preds = %entry
6    ; ...
7    tail call void (i8*, ...) @Rf_error(i8* %call14) #7
8    unreachable
```

Listing 3.28: IR instructions that show the `rpart` routine raises an error if the result from calling `Rf_asInteger` on `method2` is the missing value `NA`.

Key to the analysis here is that the result of the coercion, `%call5`, is used in a comparison instruction (`icmp`). The comparison is always false if the result is the missing value, and the target block when the comparison is false, `if.else`, raises an error. We can see this from the call to `Rf_error`, or from the `unreachable` instruction.

Unlike the first call to `Rf_asInteger` on `method2`, the second call does not lead to an error, regardless of its result. Nevertheless, the fact that the first call does lead to an error if coercion

fails is sufficient to conclude that the `method2` parameter must be or be compatible with an integer.

For the remaining two parameters, `method2` and `xvals2`, there is no code to raise an error if coercion fails. This suggests the routine can proceed normally and return a result even if these parameters cannot be coerced. Thus it is not possible to infer a type for them through static analysis. The type inference system should still record that the coercions appear in the routine, since they indicate that in typical usage, these parameters should be integers, even though the routine produces a result even if they are not. □

Listing 3.29 shows a modified version of the type inference strategy based on the example (Example 54). Calls to the coercion routines listed in Table 3.9 provide important type information.

For each parameter, get all uses. For each use, if the instruction is a call to:

- An R internals routine with restrictions on the types of its arguments. Look up the routine in the database of routines. Since the parameter is used as an argument, the restrictions also apply to the parameter.
- A coercion routine. Check if the routine raises an error if the coercion fails (that is, the result is a missing value). If the routined does, then the parameter must be or be compatible with the target type. Otherwise, just record that the parameter is coerced to the type indicated by the routine.

If more than one type is found for a parameter, the inferred type is the union of the types. Collect the condition from the nearest common predecessor of the blocks which the contain uses with different types.

Listing 3.29: The modified strategy to infer a routine's parameter types. Modifications of Listing 3.26 to handle coercion routines are shown in black.

Assertions

Tests and assertions about the type of a parameter are another source of information for type inference. Table 3.10 lists routines provided by the R Internals programming interface to test whether objects have specific types.

Assertions are a valuable source of information for type inference, because they essentially

Routine	Type	Notes
Rf_isNull	NIL	
Rf_isLogical	LGL	
Rf_isInteger	INT	
Rf_isReal	REAL	
Rf_isComplex	CPLX	
Rf_isString	STR	
Rf_isValidString	STR	Length at least 1
Rf_isSymbol	SYM	
Rf_isEnvironment	ENV	
Rf_isFunction	CLO, BUILTIN, SPECIAL	
Rf_isLanguage	NIL, LANG	
Rf_isList	NIL, LIST	
Rf_isPairList	NIL, LIST LANG, DOT	
Rf_isNewList	NIL, VEC	
Rf_isVectorList	VEC, EXPR	
Rf_isVectorAtomic	LGL, INT, REAL, CPLX, STR, RAW	
Rf_isVector	isVectorAtomic, isVectorList	
Rf_type2str		Returns name of type
TYPEOF		Returns SEXPTYPE

Table 3.10: Routines provided by the R Internals interface to test for specific SEXPTYPEs. Where the type column lists a routine, the test is true for all arguments for which that routine returns true. The last two routines return the name of their argument's type as a string or SEXPTYPE rather than true or false.

state the type (or some other characteristic) that a parameter must have. In this section, we examine how to modify the type inference strategy in Listing 3.29 to identify assertions and collect type information from them.

Example 55. The **mco** package provides functions to solve multiple criteria optimization problems (Mersmann 2020). One of these functions, `nsga2`, calls the C routine `do_nsga2`. This routine is notable for the many assertions it makes before computing anything. Here's the type signature for the routine and a sample of the assertions:

```
1 SEXP do_nsga2(SEXP s_function, SEXP s_constraint, SEXP s_env,
2               SEXP s_obj_dim, SEXP s_constr_dim, SEXP s_input_dim,
3               SEXP s_lower_bound, SEXP s_upper_bound, SEXP s_popsize,
4               SEXP s_generations, SEXP s_crossing_prob,
5               SEXP s_crossing_dist, SEXP s_mutation_prob,
6               SEXP s_mutation_dist) {
7   if (!isFunction(s_function))
```

```
8       error("Argument 's_function' is not a function.");
9     if (!isFunction(s_constraint))
10      error("Argument 's_constraint' is not a function.");
11    if (!isInteger(s_input_dim))
12      error("Argument 's_input_dim' is not an integer.");
13    if (!isInteger(s_constr_dim))
14      error("Argument 's_constr_dim' is not an integer.");
15    if (!isReal(s_lower_bound))
16      error("Argument 's_lower_bound' is not a real vector.");
17    if (!isReal(s_upper_bound))
18    // ...
19    if (!isInteger(s_mutation_dist))
20      error("Argument 's_mutation_dist' is not an integer.");
```

Listing 3.30: An excerpt of the C code for the `do_nsga2` routine in the **mco** package (Mersmann 2020).

For this routine, there is an assertion for every single parameter. If any assertion fails, an error is raised immediately. For the parameters that require integer or numeric arguments, the assertions provide complete information about the type. The other parameters require function arguments; for these, the assertions narrow down the `SEXPTYPE` to R's three different function types (`builtin`, `special`, and `closure`), but no more. Inferring which of the three is correct and the type signature still depends on inspecting the other uses. Nonetheless, the assertions greatly limit the set of possible types.

Consider the parameter `s_input_dim`. In the LLVM IR, the assertion on `s_input_dim` is translated into two blocks. The first block, `if.end4`, tests whether `s_input_dim` is an integer (`INTSXP`) with a call to `Rf_isInteger`:

```
1  if.end4:                          ; preds = %if.end
2      %call5 = tail call i32 @Rf_isInteger(
3          %struct.SEXPREC* %s_input_dim) #10
4      %tobool6.not = icmp eq i32 %call5, 0
5      br i1 %tobool6.not, label %if.then7, label %if.end8
```

Since this is a call to one of the test routines listed in Table 3.10, the type inference strategy

should follow the result `%call5` to see how it is used. In particular, the strategy should check whether a false result leads to an error.

The value of `%call5` is negated by a `not` instruction to get `%tobool6.not`, and this is then used in a branch instruction. When `s_input_dim` is not an integer, `%tobool6.not` is true, and the code branches to `%if.then7`.

Here's the `%if.then7` block:

```
1 if.then7:                       ; preds = %if.end4
2     tail call void (i8*, ...) @Rf_error(
3         i8* getelementptr inbounds ([42 x i8],
4         [42 x i8]* @.str.2, i64 0, i64 0)) #11
5     unreachable
```

As we saw in Example 54, we can identify errors either through the `unreachable` instruction or a call to `Rf_error` (the latter implies the former, but not vice-versa, because other routines can cause execution to abort). The `if.then7` block has both. Thus whenever the original call to `Rf_isInteger` returns false, the routine raises an error. This is the pattern of an assertion, and it establishes that `s_input_dim` must be an integer vector.

The pattern for the other assertions is the same, so we do not inspect their IR code here. The type inference strategy can use the same technique to collect type information from each. After inspecting the assertions, the type inference can then proceed to inspect the other uses as described in Listing 3.29. □

Listing 3.31 shows the type inference strategy modified to handle assertions.

For each parameter, get all uses. For each use, if the instruction is a call to:

- A routine for testing types (Table 3.10). Check whether a false result leads to a block that raises an error by calling `Rf_error`. If it does, the call to the test routine is part of an assertion and indicates the type of the parameter.
- An R internals routine with restrictions on the types of its arguments. Look up the routine in the database of routines. Since the parameter is used as an argument, the restrictions also apply to the parameter.
- A coercion routine. Check if the routine raises an error if the coercion fails (that is, the result is a missing value). If the routined does, then the parameter must be or be compatible with the target type. Otherwise, just record that the parameter is coerced to the type indicated by the routine.

If more than one type is found for a parameter, the inferred type is the union of the types. Collect the condition from the nearest common predecessor of the blocks which the contain uses with different types.

Listing 3.31: The modified strategy to infer a routine's parameter types. Modifications of Listing 3.29 to handle assertions are shown in black.

The routines `Rf_type2str` and `TYPEOF` in Table 3.10 do not test for a specific type on their own. Instead, they return the type of their argument as a string or a `SEXPTYPE`, respectively. These routines are included in the table because they are frequently used to test for types.

Example 56. The **RCurl** package provides the `base64` function to encode and decode strings in base 64 (Temple Lang and CRAN Team 2021). For decoding, the function calls the routine `R_base64_decode`. This routine is notable because it accepts either a raw vector (`RAWSXP`) or a character vector (`STRSXP`) for its first argument. This example describes how the type inference strategy in Listing 3.31 can be adapted to handle parameters with more than one allowed type, and also demonstrates what type checks that use the `TYPEOF` routine look like.

The routine's first parameter is called `r_text`. Here's the C code that checks the type of `r_text` and handles each case:

```
1 char *text;
2 size_t len;
```

```
3 if(TYPEOF(r_text) == STRSXP)
4     text = (char *) CHAR(STRING_ELT(r_text, 0));
5 else {
6     len = LENGTH(r_text);
7     text = R_alloc(len+1, 1); text[len] = '\0';
8     memcpy(text, RAW(r_text), len);
9 }
```

The calls to R Internals routines `STRING_ELT` and `RAW` indicate the type of `r_text` on each branch. The condition in the if-statement also indicates the type on the true branch.

In the LLVM IR, the call to `TYPEOF` and if-statement become:

```
1   %call = tail call i32 @TYPEOF(%struct.SEXPREC* %r_text) #6
2   %cmp = icmp eq i32 %call, 16
3   br i1 %cmp, label %if.then, label %if.else
```

The `icmp` instruction compares the result from `TYPEOF` to `16`, the enumeration value for `STRSXP` (Table 3.2). The IR then branches to the `%if.then` block if `r_text` is a character vector, or `%if.else` if it is not. These blocks correspond to the two branches of the if-statement in the C code.

The type inference strategy should detect when a result from `TYPEOF` is compared to an integer and look up the integer in Table 3.2 to determine which type is being checked. The strategy must also check that this is not an assertion—meaning there is no call to `Rf_error` along either branch. The type along the true branch is established by the condition, although the type inference strategy should also detect that `r_text` is used in the routine `STRING_ELT`, which only accepts character vectors.

Now consider the IR for the false branch, the block `%if.else`. It contains a call to the `RAW` routine on `r_text`:

```
1 %call5 = tail call i8* @RAW(%struct.SEXPREC* %r_text) #6
```

By inspecting this call, the type inference strategy can infer that the `r_text` parameter can also be a raw vector.

For this example, it is not actually necessary for the type inference strategy to analyze the call to `TYPEOF` in order to determine that `r_text` can be either a character vector or a raw vector.

In fact, even the core type inference strategy in Listing 3.26 correctly infers the two types for `r_text`. However, the example shows how the type inference strategy could use a call to `TYPEOF`, and examining the call would be essential if this code did not include the call to `STRING_ELT` on `r_text`. □

Besides checking for types, assertions can also check for other properties of an object. Many of these fall outside the immediate goals of type inference, but one exception is assertions about S3 and S4 classes. Table 3.11 lists routines in the R programming interface that test for classes. Although these routines check for classes instead of types, the strategy for detecting assertions that use them is the same. In some cases, these routines also reveal something about the type.

Routine	Type	Class	Notes
`Rf_isNumeric`	`LGL`, `INT`, `REAL`	Not factor	
`Rf_isMatrix`	`isVector`		Length 2 `INT` dimensions
`Rf_isArray`	`isVector`		`INT` dimensions
`Rf_isFactor`	`INT`	factor	
`Rf_isFrame`	τ	data.frame	
`Rf_inherits`	τ		
`Rf_isS4`	τ		S4 bit, not just `S4SXP`
`IS_S4_OBJECT`	τ		S4 bit, not just `S4SXP`

Table 3.11: Routines from the C interface related to testing for specific classed or class-compatible objects.

3.4.3 The `.External` and `.External2` Interfaces

Like `.Call`, the `.External` and `.External2` interfaces are designed for calls to foreign routines that use the R Internals programming interface. The primary difference is that routines called with `.External` and `.External2` must have exactly one parameter. The argument to this parameter is a pairlist (`LISTSXP`) that contains one element for each argument to be passed to the routine. The purpose of this design is to allow foreign routines to accept a variable number of arguments. Use of these interfaces is relatively uncommon: a 2021 analysis of CRAN, Temple Lang found that only 29 packages (of approximately 17,000) call `.External` or `.External2`.

Elements in the pairlist can be accessed with the `CAR` and `CDR` routines, as shown in Table 3.6. As a result, the type inference strategy for parameter types (Listing 3.31) must be modified in order to correctly infer types for with routines called with these interfaces. More specifically, the strategy should count calls to `CAR` and `CDR` in order to determine the number of parameters in the parameter list. Where the results from calling `CAR` on the parameter list are saved into

variables, the type inference strategy should treat those variables as the parameters of the function, handling them the same way as parameters in a routine called with `.Call`.

The type inference strategy for return types (Listing 3.20) still applies without modification.

3.4.4 C++ Routines

The `.Call`, `.External`, and `.External2` routines can also call C++ routines, and on CRAN in 2020, about 62% of packages with foreign routines contained C++ code. The `clang` compiler can translate C++ code into LLVM IR, so for C++ routines that use the R Internals programming interface, the type inference strategies described in Section 3.4.1 and 3.4.2 still apply.

There are, however, many C++ routines that use the **Rcpp** package (Eddelbuettel and François 2011) instead of or in addition to the R Internals interface. The **Rcpp** package provides C++ data types and routines to integrate R and C++. In other words, it provides a programming interface for manipulating R objects in C++ which is distinct from the R Internals interface. As a result, the strategies of Section 3.4 must be modified in order to correctly infer types for routines which use **Rcpp**.

Like the routines provided by the R Internals interface, many of the routines provided by **Rcpp** restrict their arguments or return value to specific types. Thus it is feasible to create a database of these routines for use in type inference, either manually or programatically.

The **Rcpp** package also makes extensive use of *templates*. A template parameterizes the types in a routine or class, so that the code can be reused for several different types. When a template is compiled, the compiler repeats the code for each requested type. In **Rcpp**, there are templated routines (such as `sum`) and classes (such as `Vector`) parameterized in terms of `SEXPTYPE`s. A preliminary investigation by Temple Lang (2021) indicates that it is possible to recover the template arguments from the LLVM IR, and use this information for type inference.

Extending the type inference strategies to support C++ routines which use **Rcpp** is an important area for future work, because **Rcpp** is in widespread use.

3.5 Connecting to R Code

The motivation for type inference on foreign routines called from R—and thus for this chapter—is to provide additional information to the type inference strategy for R code described in Chapter 2. In order to provide the information, two things must be considered: how to represent types of

R objects in foreign routines, and where type inference for foreign routines fits into the type inference strategy for R code.

Since `SEXPTYPE`s are R types, we can use the classes provided by the **typesys** package to represent types for R objects in foreign routines. These classes were described in Section 2.5.1 of the previous chapter. Should it be necessary to distinguish between types found in foreign routines and types found in R code, we can use subclasses of these classes. Due to the design of the package, making these subclasses work with functions designed to handle the types provided by the package does not require any additional effort.

In Section 3.4.1 and Section 3.4.2, we explained that type inference should not only collect type information, but also information about coercions, conditions, classes, and dimensions. The **typesys** package also provides facilities for recording these. One exception is that **typesys** does not provide a way to represent conditions in LLVM IR code. There are various ways we can approach this deficiency:

- Use strings to represent the code
- Translate the conditions into pseudo-R code
- Use file name and line numbers to refer to conditions
- Serialize the relevant LLVM IR code

We favor translating conditions into pseudo-R code. Users of the type inference system will already be familiar with R, so pseudo-R code is likely to be easier for them to understand than the other options. The drawback is that this requires an extra step during type inference. As of writing, we have not implemented this system, so it bears further investigation.

The other issue is how type inference for foreign routines fits into the R type inference strategy. As described in this chapter, the type inference strategy for foreign code can operate independently of the type inference strategy for R code. However, sharing information from type inference for R code can improve the accuracy of the results. This is illustrated in the following example.

Example 57. Example 55 examined the `do_nsga2` routine in the **mco** package (Mersmann 2020). That routine is called by the package's `nsga2` function through the `.Call` interface. In the call to `.Call`, the R code uses coercions to marshal objects to appropriate types:

```
res <- .Call(do_nsga2,
             ff, cf, sys.frame(),
             as.integer(odim),
             as.integer(cdim),
             as.integer(idim),
             lower.bounds, upper.bounds,
             as.integer(popsize), as.integer(generations),
             cprob, as.integer(cdist),
             mprob, as.integer(mdist))
```

Listing 3.32: The `.Call` expression from the `nsga2` function in the **mco** package (Mersmann 2020). Most of the objects passed as arguments are wrapped in calls to `as.integer`.

This practice is common in R packages that call foreign routines, and it provides valuable information about the types the routine requires. Since these coercions are in the R code, the type inference strategy for foreign routines does not normally take them into account. However, we can make the strategy accept initial hints about the parameter types, or we can unify these types with the results from independently carrying out type inference on the routine. Either approach takes advantage of both sources of information to refine the overall result. □

The example shows that the context of a call to a foreign routine can provide additional type information about the routine. For some routines, the return type depends on one or more argument types, so the type inference strategy can only infer a specific return type when the argument types are known. Calls to foreign routines are not always preceded by coerctions or type checks. Chapter 2 describes situations where it is or isn't possible to infer types for objects in R code. When the argument types can be inferred, the should be combined with the inferred type signature of the routine in order to solve any dependence the return type has on the argument types.

The most straightforward way to combine information from the type inference strategy for R and the type inference strategy for foreign routines is to run the latter independently for each call site, and then use the resulting information in the former. As described in Section 2.3.1 of the previous chapter, the first step of the type inference strategy for R code is constraint generation. The strategy generates constraints on the types of objects in the code based on how they are defined and used. Calls to R functions cause the strategy to generate constraints

on the types of the arguments and result, based on the type signature of the called function. When provided with type signatures, the strategy can also generate these constraints for calls to foreign routines.

Thus the approach we recommend is to provide handlers for the `.C`, `.Fortran`, `.Call`, `.External`, and `.External2` interfaces in constraint generation step of the type inference strategy for R. Each handler should use the corresponding type inference strategy for foreign routines to compute a type signature for the routine in question. The type inference strategy for R can then generate constraints for the call in the same way as for a call to an R function. The second step of the strategy—constraint solving—remains the same as it was described in Chapter 2.

3.6 Related Work

As of writing, this appears to be the first research into static type inference across R's foreign function interface. Type inference as a subject has been studied extensively, but generally not for code called via foreign function interfaces. Work related to type inference in general is discussed in Section 2.6 of the previous chapter.

Furr and Foster explored type inference across the foreign function interface for both the OCaml and Java programming languages (Furr and Foster 2005; Furr and Foster 2006). For both languages, their primary goal was to detect type errors in the called C code. Moreover, their type systems also model C types, since in the Java foreign function interface, these can sometimes be assigned directly to Java objects. Their system supports inference for user defined routines, including routines which are polymorphic. Their approach to type inference follows a two-stage design similar to what we described for R code in Chapter 2, in contrast to the one-stage code analysis strategy we've adopted for type inference for C code. In short, their system is substantially richer, as necessitated by their type safety goal, and could provide a useful reference for enhancing the system we described here.

Kalibera developed the rchk tool to analyze C code called from R in order to identify memory protection errors (Kalibera 2014). The tool is written in C++, uses LLVM directly to analyze code, and does not contain or interact with R code at all. Because R is a garbage collected language, objects allocated in C code must be protected (by calling the `Rf_protect` routine) in order to prevent accidental garbage collection. When they are no longer in use, the objects must

then be unprotected. The rchk tool programatically detects objects that should be protected but aren't, as well as imbalances in the protection stack. This is the only other tool we are aware of for analysis of C code called from R.

3.7 Conclusion

The type inference strategies proposed in this chapter address all of R's interfaces for calling foreign routines: `.C`, `.Fortran`, `.Call`, `.External`, and `.External2`. All of the strategies rely on the LLVM Compiler Infrastructure and its R binding, the **Rllvm** package, in order to analyze foreign routines. LLVM provides a rich set of tools for extracting information from code.

Routines called via `.C` and `.Fortran` do not operate directly on R objects. The strategy for these routines is to use the tools LLVM provides to collect the type signature. The `.C` and `.Fortran` interfaces do not allow routines to return a result directly, so it is customary to return results by manipulating the arguments to the routine. To determine which arguments are results, the strategy also uses LLVM to check which arguments are modified.

Routines called via `.Call`, `.External`, or `.External2` use the R Internals programming interface and operate directly on R objects. Each R object is represented by a `SEXP`, and as a result, even though C and C++ require explicit type annotations, the annotation for an R object will be `SEXP` regardless of its underlying type. Thus type inference is necessary to determine the types for R objects in these routines. The key idea of the type inference strategy that this chapter proposes for R objects of interest in these routines is to analyze the definition and uses. LLVM is especially important for this strategy, because it provides tools to find the definition and uses of any variable. The proposed type inference strategy is designed to handle R's atomic types, containers with heterogeneous elements, conditionals, loops, attributes, S3 and S4 classes, coercions, and assertions.

Type information collected from foreign routines is meant to be integrated into the type inference strategy for R code. To do this, we suggest using the proposed strategies to infer a type signature for each foreign routine called from R code. The type inference strategy for R code can then use the type signature to generate constraints in the same way it would use a type signature for an R function called in the code.

Bibliography

Adams, Lauren (2018). "Optimized Reservoir Management for Downstream Environmental Purposes." English. PhD thesis, p. 134. ISBN: 978-0-438-93009-4. URL: https://search.proquest.com/docview/2191568383?accountid=14505.

Agesen, Ole (1995). "The Cartesian Product Algorithm. Simple and Precise Type Inference Of Parametric Polymorphism." In: *Proceedings of the 9th European Conference on Object-Oriented Programming*. Ed. by Mario Tokoro and Remo Pareschi. ECOOP '95. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 2–26. ISBN: 978-3-540-49538-3. DOI: 10.1007/3-540-49538-x_2.

Aiken, Alexander, Edward L. Wimmers, and T. K. Lakshman (1994). "Soft Typing with Conditional Types." In: *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '94*. ACM Press. DOI: 10.1145/174675.177847.

Ananian, C. Scott (2001). "The static single information form." OCLC: 48072795. PhD thesis.

Bache, Stefan Milton and Hadley Wickham (2020). *magrittr: A Forward-Pipe Operator for R*. R package version 2.0.1. URL: https://CRAN.R-project.org/package=magrittr.

Bengtsson, Henrik (2018). *globals: Identify Global Objects in R Expressions*. URL: https://CRAN.R-project.org/package=globals.

Chang, Winston (2021). *R6: Encapsulated Classes with Reference Semantics*. R package version 2.4.1. URL: https://CRAN.R-project.org/package=R6.

Cooper, Keith D. and Linda Torczon (2012). *Engineering a compiler*. 2nd ed. Amsterdam ; Boston: Elsevier/Morgan Kaufmann. 800 pp. ISBN: 978-0-12-088478-0.

Csardi, Gabor (2016). *cyclocomp: Cyclomatic Complexity of R Code*. R package version 1.1.0. URL: https://CRAN.R-project.org/package=cyclocomp.

Csárdi, Gábor (2020). *pingr: Check if a Remote Computer is Up*. R package version 2.0.1. URL: https://CRAN.R-project.org/package=pingr.

Damas, Luis and Robin Milner (1982). "Principal type-schemes for functional programs." In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '82*. ACM Press. DOI: 10.1145/582153.582176.

Eddelbuettel, Dirk and Romain François (2011). "Rcpp: Seamless R and C++ Integration." In: *Journal of Statistical Software* 40.8, pp. 1–18. DOI: 10.18637/jss.v040.i08. URL: https://www.jstatsoft.org/v40/i08/.

Flang (Jan. 1, 2021). URL: https://releases.llvm.org/11.0.0/tools/flang/docs/ (visited on 08/01/2021).

Furr, Michael and Jeffrey S. Foster (June 12, 2005). "Checking type safety of foreign function calls." In: *ACM SIGPLAN Notices* 40.6, pp. 62–72. ISSN: 0362-1340. DOI: 10.1145/1064978.1065019. URL: https://doi.org/10.1145/1064978.1065019 (visited on 05/24/2021).

——. (2006). "Polymorphic Type Inference for the JNI." In: *Programming Languages and Systems*. Ed. by Peter Sestoft. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 309–324. ISBN: 978-3-540-33096-7. DOI: 10.1007/11693024_21.

Heeren, Bastiaan, Jurriaan Hage, and Doaitse Swierstra (2002). *Generalizing Hindley-Milner Type Inference Algorithms*. UU-CS-2002-031. Department of Information and Computing Sciences, Utrecht University. URL: http://www.cs.uu.nl/research/techreps/UU-CS-2002-031.html.

Hester, Jim (2017). *lintr: A 'Linter' for R Code*. R package version 1.0.2. URL: https://CRAN.R-project.org/package=lintr.

——. (2018). *covr: Test Coverage for Packages*. R package version 3.2.0. URL: https://CRAN.R-project.org/package=covr.

Johnson, Steven G. and Jelmer Ypma (2020). *The NLopt nonlinear-optimization package*. URL: https://CRAN.R-project.org/package=nloptr.

Kalibera, Tomas (2014). *rchk*. URL: https://github.com/kalibera/rchk.

Lehtosalo, Jukka et al. (Jan. 8, 2015). *python/mypy*. original-date: 2012-12-07T13:30:23Z. URL: https://github.com/python/mypy (visited on 06/30/2021).

LFortran (Jan. 1, 2021). URL: https://lfortran.org/ (visited on 08/01/2021).

Maechler, Martin et al. (2021). *cluster: Cluster Analysis Basics and Extensions*. R package version 2.1.1 — For new features, see the 'Changelog' file (in the package source). URL: https://CRAN.R-project.org/package=cluster.

Mersmann, Olaf (2020). *mco: Multiple Criteria Optimization Algorithms and Related Functions*. R package version 1.15.6. URL: https://CRAN.R-project.org/package=mco.

Milner, Robin (Dec. 1978). "A Theory of Type Polymorphism in Programming." In: *Journal of Computer and System Sciences* 17.3, pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

Morandat, Floréal et al. (June 11, 2012). "Evaluating the Design of the R Language." In: *ECOOP 2012 – Object-Oriented Programming*. European Conference on Object-Oriented Programming. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 104–131. DOI: 10.1007/978-3-642-31057-7_6. URL: https://link.springer.com/chapter/10.1007/978-3-642-31057-7_6 (visited on 06/21/2018).

Müller, Kirill and Hadley Wickham (2019). *tibble: Simple Data Frames*. R package version 2.1.3. URL: https://CRAN.R-project.org/package=tibble.

Nielson, Flemming, Chris Hankin, and Hanne Riis Nielson (2010). *Principles of Program Analysis*. OCLC: 711850939. Berlin: Springer Berlin. ISBN: 978-3-642-08474-4.

Nieweglowski, Lukasz (2020). *clv: Cluster Validation Techniques*. URL: https://CRAN.R-project.org/package=clv.

Quiroga, Jose and Francisco Ortin (Jan. 2017). "SSA Transformations to Facilitate Type Inference in Dynamically Typed Code." In: *The Computer Journal*. DOI: 10.1093/comjnl/bxw108.

R Core Team (2019a). *R Language Definition*. Vienna, Austria: R Foundation for Statistical Computing. URL: https://cran.r-project.org/doc/manuals/r-release/R-lang.html.

——. (2019b). *Writing R Extensions*. Vienna, Austria: R Foundation for Statistical Computing. URL: https://cran.r-project.org/doc/manuals/r-release/R-exts.html.

Robinson, John Alan (1965). "A Machine-Oriented Logic Based on the Resolution Principle." In: *Automation of Reasoning*. Springer Berlin Heidelberg, pp. 397–415. DOI: 10.1007/978-3-642-81952-0_26.

Sarda, Suyog and Mayur Pandey (Dec. 21, 2015). *LLVM Essentials*. Google-Books-ID: ZTblCwAAQBAJ. Packt Publishing Ltd. 166 pp. ISBN: 978-1-78355-862-9.

Schmidt, Drew (n.d.). *float: 32-Bit Floats*. R package version 0.2-5. URL: https://cran.r-project.org/package=float.

Sen, Rathijit et al. (Apr. 10, 2017). "ROSA: R Optimizations with Static Analysis." In: *arXiv:1704.02996 [cs]*. arXiv: 1704.02996. URL: http://arxiv.org/abs/1704.02996 (visited on 06/21/2018).

Siek, Jeremy G. and Walid Taha (2006). "Gradual typing for functional languages." In: *Scheme and Functional Programming Workshop*. Vol. 6, pp. 81–92.

Staiger, Stefan et al. (Oct. 2007). "Interprocedural Static Single Assignment Form." In: IEEE, pp. 1–10. ISBN: 978-0-7695-3034-5. DOI: 10.1109/WCRE.2007.31. URL: http://ieeexplore.ieee.org/document/4400146/ (visited on 06/21/2018).

Temple Lang, Duncan (2011). *RCIndex: R Interface to the clang parser's C API*. R package version 0.3-0. URL: https://github.com/omegahat/RClangSimple.

——. (May 2014). "Enhancing R with Advanced Compilation Tools and Methods." In: *Statistical Science* 29.2, pp. 181–200. ISSN: 0883-4237, 2168-8745. DOI: 10.1214/13-STS462. URL: https://projecteuclid.org/euclid.ss/1408368570 (visited on 06/21/2018).

——. (2021). "A Survey of CRAN Packages."

Temple Lang, Duncan and CRAN Team (Aug. 17, 2021). *RCurl*. R package version 1.98-1.4. URL: https://cran.r-project.org/package=RCurl.

Temple Lang, Duncan, Roger Peng, et al. (2018). *CodeDepends: Analysis of R Code for Reproducible Research and Code Comprehension*. R package version 0.6.5. URL: https://CRAN.R-project.org/package=CodeDepends.

Therneau, Terry and Beth Atkinson (2019). *rpart: Recursive Partitioning and Regression Trees*. R package version 4.1-15. URL: https://CRAN.R-project.org/package=rpart.

Tierney, Luke (2020). *codetools: Code Analysis Tools for R*. URL: https://CRAN.R-project.org/package=codetools.

Turcotte, Alexi and Jan Vitek (July 19, 2019). "Towards a Type System for R." In: *Proceedings of the 14th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS '19. New York, NY, USA: Association for Computing Machinery, pp. 1–5. ISBN: 978-1-4503-6862-9. DOI: 10.1145/3340670.3342426. URL: https://doi.org/10.1145/3340670.3342426 (visited on 06/29/2021).

Vitousek, Michael M. et al. (2014). "Design and Evaluation of Gradual Typing for Python." In: *Proceedings of the 10th ACM Symposium on Dynamic Languages*. DLS '14. New York, NY, USA: ACM, pp. 45–56. ISBN: 978-1-4503-3211-8. DOI: 10.1145/2661088.2661101. URL: http://doi.acm.org/10.1145/2661088.2661101 (visited on 06/21/2018).

Wegman, Mark N. and F. Kenneth Zadeck (Apr. 1991). "Constant Propagation with Conditional Branches." In: *ACM Trans. Program. Lang. Syst.* 13.2, pp. 181–210. ISSN: 0164-0925. DOI:

10.1145/103135.103136. URL: http://doi.acm.org/10.1145/103135.103136 (visited on 09/27/2018).

Wickham, Hadley (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. ISBN: 978-3-319-24277-4. URL: http://ggplot2.org.

Wickham, Hadley, Mara Averick, et al. (2019). "Welcome to the tidyverse." In: *Journal of Open Source Software* 4.43, p. 1686. DOI: 10.21105/joss.01686.

Wickham, Hadley, Peter Danenberg, et al. (2021). *roxygen2: In-Line Documentation for R*. R package version 7.1.2. URL: https://CRAN.R-project.org/package=roxygen2.