2013-02-05
19:46:46

Real World OCaml

Jason Hickey, Anil Madhavapeddy, and Yaron Minsky

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

2013-02-05
19:46:46

Real World OCaml
by Jason Hickey, Anil Madhavapeddy, and Yaron Minsky

Copyright © 2010 O'Reilly Media . All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mike Loukides
Production Editor:
Copyeditor:
Proofreader: FIX ME!
Indexer:
Cover Designer:
Interior Designer: FIX ME!
Illustrator: Robert Romano

March 2013: First Edition.

Revision History for the First Edition:
YYYY-MM-DD First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449323912* for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. !!FILL THIS IN!! and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-32391-2

[?]

1360111605

Table of Contents

Preface xi

Part I. Basic Concepts

1. Prologue 3
Why OCaml? 3
Why Core? 4
About the Authors 4
    Jason Hickey 4
    Anil Madhavapeddy 5
    Yaron Minsky 5

2. A Guided Tour 7
OCaml as a calculator 7
Functions and Type Inference 8
    Type inference 10
    Inferring generic types 11
Tuples, Lists, Options and Pattern-matching 13
    Tuples 13
    Lists 13
    Options 18
Records and Variants 19
Imperative programming 20
    Arrays 21
    Mutable record fields 21
    Refs 22
    For and while loops 23
A complete program 24
    Compiling and running 24
Where to go from here 25

3. Variables and Functions **27**
Variables 27
  Pattern matching and `let` 29
  `let`/`and` bindings 30
Functions 30
  Anonymous Functions 31
  Multi-argument functions 32
  Recursive functions 34
  Prefix and Infix operators 34
  Declaring functions with `function` 37
  Labeled Arguments 38
  Optional arguments 40

4. Lists, Options and Patterns **47**
Lists 47
Pattern matching 48
Options 50
Example: pretty-printing a table 52
  Computing the widths 52
  Rendering the rows 53
List performance 55
  Tail-recursion 55
  Hybrid recursion 56
Heterogenous values 57
Options 59

5. Records **63**
Patterns and exhaustiveness 64
Field punning 65
Reusing field names 66
Functional updates 69
Mutable fields 70
First-class fields 71

6. Variants **73**
Combining records and variants 76
Variants and recursive data structures 79
Polymorphic variants 82
  Example: Terminal colors redux 83
  When to use polymorphic variants 88

2013-02-05
19:46:46

7. Error Handling **89**
- Error-aware return types 89
  - Encoding errors with `Result` 90
  - `Error` and `Or_error` 91
  - `bind` and other error-handling idioms 92
- Exceptions 93
  - Exception handlers 95
  - Cleaning up in the presence of exceptions 96
  - Catching specific exceptions 96
  - Backtraces 97
  - Exceptions for control flow 98
  - From exceptions to error-aware types and back again 98

8. Functors **101**
- A trivial example 101
- A bigger example: computing with intervals 103
  - Making the functor abstract 105
  - Sharing constraints 106
  - Destructive substitution 108
  - Using multiple interfaces 109
- Extending modules 111

9. Files, Modules and Programs **115**
- Single File Programs 115
- Multi-file programs and modules 118
  - Signatures and Abstract Types 119
- More on modules and signatures 121
  - Concrete types in signatures 121
  - The `include` directive 122
  - Modules within a file 123
  - Opening modules 124
  - Common errors with modules 125

10. Imperative Programming **129**
- A simple dictionary 129
- Mutable data structures 132
  - Array-like data 132
  - Mutable record and object fields and ref cells 133
  - Foreign functions 134
- `for` and `while` loops 134
- Doubly-linked lists 135
  - Iteration 138
  - Java-style iteration 139

    Hash tables with iterators 141
  Lazy computation 144
    Memoization 145
    Hash consing 146
    Weak hash consing 148
  Concurrency 149
    Dealing with concurrency 151
    Concurrent hash tables 151
  Input and output (fix this chapter heading) 155
    Basic file input and output 158
    Formatted output with Printf 159
    Strong typing and format strings 160
    Output to channels, strings, and buffers 161
    Formatted input with Scanf 162
  Summary 162

11. Object Oriented Programming 165
  When to use objects 166
  OCaml objects 166
  Object Polymorphism 167
  Classes 169
  An Example: Cryptokit 170
  Class parameters and polymorphism 173
  Object types 175
  Immutable objects 177
  Class types 178

Part II. Practical Examples

12. Data Serialization with S-Expressions 183
  Sexplib 185
    Formatting of s-expressions 185
    Sexp converters 186
    Getting good error messages 188
    Sexp-conversion directives 189
  Bin_prot 191
  Fieldslib 192

13. Handling JSON data 195
  Parsing standard JSON with Yojson 196
    Selecting values from JSON structures 198
  Constructing JSON values 199

  Using non-standard JSON extensions 201
Automatically mapping JSON to OCaml types 202

14. XML Streams and Trees **205**
Stream parsing XML 206
Tree parsing XML 207
Constructing XML documents using syntax extensions 209
Working with XHTML 213

15. Input and Output **215**

16. Concurrent Programming with Async **217**
Example: searching definitions with DuckDuckGo 218
  URI handling 218
  Parsing JSON strings 219
  Executing an HTTP client query 219
Manipulating Async threads 221
Timing and Thread Composition 223
  Cancellation 224
A simple TCP Echo Server 224
Onto an HTTP Server 224
Binding to the Github API 224

Part III. Advanced Topics

17. First class modules **229**
Another trivial example 230
Standard vs. first-class modules 231
A more complete example -- containers 233
  Imperative containers 234
  Pure functional containers 235
Dynamically choosing a module 237
Example: A service bundle 238

18. Understanding the runtime system **243**
The garbage collector 243
  The fast minor heap 244
  The long-lived major heap 245
The representation of values 245
  Blocks and values 246
  Integers, characters and other basic types 247
  Tuples, records and arrays 247

Floating point numbers and arrays 248
Variants and lists 249
Polymorphic variants 250
String values 251
Custom heap blocks 252
Interfacing with C 252
Getting started with a "Hello World" C binding 252

19. Object Subtyping and Inheritance 255
Subtyping 255
Using more precise types to address subtyping problems 257
Using elided types to address subtyping problems 258
Narrowing 258
Binary methods 260
Private methods 262
Virtual classes and methods 264
Multiple inheritance 266
How names are resolved 266
Mixins 268

20. Managing external memory with Bigarrays 271
Bigarrays for external memory blocks 271

21. Inside the Runtime 273
Runtime Memory Management 273
Allocating on the minor heap 274
Allocating on the major heap 274
The major heap free list 275
Memory allocation strategies 275
Inter-generational pointers 277
How garbage collection works 279
Collecting the minor heap 279
Collecting the major heap 280
Marking the major heap 281
Sweeping unused blocks from the major heap 281
Compaction and defragmenting the major heap 281

22. Performance Tuning and Profiling 283
Byte code Profiling 283
Native Code Profiling 283
gdb 283
perf 283
dtrace 283

23. Packaging and Build Systems **285**
The OCaml toolchain 285
  The `ocamlc` bytecode compiler 285
  The `ocamlopt` native code compiler 286
  The `ocaml` top-level loop 286
The Findlib compiler frontend 286
Packaging applications with OASIS 286
  ocamlbuild 286
Distributing applications with OPAM 286

24. Parsing with OCamllex and OCamlyacc **287**

Appendix: Installation **289**

2013-02-05
19:46:46

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business. Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/<catalog page>

Don't forget to update the <url> attribute, too.

2013-02-05
19:46:46

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

2013-02-05
19:46:46

PART I

Basic Concepts

2013-02-05
19:46:46

CHAPTER 1

Prologue

(yminsky: this is something of a placeholder. We need a real introduction that should talk, amongst other things, about what kinds of applications OCaml is good for and why one should want to learn it. Also, some coverage of who uses OCaml successfully now.)

Why OCaml?

Programming languages matter.

The programming languages that you use affect your productivity. They affect how reliable your software is, how efficient it is, how easy it is to read, to refactor, and to extend. And the programming languages you know and use can deeply affect how you think about programming and software design.

But not all ideas about how to design a programming language are created equal. Over the last 40 years, a few key language features have emerged that together form a kind of sweet-spot in language design. These features include:

- Garbage collection
- First-class and higher-order functions
- Static type-checking
- Parametric polymorphism
- Support for programming with immutable values
- Algebraic datatypes and pattern-matching
- Type inference

Some of these features you already know and love, and some are probably new to you. But as we hope to demonstrate over the course of this book, it turns out that there is something transformative about having them all together and able to interact with each other in a single language.

Despite their importance, these ideas have made only limited inroads into mainstream languages. And when they do arrive there, like higher-order functions in C# or parametric polymorphism in Java, it's typically in a limited and awkward form. The only languages that support these ideas well are statically-typed functional programming languages like OCaml, F#, Haskell, Scala and Standard-ML.

Among this worthy set of languages, OCaml stands apart because it manages to provide a great deal of power while remaining highly pragmatic, highly performant, and comparatively simple to use and understand. It is this that makes OCaml a great choice for programmers who want to step up to a better programming language, and at the same time want to get practical work done.

Why Core?

A language on its own isn't enough. You also need a rich set of libraries to base your applications on. A common source of frustration for those learning OCaml is that the standard library that ships with the OCaml compiler is not ideal. While it's well implemented, it covers only a small subset of the functionality you expect from a standard library, and the interfaces are idiosyncratic and inconsistent.

But all is not lost! There is an effective alternative to the OCaml standard library called Core. Jane Street, a company that has been using OCaml for nearly a decade, developed Core for its own internal use, but it was designed from the start with an eye towards being a general-purpose standard library. Core is also distributed with syntax-extensions which provide essential new functionality to OCaml; and there are additional libraries, like Core_extended and Async, that provide even more useful functionality.

We believe that Core makes OCaml a better tool, and that's why we'll present OCaml and Core together.

About the Authors

Jason Hickey

Jason Hickey is a Software Engineer at Google Inc. in Mountain View, California. He is part of the team that designs and develops the global computing infrastructure used to support Google services, including the software systems for managing and scheduling massively distributed computing resources.

Prior to joining Google, Jason was an Assistant Professor of Computer Science at Caltech, where his research was in reliable and fault-tolerant computing systems, including programming language design, formal methods, compilers, and new models of distributed computation. He obtained his PhD in Computer Science from Cornell University, where he studied programming languages. He is the author of the MetaPRL system, a logical framework for design and analysis of large software systems; OMake, an advanced build system for large software projects. He is the author of the textbook, *An Introduction to Objective Caml* (unpublished).

Anil Madhavapeddy

Anil Madhavapeddy is a Senior Research Fellow at the University of Cambridge, based in the Systems Research Group. He was on the original team that developed the Xen hypervisor, and helped develop an industry-leading cloud management toolstack written entirely in OCaml. This XenServer product has been deployed on hundreds of thousands of physical hosts, and drives critical infrastructure for many Fortune 500 companies.

Prior to obtaining his PhD in 2006 from the University of Cambridge, Anil had a diverse background in industry at Network Appliance, NASA and Internet Vision. In addition to professional and academic activities, he is an active member of the open-source development community with the OpenBSD operating system, is co-chair of the Commercial Uses of Functional Programming workshop, and serves on the boards of startup companies such as Ashima Arts where OCaml is extensively used.

Yaron Minsky

Yaron Minsky heads the Technology group at Jane Street, a proprietary trading firm that is the largest industrial user of OCaml. He was responsible for introducing OCaml to the company and for managing the company's transition to using OCaml for all of its core infrastructure. Today, billions of dollars worth of securities transactions flow each day through those systems.

Yaron obtained his PhD in Computer Science from Cornell University, where he studied distributed systems. Yaron has lectured, blogged and written about OCaml for years, with articles published in Communications of the ACM and the Journal of Functional Programming. He chairs the steering committee of the Commercial Users of Functional Programming, and is a member of the steering committee for the International Conference on Functional Programming.

2013-02-05
19:46:46

CHAPTER 2

A Guided Tour

This chapter gives an overview of OCaml by walking through a series of small examples that cover most of the major features of the language. This should give a sense of what OCaml can do, without getting too deep in any one topic.

We'll present this guided tour using the `utop` OCaml toplevel, a shell that lets you type in expressions and evaluate them interactively. When you get to the point of running real programs, you'll want to leave the toplevel behind, but it's a great tool for getting to know the language.

Make sure you have a working toplevel as you go through this chapter, so you can try out the examples as you go.

Installing the interactive top-level

The easiest way to get `utop` is via the OPAM package manager, which is explained in Appendix. You'll need to also have the Core standard library installed. In a nutshell, you need to:

```
$ opam init
$ opam switch 4.00.1+short-types
$ opam install utop async core_extended
$ eval `opam config -env`
$ utop
```

You can exit `utop` by pressing `control-D` and return.

OCaml as a calculator

Let's spin up the toplevel. Throughout the book we're going to use Core, a more full-featured and capable replacement for OCaml's standard library. Accordingly, we'll start by opening the `Core.Std` module to get access to Core's libraries. If you don't open `Core.Std` many of the examples below will fail.

```
$ utop
# open Core.Std;;
```

Now that we have Core open, let's try a few simple numerical calculations.

```
# 3 + 4;;
- : int = 7
# 8 / 3;;
- : int = 2
# 3.5 +. 6.;;
- : float = 9.5
# sqrt 9.;;
- : float = 3.
```

By and large, this is pretty similar to what you'd find in any programming language, but there are a few things that jump right out at you.

- We needed to type `;;` in order to tell the toplevel that it should evaluate an expression. This is a peculiarity of the toplevel that is not required in stand-alone programs (though it is sometimes helpful to include `;;` to improve OCaml's error reporting.)
- After evaluating an expression, the toplevel spits out both the type of the result and the result itself.
- Function arguments are separated by spaces, instead of by parenthesis and commas, which is more like the UNIX shell than C or Java.
- OCaml carefully distinguishes between `float`, the type for floating point numbers and `int`. The types have different literals (`6.` instead of `6`) and different infix operators (`+.` instead of `+`), and OCaml doesn't do any automated casting between the types. This can be a bit of a nuisance, but it has its benefits, since it prevents some kinds of bugs that arise in other languages due to unexpected differences between the behavior of `int` and `float`.

We can also a create variable to name the value of a given expression, using a `let` binding.

```
# let x = 3 + 4;;
val x : int = 7
# let y = x + x;;
val y : int = 14
```

After a new variable is created, the toplevel tells us the name of the variable, in addition to its type and value.

Functions and Type Inference

The `let` syntax can also be used for creating functions:

```
# let square x = x * x ;;
val square : int -> int = <fun>
# square (square 2);;
- : int = 16
```

When using `let` to define a function, the first identifier after the `let` is the function name, and each subsequent identifier is a different argument to the function. Thus, `square` is a function with a single argument.

Now that we're creating more interesting values like functions, the types have gotten more interesting too. `int -> int` is a function type, in this case indicating a function that takes an `int` and returns an `int`. We can also write functions that take multiple arguments.

```
# let ratio x y =
     Float.of_int x /. Float.of_int y
  ;;
val ratio : int -> int -> float = <fun>
# ratio 4 7;;
- : float = 0.571428571428571397
```

The notation for the type-signature of a multi-argument functions may be a little surprising at first, but we'll explain where it comes from when we get to function currying in Chapter 3. For the moment, think of the arrows as separating different arguments of the function, with the type after the final arrow being the return value of the function. Thus,

```
int -> int -> float
```

describes a function that takes two `int` arguments and returns a `float`.

We can even write functions that take other functions as arguments. Here's an example of a function that takes three arguments: a test function and two integer arguments. The function returns the sum of the integers that pass the test.

```
# let sum_if_true test x y =
     if test x then x else 0
     + if test y then y else 0
  ;;
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>
```

If we look at the inferred type signature in detail, we see that the first argument is a function that takes an int and returns a boolean, and that the remaining two arguments are integers. Here's an example of this function in action.

```
# let even x = x mod 2 = 0 ;;
val even : int -> bool = <fun>
# sum_if_true even 3 4;;
- : int = 4
```

```
# sum_if_true even 2 4;;
- : int = 6
```

Type inference

As the types we encounter get more complicated, you might ask yourself how OCaml is able to figure them out, given that we didn't write down any explicit type information.

OCaml determines the type of an expression using a technique called *type inference*, by which it infers the type of a given expression based on what it already knows about the types of other related variables, and on constraints on the types that arise from the structure of the code.

As an example, let's walk through the process of inferring the type of `sum_if_true`.

- OCaml requires that both arms of an `if` statement return the same type, so the expression `if test x then x else 0` requires that `x` must be the same type as `0`, which is `int`. By the same logic we can conclude that `y` has type `int`.
- `test` is passed `x` as an argument. Since `x` has type `int`, the input type of `test` must be `int`.
- `test x` is used as the condition in an `if` statement, so the return type of `test` must be `bool`.
- The fact that `+` returns an int implies that the return value of `sum_if_true` must be int.

Together, that nails down the the types of all the variables, which determines the overall type of `sum_if_true`.

Over time, you'll build a rough intuition for how the OCaml inference engine works, which makes it easier to reason through your programs. One way of making it easier to understand the types is to add explicit type annotations. These annotations never change the behavior of an OCaml program, but they can serve as useful documentation, as well as catch unintended type changes. Here's an annotated version of `sum_if_true`:

```
# let sum_if_true (test : int -> bool) (x:int) (y:int) : int =
     if test x then x else 0
     + if test y then y else 0
  ;;
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>
```

In the above, we've marked every argument to the function with its type, with the final annotation indicating the type of the return value. Such type annotations can actually go around any value in an OCaml program, and can be useful for figuring out why a given program is failing to compile.

Inferring generic types

Sometimes, there isn't enough information to fully determine the concrete type of a given value. Consider this function.

```
# let first_if_true test x y =
    if test x then x else y
  ;;
```

`first_if_true` takes as its arguments a function `test`, and two values, `x` and `y`, where `x` is to be returned if `test x` evaluates to `true`, and `y` otherwise. So what's the type of `first_if_true`? There are no obvious clues such as arithmetic operators or literals to tell you what the type of `x` and `y` are. That makes it seem like one could use this `first_if_true` on values of any type. Indeed, if we look at the type returned by the toplevel:

```
val first_if_true : ('a -> bool) -> 'a -> 'a -> 'a = <fun>
```

we see that rather than choose a single concrete type, OCaml has introduced a *type variable* `'a` to express that the type is generic. In particular, the type of the `test` argument is `('a -> bool)`, which means that test is a one-argument function whose return value is `bool`, and whose argument could be of any type `'a`. But, whatever type `'a` is, it has to be the same as the type of the other two arguments, `x` and `y`.

This genericity means that we can write:

```
# let long_string s = String.length s > 6;;
val long_string : string -> bool = <fun>
# first_if_true long_string "short" "loooooong";;
- : string = "loooooong"
```

And we can also write:

```
# let big_number x = x > 3;;
val big_number : int -> bool = <fun>
# first_if_true big_number 4 3;;
- : int = 4
```

Both `long_string` and `big_number` are functions, and each is passed to `first_if_true` with two other arguments of the appropriate type (strings in the first example, and integers in the second). But we can't mix and match two different concrete types for `'a` in the same use of `first_if_true`.

```
# first_if_true big_number "short" "loooooong";;
Characters 25-30:
  first_if_true big_number "short" "loooooong";;
                           ^^^^^^^
Error: This expression has type string but
    an expression was expected of type int
```

In this example, `big_number` requires that `'a` be of type `int`, whereas `"short"` and `"loooooong"` require that `'a` be of type `string`, and they can't all be right at the same time. This kind of genericity is called *parametric polymorphism*, and is very similar to generics in C# and Java.

Type errors vs exceptions

There's a big difference in OCaml (and really in any compiled language) between errors that are caught at compile time and those that are caught at run-time. It's better to catch errors as early as possible in the development process, and compilation time is best of all.

Working in the top-level somewhat obscures the difference between run-time and compile time errors, but that difference is still there. Generally, type errors, like this one:

```
# 3 + "potato";;
Characters 4-12:
  3 + "potato";;
      ^^^^^^^^
Error: This expression has type string but an expression was expected of type
         int
```

are compile-time errors, whereas an error that can't be caught by the type system, like division by zero, leads to a runtime exception.

```
# 3 / 0;;
Exception: Division_by_zero.
```

One important distinction is that type errors will stop you whether or not the offending code is ever actually executed. Thus, you get an error from typing in this code:

```
# if 3 < 4 then 0 else 3 + "potato";;
Characters 25-33:
  if 3 < 4 then 0 else 3 + "potato";;
                           ^^^^^^^^
Error: This expression has type string but an expression was expected of type
         int
```

but the following code runs incident, even though it contains a branch that would throw an exception if it were ever reached.

```
# if 3 < 4 then 0 else 3 / 0;;
- : int = 0
```

Tuples, Lists, Options and Pattern-matching

Tuples

So far we've encountered a handful of basic types like `int`, `float` and `string` as well as function types like `string -> int`. But we haven't yet talked about any data structures. We'll start by looking at a particularly simple data structure, the tuple. You can create a tuple by joining values together with a comma:

```
# let tup = (3,"three");;
val tup : int * string = (3, "three")
```

For the mathematically inclined, the `*` character is used because the set of all pairs of type `t * s` corresponds to the Cartesian product of the set of elements of type `t` and the set of elements of type `s`.

You can extract the components of a tuple using OCaml's pattern-matching syntax. For example:

```
# let (x,y) = tup;;
val x : int = 3
val y : string = "three"
```

Here, the `(x,y)` on the left-hand side of the `let` binding is the pattern. This pattern lets us mint the new variables `x` and `y`, each bound to different components of the value being matched. Note that the same syntax is used both for constructing and for pattern-matching on tuples.

Pattern matching can also show up in function arguments. Here's a function for computing the distance between two points on the plane, where each point is represented as a pair of `float`s. The pattern matching syntax lets us get at the values we need with a minimum of fuss.

```
# let distance (x1,y1) (x2,y2) =
    sqrt ((x1 -. x2) ** 2. +. (y1 -. y2) ** 2.)
;;
```

This is just a first taste of pattern matching. Pattern matching is a pervasive tool in OCaml, and as you'll see, it has surprising power.

Lists

Where tuples let you combine a fixed number of items, potentially of different types, lists let you hold any number of items of the same type. For example:

```
# let languages = ["OCaml";"Perl";"C"];;
val languages : string list = ["OCaml"; "Perl"; "C"]
```

Note that you can't mix elements of different types on the same list, as we did with tuples.

```
# let numbers = [3;"four";5];;
Characters 17-23:
  let numbers = [3;"four";5];;
                   ^^^^^^
Error: This expression has type string but an expression was expected of type
         int
```

The `List` module

OCaml comes with a `List` module that has a rich collection of functions for working with lists. We can access values from within a module by using dot-notation. Here, for example, is how we compute the length of a list.

```
# List.length languages;;
- : int = 3
```

Here's something a little more complicated. We can compute the list of the lengths of each language as follows.

```
# List.map languages ~f:String.length;;
- : int list = [5; 4; 1]
```

`List.map` takes two arguments: a list, and a function for transforming the elements of that list. Note that `List.map` creates a new list and does not modify the original.

In this example, the function `String.length` is passed using under the *labeled argument* `~f`. Labels allow you to specify function arguments by name rather than by position. As you can see below, we can change the order of labeled arguments without changing the function's behavior.

```
# List.map ~f:String.length languages;;
- : int list = [5; 4; 1]
```

We'll learn more about labeled arguments and why they're important in Chapter 3.

Constructing lists with `::`

In addition to constructing lists using brackets, we can use the operator `::` for adding elements to the front of a list.

```
# "French" :: "Spanish" :: languages;;
- : string list = ["French"; "Spanish"; "OCaml"; "Perl"; "C"]
```

Here, we're creating a new extended list, not changing the list we started with, as you can see below.

```
# languages;;
- : string list = ["OCaml"; "Perl"; "C"]
```

The bracket notation for lists is really just syntactic sugar for `::`. Thus, the following declarations are all equivalent. Note that `[]` is used to represent the empty list.

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
# 1 :: (2 :: (3 :: []));;
- : int list = [1; 2; 3]
# 1 :: 2 :: 3 :: [];;
- : int list = [1; 2; 3]
```

List patterns using `match`

The elements of a list can be accessed through pattern-matching. List patterns are based on the two list constructors, `[]` and `::`. Here's a simple example.

```
# let my_favorite_language (my_favorite :: the_rest) =
    my_favorite
 ;;
```

By pattern matching using `::`, we've broken off the first element of `languages` from the rest of the list. If you know Lisp or Scheme, what we've done is the equivalent of using `car` to grab the first element of a list.

If you try the above example in the toplevel, however, you'll see that it spits out an error:

```
  Characters 25-69:
 ........................(my_favorite :: the_rest) =
     my_favorite
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val my_favorite_language : 'a list -> 'a = <fun>
```

The warning comes because the compiler can't be certain that the pattern match won't lead to a runtime error. Indeed, the warning gives an example of a pattern that won't match, the empty list, `[]`. We can see this in action below.

```
# my_favorite_language ["English";"Spanish";"French"];;
- : string = "English"
# my_favorite_language [];;
Exception: Match_failure ("//toplevel//", 11, 10).
```

You can avoid these warnings, and more importantly make sure that your code actually handles all of the possible cases, by using a `match` statement instead.

A `match` statement is a kind of juiced-up version of the switch statement found in `C` and `Java`. It essentially lets you list a sequence of patterns (separated by | characters --- the one before the first case is optional), and the compiler then dispatches to the code following the first matched pattern. And, as we've already seen, we can name new variables in our patterns that correspond to sub-structures of the value being matched.

Here's a new version of `my_favorite_language` that uses `match`, and doesn't trigger a compiler warning.

```
# let my_favorite_language languages =
    match languages with
    | first :: the_rest -> first
    | [] -> "OCaml" (* A good default! *)
 ;;
val my_favorite_language : string list -> string = <fun>
# my_favorite_language ["English";"Spanish";"French"];;
- : string = "English"
# my_favorite_language [];;
- : string = "OCaml"
```

Note that we included a comment in the above code. OCaml comments are bounded by `(*` and `*)`, and can be nested arbitrarily and cover multiple lines.

The first pattern, `first :: the_rest`, covers the case where `languages` has at least one element, since every list except for the empty list can be written down with one or more `::`'s. The second pattern, `[]`, matches only the empty list. These cases are exhaustive (every list is either empty, or has at least one element), and the compiler can detect that exhaustiveness, which is why it doesn't spit out a warning.

Recursive list functions

Recursive functions, or, functions that call themselves, are an important technique in OCaml and in any functional language. The typical approach to designing a recursive function is to separate the logic into a set of *base cases*, that can be solved directly, and a set of *inductive cases*, where the function breaks the problem down into smaller pieces and then calls itself to solve those smaller problems.

When writing recursive list functions, this separation between the base cases and the inductive cases is often done using pattern matching. Here's a simple example of a function that sums the elements of a list.

```
# let rec sum l =
    match l with
    | [] -> 0
    | hd :: tl -> hd + sum tl
  ;;
val sum : int list -> int
# sum [1;2;3;4;5];;
- : int = 15
```

Note that we had to use the `rec` keyword to allow `sum` to refer to itself. And, as you might imagine, the base case and inductive case are different arms of the match. In particular, the base case is that of the empty list, and the inductive case is that of a list of zero or more elements.

We can introduce more complicated list patterns as well. Here's a function for destuttering a list, *i.e.*, for removing sequential duplicates.

```
# let rec destutter list =
    match list with
    | [] -> []
    | hd1 :: hd2 :: tl ->
      if hd1 = hd2 then destutter (hd2 :: tl)
      else hd1 :: destutter (hd2 :: tl)
  ;;
```

Again, the first arm of the match is the base case, and the second is the inductive. Unfortunately, this code has a problem. If you type it into the top-level, you'll see this error:

```
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
_::[]
```

This indicates that we're missing a case, in particular we don't handle one-element lists. That's easy enough to fix by adding another case to the match:

```
# let rec destutter list =
    match list with
    | [] -> []
    | [hd] -> [hd]
    | hd1 :: (hd2 :: tl) ->
      if hd1 = hd2 then destutter (hd2 :: tl)
      else hd1 :: destutter (hd2 :: tl)
  ;;
val destutter : 'a list -> 'a list = <fun>
# destutter ["hey";"hey";"hey";"man!"];;
- : string list = ["hey"; "man!"]
```

Note that this code used another variant of the list pattern, `[hd]`, to match a list with a single element. We can do this to match a list with any fixed number of elements, *e.g.*, `[x;y;z]` will match any list with exactly three elements, and will bind those elements to the variables `x`, `y` and `z`.

In the last few examples, our list processing code involved a lot of recursive functions. In practice, this isn't usually necessary. Most of the time, you'll find yourself happy to use the iteration functions found in the `List` module. But it's good to know how to use recursion when you need to do something new that's not already supported.

Options

Another common data structure in OCaml is the option. An option is used to express that a value might or might not be present. For example,

```
# let divide x y =
    if y = 0 then None else Some (x/y) ;;
val divide : int -> int -> int option = <fun>
```

The function `divide` either returns `None`, if the divisor is zero, or `Some` of the result of the division, otherwise. `Some` and `None` are constructors, like `::` and `[]` for lists, which let you build optional values. You can think of an option as a specialized list that can only have zero or one element.

To get a value out of an option, we use pattern matching, as we did with tuples and lists. Consider the following simple function for printing a log entry given an optional time and a message. If no time is provided (*i.e.*, if the time is `None`), the current time is computed and used in its place.

```
# let print_log_entry maybe_time message =
    let time =
      match maybe_time with
      | Some x -> x
      | None -> Time.now ()
    in
    printf "%s: %s\n" (Time.to_string time) message ;;
val print_log_entry : Time.t option -> string -> unit
```

Here, we again use a `match` statement for handling the two possible states of an option. It's worth noting that we don't necessarily need to use an explicit `match` statement in this case. We can instead use some built in functions from the `Option` module, which, like the `List` module for lists, is a place where you can find a large collection of useful functions for working with options.

In this case, we can rewrite `print_log_entry` using `Option.value`, which returns the content of an option if the option is `Some`, or a default value if the option is `None`.

```
# let print_log_entry maybe_time message =
    let time = Option.value ~default:(Time.now ()) maybe_time in
    printf "%s: %s\n" (Time.to_string time) message ;;
```

Options are important because they are the standard way in OCaml to encode a value that might not be there. This is different from most other languages, including Java and C#, where most if not all datatypes are *nullable*, meaning that, whatever their type is, any given value also contains the possibility of being a null value. In such languages, null is lurking everywhere.

In OCaml, however, nulls are explicit. A value of type `string * string` always actually contains two two well-defined values of type `string`. If you want to allow, say, the first of those, to possibly be absent, then you need to change the type to something like `string option * string`. As we'll see, this explicitness allows the compiler to provide a great deal of help in making sure you're correctly handing the possibility of missing data.

Records and Variants

So far, we've looked only at data structures that were predefined in the language, like lists and tuples. But OCaml also allows us to define new datatypes. Here's a toy example of a datatype representing a point in 2-dimensional space:

```
# type point2d = { x : float; y : float };;
type point2d = { x : float; y : float; }
```

`point2d` is a *record* type, which you can think of as a tuple where the individual fields are named, rather than being defined positionally. Record types are easy enough to construct:

```
# let p = { x = 3.; y = -4. };;
val p : point2d = {x = 3.; y = -4.}
```

And we can get access to the contents of these types using pattern matching:

```
# let magnitude { x = x_pos; y = y_pos } =
    sqrt (x_pos ** 2. +. y_pos ** 2.);;
val magnitude : point2d -> float = <fun>
```

We can write the pattern match even more tersely, using what's called *field punning*. In particular, when the name of the field and the name of the variable coincide, we don't have to write them both down. Thus, the magnitude function can be rewritten as follows.

```
# let magnitude { x; y } = sqrt (x ** 2. +. y ** 2.);;
```

We can also use dot-notation for accessing record fields:

```
# let distance v1 v2 =
     magnitude { x = v1.x -. v2.x; y = v1.y -. v2.y };;
val distance : point2d -> point2d -> float = <fun>
```

And we can of course include our newly defined types as components in larger types, as in the following types, each of which is a description of a different geometric object.

```
# type circle_desc  = { center: point2d; radius: float }
  type rect_desc    = { lower_left: point2d; width: float; height: float }
  type segment_desc = { endpoint1: point2d; endpoint2: point2d } ;;
```

Now, imagine that you want to combine multiple objects of these types together as a description of a multi-object scene. You need some unified way of representing these objects together in a single type. One way of doing this is using a *variant* type:

```
# type scene_element =
    | Circle  of circle_desc
    | Rect    of rect_desc
    | Segment of segment_desc
  ;;
```

The | character separates the different cases of the variant (the first | is optional), and each case has a tag, like `Circle`, `Rect` and `Scene`, to distinguish that case from the others. Here's how we might write a function for testing whether a point is in the interior of some element of a list of `scene_element`s.

```
# let is_inside_scene_element point scene_element =
     match scene_element with
     | Circle { center; radius } ->
       distance center point < radius
     | Rect { lower_left; width; height } ->
       point.x > lower_left.x && point.x < lower_left.x +. width
       && point.y > lower_left.y && point.y < lower_left.y +. height
     | Segment { endpoint1; endpoint2 } -> false
  ;;
val is_inside_scene_element : point2d -> scene_element -> bool = <fun>
# let is_inside_scene point scene =
     List.for_all scene
       ~f:(fun el -> is_inside_scene_element point el)
   ;;
val is_inside_scene : point2d -> scene_element list -> bool = <fun>
```

You might at this point notice that the use of `match` here is reminiscent of how we used `match` with `option` and `list`. This is no accident: `option` and `list` are really just examples of variant types that happen to be important enough to be defined in the standard library (and in the case of lists, to have some special syntax).

We also made our first use of an *anonymous function* in the call to `List.for_all`. An anonymous function is a function that is defined but not named, in this case, using the `fun` keyword. Anonymous functions are common in OCaml, particularly when using iteration functions like like `List.for_all`.

Imperative programming

So far, we've only written so-called *pure* or *functional* code, meaning that we didn't write any code that modified a variable or value after its creation. This is a quite different style from *imperative* programming, where computations are structured as sequences of instructions that operate by modifying state as they go.

Functional code is the default in OCaml, with variable bindings and most data structures being immutable. But OCaml also has excellent support for imperative programming, including mutable data structures like arrays and hashtables and control-flow constructs like for and while loops.

Arrays

Perhaps the simplest mutable data structure in OCaml is the array. Arrays in OCaml are very similar to arrays in other languages like C: they are fixed width, indexing starts at 0, and accessing or modifying an array element is a constant-time operation. Arrays are more compact in terms of memory utilization than most other data structures in OCaml, including lists. OCaml uses three words per element of a list, but only one per element of an array.

Here's an example.

```
# let numbers = [| 1;2;3;4 |];;
val numbers : int array = [|1; 2; 3; 4|]
# numbers.(2) <- 4;;
- : unit = ()
# numbers;;
- : int array = [|1; 2; 4; 4|]
```

the `.(i)` syntax is used to refer to an element of an array, and the `<-` syntax is for modification. Because the elements of the array are counted starting at zero, element `.(2)` is the third element.

Mutable record fields

The array is an important mutable data structure, but it's not the only one. Records, which are immutable by default, can be declared with specific fields as being mutable. Here's a small example of a data structure for storing a running statistical summary of a collection of numbers. Here's the basic data structure:

```
# type running_sum =
   { mutable sum: float;
     mutable sum_sq: float; (* sum of squares *)
     mutable samples: int;
   }
  ;;
```

The fields in `running_sum` are designed to be easy to extend incrementally, and sufficient to compute means and standard deviations, as shown below.

```
# let mean rsum = rsum.sum /. float rsum.samples
  let stdev rsum =
     sqrt (rsum.sum_sq /. float rsum.samples
           -. (rsum.sum /. float rsum.samples) ** 2.) ;;
```

```
val mean : running_sum -> float = <fun>
val stdev : running_sum -> float = <fun>
```

We also need functions to create and update `running_sum`s:

```
# let create () = { sum = 0.; sum_sq = 0.; samples = 0 }
  let update rsum x =
     rsum.samples <- rsum.samples + 1;
     rsum.sum     <- rsum.sum     +. x;
     rsum.sum_sq  <- rsum.sum_sq  +. x *. x
  ;;
val create : unit -> running_sum = <fun>
val update : running_sum -> float -> unit = <fun>
```

`create` returns a `running_sum` corresponding to the empty set, and `update rsum x` changes `rsum` to reflect the addition of `x` to its set of samples, by updating the number of samples, the sum, and the sum of squares.

Note the use in the above code of single semi-colons to sequence operations. When we were working purely functionally, this wasn't necessary, but you start needing it when your code is acting by side-effect.

A new and somewhat odd type has cropped up in this example: `unit`. What makes `unit` different is that there is only one value of type `unit`, which is written `()`. Because there is only one value of type `unit` that value doesn't really convey any information.

If it doesn't convey any information, then what is `unit` good for? Most of the time, `unit` acts as a placeholder. Thus, we use `unit` for the return value of a function like `update` that operates by side effect rather than by returning a value, and for the argument to a function like `create` that doesn't require any information to be passed into it in order to run.

Here's an example of `create` and `update` in action.

```
# let rsum = create ();;
val rsum : running_sum = {sum = 0.; sum_sq = 0.; samples = 0}
# List.iter [1.;3.;2.;-7.;4.;5.] ~f:(fun x -> update rsum x);;
- : unit = ()
# mean rsum;;
- : float = 1.33333333333333326
# stdev rsum;;
- : float = 3.94405318873307698
```

Refs

We can declare a single mutable value by using a `ref`, which is a record type with a single mutable field that is defined in the standard library.

```
# let x = { contents = 0 };;
val x : int ref = {contents = 0}
# x.contents <- x.contents + 1;;
```

```
- : unit = ()
# x;;
- : int ref = {contents = 1}
```

There are a handful of useful functions and operators defined for refs to make them more convenient to work with.

```
# let x = ref 0 ;; (* create a ref, i.e., { contents = 0 } *)
val x : int ref = {contents = 0}
# !x ;;            (* get the contents of a ref, i.e., x.contents *)
- : int = 0
# x := !x + 1 ;;   (* assignment, i.e., x.contents <- ... *)
- : unit = ()
# incr x ;;        (* increment, i.e., x := !x + 1 *)
- : unit = ()
# !x ;;
- : int = 2
```

A ref is really just an example of a mutable record, but in practice, it's the standard way of dealing with a single mutable value in a computation.

For and while loops

Along with mutable data structures, OCaml gives you constructs like `while` and `for` loops for interacting with them. Here, for example, is a piece of imperative code for permuting an array. Here, we use the `Random` module as our source of randomness. (`Random` starts out with a deterministic seed, but you can call `Random.self_init` to get a new random seed chosen.)

```
# let permute ar =
    for i = 0 to Array.length ar - 2 do
       (* pick a j that is after i and before the end of the list *)
       let j = i + 1 + Random.int (Array.length ar - i - 1) in
       (* Swap i and j *)
       let tmp = ar.(i) in
       ar.(i) <- ar.(j);
       ar.(j) <- tmp
    done
  ;;
val permute : 'a array -> unit = <fun>
```

Note that the semi-colon after the first array assignment doesn't terminate the scope of the let-binding, so the variable `j` remains in scope until the end of the body of the for loop.

Here's an example run of this code.

```
# let ar = Array.init 20 ~f:(fun i -> i);;
val ar : int array =
  [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19|]
# permute ar;;
```

```
- : unit = ()
# ar;;
- : int array =
[|14; 13; 1; 3; 2; 19; 17; 18; 9; 16; 15; 7; 12; 11; 4; 10; 0; 5; 6; 8|]
```

A complete program

So far, we've played with the basic features of the language using the toplevel. Now we'll create a simple, complete stand-along program that does something useful: sum up a list of numbers read in from the UNIX standard input.

Here's the code, which you can save in a file called `sum.ml`.

```
(* file: sum.ml *)

open Core.Std

let rec read_and_accumulate accum =
  let line = In_channel.input_line stdin in
  match line with
  | None -> accum
  | Some x -> read_and_accumulate (accum +. Float.of_string x)

let () =
  printf "Total: %F\n" (read_and_accumulate 0.)
```

This is our first use of OCaml's input and output routines. The function `read_and_accumulate` uses `In_channel.input_line` to read in lines one by one from the standard input, adding each number to its accumulated sum as it goes. Note that `input_line` returns an optional value, with `None` indicating the end of the input. Note that `read_and_accumulate` is a recursive function, invoking itself to read the next line, until the last line is reached.

After `read_and_accumulate` returns, the total needs to be printed. This is done using the `printf` command, which provides support for type-safe format strings, similar to what you'll find in a variety of languages. The format string is parsed by the compiler and used to determine the number and type of the remaining arguments that are required for `printf`. In this case, there is a single formatting directive, `%F`, so `printf` expects one additional argument of type `float`.

Compiling and running

We can use `ocamlbuild` to compile the program. We'll need to create a file, in the same directory as `sum.ml`, called `_tags`. We can put the following in `_tags` to indicate that we're building against Core, and that threads should be enabled, which is a required by Core.

```
true:package(core),thread
```

With our `_tags` file in place, we can build our executable by issuing this command.

```
ocamlbuild -use-ocamlfind sum.native
```

The `.native` suffix indicates that we're building a native-code executable, which we'll discuss more in Chapter 9. Once the build completes, we can use the resulting program like any command-line utility. In this example, we can just type in a sequence of numbers, one per line, hitting control-d to exit when the input is complete.

```
max $ ./sum.native
1
2
3
94.5
Total: 100.5
```

More work is needed to make a really usable command-line programming, including a proper command-line parsing interface and better error handling.

Where to go from here

That's it for our guided tour! There are plenty of features left to touch upon and lots of details to explain, but the hope is that this has given you enough of a feel for the language that you have a sense as to what to expect, and will be comfortable reading examples in the rest of the book.

2013-02-05
19:46:46

CHAPTER 3

Variables and Functions

Variables and functions are fundamental ideas that show up in virtually all programming languages. But OCaml has a different take on these basic concepts, and so we'll spend some time digging into the details of OCaml's variables and functions differ from what you may have seen elsewhere.

Variables

At its simplest, a variable is an identifier whose meaning is bound to a particular value. In OCaml these bindings are often introduced using the `let` keyword. When typed in at the prompt of the interpreter, a let binding has the following syntax.

```
let <identifier> = <expr>
```

As we'll see when we get to the module system in Chapter 9, this same syntax is used for top-level definitions in a module.

Every variable binding has a *scope*, which is the portion of the code that can refer to that binding. The scope of a top-level let binding is everything that follows it in the top-level session (or in the remainder of the module).

Here's a simple example.

```
# let x = 3;;
val x : int = 3
# let y = 4;;
val y : int = 4
# let z = x + y;;
val z : int = 7
```

`let` can also be used to create a variable binding whose scope is limited to a particular expression, using the following syntax.

```
let <identifier> = <expr1> in <expr2>
```

This first evaluates *`expr1`* and then evaluates *`expr2`* with *`identifier`* bound to whatever value was produced by the evaluation of *`expr1`*. Here's how it looks in practice.

```
# let languages = "OCaml,Perl,C++,C";;
val languages : string = "OCaml,Perl,C++,C"
# let dashed_languages =
    let language_list = String.split languages ~on:',' in
    String.concat ~sep:"-" language_list
  ;;
val dashed_languages : string = "OCaml-Perl-C++-C"
```

Note that the scope of `language_list` is just the expression `String.concat ~sep:"-" language_list`, and is not available at the top-level, as we can see if we try to access it now.

```
# language_list;;
Characters 0-13:
  language_list;;
  ^^^^^^^^^^^^^
Error: Unbound value language_list
```

A let binding in an inner scope can *shadow*, or hide, the definition from an outer scope. So, for example, we could have written the `dashed_languages` example as follows:

```
# let languages = "OCaml,Perl,C++,C";;
val languages : string = "OCaml,Perl,C++,C"
# let dashed_languages =
     let languages = String.split languages ~on:',' in
     String.concat ~sep:"-" languages
  ;;
val dashed_languages : Core.Std.String.t = "OCaml-Perl-C++-C"
```

This time, in the inner scope we called the list of strings `languages` instead of `language_list`, thus hiding the original definition of `languages`. But once the definition of `dashed_languages` is complete, the inner scope has closed and the original definition of languages reappears.

```
# languages;;
- : string = "OCaml,Perl,C++,C"
```

One common idiom is to use a series of nested `let`/`in` expressions to build up the components of a larger computation. Thus, we might write:

```
# let area_of_ring inner_radius outer_radius =
     let pi = acos (-1.) in
     let area_of_circle r = pi *. r *. r in
     area_of_circle outer_radius -. area_of_circle inner_radius
  ;;
# area_of_ring 1. 3.;;
- : float = 25.1327412287183449
```

It's important not to confuse this sequence of let bindings with the modification of a mutable variable. How would `area_of_ring` be different, for example, if we had instead written this purposefully confusing bit of code:

```
# let area_of_ring inner_radius outer_radius =
     let pi = acos (-1.) in
     let area_of_circle r = pi *. r *. r in
     let pi = 0. in
     area_of_circle outer_radius -. area_of_circle inner_radius
  ;;
```

Here, we redefined `pi` to be zero after the definition of `area_of_circle`. You might think that this would mean that the result of the computation would now be zero, but you'd be wrong. In fact, the behavior of the function is unchanged. That's because the original definition of `pi` wasn't changed, it was just shadowed, so that any subsequent reference to `pi` would see the new definition of `pi` as zero. But there is no later use of `pi`, so the binding doesn't make a difference. Indeed, if you type the example I gave above into the toplevel, OCaml will warn you that the definition is unused.

```
Characters 126-128:
    let pi = 0. in
        ^^
Warning 26: unused variable pi.
```

In OCaml, let bindings are immutable. As we'll see in "Imperative programming" on page 20, there are mutable values in OCaml, but no mutable variables.

Pattern matching and `let`

Another useful feature of let bindings is that they support the use of patterns on the left-hand side of the bind. Consider the following code, which uses `List.unzip`, a function for converting a list of pairs into a pair of lists.

```
# let (ints,strings) = List.unzip [(1,"one"); (2,"two"); (3,"three")]
val ints : int list = [1; 2; 3]
val strings : string list = ["one"; "two"; "three"]
```

This actually binds two variables, one for each element of the pair. Using a pattern in a let-binding makes the most sense for a pattern that is *irrefutable*, *i.e.*, where any value of the type in question is guaranteed to match the pattern. Tuple and record patterns are irrefutable, but list patterns are not. Consider the following code that implements a function for up-casing the first element of a comma-separate list.

```
# let upcase_first_entry line =
     let (key :: values) = String.split ~on:',' line in
     String.concat ~sep:"," (String.uppercase key :: values)
  ;;
      Characters 40-53:
       let (key :: values) = String.split ~on:',' line in
            ^^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val upcase_first_entry : string -> string = <fun>
```

This case can't really come up in practice, because `String.split` always returns a list with at least one element. But the compiler doesn't know this, and so it emits the warning. It's generally better to use a match statement to handle such cases explicitly:

```
# let upcase_first_entry line =
     match String.split ~on:',' line with
     | [] -> assert false (* String.split returns at least one element *)
     | first :: rest -> String.concat ~sep:"," (String.uppercase first :: rest)
  ;;
val upcase_first_entry : string -> string = <fun>
```

`let`/`and` bindings

Another variant on the let binding is the use of `and` to join multiple variable definitions into a single declaration. For example, we can write:

```
# let x = 100 and y = 3.5;;
val x : int = 100
val y : float = 3.5
```

This can be useful when you want to create a number of new let bindings at once, without having each definition affect the next. So, if we wanted to create new bindings that swapped the values of `x` and `y`, we could write:

```
# let x = y and y = x ;;
val x : float = 3.5
val y : int = 100
```

This use-case doesn't come up that often. Most of the time that `and` comes into play, it's used to define multiple mutually recursive values, which we'll learn about later in the chapter.

Note that when doing a `let`/`and` style declaration, the order of execution of the right-hand side of the binds is undefined by the language definition, so one should not write code that relies on it.

Functions

OCaml being a functional language, it's no surprise that functions are an important and pervasive element of programming in OCaml. Indeed, we've seen functions pop up already in many of the examples we've looked at thus far. But while we've introduced the basics of functions, we're now going to cover them in more depth, starting from the foundations.

Anonymous Functions

We'll start by looking at the most basic form of OCaml function, the *anonymous* function. Anonymous functions are declared using the `fun` keyword. Here's a simple anonymous function for incrementing an integer.

```
# (fun x -> x + 1);;
- : int -> int = <fun>
```

Anonymous functions aren't named, but they can be used for many different purposes nonetheless. You can, for example, apply an anonymous function to an argument:

```
# (fun x -> x + 1) 7;;
- : int = 8
```

Or pass it to another function.

```
# List.map ~f:(fun x -> x + 1) [1;2;3];;
- : int list = [2; 3; 4]
```

Or even stuff them into a data structure.

```
# let increments = [ (fun x -> x + 1); (fun x -> x + 2) ] ;;
val increments : (int -> int) list = [<fun>; <fun>]
# List.map ~f:(fun f -> f 5) increments;;
- : int list = [6; 7]
```

It's worth stopping for a moment to puzzle this example out, since this kind of higher-order use of functions can be a bit obscure at first. The first thing to understand is the function `(fun f -> f 5)`, which takes a function as its argument and applies that function to the number `5`. The invocation of `List.map` applies `(fun f -> f 5)` to the elements of the `increments` list (which are themselves functions) and returns the list containing the results of these function applications.

The key thing to understand is that functions are ordinary values in OCaml, and you can do everything with them that you'd do with an ordinary value, including passing them to and returning them from other functions and storing them in data structures. We even name functions in the same way that we name other values, by using a let binding.

```
# let plusone = (fun x -> x + 1);;
val plusone : int -> int = <fun>
# plusone 3;;
- : int = 4
```

Defining named functions is so common that there is some built in syntactic sugar for it. Thus, we can write:

```
# let plusone x = x + 1;;
val plusone : int -> int = <fun>
```

This is the most common and convenient way to declare a function, but syntactic niceties aside, the two styles of function definition are entirely equivalent.

`let` and `fun`

Functions and let bindings have a lot to do with each other. In some sense, you can think of the argument of a function as a variable being bound to the value passed by the caller. Indeed, the following two expressions are nearly equivalent:

```
# (fun x -> x + 1) 7;;
- : int = 8
# let x = 7 in x + 1;;
- : int = 8
```

This connection is important, and will come up more when programming in a monadic style, as we'll see in Chapter 16.

Multi-argument functions

OCaml of course also supports multi-argument functions. Here's an example that came up in Chapter 2.

```
# let abs_diff x y = abs (x - y);;
val abs_diff : int -> int -> int = <fun>
# abs_diff 3 4;;
- : int = 1
```

You may find the type signature of `abs_diff` with all of its arrows a little hard to parse. To understand what's going on, let's rewrite `abs_diff` in an equivalent form, using the `fun` keyword:

```
# let abs_diff =
    (fun x -> (fun y -> abs (x - y)));;
val abs_diff : int -> int -> int = <fun>
```

This rewrite makes it explicit that `abs_diff` is actually a function of one argument that returns another function of one argument, which itself returns the final computation. Because the functions are nested, the inner expression `abs (x - y)` has access to both `x`, which was captured by the first function application, and `y`, which was captured by the second one.

This style of function is called a *curried* function. (Currying is named after Haskell Curry, a famous logician who had a significant impact on the design and theory of programming languages.) The key to interpreting the type signature of a curried function is the observation that `->` is right-associative. The type signature of `abs_diff` can therefore be parenthesized as follows. This doesn't change the meaning of the signature, but it makes it easier to see how the currying fits in.

```
val abs_diff : int -> (int -> int)
```

Currying is more than just a theoretical curiosity. You can make use of currying to specialize a function by feeding in some of the arguments. Here's an example where we create a specialized version of `abs_diff` that measures the distance of a given number from `3`.

```
# let dist_from_3 = abs_diff 3;;
val dist_from_3 : int -> int = <fun>
# dist_from_3 8;;
- : int = 5
# dist_from_3 (-1);;
- : int = 4
```

The practice of applying some of the arguments of a curried function to get a new function is called *partial application*.

Note that the `fun` keyword supports its own syntactic sugar for currying, so we could also have written `abs_diff` as follows.

```
# let abs_diff = (fun x y -> abs (x - y));;
```

You might worry that curried functions are terribly expensive, but this is not the case. In OCaml, there is no penalty for calling a curried function with all of its arguments. (Partial application, unsurprisingly, does have a small extra cost.)

Currying is not the only way of writing a multi-argument function in OCaml. It's also possible to use the different arms of a tuple as different arguments. So, we could write:

```
# let abs_diff (x,y) = abs (x - y)
val abs_diff : int * int -> int = <fun>
# abs_diff (3,4);;
- : int = 1
```

OCaml handles this calling convention efficiently as well. In particular it does not generally have to allocate a tuple just for the purpose of sending arguments to a tuple-style function.

There are small tradeoffs between these two approaches, but most of the time, one should stick to currying, since it's the default style in the OCaml world.

Recursive functions

A function is *recursive* if it refers to itself in its definition. Recursion is important in any programming language, but is particularly important in functional languages.

In order to define a recursive function, you need to mark the let binding as recursive with the `rec` keyword, as shown in this example:

```
# let rec find_first_stutter = function
     | [] | [_] ->
       (* only zero or one elements, so no repeats *)
       None
     | x :: y :: tl ->
       if x = y then Some x else find_first_stutter (y::tl)
val find_first_stutter : 'a list -> 'a option = <fun>
```

We can also define multiple mutually recursive values by using `let rec` and `and` together, as in this (gratuitously inefficient) example.

```
# let rec is_even x =
    if x = 0 then true else is_odd (x - 1)
  and is_odd x =
    if x = 0 then false else is_even (x - 1)
 ;;
val is_even : int -> bool = <fun>
val is_odd : int -> bool = <fun>
# List.map ~f:is_even [0;1;2;3;4;5];;
- : bool Core.Std.List.t = [true; false; true; false; true; false]
# List.map ~f:is_odd [0;1;2;3;4;5];;
- : bool Core.Std.List.t = [false; true; false; true; false; true]
```

Note that in the above example, we take advantage of the fact that the right hand side of || is evaluated lazily, only being executed if the left hand side evaluates to false.

Prefix and Infix operators

So far, we've seen examples of functions used in both prefix and infix style:

```
# Int.max 3 4;;  (* prefix *)
- : int = 4
# 3 + 4;;        (* infix  *)
- : int = 7
```

You might not have thought of the second example as an ordinary function, but it very much is. Infix operators like `+` really only differ syntactically from other functions. In fact, if we put parenthesis around an infix operator, you can use it as an ordinary prefix function.

```
# (+) 3 4;;
- : int = 7
```

```
# List.map ~f:((+) 3) [4;5;6];;
- : int list = [7; 8; 9]
```

In the second expression above, we've partially applied `(+)` to gain a function that increments its single argument by `3`, and then applied that to all the elements of a list.

A function is treated syntactically as an operator if the name of that function is chosen from one of a specialized set of identifiers. This set includes any identifier that is a sequence of characters from the following set

```
! $ % & * + - . / : < = > ? @ ^ | ~
```

or is one of a handful of pre-determined strings, including `mod`, the modulus operator, and `lsl`, for "logical shift left", a bit-shifting operation.

We can define (or redefine) the meaning of an operator as follows. Here's an example of a simple vector-addition operator on int pairs.

```
# let (+!) (x1,y1) (x2,y2) = (x1 + x2, y1 + y2)
val ( +! ) : int * int -> int * int -> int *int = <fun>
# (3,2) +! (-2,4);;
- : int * int = (1,6)
```

The syntactic role of an operator work is determined by its first character. This table describes how, and lists the operators from highest to lowest precedence.

First character	Usage
! ? ~	Prefix and unary
**	Infix, right associative
+ -	Infix, left associative
@ ^	Infix, right associative
= < > | & $	Infix, left associative

There's one important special case: `-` and `-.`, which are the integer and floating point subtraction operators, can act as both prefix operators (for negation) and infix operators (for subtraction), So, you both `-x` and `x - y` are meaningful expressions.

Here's an example of a very useful operator that's defined in Core, following these rules. Here's the definition:

```
# let (|!) x f = f x ;;
val ( |! ) : 'a -> ('a -> 'b) -> 'b = <fun>
```

It's not quite obvious at first what the purpose of this operator is: it just takes some value and a function, and applies the function to the value. But its utility is clearer when you see it in action. It works as a kind of sequencing operator, similar in spirit to using pipe in the UNIX shell. Consider, for example, the following code for printing out the unique elements of your `PATH`. Note that `List.dedup` below removes duplicates from a list by sorting the list using the provided comparison function.

```
# Sys.getenv_exn "PATH"
  |! String.split ~on:':'
  |! List.dedup ~compare:String.compare
  |! List.iter ~f:print_endline
  ;;
/bin
/opt/local/bin
/usr/bin
/usr/local/bin
- : unit = ()
```

An important part of what's happening here is partial application. Normally, `List.iter` takes two arguments: a function to be called on each element of the list, and the list to iterate over. We can call `List.iter` with all it's arguments:

```
# List.iter ~f:print_endline ["Two"; "lines"];;
Two
lines
- : unit = ()
```

Or, we can pass it just the function argument, leaving us with a function for printing out a list of strings.

```
# List.iter ~f:print_endline;;
- : string list -> unit = <fun>
```

It is this later form that we're using in the `|!` pipeline above.

Note that `|!` only works in the intended way because it is left-associative. Indeed, let's see what happens if we try using a right associative operator, like (^!).

```
# let (^!) = (|!);;
val ( ^! ) : 'a -> ('a -> 'b) -> 'b = <fun>
# Sys.getenv_exn "PATH"
  ^! String.split ~on:':'
  ^! List.dedup ~compare:String.compare
  ^! List.iter ~f:print_endline
  ;;
        Characters 93-119:
    ^! List.iter ~f:print_endline
       ^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: This expression has type string list -> unit
       but an expression was expected of type
         (string list -> string list) -> 'a
```

The above type error is a little bewildering at first glance. What's going on is that, because `^!` is right associative, the operator is trying to feed the value `List.dedup ~com` `pare:String.compare` to the function `List.iter ~f:print_endline`. But `List.iter ~f:print_endline` expects a list of strings as its input, not a function.

The type error aside, this example highlights the importance of choosing the operator you use with care, particularly with respect to associativity.

Declaring functions with `function`

Another way to define a function is using the `function` keyword. Instead of having syntactic support for declaring multi-argument (curried) functions, `function` has built-in pattern matching. Here's an example:

```
# let some_or_zero = function
     | Some x -> x
     | None -> 0
  ;;
val some_or_zero : int option -> int = <fun>
# List.map ~f:some_or_zero [Some 3; None; Some 4];;
- : int list = [3; 0; 4]
```

This is equivalent to combining a `fun` with `match`, as follows:

```
# let some_or_zero num_opt =
    match num_opt with
    | Some x -> x
    | None -> 0
  ;;
val some_or_zero : int option -> int = <fun>
```

We can also combine the different styles of function declaration together, as in the following example where we declare a two argument (curried) function with a pattern-match on the second argument.

```
# let some_or_default default = function
     | Some x -> x
     | None -> default
  ;;
# some_or_default 3 (Some 5);;
- : int = 5
# List.map ~f:(some_or_default 100) [Some 3; None; Some 4];;
- : int list = [3; 100; 4]
```

Also, note the use of partial application to generate the function passed to `List.map`. In other words, `some_or_default 100` is a function that was created by feeding just the first argument to `some_or_default`.

Labeled Arguments

Up until now, we've written functions where the arguments are specified positionally, *i.e.*, by the order in which the arguments are passed to the function. OCaml also supports labeled arguments, which let you identify a function argument by name. Labels are marked by a leading tilde, and a label (followed by a colon) are put in front of the variable to be labeled.

```
# let f ~foo:a ~bar:b = a + b;;
val f : foo:int -> bar:int -> int = <fun>
```

We can then provide a labeled argument using a similar convention. As you can see, the arguments can be provided in any order.

```
# f ~foo:3 ~bar:10;;
- : int = 13
# f ~bar:10 ~foo:3;;
- : int = 13
```

OCaml also supports *label punning*, meaning that you get to drop the text after the `:` if the name of the label and the name of the variable being used are the same. Label punning works in both function declaration and function invocation, as shown in these examples:

```
# let foo = 3;;
# let bar = 4;;
# f ~foo ~bar;;
- : int = 7
```

Labeled arguments are useful in a few different cases:

- When defining a function with lots of arguments. Beyond a certain number, arguments are easier to remember by name than by position.
- When defining functions that have multiple arguments that might get confused with each other. This is most at issue when the arguments are of the same type. For example, consider this signature for a function for extracting a substring of another string.

  ```
  val substring: string -> int -> int -> string
  ```

  where the two ints are the starting position and length of the substring to extract. Labeled arguments can make this signature clearer:

  ```
  val substring: string -> pos:int -> len:int -> string
  ```

  This improves the readability of both the signature and of client code that makes use of `substring`, and makes it harder to accidentally swap the position and the length.

- When the meaning of a particular argument is unclear from the type alone. For example, consider a function for creating a hashtable where the first argument is the initial size of the table, and the second argument is a flag which, when true, indicates that the hashtable will reduce its size when the hashtable contains few elements. The following signature doesn't give you much of a hint as to the meaning of the arguments.

  ```
  val create_hashtable : int -> bool -> ('a,'b) Hashtable.t
  ```

  but with labeled arguments, we can make the intent much clearer.

  ```
  val create_hashtable : init_size:int -> allow_shrinking:bool -> ('a,'b) Hashtable.t
  ```

- When you want flexibility on the order in which arguments are passed. Consider a function like `List.iter`, that takes two arguments: a function, and a list of elements to call that function on. A common pattern is to partially apply `List.iter` by giving it just the function, as in the following example from earlier in the chapter. This requires putting the function argument first.

  ```
  # Sys.getenv_exn "PATH"
    |! String.split ~on:':'
    |! List.dedup ~compare:String.compare
    |! List.iter ~f:print_endline
    ;;
  ```

  In other cases, you want to put the function argument second. One common reason is readability. In particular, a function that spans multiple lines is easiest to read when it's the last argument provided.

Higher-order functions and labels

One surprising gotcha with labeled arguments is that while order doesn't matter when calling a function with labeled arguments, it does matter in a higher-order context, *e.g.*, when passing a function with labeled arguments to another function. Here's an example.

```
# let apply_to_tuple f (first,second) = f ~first ~second;;
val apply_to_tuple : (first:'a -> second:'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

Here, the definition of `apply_to_tuple` sets up the expectation that its first argument is a function with two labeled arguments, `first` and `second`, listed in that order. We could have defined `apply_to_tuple` differently to change the order in which the labeled arguments were listed.

```
# let apply_to_tuple f (first,second) = f ~second ~first;;
val apply_to_tuple : (second:'a -> first:'b -> 'c) -> 'b * 'a -> 'c = <fun>
```

It turns out this order of listing matters. In particular, if we define a function that has a different order

```
# let divide ~first ~second = first / second;;
val divide : first:int -> second:int -> int = <fun>
```

we'll find that it can't be passed in to `apply_to_tuple`.

```
# apply_to_tuple divide (3,4);;
Characters 15-21:
  apply_to_tuple divide (3,4);;
                 ^^^^^^
Error: This expression has type first:int -> second:int -> int
       but an expression was expected of type second:'a -> first:'b -> 'c
```

But, if we go back to the original definition of `apply_to_tuple`, things will work smoothly.

```
# let apply_to_tuple f (first,second) = f ~first ~second;;
val apply_to_tuple : (first:'a -> second:'b -> 'c) -> 'a * 'b -> 'c = <fun>
# apply_to_tuple divide (3,4);;
- : int = 0
```

So, even though the order of labeled arguments usually doesn't matter, it will sometimes bite you in higher-ordered contexts, where you're doing things like passing functions as arguments to other functions.

Optional arguments

An optional argument is like a labeled argument that the caller can choose whether or not to provide. Optional arguments are passed in using the same syntax as labeled arguments, and, similarly to labeled arguments, optional arguments can be provided in any order.

Here's an example of a string concatenation function with an optional separator.

```
# let concat ?sep x y =
     let sep = match sep with None -> "" | Some x -> x in
     x ^ sep ^ y
  ;;
val concat : ?sep:string -> string -> string -> string = <fun>
# concat "foo" "bar";;             (* without the optional argument *)
- : string = "foobar"
# concat ~sep:":" "foo" "bar";;    (* with the optional argument    *)
- : string = "foo:bar"
```

Here, `?` is used in the definition of the function to mark `sep` as optional. And while the caller can pass a value of type `string` for `sep`, internally to the function, `sep` is seen as a `string option`, with `None` appearing when `sep` is not provided by the caller.

In the above example, we had a bit of code to substitute in the empty string when no argument was provided. This is a common enough pattern that there's an explicit syntax for providing a default value, which allows us to write `concat` even more concisely.

```
# let concat ?(sep="") x y = x ^ sep ^ y ;;
val concat : ?sep:string -> string -> string -> string = <fun>
```

Optional arguments are very useful, but they're also easy to abuse. The key advantage of optional arguments is that they let you write functions with multiple arguments that users can ignore most of the time, only worrying about them when they specifically want to invoke those options.

The downside is that the caller may be unaware that there is a choice to be made, and so may unknowingly (and wrongly) pick that default behavior. Optional arguments really only make sense when the extra concision of omitting the argument overwhelms the corresponding loss of explicitness.

This means that rarely used functions should not have optional arguments. A good rule of thumb is not to use optional arguments for functions internal to a module, *i.e.*, functions that are not included in the `mli`.

Explicit passing of an optional argument

Under the covers, a function with an optional argument receives `None` when the caller doesn't provide the argument, and `Some` when it does. But the `Some` and `None` are normally not explicitly passed in by the caller.

But sometimes, passing in `Some` or `None` explicitly is exactly what you want. OCaml lets you do this by using `?` instead of `~` to mark the argument. Thus, the following two lines are equivalent ways of specifying the `sep` argument to concat.

```
# concat ~sep:":" "foo" "bar";; (* provide the optional argument *)
- : string = "foo:bar"
# concat ?sep:(Some ":") "foo" "bar";; (* pass an explicit [Some] *)
- : string = "foo:bar"
```

And the following two lines are equivalent ways of calling `concat` without specifying `sep`.

```
# concat "foo" "bar";; (* don't provide the optional argument *)
- : string = "foobar"
# concat ?sep:None "foo" "bar";; (* explicitly pass `None` *)
- : string = "foobar"
```

One use-case for this is when you want to define a wrapper function that mimics the optional arguments of the function it's wrapping. For example, imagine we wanted to create a function called `uppercase_concat`, which is the same as `concat` except that it converts the first string that it's passed to uppercase. We could write the function as follows.

```
# let uppercase_concat ?(sep="") a b = concat ~sep (String.uppercase a) b ;;
val uppercase_concat : ?sep:string -> string -> string -> string = <fun>
# uppercase_concat "foo" "bar";;
- : string = "FOObar"
# uppercase_concat "foo" "bar" ~sep:":";;
- : string = "FOO:bar"
```

In the way we've written it, we've been forced to separately make the decision as to what the default separator is. Thus, if we later change `concat`'s default behavior, we'll need to remember to change `uppercase_concat` to match it.

Instead, we can have `uppercase_concat` simply pass through the optional argument to `concat` using the `?` syntax.

```
# let uppercase_concat ?sep a b = concat ?sep (String.uppercase a) b ;;
val uppercase_concat : ?sep:string -> string -> string -> string = <fun>
```

Now, if someone calls `uppercase_concat` without an argument, an explicit `None` will be passed to `concat`, leaving `concat` to decide what the default behavior should be.

Inference of labeled and optional arguments

One subtle aspect of labeled and optional arguments is how they are inferred by the type system. Consider the following example for computing numerical derivatives of a function of two dimensions. The function takes an argument `delta` which determines the scale at which to compute the derivative, values `x` and `y` which determine which point to compute the derivative at, and the function `f` whose derivative is being computed. The function `f` itself takes two labeled arguments `x` and `y`. Note that you can use an apostrophe as part of a variable name, so `x'` and `y'` are just ordinary variables.

```
# let numeric_deriv ~delta ~x ~y ~f =
    let x' = x +. delta in
    let y' = y +. delta in
    let base = f ~x ~y in
    let dx = (f ~x:x' ~y -. base) /. delta in
    let dy = (f ~x ~y:y' -. base) /. delta in
    (dx,dy)
  ;;
val numeric_deriv :
  delta:float ->
  x:float -> y:float -> f:(x:float -> y:float -> float) -> float * float =
  <fun>
```

In principle, it's not obvious how the order of the arguments to `f` should be chosen. Since optional arguments can be passed in arbitrary order, it seems like it could as well be `y:float -> x:float -> float` as it is `x:float -> y:float -> float`.

Even worse, it would be perfectly consistent for `f` to take an optional argument instead of a labeled one, which could lead to this type signature for `numeric_deriv`:

```
val numeric_deriv :
  delta:float ->
  x:float -> y:float -> f:(?x:float -> y:float -> float) -> float * float =
  <fun>
```

Since there are multiple plausible types to choose from, OCaml needs some heuristic for choosing between them. The heuristic the compiler uses is to prefer labels to options, and to choose the order of arguments that shows up in the source code.

Note that these heuristics might at different points in the source suggest different types. Here's a version of `numeric_deriv` where the invocations of `f` were changes so they list the arguments in different orders.

```
# let numeric_deriv ~delta ~x ~y ~f =
    let x' = x +. delta in
    let y' = y +. delta in
    let base = f ~x ~y in
    let dx = (f ~y ~x:x' -. base) /. delta in
    let dy = (f ~x ~y:y' -. base) /. delta in
    (dx,dy)
  ;;
Characters 131-132:
      let dx = (f ~y ~x:x' -. base) /. delta in
                ^
Error: This function is applied to arguments
in an order different from other calls.
This is only allowed when the real type is known.
```

As suggested by the error message, we can get OCaml to accept the fact that `f` is used with different argument orders if we provide explicit type information. Thus, the following code compiles without error, due to the type annotation on `f`.

```
# let numeric_deriv ~delta ~x ~y ~(f: x:float -> y:float -> float) =
    let x' = x +. delta in
    let y' = y +. delta in
    let base = f ~x ~y in
    let dx = (f ~y ~x:x' -. base) /. delta in
    let dy = (f ~x ~y:y' -. base) /. delta in
    (dx,dy)
  ;;
val numeric_deriv :
  delta:float ->
  x:float -> y:float -> f:(x:float -> y:float -> float) -> float * float =
  <fun>
```

Optional arguments and partial application

Optional arguments can be tricky to think about in the presence of partial application. We can of course partially apply the optional argument itself:

```
# let colon_concat = concat ~sep:":";;
```

```
val colon_concat : string -> string -> string = <fun>
# colon_concat "a" "b";;
- : string = "a:b"
```

But what happens if we partially apply just the first argument?

```
# let prepend_pound = concat "# ";;
val prepend_pound : string -> string = <fun>
# prepend_pound "a BASH comment";;
- : string = "# a BASH comment"
```

Note that the optional argument `?sep` has now disappeared, or *erased*. So when does OCaml decide to erase an optional argument?

The rule is: an optional argument is erased as soon as the first positional argument defined *after* the optional argument is passed in. That explains the behavior of `prepend_pound` above. But if we had instead defined `concat` with the optional argument in the second position:

```
# let concat x ?(sep="") y = x ^ sep ^ y ;;
val concat : string -> ?sep:string -> string -> string = <fun>
```

then application of the first argument would not cause the optional argument to be erased.

```
# let prepend_pound = concat "# ";;
val prepend_pound : ?sep:string -> string -> string = <fun>
# prepend_pound "a BASH comment";;
- : string = "# a BASH comment"
# prepend_pound "a BASH comment" ~sep:"--- ";;
- : string = "# --- a BASH comment"
```

However, if all arguments to a function are presented at once, then erasure of optional arguments isn't applied until all of the arguments are passed in. This preserves our ability to pass in optional arguments anywhere on the argument list. Thus, we can write:

```
# concat "a" "b" ~sep:"=";;
- : string = "a=b"
```

An optional argument that doesn't have any following positional arguments can't be erased at all, which leads to a compiler warning.

```
# let concat x y ?(sep="") = x ^ sep ^ y ;;
Characters 15-38:
  let concat x y ?(sep="") = x ^ sep ^ y ;;
                 ^^^^^^^^^^^^^^^^^^^^^^^
Warning 16: this optional argument cannot be erased.
val concat : string -> string -> ?sep:string -> string = <fun>
```

And indeed, when we provide the two positions arguments, the `sep` argument is not erased, instead returning a function that expects the `sep` argument to be provided.

```
# concat "a" "b";;
- : ?sep:string -> string = <fun>
```

2013-02-05
19:46:46

CHAPTER 4

Lists, Options and Patterns

(Note, this chapter is incomplete. jyh is working on it.)

Lists

As with any programming language, we need a way to represent *data*, things like numbers, words, images, etc., and we need a way to define *aggregates* that bring together related values that represent some concept.

Lists are one of the most common ways to aggregate data in OCaml. You can construct a list of values by enclosing them in square brackets, and separating the elements with semicolons. The list elements must all have the same type.

```
# let l1 = ["Chicago"; "Paris"; "Tokyo"];;
val l1 : string list = ["Chicago"; "Paris"; "Tokyo"]
# List.nth l1 1;;
- : string option = Some "Paris"
# List.nth l1 2;;
- : string option = Some "Tokyo"
```

The square bracket syntax is really just a shorthand. There are really just two ways to construct a list value.

- [] is the *empty* list.
- If `x` is a value and `l` is a list, then the expression `x :: l` constructs a new list where the first element is `x`, and the rest is `l`. The value corresponding to `x :: l` is commonly called a *cons*-cell (the term comes from Lisp, where "cons" is short for "constructor").

The bracket syntax `["Chicago"; "Paris"; "Tokyo"]` is just another way to write a list with 3 cons-cells, `"Chicago" :: "Paris" :: "Tokyo" :: []`. Each cell has two parts: a value, and a pointer to the rest of the list. The final pointer refers to the special value `[]` representing the empty list.

TODO: IMAGE figures/04-list-01.svg

```
# let l2 = "Chicago" :: "Paris" :: "Tokyo" :: [];;
val l2 : string list = ["Chicago"; "Paris"; "Tokyo"]
# List.hd l2;;
- : string option = Some "Chicago"
# List.tl l2;;
- : string list option = Some ["Paris"; "Tokyo"]
```

Pattern matching

Constructing a list is really only half the story -- it would be pretty useless to construct lists unless we can also pull them apart. We need destructors, and for this we use *pattern matching*.

For a list, there are two possible shapes: the empty list `[]` or a cons-cell `h :: t`. We can use a `match` expression to perform the pattern matching. In the case of a cons-cell, the variables `h` and `t` in the pattern are bound to the corresponding values in the list when the match is performed.

For example, suppose we want to define a function to add 1 to each element of a list. We have to consider two cases, where the list is empty, or where it is a cons-cell `h :: t`. In the latter case, we add one to `h`, and then recursively compute over the tail of the list `t`.

```
# let rec add1 l =
    match l with
    | [] -> []
    | h :: t -> (h + 1) :: (add1 t);;
val add1 : int list -> int list = <fun>
# add1 [5; 3; 7];;
- : int list = [6; 4; 8]
```

Patterns are not limited to just one constructor. They can be aribitrarily nested. For example, if we want to extract the third element of a list, we can use a pattern `_ :: _ :: x :: _`. The underscore `_` is a special pattern that matches (and ignores) anything. The first two underscores match the first two elements of the list, the `x` matches the third element, and the final underscore matches the rest of the list.

```
# let third l =
    match l with
    | _ :: _ :: x :: _ -> Some x
    | _ -> None;;
val third : 'a list -> 'a option = <fun>
# third ["A"; "B"; "C"; "D"];;
- : string option = Some "C"
# third ["A"; "B"];;
- : string option = None
```

The pattern `_ :: _ :: x :: _` is not exhaustive, because it doesn't match lists with fewer than three elements, so we have included a second "wildcard" pattern, a single underscore `_`, that matches anything else. We're using the `option` type for the return value, where `Some x` means that the third element of the list was `x`, and `None` means that the list had fewer than three elements. We'll discuss options more in the next section.

Patterns are matched in left-to-right order (or top-to-bottom, in this case), so the result is determined by the first pattern to match. If we had listed the patterns in opposite order, the wildcard pattern would match everything, and the second pattern would be ignored.

```
# let broken_third l =
    match l with
    | _ -> None
    | _ :: _ :: x :: _ -> Some x;;
Characters 47-63:
    | _ :: _ :: x :: _ -> Some x;;
      ^^^^^^^^^^^^^^^^
Warning 11: this match case is unused.
val broken_third : 'a list -> 'a option = <fun>
# broken_third [17; 21; 7; 34];;
- : int option = None
```

Pattern ordering can be an issue, but a more important concern is that pure wildcard patterns match anything, making it easier to forget "important" cases. When a pattern matching is compiled, OCaml performs an exhaustiveness check, printing a warning if it is not exhaustive, along with an example of a missing pattern.

```
# let inexhaustive_third l =
    match l with
    | _ :: _ :: x :: _ -> Some x;;
    Characters 24-68:
  ...match l with
     | _ :: _ :: x :: _ -> Some x..
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val inexhaustive_third : 'a list -> 'a option = <fun>
# inexhaustive_third [1; 2; 3; 4];;
- : int option = Some 3
# inexhaustive_third [1; 2];;
Exception: (Match_failure //toplevel// 14 3).
```

This exhaustiveness checking can be tremendously useful in ensuring that the appropriate cases are all considered. Generally speaking, it is usually better to be as specific as possible when writing patterns. This helps exhaustiveness checking point out missing cases if they exist, and also means that pattern ordering is not as important.

```
# let third l =
    match l with
    | [] | [_] | [_; _] -> None
    | _ :: _ :: x :: _ -> Some x;;
val third : 'a list -> 'a option = <fun>
```

Options

Note that the `List` functions like `List.hd` (for "head" of the list), `List.tl` (for "tail" of the list), and `List.nth` return values of `option` type. That's because, in some cases, there is no value to return. For example, the empty list has no head or tail.

```
# List.hd ["Chicago"; "Paris"; "Tokyo"];;
- : string option = Some "Chicago"
# List.tl ["Chicago"; "Paris"; "Tokyo"];;
- : string list option = Some ["Paris"; "Tokyo"]
# List.nth ["Chicago"; "Paris"; "Tokyo"] 2;;
- : string option = Some "Tokyo"
# List.hd [];;
- : 'a option = None
# List.tl [];;
- : 'a list option = None
# List.nth ["Chicago"; "Paris"; "Tokyo"] 5;;
- : string option = None
```

A value of `option` type is "optional" -- it is either `None`, meaning "no value"; or `Some v` for some value `v`. For example, `List.hd ["Chicago"; "Paris"; "Tokyo"]` is `Some "Chicago"` because the head of the list exists ("Chicago"), but `List.hd []` is `None` because there is no head of the empty list.

The type definition for `option` has the following form.

```
type 'a option =
  | Some of 'a
  | None
```

As with lists, the way to take apart an `option` value is to use pattern matching, with patterns for the two cases.

```
# let optional_default ~default opt =
    match opt with
    | Some v -> v
    | None -> default;;
val optional_default : default:'a -> 'a option -> 'a = <fun>
# optional_default ~default:10 (Some 17);;
- : int = 17
# optional_default ~default:10 None;;
- : int = 10
```

One annoying thing about the use of options is that functions do not compose directly. For example, if we wanted to compute the third element of a list using the `List.hd` and `List.tl` functions, we might try to compose them like this.

```
# let third l = List.hd (List.tl (List.tl l));;
Characters 32-41:
  let third l = List.hd (List.tl (List.tl l));;
                                  ^^^^^^^^^
```

```
Error: This expression has type 'a list option
       but an expression was expected of type 'b list
```

Unfortunately, this doesn't work because the functions return options, not lists. We can use pattern matching to perform the composition, but the code is pretty tedious.

```
# let tedious_third l1 =
    match List.tl l1 with
    | None -> None
    | Some l2 ->
      match List.tl l2 with
      | None -> None
      | Some l3 ->
        List.hd l3;;
val tedious_third : 'a list -> 'a option = <fun>
# tedious_third [7; 21; 12; 19];;
- : int option = Some 12
# tedious_third [1; 2];;
- : int option = None
```

One solution for abbreviating the code is to define a composition operator `v >>= f` that takes an option value `v`, performs the pattern match, and passes the value to function `f` if there is one.

```
# let (>>=) v f =
    match v with
    | None -> None
    | Some x -> f x;;
val ( >>= ) : 'a option -> ('a -> 'b option) -> 'b option = <fun>
# let third l = Some l >>= List.tl >>= List.tl >>= List.hd;;
val third : 'a list -> 'a option = <fun>
# third [7; 21; 12; 19];;
- : int option = Some 12
# third [7; 21];;
- : int option = None
```

The operator `>>=` has infix syntax, and the expression `Some l >>= List.tl >>= List.tl >>= List.hd` should be read left-to-right: pass `Some l` to the `List.tl` function, then to another `List.tl`, then to `List.hd`. Since `>>=` is an infix operator, the function definition uses parenthesis `(>>=)` to define the function in prefix form.

All we have done here is to hoist the pattern matching into a separate function `>>=`, but the result has a convenient left-to-right sequential reading. This is a common pattern in Core, captured by the `Monad.S` module signature. In fact, The `Core.Std.Option` module already implements the `Monad.S` signature. A more conventional way to write the function would be to use function `Option.(>>=)` directly.

```
# let third l =
    let (>>=) = Option.(>>=) in
    Some l >>= List.tl >>= List.tl >>= List.hd;;
val third : 'a list -> 'a option = <fun>
```

Example: pretty-printing a table

_(jyh: I'm revising the examples. This section is temporary.)

(yminsky: I wonder if the whole example should go later in the section, after we've done more of the basis of lists in this chapter.)

One common programming task is displaying tabular data. In this example, we will go over the design of a simple library to do just that.

(yminsky: This is the first appearance of an mli file. If we're going to introduce it here, we need to do a little more explanation.)

We'll start with the interface. The code will go in a new module called `Text_table` whose `.mli` contains just the following function:

```
(* [render headers rows] returns a string containing a formatted
   text table, using Unix-style newlines as separators *)
val render
   :  string list         (* header *)
   -> string list list    (* data *)
   -> string
```

If you invoke `render` as follows:

```
let () =
  print_string (Text_table.render
     ["language";"architect";"first release"]
     [ ["Lisp" ;"John McCarthy" ;"1958"] ;
       ["C"    ;"Dennis Ritchie";"1969"] ;
       ["ML"   ;"Robin Milner"  ;"1973"] ;
       ["OCaml";"Xavier Leroy"  ;"1996"] ;
     ])
```

you'll get the following output:

```
| language | architect      | first release |
|----------+----------------+---------------|
| Lisp     | John McCarthy  | 1958          |
| C        | Dennis Ritchie | 1969          |
| ML       | Robin Milner   | 1973          |
| OCaml    | Xavier Leroy   | 1996          |
```

Now that we know what `render` is supposed to do, let's dive into the implementation.

Computing the widths

To render the rows of the table, we'll first need the width of the widest entry in each column. The following function does just that.

```
let max_widths header rows =
let to_lengths l = List.map ~f:String.length l in
List.fold rows
  ~init:(to_lengths header)
  ~f:(fun acc row ->
    List.map2_exn ~f:Int.max acc (to_lengths row))
```

In the above we define a helper function, `to_lengths` which uses `List.map` and `String.length` to convert a list of strings to a list of string lengths. Then, starting with the lengths of the headers, we use `List.fold` to join in the lengths of the elements of each row by `max`'ing them together element-wise.

Note that this code will throw an exception if any of the rows has a different number of entries than the header. In particular, `List.map2_exn` throws an exception when its arguments have mismatched lengths.

Rendering the rows

Now we need to write the code to render a single row. There are really two different kinds of rows that need to be rendered; an ordinary row:

```
| Lisp     | John McCarthy  | 1962          |
```

and a separator row:

```
|----------+----------------+---------------|
```

Let's start with the separator row, which we can generate as follows:

```
let render_separator widths =
  let pieces = List.map widths
    ~f:(fun w -> String.make (w + 2) '-')
  in
  "|" ^ String.concat ~sep:"+" pieces ^ "|"
```

We need the extra two-characters for each entry to account for the one character of padding on each side of a string in the table.

Performance of `String.concat` and `^`

In the above, we're using two different ways of concatenating strings, `String.concat`, which operates on lists of strings, and `^`, which is a pairwise operator. You should avoid `^` for joining long numbers of strings, since, it allocates a new string every time it runs. Thus, the following code:

```
let s = "." ^ "."  ^ "."  ^ "."  ^ "."  ^ "."  ^ "."
```

will allocate a string of length 2, 3, 4, 5, 6 and 7, whereas this code:

```
let s = String.concat [".";".";".";".";".";".";"."]
```

allocates one string of size 7, as well as a list of length 7. At these small sizes, the differences don't amount to much, but for assembling of large strings, it can be a serious performance issue.

We can write a very similar piece of code for rendering the data in an ordinary row.

```
let pad s length =
  if String.length s >= length then s
  else s ^ String.make (length - String.length s) ' '

let render_row row widths =
  let pieces = List.map2 row widths
    ~f:(fun s width -> " " ^ pad s width ^ " ")
  in
  "|" ^ String.concat ~sep:"|" pieces ^ "|"
```

You might note that `render_row` and `render_separator` share a bit of structure. We can improve the code a bit by factoring that repeated structure out:

```
let decorate_row ~sep row = "|" ^ String.concat ~sep row ^ "|"

let render_row widths row =
  decorate_row ~sep:"|"
    (List.map2_exn row widths ~f:(fun s w -> " " ^ pad s w ^ " "))

let render_separator widths =
  decorate_row ~sep:"+"
    (List.map widths ~f:(fun width -> String.make (width + 2) '-'))
```

And now we can write the function for rendering a full table.

```
let render header rows =
  let widths = max_widths header rows in
  String.concat ~sep:"\n"
    (render_row widths header
     :: render_separator widths
```

```
   :: List.map rows ~f:(fun row -> render_row widths row)
  )
```

List performance

Lists are ubiquitous in OCaml programs. They are easy to use and reasonably efficient for small lists, but large lists can have significant performance problems. The issue is that lists are formed from separately allocated cons-cells. This has space overhead because each value in the list is paired with a pointer to the rest of the list. The separate allocation also reduces locality, so it can result in poor cache behavior.

Perhaps more important than those concerns is that naive list traversal takes time linear in the length of the list. For example, the following `length` function takes linear time to count the number of elements in the list.

```
let rec length = function [] -> 0 | _ :: t -> (length t) + 1;;
```

In fact, this implementation of the function `length` is worse than that, because the function is recursive. In this implementation of the function, the recursive call to `length t` is active at the same time as the outer call, with the result that the runtime needs to allocate stack frames for each recursive call, so this function also takes linear space. For large lists, this is not only inefficient, it can also result in stack overflow.

Tail-recursion

We can't do anything about `length` taking linear time -- singly-linked lists of this kind don't have an efficient `length` operation. However, we can address the space problem using *tail recursion*.

Tail recursion occurs whenever the result of the recursive call is returned immediately by the calling function. In this case, the compiler optimizes the call by skipping the allocation of a new stack frame, instead branching directly to the called procedure.

In the definition of `length` above, the expression containing the recursive call `(length t) + 1` is *not* tail recursive because 1 is added to the result. However, it is easy to transform the function so that it is properly tail recursive.

```
let length l =
  let rec tail_recursive_length len = function
    | [] -> len
    | _ :: t -> tail_recursive_length (len + 1) t
  in tail_recursive_length 0 l;;
```

To preserve the type of the `length` function, we hide the tail-recursive implementation by nesting it. The tail-recursive implementation performs the addition *before* the recursive call, instead of afterwards. Since the result of the recursive call is returned without modification, the compiler branches directly to the called procedure rather than allocating a new stack frame.

In other cases, it can be more problematic to use tail-recursion. For example, consider the non tail-recursive implemenation of `map` function, listed above. The code is simple, but not efficient.

```
let rec map f = function
 | [] -> []
 | h :: t -> f h :: map f t;;
```

If we use the same trick as we used for the `length` method, we need to accumulate the result *before* the recursive call, but this collects the result in reverse order. One way to address it is to construct the reserved result, then explicitly correct it before returning.

```
let rev l =
  let rec tail_recursive_rev result = function
   | [] -> result
   | h :: t -> tail_recursive_rev (h :: result) t
  in tail_recursive_rev [] l;;

let rev_map l ~f =
  let rec rmap accu = function
   | [] -> accu
   | h :: t -> rmap (f h :: accu) l
  in rmap [] l;;

let map l ~f = rev (rev_map l ~f);;
```

The functions `tail_recursive_rev` and `rev_map` are both tail-recursive, which means that the function `map` is tail-recursive also. The cost of doing so is that we construct an intermediate reversed list that is immediately discarded. One way to think of it is that instead of allocating a linear number of stack frames, we allocate a linear number of cons-cells.

Allocation of short-lived data in OCaml is quite cheap, so the intermediate list is not very expensive. The performance of the two implementations is not significantly different, with one exception: the tail-recursive implementation will not cause a stack overflow for large lists, while the simple non-tail-recursive implementation will have problems with large lists.

Hybrid recursion

In general, the choice of whether to use regular recursion vs. tail recursion is not immediately obvious. Regular recursion is often better for small lists (and other data structures), but it is better to use tail recursion for very large lists -- especially because stack sizes limit the number of recursive calls.

Core takes a hybrid approach that can be illustrated with the implementation of the function `Core_list.map`.

```
let map_slow l ~f = rev (rev_map l ~f);;

let rec count_map ~f l ctr =
  match l with
  | [] -> []
  | [x1] -> let f1 = f x1 in [f1]
  | [x1; x2] -> let f1 = f x1 in let f2 = f x2 in [f1; f2]
  | [x1; x2; x3] ->
    let f1 = f x1 in
    let f2 = f x2 in
    let f3 = f x3 in
    [f1; f2; f3]
  | [x1; x2; x3; x4] ->
    let f1 = f x1 in
    let f2 = f x2 in
    let f3 = f x3 in
    let f4 = f x4 in
    [f1; f2; f3; f4]
  | x1 :: x2 :: x3 :: x4 :: x5 :: tl ->
    let f1 = f x1 in
    let f2 = f x2 in
    let f3 = f x3 in
    let f4 = f x4 in
    let f5 = f x5 in
    f1 :: f2 :: f3 :: f4 :: f5 ::
      (if ctr > 1000 then map_slow ~f tl else count_map ~f tl (ctr + 1));;

let map l ~f = count_map ~f l 0;;
```

For performance, there are separate patterns for small lists with up to 4 elements, then a recursive case for lists with five or more elements. The `ctr` value limits the recursion -- regular recursion is used for up to 1000 recursive calls (which includes lists with up to 4000 elements), then the tail-recursive function `map_slow` is used for any remainder.

As an aside, you might wonder why this implementation uses explicit let-definitions for the result values `f1`, `f2`, etc. The reason is to force the order of evaluation, so that the the function `f` is always applied to the list values left-to-right (starting with the first element in the list). In an expression like `[f x1; f x2; f x3]` the order of evaluation is not specified by the language, any of the subexpressions might be evaluated first (though we would often expect evaluation order to be either left-to-right or right-to-left). For functions that perform I/O, or have other side-effects, left-to-right evaluation order is important (and required).

Heterogenous values

Lists are fairly general, but there are several reasons why you might not want to use them.

- Large lists often have poor performance.
- The list length is variable, not fixed.
- The data in a list must have the same type.

In the tabulaton example that we used to start this chapter, the `List` is not a good choice for each entry in the table. Now, let's think about how you might actually use this interface in practice. Usually, when you have data to render in a table, the data entries are described more precisely by a record. So, imagine that you start off with a record type for representing information about a given programming language:

```
type style =
    Object_oriented | Functional | Imperative | Logic

type prog_lang = { name: string;
                   architect: string;
                   year_released: int;
                   style: style list;
                 }
```

If we then wanted to render a table from a list of languages, we might write something like this:

```
let print_langs langs =
   let headers = ["name";"architect";"year released"] in
   let to_row lang =
     [lang.name; lang.architect; Int.to_string lang.year_released ]
   in
   print_string (Text_table.render headers (List.map ~f:to_row langs))
```

This is OK, but as you consider more complicated tables with more columns, it becomes easier to make the mistake of having a mismatch in between `headers` and `to_row`. Also, adding, removing and reordering columns becomes awkward, because changes need to be made in two places.

We can improve the table API by adding a type that is a first-class representative for a column. We'd add the following to the interface of `Text_table`:

```
(** An ['a column] is a specification of a column for rending a table
    of values of type ['a] *)
type 'a column

(** [column header to_entry] returns a new column given a header and a
    function for extracting the text entry from the data associated
    with a row *)
val column : string -> ('a -> string) -> 'a column

(** [column_render columns rows] Renders a table with the specified
    columns and rows *)
val column_render :
  'a column list -> 'a list -> string
```

Thus, the `column` functions creates a `column` from a header string and a function for extracting the text for that column associated with a given row. Implementing this interface is quite simple:

```
type 'a column = string * ('a -> string)
let column header to_string = (header,to_string)

let column_render columns rows =
  let header = List.map columns ~f:fst in
  let rows = List.map rows ~f:(fun row ->
    List.map columns ~f:(fun (_,to_string) -> to_string row))
  in
  render header rows
```

And we can rewrite `print_langs` to use this new interface as follows.

```
let columns =
  [ Text_table.column "Name"      (fun x -> x.name);
    Text_table.column "Architect" (fun x -> x.architect);
    Text_table.column "Year Released"
       (fun x -> Int.to_string x.year_released);
  ]

let print_langs langs =
  print_string (Text_table.column_render columns langs)
```

The code is a bit longer, but it's also less error prone. In particular, several errors that might be made by the user are now ruled out by the type system. For example, it's no longer possible for the length of the header and the lengths of the rows to be mismatched.

The simple column-based interface described here is also a good starting for building a richer API. You could for example build specialized columns with different formatting and alignment rules, which is easier to do with this interface than with the original one based on passing in lists-of-lists.

Options

OCaml has no "NULL" or "nil" values. Programmers coming from other languages are often surprised and annoyed by this -- it seems really convenient to have a special `NULL` value that represents concepts like "end of list" or "leaf node in a tree." The possible benefit is that *every* pointer type has a extra NULL value; the problem is that using the NULL value as if it were a real value has weak or undefined semantics.

How do we get similar semantics in OCaml? The ubiquitous technique is to use the `option` type, which has the following definition.

```
type 'a option = None | Some of 'a;;
```

That is, a value of type `'a option` is either `None`, which means "no value;" or it is `Some v`, which represents a value `v`. There is nothing special about the `option` type -- it is a variant type just like any other. What it means is that checking for `None` is *explicit*, it is not possible to use `None` in a place where `Some x` is expected.

In the most direct form, we can use an `option` wherever some value is "optional," with the usual meaning. For example, if the architect of a programming language is not always known, we could use a special string like `"unknown"` to represent the architect's name, but we might accidentally confuse it with the name of a person. The more explicit alternative is to use an option.

```
type prog_lang = { name: string;
                   architect: string option;
                   year_released: int;
                   style: style list;
                 }

let x86 = { name = "x86 assembly";
            architect = None;
            year_released = 1980;
            style = Imperative
          };;
```

We can also represent a data structure with NULL-pointers using the `option` type. For example, let's build an imperative singly-linked list, where new values are added to the *end* of the list. In a standard imperative language (like in the C++ Standard Template Library), NULL is used to represent "end of list." We'll use the `option` type instead.

```
type 'a slist = { mutable head : 'a elem option; mutable tail : 'a elem option }
and 'a elem = { value : 'a; mutable next : 'a elem option };;

let new_slist () = { head = None; tail = None };;

let push_back l x =
  let elem = { value = x; next = None } in
  match l.tail with
   | None -> l.head <- Some elem; l.tail <- Some elem
   | Some last -> last.next <- Some elem;;
```

Similarly, if we're defining a type of binary trees, one choice is to use `option` for the child node references. In a binary search tree, each node in the tree is labeled with a value and it has up to two children. The nodes in the tree follow *prefix* order, meaning that the label of the left child is smaller than the label of its parent, and the label of the right child is larger than the label of the parent.

```
type 'a node = { label : 'a; left : 'a binary_tree; right : 'a binary_tree }
and 'a binary_tree = 'a node option;;

let new_binary_tree () : 'a binary_tree = None;;
let rec insert x = function
 | Some { label = label; left = left; right = right } as tree ->
   if x < label then
     Some { label = label; left = insert x left; right = right }
   else if x > label then
     Some { label = label; left = left; right = insert x right }
   else
     tree
 | None -> Some { label = x; left = None; right = None };;
```

This representation is perfectly adequate, but many OCaml programmers would prefer a representation where the `option` is "hoisted" to the `node` type, meaning that we have two kinds of nodes. In this case, the code is somewhat more succinct. In the end, of course, the two versions are isomorphic.

```
type 'a binary_tree =
 | Leaf
 | Interior of 'a * 'a binary_tree * 'a binary_tree;;

let new_binary_tree () : 'a binary_tree = Leaf;;

let rec insert x = function
 | Interior (label, left, right) as tree ->
   if x < label then Interior (label, insert x left, right)
   else if x > label then Interior (label, left, insert x right)
   else tree
 | Leaf -> Interior (x, Leaf, Leaf);;
```

2013-02-05
19:46:46

CHAPTER 5

Records

One of OCaml's best features is its concise and expressive system for declaring new datatypes. Two key elements of that system are *records* and *variants*, both of which we discussed briefly in Chapter 2. In this chapter we'll cover records in more depth, covering more of the details of how they work, as well as advice on how to use them effectively in your software designs.

A record represents a collection of values stored together as one, where each component is identified by a different field name. The basic syntax for a record type declaration is as follows.

```
type <record-name> =
  { <field-name> : <type-name> ;
    <field-name> : <type-name> ;
    ...
  }
```

Here's a simple example, a `host_info` record that summarizes information about a given computer.

```
# type host_info =
    { hostname   : string;
      os_name    : string;
      os_release : string;
      cpu_arch   : string;
    };;
```

We can construct a `host_info` just as easily. The following code uses the `Shell` module from `Core_extended` to dispatch commands to the shell to extract the information we need about the computer we're running on.

```
# open Core_extended.Std;;
# let my_host = { hostname   = Shell.sh_one "hostname";
                  os_name    = Shell.sh_one "uname -s";
                  os_release = Shell.sh_one "uname -r";
                  cpu_arch   = Shell.sh_one "uname -p";
```

```
                };;
val my_host : host_info =
  {hostname = "Yarons-MacBook-Air.local"; os_name = "Darwin";
   os_release = "11.4.0"; cpu_arch = "i386"}
```

You might wonder how the compiler inferred that `my_host` is of type `host_info`. The hook that the compiler uses in this case to figure out the type is the record field names. It turns out that, within a given scope, each record field name is associated with a unique record type. Later in the chapter, we'll talk about what to do when you want to have the same record fields in multiple records.

Once we have a record value in hand, we can extract elements from the record field using dot-notation.

```
# my_host.cpu_arch;;
- : string = "i386"
```

Patterns and exhaustiveness

Another way of getting information out of a record is by using a pattern match, as in the definition of `host_info_to_string` below.

```
# let host_info_to_string { hostname = h; os_name = os;
                            os_release = r; cpu_arch = c } =
       sprintf "%s (%s %s / %s)" h os r c;;
    val host_info_to_string : host_info -> string = <fun>
# host_info_to_string my_host;;
- : string = "Yarons-MacBook-Air.local (Darwin 11.4.0 / i386)"
```

Note that the pattern that we used had only a single case, rather than using several cases separated by |s. We only needed a single pattern because record patterns are *irrefutable*, meaning that, because the layout of a record is always the same, a record pattern match will never fail at runtime. In general, types with a fixed structure, like records and tuples, have irrefutable patterns, whereas types with variable structure, like lists and variants, do not.

Another important characteristic of record patterns is that they don't need to be complete; a pattern can mention only a subset of the fields in the record. This can be convenient, but it's can also be error prone. In particular, this means that when new fields are added to the record, code that should be updated to react to the presence of those new fields will not be flagged by the compiler.

As an example, imagine that we wanted to add a new field to our `host_info` record called `os_version`, as shown below.

```
# type host_info =
    { hostname   : string;
      os_name    : string;
      os_release : string;
```

```
   cpu_arch   : string;
   os_version : string;
  };;
```

The code for `host_info_to_string` would continue to compile without change. In this particular case, it's pretty clear that you might want to update `host_info_to_string` in order to take into account the new field, and it would be nice if the type system would give you a warning about the change.

Happily, OCaml does offer an optional warning for missing fields in a record pattern. With that warning turned on (which you can do in the toplevel by typing `#warnings "+9"`), the compiler will warn about the missing field.

```
# warnings "+9";;
# let host_info_to_string { hostname = h; os_name = os;
                            os_release = r; cpu_arch = c } =
       sprintf "%s (%s %s / %s)" h os r c;;
    Characters 24-112:
  ........................{ hostname = h; os_name = os;
                              os_release = r; cpu_arch = c }..
Warning 9: the following labels are not bound in this record pattern:
os_version
Either bind these labels explicitly or add '; _' to the pattern.
val host_info_to_string : host_info -> string = <fun>
```

We can disable the warning for a given pattern by explicitly acknowledging that we are ignoring extra fields. This is done by adding an underscore to the pattern, as shown below.

```
# let host_info_to_string { hostname = h; os_name = os;
                            os_release = r; cpu_arch = c; _ } =
       sprintf "%s (%s %s / %s)" h os r c;;
    val host_info_to_string : host_info -> string = <fun>
```

Generally, the right default is to turn the warning for incomplete record matches on, and to explicitly disable it with an _ where necessary.

Field punning

When the name of a variable coincides with the name of a record field, OCaml provides some handy syntactic shortcuts. For example, the pattern in the following function binds all of the fields in question to variables of the same name. This is called *field punning*.

```
# let host_info_to_string { hostname; os_name; os_release; cpu_arch } =
     sprintf "%s (%s %s / %s)" hostname os_name os_release cpu_arch;;
  val host_info_to_string : host_info -> string = <fun>
```

Field punning can also be used to construct a record. Consider the following code for generating a `host_info` record.

```
# let my_host =
    let hostname   = Shell.sh_one "hostname" in
    let os_name    = Shell.sh_one "uname -s" in
    let os_release = Shell.sh_one "uname -r" in
    let cpu_arch   = Shell.sh_one "uname -p" in
    { hostname; os_name; os_release; cpu_arch };;
val my_host : host_info =
  {hostname = "Yarons-MacBook-Air.local"; os_name = "Darwin";
   os_release = "11.4.0"; cpu_arch = "i386"}
```

In the above code, we defined variables corresponding to the record fields first, and then the record declaration itself simply listed the fields that needed to be included.

You can take advantage of both field punning and label punning when writing a function for constructing a record from labeled arguments, as shown below.

```
# let create_host_info ~hostname ~os_name ~os_release ~cpu_arch =
    let hostname = String.lowercase hostname in
    { hostname; os_name; os_release; cpu_arch };;
```

This is considerably more concise than what you would get without punning at all.

```
let create_host_info ~hostname:hostname ~os_name:os_name
   ~os_release:os_release ~cpu_arch:cpu_arch =
    let hostname = String.lowercase hostname in
    { hostname = hostname ; os_name = os_name;
      os_release = os_release; cpu_arch = cpu_arch };;
```

Together, labeled arguments, field names, and field and label punning, encourage a style where you propagate the same names throughout your code-base. This is generally good practice, since it encourages consistent naming, which makes it easier for new people to navigate your source.

Reusing field names

Defining records with the same field names can be problematic. Let's consider a simple example: building types to represent the protocol used for a logging server. The following types represent messages a server might receive from a client.

Below, the `log_entry` message is used to deliver a log entry to the server for processing. The `logon` message is sent when a client initiates a connection, and includes the identity of the user connecting and credentials used for authentication. Finally, the `heartbeat` message is periodically sent by the client to demonstrate to the server that the client is alive and connected. All of these messages include a session id and the time the message was generated.

```
# type log_entry =
    { session_id: string;
      time: Time.t;
```

2013-02-05
19:46:46

```
      important: bool;
      message: string;
    }
  type heartbeat =
    { session_id: string;
      time: Time.t;
      status_message: string;
    }
  type logon =
    { session_id: string;
      time: Time.t;
      user: string;
      credentials: string;
    }
;;
```

The fact that we reused field names will cause trouble when we try to construct a message.

```
# let create_log_entry ~session_id ~important message =
     { time = Time.now (); session_id; important; message }
  ;;
    Characters 75-129:
       { time = Time.now (); session_id; important; message }
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: The record field label important belongs to the type log_entry
       but is mixed here with labels of type logon
```

The problem is that the declaration of `logon` (and `heartbeat`) shadowed some of the fields of `log_entry`. As a result, the fields `time` and `session_id` are assumed to be fields of `logon`, and `important` and `message`, which were not shadowed, are assumed to be fields of `log_entry`. The compiler therefore complains that we're trying to construct a record with fields from two different record types.

There are two common solutions to this problem. The first is to add a prefix to each field name to make it unique, as shown below.

```
# type log_entry =
    { log_entry_session_id: string;
      log_entry_time: Time.t;
      log_entry_important: bool;
      log_entry_message: string;
    }
  type heartbeat =
    { heartbeat_session_id: string;
      heartbeat_time: Time.t;
      heartbeat_status_message: string;
    }
  type logon =
    { logon_session_id: string;
      logon_time: Time.t;
      logon_user: string;
      logon_credentials: string;
    }
;;
```

This eliminates the collisions and is simple enough to do. But it leaves you with awkwardly named record fields, and adds needless repetition and verbosity to your code.

Another approach is to mint a module for each type. This is actually a broadly useful idiom, providing for each type a namespace within which to put related values. Using this style we would write:

```
# module Log_entry = struct
    type t =
      { session_id: string;
        time: Time.t;
        important: bool;
        message: string;
      }
  end
  module Heartbeat = struct
    type t =
      { session_id: string;
        time: Time.t;
        status_message: string;
      }
  end
  module Logon = struct
    type t =
      { session_id: string;
        time: Time.t;
        user: string;
        credentials: string;
      }
  end;;
```

Now, our heartbeat-creation function can be rendered as follows.

```
# let create_log_entry ~session_id ~important message =
     { Log_entry.time = Time.now (); Log_entry.session_id;
       Log_entry.important; Log_entry.message }
  ;;
val create_log_entry :
  session_id:string -> important:bool -> string -> Log_entry.t = <fun>
```

The module name `Log_entry` is required to qualify the fields, because this function is outside of the `Log_entry` module where the record was defined. OCaml only requires the module qualification for one record field, however, so we can write this more concisely.

```
# let create_log_entry ~session_id ~important message =
     { Log_entry. time = Time.now (); session_id; important; message }
  ;;
val create_log_entry :
  session_id:string -> important:bool -> string -> Log_entry.t = <fun>
```

For functions defined within the module where a given record is defined, the module qualification goes away entirely. And indeed, for things like constructors, defining it within the module is often the best solution.

Functional updates

Fairly often, you will find yourself wanting to create a new record that differs from an existing record in only a subset of the fields. For example, imagine our logging server had a record type for representing the state of a given client, including when the last heartbeat was received from that client. The following defines a type for representing this information, as well as a function for updating the client information when a new heartbeat arrives.

```
# type client_info =
   { addr: Unix.Inet_addr.t;
     port: int;
     user: string;
     credentials: string;
     last_heartbeat_time: Time.t;
   };;
# let register_heartbeat t hb =
      { addr = t.addr;
        port = t.port;
        user = t.user;
        credentials = t.credentials;
        last_heartbeat_time = hb.Heartbeat.time;
      };;
val register_heartbeat : client_info -> Heartbeat.t -> client_info = <fun>
```

This is fairly verbose, given that there's only one field that we actually want to change, and all the others are just being copied over from `t`. We can use OCaml's *functional update* syntax to do this more tersely. The syntax of a functional update is as follows.

```
{ <record-value> with <field> = <value>;
                      <field> = <value>;
                      ...
}
```

The purpose of the functional update is to create a new record based on an existing one, with a set of field changes layered on top.

Given this, we can rewrite `register_heartbeat` more concisely.

```
# let register_heartbeat t hb =
    { t with last_heartbeat_time = hb.Heartbeat.time };;
```

Functional updates make your code independent of the identity of the fields in the record that are not changing. This is often what you want, but it has downsides as well. In particular, if you change the definition of your record to have more fields, the type system will not prompt you to reconsider whether your update code should affect those fields. Consider what happens if we decided to add a field for the status message received on the last heartbeat.

```
# type client_info =
   { addr: Unix.Inet_addr.t;
     port: int;
     user: string;
     credentials: string;
     last_heartbeat_time: Time.t;
     last_heartbeat_status: string;
   };;
```

The original implementation of `register_heartbeat` would now be invalid, and thus the compiler would warn us to think about how to handle this new field. But the version using a functional update continues to compile as is, even though it incorrectly ignores the new field. The correct thing to do would be to update the code as follows.

```
# let register_heartbeat t hb =
    { t with last_heartbeat_time   = hb.Heartbeat.time;
             last_heartbeat_status = hb.Heartbeat.status_message;
    };;
```

Mutable fields

Like most OCaml values, records are immutable by default. You can, however, declare individual record fields as mutable. For example, we could take the `client_info` type and make the fields that may need to change over time mutable, as follows.

```
# type client_info =
   { addr: Unix.Inet_addr.t;
     port: int;
     user: string;
     credentials: string;
     mutable last_heartbeat_time: Time.t;
     mutable last_heartbeat_status: string;
   };;
```

We then use the `<-` operator for actually changing the state. The side-effecting version of `register_heartbeat` would be written as follows.

```
# let register_heartbeat t hb =
    t.last_heartbeat_time   <- hb.Heartbeat.time;
    t.last_heartbeat_status <- hb.Heartbeat.status_message
  ;;
val register_heartbeat : client_info -> Heartbeat.t -> unit = <fun>
```

Note that `<-` is not needed for initialization, because all fields of a record, including mutable ones, must be specified when the record is created.

OCaml's policy of immutable-by-default is a good one, but imperative programming does have its place. We'll discuss more about how (and when) to use OCaml's imperative features in “Imperative programming” on page 20.

First-class fields

Consider the following function for extracting the usernames from a list of `Logon` messages.

```
# let get_users logons =
     List.dedup (List.map logons ~f:(fun x -> x.Logon.user));;
  val get_hostnames : Logon.t list -> string list = <fun>
```

Here, we wrote a small function `(fun x -> x.Logon.user)` to access the `user` field. This kind of accessor function is a common enough pattern that that it would be convenient to generate them automatically. The `fieldslib` syntax extension that ships with `Core` does just that.

`fieldslib` is invoked by putting the `with fields` annotation at the end of the declaration of a record type. So, for example, we could have defined `Logon` as follows.

```
# module Logon = struct
    type t =
      { session_id: string;
        time: Time.t;
        user: string;
        credentials: string;
      }
    with fields
  end;;
```

Given that definition, we can use the function `Logon.user` to extract the user field from a logon message.

```
# let get_users logons = List.dedup (List.map logons ~f:Logon.user);;
val get_users : Logon.t list -> string list = <fun>
```

In addition to generating field accessor functions, `fieldslib` also creates a sub-module called `Fields` that contains a first class representative of each field, in the form of a value of type `Field.t`. A `Field.t` bundles up the following functionality of a record filed:

- The name of the field as a string
- The ability to extract the field
- The ability to do a functional update of that field
- The (optional) ability to set the record field, which is present only if the field is mutable.

We can use these first class fields to do things like write a generic function for displaying a record field. The function `show_field` takes three arguments: the `Field.t`, a function for converting the contents of the field in question to a string, and the record type.

```
# let show_field field to_string record =
     sprintf "%s: %s" (Field.name field) (Field.get field record |! to_string);;
val show_field : ('a, 'b) Field.t -> ('b -> string) -> 'a -> string =
  <fun>
```

Here's an example of `show_field` in action.

```
# let logon = { Logon.
                session_id = "26685";
                time = Time.now ();
                user = "yminsky";
                credentials = "Xy2d9W"; }
  ;;
# show_field Logon.Fields.user Fn.id logon;;
- : string = "user: yminsky"
# show_field Logon.Fields.time Time.to_string logon;;
- : string = "time: 2012-06-26 18:44:13.807826"
```

`fieldslib` also provides higher-level operators, like `Fields.fold` and `Fields.iter`, which let you iterate over all the fields of a record. The following function uses `Logon.Fields.iter` and `show_field` to print out all the fields of a `Logon` record.

```
# let print_logon logon =
    let print to_string field =
      printf "%s\n" (show_field field to_string logon)
    in
    Logon.Fields.iter
      ~session_id:(print Fn.id)
      ~time:(print Time.to_string)
      ~user:(print Fn.id)
      ~credentials:(print Fn.id)
  ;;
val print_logon : Logon.t -> unit = <fun>
# print_logon logon;;
session_id: 26685
time: 2012-06-26 18:44:13.807826
user: yminsky
credentials: Xy2d9W
- : unit = ()
```

The advantage of using field iterators is that when the definition of `Logon` changes, `iter` will change along with it, prompting you to handle whatever new cases arise.

Field iterators are useful for a variety of tasks, from building validation functions to scaffolding the definition of a web-form based on a record type, all with a guarantee that you've exhaustively considered all elements of the field.

CHAPTER 6
Variants

Variant types are one of the most useful features of OCaml, and also one of the most unusual. They let you represent data that may take on multiple different forms, where each form is marked by an explicit tag. As we'll see, when combined with pattern matching, variants give you a powerful way of representing complex data and of organizing the case-analysis on that information.

Let's consider a concrete example of how variants can be useful. Almost all terminals support a set of 8 basic colors, and we can represent those colors using a variant. Each color is declared as a simple tag, with pipes used to separate the different cases.

```
# type basic_color =
    Black | Red | Green | Yellow | Blue | Magenta | Cyan | White ;;
# Cyan ;;
- : basic_color = Cyan
# [Blue; Magenta; Red] ;;
- : basic_color list = [Blue; Magenta; Red]
```

The following function uses pattern matching to convert a `basic_color` to a corresponding integer. Note that the exhaustiveness checking on pattern matches means that the compiler will warn us if we miss a color.

```
# let basic_color_to_int = function
  | Black -> 0 | Red     -> 1 | Green -> 2 | Yellow -> 3
  | Blue  -> 4 | Magenta -> 5 | Cyan  -> 6 | White  -> 7 ;;
val basic_color_to_int : basic_color -> int = <fun>
# List.map ~f:basic_color_to_int [Blue;Red];;
- : int list = [4; 1]
```

Using the above, we can generate escape codes to change the color of a given string displayed in a terminal.

```
# let color_by_number number text =
    sprintf "\027[38;5;%dm%s\027[0m" number text;;
  val color_by_number : int -> string -> string = <fun>
# let blue = color_by_number (basic_color_to_int Blue) "Blue";;
```

```
val blue : string = "\027[38;5;4mBlue\027[0m"
# printf "Hello %s World!\n" blue;;
Hello Blue World!
- : unit = ()
```

On most terminals, that word "Blue" will be rendered in blue.

In this example, the cases of the variant are simple tag with no associated data. This is substantively the same as the enumerations found in languages like C and Java. But as we'll see, variants can do considerably more than represent a simple enumeration. Indeed, an enumeration isn't enough to effectively describe the full set of colors that a modern terminal can display. Many terminals, including the venerable `xterm`, support 256 different colors, broken up into the following groups.

- The 8 basic colors, in regular and bold versions.
- A 6 × 6 × 6 RGB color cube
- A 24-level grayscale ramp

We'll also represent this more complicated color-space as a variant, but this time, the different tags will have arguments which describe the data available in each case.

```
# type weight = Regular | Bold
  type color =
  | Basic of basic_color * weight (* basic colors, regular and bold *)
  | RGB   of int * int * int       (* 6x6x6 color cube *)
  | Gray  of int                   (* 24 grayscale levels *)
;;
# [RGB (250,70,70); Basic (Green, Regular)];;
- : color list = [RGB (250, 70, 70); Basic (Green, Regular)]
```

Once again, we'll use pattern matching to convert a color to a corresponding integer. But in this case, the pattern matching does more than separate out the different cases; it also allows us to extract the data associated with each tag.

```
# let color_to_int = function
    | Basic (basic_color,weight) ->
      let base = match weight with Bold -> 8 | Regular -> 0 in
      base + basic_color_to_int basic_color
    | RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | Gray i -> 232 + i ;;
val color_to_int : color -> int = <fun>
```

Now, we can print text using the full set of available colors.

```
# let color_print color s =
     printf "%s\n" (color_by_number (color_to_int color) s);;
val color_print : color -> string -> unit = <fun>
# color_print (Basic (Red,Bold)) "A bold red!";;
A bold red!
- : unit = ()
# color_print (Gray 4) "A muted gray...";;
```

```
A muted gray...
- : unit = ()
```

Catch-all cases and refactoring

OCaml's type system can act as a refactoring tool, by warning you of places where your code needs to be updated to match an interface change. This is particularly valuable in the context of variants.

Consider what would happen if we were to change the definition of `color` to the following.

```
# type color =
  | Basic of basic_color     (* basic colors *)
  | Bold  of basic_color     (* bold basic colors *)
  | RGB   of int * int * int (* 6x6x6 color cube *)
  | Gray  of int             (* 24 grayscale levels *)
;;
```

We've essentially broken out the `Basic` case into two cases, `Basic` and `Bold`, and `Basic` has changed from having two arguments to one. `color_to_int` as we wrote it still expects the old structure of the variant, and if we try to compile that same code again, the compiler will notice the discrepancy.

```
# let color_to_int = function
    | Basic (basic_color,weight) ->
      let base = match weight with Bold -> 8 | Regular -> 0 in
      base + basic_color_to_int basic_color
    | RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | Gray i -> 232 + i ;;
Characters 40-60:
Error: This pattern matches values of type 'a * 'b
       but a pattern was expected which matches values of type basic_color
```

Here, the compiler is complaining that the `Basic` tag is used with the wrong number of arguments. If we fix that, however, the compiler flag will flag a second problem, which is that we haven't handled the new `Bold` tag.

```
# let color_to_int = function
    | Basic basic_color -> basic_color_to_int basic_color
    | RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | Gray i -> 232 + i ;;
Characters 19-154:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Bold _
val color_to_int : color -> int = <fun>
```

Fixing this now leads us to the correct implementation.

```
# let color_to_int = function
    | Basic basic_color -> basic_color_to_int basic_color
    | Bold  basic_color -> 8 + basic_color_to_int basic_color
    | RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | Gray i -> 232 + i ;;
val color_to_int : color -> int = <fun>
```

As we've seen, the type errors identified the things that needed to be fixed to complete the refactoring of the code. This is fantastically useful, but for it to work well and reliably, you need to write your code in a way that maximizes the compiler's chances of helping you find the bugs. One important rule of thumb to follow to maximize what the compiler can do for you is to avoid catch-all cases in pattern matches.

Here's an example of how a catch-all case plays in. Imagine we wanted a version of `color_to_int` that works on older terminals by rendering the first 16 colors (the 8 `basic_color`s in regular and bold) in the normal way, but rendering everything else as white. We might have written the function as follows.

```
# let oldschool_color_to_int = function
    | Basic (basic_color,weight) ->
      let base = match weight with Bold -> 8 | Regular -> 0 in
      base + basic_color_to_int basic_color
    | _ -> basic_color_to_int White;;
val oldschool_color_to_int : color -> int = <fun>
```

But because the catch-all case encompasses all possibilities, the type system will no longer warn us that we have missed the new `Bold` case when we change the type to include it. We can get this check back by being more explicit about what we're ignoring. We haven't changed the behavior of the code, but we have improved our robustness to change.

Combining records and variants

Records and variants are most effective when used in concert. Consider again the type `Log_entry.t` Chapter 5:

```
module Log_entry = struct
  type t =
    { session_id: string;
      time: Time.t;
      important: bool;
      message: string;
    }
end
```

This record type combines multiple pieces of data into one value. In particular, a single `Log_entry.t` has a `session_id` *and* a `time` *and* an `important` flag *and* a `message`. More generally, you can think of record types as acting as conjunctions. Variants, on the other hand, are disjunctions, letting you represent multiple possibilities, as in the following example.

```
type client_message = | Logon of Logon.t
```

```
                          | Heartbeat of Heartbeat.t
                          | Log_entry of Log_entry.t
```

A `client_message` is a `Logon` *or* a `Heartbeat` *or* a `Log_entry`. If we want to write code that processes messages generically, rather than code specialized to a fixed message type, we need something like `client_message` to act as one overarching type for the different possible messages.

You can increase the precision of your types by using variants to represent differences between types, and records to represent shared structure. Consider the following function that takes a list of `client_message`s and returns all messages generated by a given user. The code in question is implemented by folding over the list of messages, where the accumulator is a pair of:

- the set of session identifiers for the user that have been seen thus far.
- the set of messages so far that are associated with the user.

Here's the concrete code.

```
let messages_for_user user messages =
  let (user_messages,_) =
    List.fold messages ~init:([],String.Set.empty)
      ~f:(fun ((messages,user_sessions) as acc) message ->
        match message with
        | Logon m ->
          if m.Logon.user = user then
            (message::messages, Set.add user_sessions m.Logon.session_id)
          else acc
        | Heartbeat _ | Log_entry _ ->
          let session_id = match message with
            | Logon     m -> m.Logon.session_id
            | Heartbeat m -> m.Heartbeat.session_id
            | Log_entry m -> m.Log_entry.session_id
          in
          if Set.mem user_sessions session_id then
            (message::messages,user_sessions)
          else acc
      )
  in
  List.rev user_messages
```

There's one awkward bit about the code above, which is the calculation of the session ids. In particular, we have the following repetitive snippet of code:

```
let session_id = match message with
  | Logon     m -> m.Logon.session_id
  | Heartbeat m -> m.Heartbeat.session_id
  | Log_entry m -> m.Log_entry.session_id
in
```

This code effectively computes the session id for each underlying message type. The repetition in this case isn't that bad, but would become problematic in larger and more complicated examples.

We can improve the code by refactoring our types to explicitly separate which parts are shared and which are common. The first step is to cut down the definitions of the per-message records to just contain the unique components of each message.

```
module Log_entry = struct
  type t = { important: bool;
             message: string;
           }
end

module Heartbeat = struct
  type t = { status_message: string; }
end

module Logon = struct
  type t = { user: string;
             credentials: string;
           }
end
```

We can then define a variant type that covers the different possible unique components.

```
type details =
| Logon of Logon.t
| Heartbeat of Heartbeat.t
| Log_entry of Log_entry.t
```

Separately, we need a record that contains the fields that are common across all messages.

```
module Common = struct
  type t = { session_id: string;
             time: Time.t;
           }
end
```

A full message can then represented as a pair of a `Common.t` and a `details`. Using this, we can rewrite our example above as follows:

```
let messages_for_user user messages =
  let (user_messages,_) =
    List.fold messages ~init:([],String.Set.empty)
      ~f:(fun ((messages,user_sessions) as acc) ((common,details) as message) ->
        let session_id = common.Common.session_id in
        match details with
        | Logon m ->
          if m.Logon.user = user then
            (message::messages, Set.add user_sessions session_id)
            else acc
          | Heartbeat _ | Log_entry _ ->
            if Set.mem user_sessions session_id then
              (message::messages,user_sessions)
            else acc
        )
    in
    List.rev user_messages
```

Note that the more complex match statement for computing the session id has been replaced with the simple expression `common.Common.session_id`.

In addition, this design allows us to essentially downcast to the specific message type once we know what it is, and then dispatch code to handle just that message type. In particular, while we use the type `Common.t * details` to represent an arbitrary message, we can use `Common.t * Logon.t` to represent a logon message. Thus, if we had functions for handling individual message types, we could write a dispatch function as follows.

```
let handle_message server_state (common,details) =
  match details with
  | Log_entry m -> handle_log_entry server_state (common,m)
  | Logon     m -> handle_logon     server_state (common,m)
  | Heartbeat m -> handle_heartbeat server_state (common,m)
```

And it's explicit at the type level that `handle_log_entry` sees only `Log_entry` messages, `handle_logon` sees only `Logon` messages, etc.

Variants and recursive data structures

Another common application of variants is to represent tree-like recursive data-structures. We'll show how this can be done by walking through the design of a simple Boolean expression language. Such a language can be useful anywhere you need to specify filters, which are used in everything from packet analyzers to mail clients.

An expression in this language will be defined by the variant `blang` (short for "boolean language") with one tag for each kind of expression we want to support.

```
# type 'a blang =
  | Base  of 'a
  | Const of bool
  | And   of 'a blang list
  | Or    of 'a blang list
  | Not   of 'a blang
  ;;
```

Note that the definition of the type `blang` is recursive, meaning that a `blang` may contain other `blang`s.

The purpose of each tag is pretty straightforward. `And`, `Or` and `Not` are the basic operators for building up boolean expression, and `Const` lets you enter constants `true` and `false`.

The `Base` tag is what allows you to tie the `blang` to your application, by letting you specify an element of some base predicate type, whose whose truth or falsehood is determined by your application. If you were writing a filter language for an email processor, your base predicates might specify the tests you would run against an email, as in the following example.

```
# type mail_field = To | From | CC | Date | Subject
  type mail_predicate = { field: mail_field;
                          contains: string }
  ;;
```

Using the above, we can construct a simple expression with `mail_predicate` as its base predicate.

```
# let test field contains = Base { field; contains };;
val test : mail_field -> string -> mail_predicate blang = <fun>
# And [ Or [ test To "doligez"; test CC "doligez" ];
        test Subject "runtime";
      ]
  ;;
- : mail_predicate blang =
And
 [Or
   [Base {field = To; contains = "doligez"};
    Base {field = CC; contains = "doligez"}];
  Base {field = Subject; contains = "runtime"}]
```

Being able to construct such expressions isn't enough: we also need to be able to evaluate such an expression. The following code shows how you could write a general-purpose evaluator for `blang`'s.

```
# let rec eval blang base_eval =
    (* a shortcut, so we don't need to repeatedly pass [base_eval]
       explicitly to [eval] *)
    let eval' blang = eval blang base_eval in
    match blang with
    | Base  base   -> base_eval base
    | Const bool   -> bool
    | And   blangs -> List.for_all blangs ~f:eval'
    | Or    blangs -> List.exists  blangs ~f:eval'
    | Not   blang  -> not (eval' blang)
  ;;
val eval : 'a blang  -> ('a -> bool) -> bool = <fun>
```

The structure of the code is pretty straightforward --- we're just pattern-matching over the structure of the data, doing the appropriate calculation based on which tag we see. To use this evaluator on a concrete example, we just need to write the `base_eval` function which is capable of evaluating a base predicate.

Another useful operation to be able to do on expressions is simplification. The following function applies some basic simplification rules, most of the simplifications being driven by the presence of constants.

```
# let rec simplify = function
    | Base _ | Const _ as x -> x
    | And blangs ->
      let blangs = List.map ~f:simplify blangs in
      if List.exists blangs ~f:(function Const false -> true | _ -> false)
      then Const false
      else And blangs
    | Or blangs ->
      let blangs = List.map ~f:simplify blangs in
      if List.exists blangs ~f:(function Const true -> true | _ -> false)
      then Const true else Or blangs
    | Not blang ->
      match simplify blang with
      | Const bool -> Const (not bool)
      | blang -> Not blang
  ;;
val simplify : 'a blang -> 'a blang = <fun>
```

One thing to notice about the above code is that it uses a catch-all case in the very last line within the `Not` case. It's generally better to be explicit about the cases you're ignoring. Indeed, if we change this snippet of code to be more explicit:

```
    | Not blang ->
      match simplify blang with
      | Const bool -> Const (not bool)
      | (And _ | Or _ | Base _ | Not _) -> Not blang
```

it's easy to see that we've missed an important case: double-negation.

```
    | Not blang ->
      match simplify blang with
      | Const b -> Const (not b)
      | Not blang -> blang
      | (And _ | Or _ | Base _ ) -> Not blang
```

This example is more than a toy. There's a module very much in this spirit already exists as part of Core, and gets a lot of practical use in a variety of applications.

More generally, using variants to build recursive data-structures is a common technique, and shows up everywhere from designing little languages to building efficient data-structures.

Polymorphic variants

In addition to the ordinary variants we've seen so far, OCaml also supports so-called *polymorphic variants*. As we'll see, polymorphic variants are more flexible and syntactically more lightweight than ordinary variants, but that extra power comes at a cost.

Syntactically, polymorphic variants are distinguished from ordinary variants by the leading backtick. And unlike ordinary variants, polymorphic variants can be used without an explicit type declaration.

```
# let three = `Int 3;;
val three : [> `Int of int ] = `Int 3
# let four = `Float 4.;;
val four : [> `Float of float ] = `Float 4.
# let nan = `Not_a_number;;
val nan : [> `Not_a_number ] = `Not_a_number
# [three; four; nan];;
- : [> `Float of float | `Int of int | `Not_a_number ] list =
[`Int 3; `Float 4.; `Not_a_number]
```

As you can see, polymorphic variant types are inferred automatically, and when we combine variants with different tags, the compiler infers a new type that knows about all of those tags.

The type system will complain, however, if it sees incompatible uses of the same tag:

```
# let five = `Int "five";;
val five : [> `Int of string ] = `Int "five"
# [three; four; five];;
Characters 14-18:
  [three; four; five];;
                ^^^^
Error: This expression has type [> `Int of string ]
       but an expression was expected of type
         [> `Float of float | `Int of int ]
       Types for tag `Int are incompatible
```

The `>` at the beginning of the variant types above is critical, because it marks the types as being open to combination with other variant types. We can read the type ``[> `Int of string | `Float of float]`` as describing a variant whose tags include `` `Int of string `` and `` `Float of float ``, but may include more tags as well. In other words, you can roughly translate `>` to mean: "these tags or more".

OCaml will in some cases infer a variant type with `<`, to indicate "these tags or less", as in the following example.

```
# let is_positive = function
     | `Int   x -> x > 0
     | `Float x -> x > 0.
  ;;
val is_positive : [< `Float of float | `Int of int ] -> bool = <fun>
```

The `<` is there because `is_positive` has no way of dealing with values that have tags other than `` `Float of float `` or `` `Int of int ``.

We can think of these `<` and `>` markers as indications of upper and lower bounds on the tags involved. If the same set of tags are both an upper and a lower bound, we end up with an *exact* polymorphic variant type, which has neither marker. For example:

```
# let exact = List.filter ~f:is_positive [three;four];;
val exact : [ `Float of float | `Int of int ] list
   = [`Int 3; `Float 4.]
```

Perhaps surprisingly, we can also create polymorphic variant types that have different upper and lower bounds.

```
let is_positive = function
     | `Int   x -> Ok (x > 0)
     | `Float x -> Ok (x > 0.)
     | `Not_a_number -> Error "not a number";;
val is_positive :
  [< `Float of float | `Int of int | `Not_a_number ] ->
  (bool, string) Result.t = <fun>
# List.filter [three; four] ~f:(fun x ->
     match is_positive x with Error _ -> false | Ok b -> b);;
  - : [< `Float of float | `Int of int | `Not_a_number > `Float `Int ]
    List.t
= [`Int 3; `Float 4.]
```

Here, the inferred type states that the tags can be no more than `` `Float ``, `` `Int `` and `` `Not_a_number ``, and must contain at least `` `Float `` and `` `Int ``. As you can already start to see, polymorphic variants can lead to fairly complex inferred types.

Example: Terminal colors redux

To see how to use polymorphic variants in practice, we'll return to terminal colors. Imagine that we have a new terminal type that adds yet more colors, say, by adding an alpha channel so you can specify translucent colors. We could model this extended set of colors as follows, using an ordinary variant.

```
# type extended_color =
  | Basic of basic_color * weight  (* basic colors, regular and bold *)
  | RGB   of int * int * int       (* 6x6x6 color space *)
  | Gray  of int                   (* 24 grayscale levels *)
  | RGBA  of int * int * int * int (* 6x6x6x6 color space *)
  ;;
```

We want to write a function `extended_color_to_int`, that works like `color_to_int` for all of the old kinds of colors, with new logic only for handling colors that include an alpha channel. One might try to write such a function as follows.

```
# let extended_color_to_int = function
    | RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
    | (Basic _ | RGB _ | Gray _) as color -> color_to_int color
  ;;
```

This looks reasonable enough, but it leads to the following type error.

```
Characters 93-98:
    | (Basic _ | RGB _ | Gray _) as color -> color_to_int color
                                                          ^^^^^
Error: This expression has type extended_color
       but an expression was expected of type color
```

The problem is that `extended_color` and `color` are in the compiler's view distinct and unrelated types. The compiler doesn't, for example, recognize any equality between the `Basic` tag in the two types.

What we want to do is to share tags between two different variant types, and polymorphic variants let us do this in a natural way. First, let's rewrite `basic_color_to_int` and `color_to_int` using polymorphic variants. The translation here is pretty straightforward.

```
# let basic_color_to_int = function
    | `Black -> 0 | `Red     -> 1 | `Green -> 2 | `Yellow -> 3
    | `Blue  -> 4 | `Magenta -> 5 | `Cyan  -> 6 | `White  -> 7

  let color_to_int = function
    | `Basic (basic_color,weight) ->
      let base = match weight with `Bold -> 8 | `Regular -> 0 in
      base + basic_color_to_int basic_color
    | `RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | `Gray i -> 232 + i
 ;;
val basic_color_to_int :
  [< `Black | `Blue | `Cyan | `Green | `Magenta | `Red | `White | `Yellow ] ->
  int = <fun>
val color_to_int :
  [< `Basic of
       [< `Black | `Blue | `Cyan | `Green | `Magenta | `Red
        | `White | `Yellow ] *
       [< `Bold | `Regular ]
   | `Gray of int
   | `RGB of int * int * int ] ->
  int = <fun>
```

Now we can try writing `extended_color_to_int`. The key issue with this code is that `extended_color_to_int` needs to invoke `color_to_int` with a narrower type, *i.e.*, one that includes fewer tags. Written properly, this narrowing can be done via a pattern match. In particular, in the following code, the type of the variable `color` includes only the tags `` `Basic ``, `` `RGB `` and `` `Gray ``, and not `` `RGBA ``.

```
# let extended_color_to_int = function
    | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
    | (`Basic _ | `RGB _ | `Gray _) as color -> color_to_int color
  ;;
val extended_color_to_int :
  [< `Basic of
       [< `Black | `Blue | `Cyan | `Green | `Magenta | `Red
        | `White | `Yellow ] *
       [< `Bold | `Regular ]
   | `Gray of int
   | `RGB of int * int * int
   | `RGBA of int * int * int * int ] ->
  int = <fun>
```

The above code is more delicately balanced than one might imagine. In particular, if we use a catch-all case instead of an explicit enumeration of the cases, the type is no longer narrowed, and so compilation fails.

```
# let extended_color_to_int = function
    | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
    | color -> color_to_int color
  ;;
      Characters 125-130:
      | color -> color_to_int color
                              ^^^^^
Error: This expression has type [> `RGBA of int * int * int * int ]
       but an expression was expected of type
         [< `Basic of
              [< `Black | `Blue | `Cyan | `Green | `Magenta | `Red
               | `White | `Yellow ] *
              [< `Bold | `Regular ]
          | `Gray of int
          | `RGB of int * int * int ]
       The second variant type does not allow tag(s) `RGBA
```

Polymorphic variants and catch-all cases

As we saw with the definition of `is_positive`, a match statement can lead to the inference of an upper bound on a variant type, limiting the possible tags to those that can be handled by the match. If we add a catch-all case to our match statement, we end up with a function with a lower bound.

```
# let is_positive_permissive = function
     | `Int   x -> Ok (x > 0)
     | `Float x -> Ok (x > 0.)
     | _ -> Error "Unknown number type"
  ;;
val is_positive_permissive :
  [> `Float of float | `Int of int ] -> (bool, string) Core.Std._result =
  <fun>
# is_positive_permissive (`Int 0);;
- : (bool, string) Result.t = Ok false
```

```
# is_positive_permissive (`Ratio (3,4));;
- : (bool, string) Result.t = Error "Unknown number type"
```

Catch-all cases are error-prone even with ordinary variants, but they are especially so with polymorphic variants. That's because you have no way of bounding what tags your function might have to deal with. Such code is particularly vulnerable to typos. For instance, if code that uses `is_positive_permissive` passes in `Float` misspelled as `Floot`, the erroneous code will compile without complaint.

```
# is_positive_permissive (`Floot 3.5);;
- : (bool, string) Result.t = Error "Unknown number type"
```

With ordinary variants, such a typo would have been caught as an unknown tag. As a general matter, one should be wary about mixing catch-all cases and polymorphic variants.

Let's consider how we might turn our code into a proper library with an `mli`. Here's what the interface to this file might look like.

```
(* file: terminal_color.mli *)

open Core.Std

type basic_color =
  [ `Black   | `Blue | `Cyan  | `Green
  | `Magenta | `Red  | `White | `Yellow ]

type color =
  [ `Basic of basic_color * [ `Bold | `Regular ]
  | `Gray of int
  | `RGB  of int * int * int ]

type extended_color =
  [ color
  | `RGBA of int * int * int * int ]

val color_to_int          : color -> int
val extended_color_to_int : extended_color -> int
```

Here, `extended_color` is defined as an explicit extension of `color`. Also, notice that we defined all of these types as exact variants. We can implement this library as follows.

```
open Core.Std

type basic_color =
  [ `Black   | `Blue | `Cyan  | `Green
  | `Magenta | `Red  | `White | `Yellow ]

type color =
  [ `Basic of basic_color * [ `Bold | `Regular ]
  | `Gray of int
  | `RGB  of int * int * int ]

type extended_color =
  [ color
  | `RGBA of int * int * int * int ]

let basic_color_to_int = function
  | `Black -> 0 | `Red     -> 1 | `Green -> 2 | `Yellow -> 3
  | `Blue  -> 4 | `Magenta -> 5 | `Cyan  -> 6 | `White  -> 7

let color_to_int = function
  | `Basic (basic_color,weight) ->
    let base = match weight with `Bold -> 8 | `Regular -> 0 in
    base + basic_color_to_int basic_color
  | `RGB (r,g,b) -> 16 + b + g * 6 + r * 36
  | `Gray i -> 232 + i

let extended_color_to_int = function
  | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
  | `Grey x -> 2000 + x
  | (`Basic _ | `RGB _ | `Gray _) as color -> color_to_int color
```

In the above code, we did something funny to the definition of `extended_color_to_int`, that underlines some of the downsides of polymorphic variants. In particular, we added some special-case handling for the color gray, rather than using `color_to_int`. Unfortunately, we misspelled `Gray` as `Grey`. This is exactly the kind of error that the compiler would catch with ordinary variants, but with polymorphic variants, this compiles without issue. All that happened was that the compiler inferred a wider type for `extended_color_to_int`, which happens to be compatible with the narrower type that was listed in the mli.

If we add an explicit type annotation to the code itself (rather than just in the mli), then the compiler has enough information to warn us.

```
let extended_color_to_int : extended_color -> int = function
  | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
  | `Grey x -> 2000 + x
  | (`Basic _ | `RGB _ | `Gray _) as color -> color_to_int color
```

In particular, the compiler will complain that the `` `Grey `` case as unused.

```
File "terminal_color.ml", line 29, characters 4-11:
Warning 11: this match case is unused.
```

Once we have type definitions at our disposal, we can revisit the question of how we write the pattern-match that narrows the type. In particular, we can explicitly use the type name as part of the pattern match, by prefixing it with a `#`.

```
let extended_color_to_int : extended_color -> int = function
  | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
  | #color as color -> color_to_int color
```

This is useful when you want to narrow down to a type whose definition is long, and you don't want the verbosity of writing the tags down explicitly in the match.

When to use polymorphic variants

At first glance, polymorphic variants look like a strict improvement over ordinary variants. You can do everything that ordinary variants can do, plus it's more flexible and more concise. What's not to like?

In reality, regular variants are the more pragmatic choice most of the time. That's because the flexibility of polymorphic variants comes at a price. Here are some of the downsides.

- *Complexity:* As we've seen, the typing rules for polymorphic variants are a lot more complicated than they are for regular variants. This means that heavy use of polymorphic variants can leave you scratching your head trying to figure out why a given piece of code did or didn't compile. It can also lead to absurdly long and hard to decode error messages.
- *Error-finding:* Polymorphic variants are type-safe, but the typing discipline that they impose is, by dint of its flexibility, less likely to catch bugs in your program.
- *Efficiency:* This isn't a huge effect, but polymorphic variants are somewhat heavier than regular variants, and OCaml can't generate code for matching on polymorphic variants that is quite as efficient as what is generated for regular variants.

All that said, polymorphic variants are still a useful and powerful feature, but it's worth understanding their limitations, and how to use them sensibly and modestly.

Probably the safest and most common use-case for polymorphic variants is for cases where ordinary variants would be sufficient, but are syntactically too heavyweight. For example, you often want to create a variant type for encoding the inputs or outputs to a function, where it's not worth declaring a separate type for it. Polymorphic variants are very useful here, and as long as there are type annotations that constrain these to have explicit, exact types, this tends to work well.

Variants are most problematic exactly where you take full advantage of their power; in particular, when you take advantage of the ability of polymorphic variant types to overlap in the tags they support. This ties into OCaml's support for subtyping. As we'll discuss further when we cover objects in Chapter 11, subtyping brings in a lot of complexity, and most of the time, that's complexity you want to avoid.

CHAPTER 7

Error Handling

Nobody likes dealing with errors. It's tedious, it's easy to get wrong, and it's usually just not as fun as planning out how your program is going to succeed. But error handling is important, and however much you don't like thinking about it, having your software fail due to poor error handling code is worse.

Thankfully, OCaml has powerful tools for handling errors reliably and with a minimum of pain. In this chapter we'll discuss some of the different approaches in OCaml to handling errors, and give some advice on how to design interfaces that make error handling easier.

We'll start by describing the two basic approaches for reporting errors in OCaml: error-aware return types and exceptions.

Error-aware return types

The best way in OCaml to signal an error is to include that error in your return value. Consider the type of the `find` function in the `List` module.

```
# List.find;;
- : 'a list -> f:('a -> bool) -> 'a option
```

The option in the return type indicates that the function may not succeed in finding a suitable element, as you can see below.

```
# List.find [1;2;3] ~f:(fun x -> x >= 2) ;;
- : int option = Some 2
# List.find [1;2;3] ~f:(fun x -> x >= 10) ;;
- : int option = None
```

Having errors be explicit in the return values of your functions tells the caller that there is an error that needs to be handled. The caller can then handle the error explicitly, either recovering from the error or propagating it onward.

Consider the `compute_bounds` function defined below. The function takes a list and a comparison function, and returns upper and lower bounds for the list by finding the smallest and largest element on the list. `List.hd` and `List.last`, which return `None` when they encounter an empty list, are used to extract the largest and smallest element of the list.

```
# let compute_bounds ~cmp list =
    let sorted = List.sort ~cmp list in
    match List.hd sorted, List.last sorted with
    | None,_ | _, None -> None
    | Some x, Some y -> Some (x,y)
  ;;
val compute_bounds :
  cmp:('a -> 'a -> int) -> 'a list -> ('a * 'a) option = <fun>
```

The match statement is used to handle the error cases, propagating a None in `hd` or `last` into the return value of `compute_bounds`. On the other hand, in `find_mismatches` below, errors encountered during the computation do not propagate to the return value of the function. `find_mismatches` takes two hashtables as its arguments and tries to find keys that are stored in both. As such, a failure to find a key in one of the tables isn't really an error.

```
# let find_mismatches table1 table2 =
     Hashtbl.fold table1 ~init:[] ~f:(fun ~key ~data mismatches ->
        match Hashtbl.find table2 key with
        | Some data' when data' <> data -> key :: mismatches
        | _ -> mismatches
     )
 ;;
val find_mismatches :
  ('a, 'b) Hashtbl.t -> ('a, 'b) Hashtbl.t -> 'a list = <fun>
```

The use of options to encode errors underlines the fact that it's not clear whether a particular outcome, like not finding something on a list, is really an error, or just another valid outcome of your function. This turns out to be very context-dependent, and error-aware return types give you a uniform way of handling the result that works well for both situations.

Encoding errors with `Result`

Options aren't always a sufficiently expressive way to report errors. Specifically, when you encode an error as `None`, there's nowhere to say anything about the nature of the error.

`Result.t` is meant to address this deficiency. The type is defined as follows.

```
module Result : sig
   type ('a,'b) t = | Ok of 'a
```

```
                          | Error of 'b
end
```

A `Result.t` is essentially an option augmented with the ability to store other information in the error case. Like `Some` and `None` for options, the constructors `Ok` and `Error` are promoted to the top-level by `Core.Std`. As such, we can write:

```
# [ Ok 3; Error "abject failure"; Ok 4 ];;
- : (int, string) Core.Result.t list = [Ok 3; Error "abject failure"; Ok 4]
```

without first opening the `Result` module.

`Error` and `Or_error`

`Result.t` gives you complete freedom to choose the type of value you use to represent errors, but it's often useful to standardize on an error type. Among other things, this makes it easier to write utility functions to automate common error handling patterns.

But which type to choose? Is it better to represent errors as strings? Or S-expressions? Or something else entirely?

Core's answer to this question is the `Error.t` type, which tries to forge a good compromise between efficiency, convenience and control over the presentation of errors.

It might not be obvious at first why efficiency is an issue at all. But generating error messages is an expensive business. An ASCII representation of a value can be quite time-consuming to construct, particularly if it includes expensive-to-convert numerical data.

`Error` gets around this issue through laziness. In particular, an `Error.t` allows you to put off generation of the error string until you need it, which means a lot of the time you never have to construct it at all. You can of course construct an error directly from a string:

```
# Error.of_string "something went wrong";;
- : Core.Std.Error.t = "something went wrong"
```

A more interesting construction message from a performance point of view is to construct an `Error.t` from a thunk:

```
# Error.of_thunk (fun () ->
    sprintf "something went wrong: %f" 32.3343);;
  - : Core.Std.Error.t = "something went wrong: 32.334300"
```

In this case, we can benefit from the laziness of `Error`, since the thunk won't be called until the `Error.t` is converted to a string.

We can also create an `Error.t` based on an s-expression converter. This is probably the most common idiom in Core.

```
# Error.create "Something failed a long time ago" Time.epoch Time.sexp_of_t;;
- : Core.Std.Error.t =
"Something failed a long time ago: (1969-12-31 19:00:00.000000)"
```

Here, the value `Time.epoch` is included in the error, but that value isn't converted into an s-expression until the error is printed out. Using the Sexplib syntax-extension, which is discussed in more detail in chapter xref (*data-serialization-with-json-xml-and-s-expressions*), we can create an s-expression converter for a new type, thus allowing us to conveniently register multiple pieces of data in an `Error.t` as a tuple.

```
# Error.create "Something went terribly wrong"
    (3.5, ["a";"b";"c"], 6034)
    <:sexp_of<float * string list * int>> ;;
- : Core.Std.Error.t = "Something went terribly wrong: (3.5(a b c)6034)"
```

The above declaration of `<:sexp_of<float * string list * int>>` is interpreted by sexplib as a sexp-converter for the tuple.

`Error` also supports operations for transforming errors. For example, it's often useful to augment an error with some extra information about the context of the error, or to combine multiplier errors together. `Error.tag` and `Error.of_list` fulfill these roles.

The type `'a Or_error.t` is just a shorthand for `('a,Error.t) Result.t`, and it is, after `option`, the most common way of returning errors in Core.

`bind` and other error-handling idioms

As you write more error handling code in OCaml, you'll discover that certain patterns start to emerge. A number of these common patterns been codified in the interfaces of modules like `Option` and `Result`. One particularly useful one is built around the function `bind`, which is both an ordinary function and an infix operator `>>=`, both with the same type signature:

```
val bind : 'a option -> ('a -> 'b option) -> 'b option
```

`bind` is a way of sequencing together error-producing functions so that the first one to produce an error terminates the computation. In particular, `bind None f` returns `None` without calling `f`, and `bind (Some x) f` returns `f x`. We can use a nested sequence of these binds to express a multi-stage computation that can fail at any stage. Here's a rewrite `compute_bounds` in this style.

```
# let compute_bounds ~cmp list =
    let sorted = List.sort ~cmp list in
    Option.bind (List.hd sorted) (fun first ->
      Option.bind (List.last sorted) (fun last ->
        Some (first,last)))
  ;;
```

```
val compute_bounds : cmp:('a -> 'a -> int) -> 'a list -> ('a * 'a) option =
  <fun>
```

The above code is a little bit hard to swallow, however, on a syntactic level. We can make it a bit easier to read, and drop some of the parenthesis, by using the infix operator form of bind. Note that we locally open the `Option.Monad_infix` module to get access to the operators. (The module is called `Monad_infix` because the bind operator is part of a sub-interface called `Monad`, which we'll talk about more in Chapter 16.

```
# let compute_bounds ~cmp list =
    let open Option.Monad_infix in
    let sorted = List.sort ~cmp list in
    List.hd sorted   >>= fun first ->
    List.last sorted >>= fun last  ->
    Some (first,last)
```

There are other useful idioms encoded in the functions in `Option`. Another example is `Option.both`, which takes two optional values and produces a new optional pair that is `None` if either of its arguments are `None`. Using `Option.both`, we can make `compute_bounds` even shorter.

```
# let compute_bounds ~cmp list =
    let sorted = List.sort ~cmp list in
    Option.both (List.hd sorted) (List.last sorted)
```

These error-handling functions are valuable because they let you express your error handling both explicitly and concisely. We've only discussed these functions in the context of the `Option` module, but similar functionality is available in both `Result` and `Or_error`.

Exceptions

Exceptions in OCaml are not that different from exceptions in many other languages, like Java, C# and Python. Exceptions are a way to terminate a computation and report an error, while providing a mechanism to catch and handle (and possibly recover from) exceptions that are triggered by sub-computations.

You can trigger an exception triggered in OCaml by, for example, dividing an integer by zero:

```
# 3 / 0;;
Exception: Division_by_zero.
```

And an exception can terminate a computation even if it happens nested a few levels deep in a computation.

```
# List.map ~f:(fun x -> 100 / x) [1;3;0;4];;
Exception: Division_by_zero.
```

```
If we put a `printf` in the middle of the computation, we can see that
the `List.map` is interrupted part way through it's execution:

# List.map ~f:(fun x -> printf "%d\n%!" x; 100 / x) [1;3;0;4];;
1
3
0
Exception: Division_by_zero.
```

In addition to built-in exceptions like `Divide_by_zero`, OCaml lets you define your own.

```
# exception Key_not_found of string;;
exception Key_not_found of string
# raise (Key_not_found "a");;
Exception: Key_not_found("a").
```

Here's an example of a function for looking up a key in an *association list*, *i.e.* a list of key/value pairs which uses this newly-defined exception:

```
# let rec find_exn alist key = match alist with
    | [] -> raise (Key_not_found key)
    | (key',data) :: tl -> if key = key' then data else find_exn tl key
  ;;
val find_exn : (string * 'a) list -> string -> 'a = <fun>
# let alist = [("a",1); ("b",2)];;
val alist : (string * int) list = [("a", 1); ("b", 2)]
# find_exn alist "a";;
- : int = 1
# find_exn alist "c";;
Exception: Key_not_found("c").
```

Note that we named the function `find_exn` to warn the user that the function routinely throws exceptions, a convention that is used heavily in Core.

In the above example, `raise` throws the exception, thus terminating the computation. The type of raise is a bit surprising when you first see it:

```
# raise;;
- : exn -> 'a = <fun>
```

The return type of `'a` suggests that `raise` could return a value of any type. That seems impossible, and it is. Really, `raise` has this type because it never returns at all. This behavior isn't restricted to functions like `raise` that terminate by throwing exceptions. Here's another example of a function that doesn't return a value.

```
# let rec forever () = forever ();;
val forever : unit -> 'a = <fun>
```

`forever` doesn't return a value for a different reason: it is an infinite loop.

This all matters because it means that the return type of `raise` can be whatever it needs to be to fit in to the context it is called in. Thus, the type system will let us throw an exception anywhere in a program.

Declaring exceptions with `with sexp`

OCaml can't always generate a useful textual representation of your exception, for example:

```
# exception Wrong_date of Date.t;;
exception Wrong_date of Date.t
# Wrong_date (Date.of_string "2011-02-23");;
- : exn = Wrong_date(_)
```

But if you declare the exception using `with sexp` (and the constituent types have sexp converters), we'll get something with more information.

```
# exception Wrong_date of Date.t with sexp;;
exception Wrong_date of Core.Std.Date.t
# Wrong_date (Date.of_string "2011-02-23");;
- : exn = (.Wrong_date 2011-02-23)
```

The period in front of `Wrong_date` is there because the representation generated by `with sexp` includes the full module path of the module where the exception in question is defined. This is quite useful in tracking down which precise exception is being reported. In this case, since we've declared the exception at the toplevel, that module path is trivial.

Exception handlers

So far, we've only seen exceptions fully terminate the execution of a computation. But often, we want a program to be able to respond to and recover from an exception. This is achieved through the use of *exception handlers*.

In OCaml, an exception handler is declared using a `try`/`with` statement. Here's the basic syntax.

```
try <expr> with
| <pat1> -> <expr1>
| <pat2> -> <expr2>
...
```

A `try`/`with` clause would first evaluate `<expr>`, and if that evaluation completes without returning an exception, then the value of the overall expression is the value of `<expr>`.

But if evaluating `<expr>` leads to an exception being thrown, then the exception will be fed to the pattern match statements following the `with`. If the exception matches a pattern, then the expression on the right hand side of that pattern will be evaluated. Otherwise, the original exception continues up the call stack, to be handled by the next outer exception handler, or terminate the program if there is none.

Cleaning up in the presence of exceptions

One headache with exceptions is that they can terminate your execution at unexpected places, leaving your program in an awkward state. Consider the following code snippet:

```
let load_config filename =
  let inc = In_channel.create filename in
  let config = Config.t_of_sexp (Sexp.input_sexp inc) in
  In_channel.close inc;
  config
```

The problem with this code is that the function that loads the s-expression and parses it into a `Config.t` might throw an exception if the config file in question is malformed. Unfortunately, that means that the `In_channel.t` that was opened will never be closed, leading to a file-descriptor leak.

We can fix this using Core's `protect` function. The basic purpose of `protect` is to ensure that the `finally` thunk will be called when `f` exits, whether it exited normally or with an exception. Here's how it could be used to fix `load_config`.

```
let load_config filename =
  let inc = In_channel.create filename in
  protect ~f:(fun () -> Config.t_of_sexp (Sexp.input_sexp inc)
    ~finally:(fun () -> In_channel.close inc)
```

Catching specific exceptions

OCaml's exception-handling system allows you to tune your error-recovery logic to the particular error that was thrown. For example, `List.find_exn` always throws `Not_found`. You can take advantage of this in your code, for example, let's define a function called `lookup_weight`, with the following signature:

```
(** [lookup_weight ~compute_weight alist key] Looks up a
    floating-point weight by applying [compute_weight] to the data
    associated with [key] by [alist].  If [key] is not found, then
    return 0.
*)
val lookup_weight :
  compute_weight:('data -> float) -> ('key * 'data) list -> 'key -> float
```

We can implement such a function using exceptions as follows:

```
# let lookup_weight ~compute_weight alist key =
    try
      let data = List.Assoc.find_exn alist key in
      compute_weight data
    with
      Not_found -> 0. ;;
val lookup_weight :
  compute_weight:('a -> float) -> ('b * 'a) list -> 'b -> float =
  <fun>
```

This implementation is more problematic than it looks. In particular, what happens if `compute_weight` itself throws an exception? Ideally, `lookup_weight` should propagate that exception on, but if the exception happens to be `Not_found`, then that's not what will happen:

```
# lookup_weight ~compute_weight:(fun _ -> raise Not_found)
    ["a",3; "b",4] "a" ;;
- : float = 0.
```

This kind of problem is hard to detect in advance, because the type system doesn't tell us what kinds of exceptions a given function might throw. Because of this kind of confusion, it's usually better to avoid catching specific exceptions. In this case, we can improve the code by catching the exception in a narrower scope.

```
# let lookup_weight ~compute_weight alist key =
    match
      try Some (List.Assoc.find_exn alist key) with
      | Not_found -> None
    with
    | None -> 0.
    | Some data -> compute_weight data ;;
```

At which point, it makes sense to simply use the non-exception throwing function, `List.Assoc.find`, instead.

Backtraces

A big part of the point of exceptions is to give useful debugging information. But at first glance, OCaml's exceptions can be less than informative. Consider the following simple program.

```
(* exn.ml *)

open Core.Std
exception Empty_list

let list_max = function
  | [] -> raise Empty_list
  | hd :: tl -> List.fold tl ~init:hd ~f:(Int.max)

let () =
  printf "%d\n" (list_max [1;2;3]);
  printf "%d\n" (list_max [])
```

If we build and run this program, we'll get a pretty uninformative error:

```
$ ./exn
```

```
3
Fatal error: exception Exn.Empty_list
```

The example in question is short enough that it's quite easy to see where the error came from. But in a complex program, simply knowing which exception was thrown is usually not enough information to figure out what went wrong.

We can get more information from OCaml if we turn on stack traces. This can be done by setting the `OCAMLRUNPARAM` environment variable, as shown:

```
exn $ export OCAMLRUNPARAM=b
exn $ ./exn
3
Fatal error: exception Exn.Empty_list
Raised at file "exn.ml", line 7, characters 16-26
Called from file "exn.ml", line 12, characters 17-28
```

Backtraces can also be obtained at runtime. In particular, `Exn.backtrace` will return the backtrace of the most recently thrown exception.

Exceptions for control flow

From exceptions to error-aware types and back again

Both exceptions and error-aware types are necessary parts of programming in OCaml. As such, you often need to move between these two worlds. Happily, Core comes with some useful helper functions to help you do just that. For example, given a piece of code that can throw an exception, you can capture that exception into an option as follows:

```
# let find alist key =
    Option.try_with (fun () -> find_exn alist key) ;;
val find : (string * 'a) list -> string -> 'a option = <fun>
# find ["a",1; "b",2] "c";;
- : int Core.Std.Option.t = None
# find ["a",1; "b",2] "b";;
- : int Core.Std.Option.t = Some 2
```

And `Result` and `Or_error` have similar `try_with` functions. So, we could write:

```
# let find alist key =
    Result.try_with (fun () -> find_exn alist key) ;;
val find : (string * 'a) list -> string -> ('a, exn) Result.t = <fun>
# find ["a",1; "b",2] "c";;
- : (int, exn) Result.t = Result.Error Key_not_found("c")
```

And then we can re-raise that exception:

```
# Result.ok_exn (find ["a",1; "b",2] "b");;
```

2013-02-05
19:46:47

```
- : int = 2
# Result.ok_exn (find ["a",1; "b",2] "c");;
Exception: Key_not_found("c").
```

2013-02-05
19:46:47

CHAPTER 8

Functors

Up until now, we've seen OCaml's module system play an important but limited role. In particular, we've seen them as a mechanism for organizing code into units with specified interfaces. But modules can do much more than that, acting as a powerful toolset for building generic code and structuring large-scale systems. Much of that power comes from functors.

Functors are, roughly speaking, functions from modules to modules, and they can be used to solve a variety of code-structuring problems, including:

- *Dependency injection*, or making the implementations of some components of a system swappable. This is particularly useful when you want to mock up parts of your system for testing and simulation purposes.
- *Auto-extension of modules*. Sometimes, there is some functionality that you want to build in a standard way for different types, in each case based on a some piece of type-specific logic. For example, you might want to add a slew of comparison operators derived from a base comparison function. To do this by hand would require a lot of repetitive code for each type, but functors let you write this logic just once and apply it to many different types.
- *Instantiating modules with state*. Modules can contain mutable state, and that means that you'll occasionally want to have multiple instantiations of a particular module, each with its own separate and independent mutable state. Functors let you automate the construction of such modules.

A trivial example

We'll start by considering the simplest possible example: a functor for incrementing an integer.

More precisely, we'll create a functor that takes a module containing a single integer variable `x`, and returns a new module with `x` incremented by one.

```
# module type X_int = sig val x : int end;;
module type X_int = sig val x : int end
# module Increment (M:X_int) : X_int = struct
    let x = M.x + 1
  end;;
module Increment : functor (M : X_int) -> X_int
```

One thing that immediately jumps out about functors is that they're considerably more heavyweight syntactically than ordinary functions. For one thing, functors require explicit (module) type annotations, which ordinary functions do not. Here, we've specified the module type `X_int` for both the input and output of the functor. Technically, only the type on the input is mandatory, although in practice, one often specifies both.

The following shows what happens when we omit the module type for the output of the functor.

```
# module Increment (M:X_int) = struct
    let x = M.x + 1
  end;;
module Increment : functor (M : X_int) -> sig val x : int end
```

We can see that the inferred module type of the output is now written out explicitly, rather than being a reference to the named signature `X_int`.

We can now use `Increment` to define new modules.

```
# module Three = struct let x = 3 end;;
  module Three : sig val x : int end
# module Four = Increment(Three);;
module Four : sig val x : int end
# Four.x - Three.x;;
- : int = 1
```

In this case, we applied `Increment` to a module whose signature is exactly equal to `X_int`. But we can apply `Increment` to any module that satisfies the interface `X_int`, in the same way that a the contents of an `ml` file can satisfy the `mli`. That means that the module type can omit some information available in the module, either by dropping fields or by leaving some fields abstract. Here's an example:

```
# module Three_and_more = struct
    let x = 3
    let y = "three"
  end;;
module Three_and_more : sig val x : int val x_string : string end
# module Four = Increment(Three_and_more);;
module Four : sig val x : int end
```

A bigger example: computing with intervals

Let's now consider a more realistic example of how to use functors: a library for computing with intervals. This library will be functorized over the type of the endpoints of the intervals and the ordering of those endpoints.

First we'll define a module type that captures the information we'll need about the endpoint type. This interface, which we'll call `Comparable`, contains just two things: a comparison function, and the type of the values to be compared.

```
# module type Comparable = sig
    type t
    val compare : t -> t -> int
  end ;;
module type Comparable = sig type t val compare : t -> t -> int end
```

The comparison function follows the standard OCaml idiom for such functions, returning `0` if the two elements are equal, a positive number if the first element is larger than the second, and a negative number if the first element is smaller than the second. Thus, we could rewrite the standard comparison functions on top of `compare` as shown below.

```
compare x y < 0     (* x < y *)
compare x y = 0     (* x = y *)
compare x y > 0     (* x > y *)
```

The functor for creating the interval module is shown below. We represent an interval with a variant type, which is either `Empty` or `Interval (x,y)`, where `x` and `y` are the bounds of the interval.

```
# module Make_interval(Endpoint : Comparable) = struct

    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty

    let create low high =
      if Endpoint.compare low high > 0 then Empty
      else Interval (low,high)

    let is_empty = function
      | Empty -> true
      | Interval _ -> false

    let contains t x =
      match t with
      | Empty -> false
      | Interval (l,h) ->
        Endpoint.compare x l >= 0 && Endpoint.compare x h <= 0

    let intersect t1 t2 =
      let min x y = if Endpoint.compare x y <= 0 then x else y in
      let max x y = if Endpoint.compare x y >= 0 then x else y in
      match t1,t2 with
      | Empty, _ | _, Empty -> Empty
      | Interval (l1,h1), Interval (l2,h2) ->
        create (max l1 l2) (min h1 h2)

  end ;;
module Make_interval :
  functor (Endpoint : Comparable) ->
    sig
      type t = Interval of Endpoint.t * Endpoint.t | Empty
      val create : Endpoint.t -> Endpoint.t -> t
      val is_empty : t -> bool
      val contains : t -> Endpoint.t -> bool
      val intersect : t -> t -> t
    end
```

We can instantiate the functor by applying it to a module with the right signature. In the following, we provide the functor input as an anonymous module.

```
# module Int_interval =
    Make_interval(struct
      type t = int
      let compare = Int.compare
    end);;
module Int_interval :
  sig
    type t = Interval of int * int | Empty
    val create : int -> int -> t
    val is_empty : t -> bool
    val contains : t -> int -> bool
    val intersect : t -> t -> t
  end
```

When we choose our interfaces that are aligned with the standards of our libraries, we don't need to construct a custom module to feed to the functor. In this case, for example, we can directly use the `Int` or `String` modules provided by Core.

```
# module Int_interval = Make_interval(Int) ;;
# module String_interval = Make_interval(String) ;;
```

This works because many modules in Core, including `Int` and `String`, satisfy an extended version of the `Comparable` signature described above. Standardized signatures are generally good practice, both because they makes functors easier to use, and because they make the codebase generally easier to navigate.

Now we can use the newly defined `Int_interval` module like any ordinary module.

```
# let i1 = Int_interval.create 3 8;;
val i1 : Int_interval.t = Int_interval.Interval (3, 8)
# let i2 = Int_interval.create 4 10;;
val i2 : Int_interval.t = Int_interval.Interval (4, 10)
```

```
# Int_interval.intersect i1 i2;;
- : Int_interval.t = Int_interval.Interval (4, 8)
```

This design gives us the freedom to use any comparison function we want for comparing the endpoints. We could, for example, create a type of int interval with the order of the comparison reversed, as follows:

```
# module Rev_int_interval =
    Make_interval(struct
      type t = int
      let compare x y = Int.compare y x
    end);;
```

The behavior of `Rev_int_interval` is of course different from `Int_interval`, as we can see below.

```
# let interval = Int_interval.create 4 3;;
val interval : Int_interval.t = Int_interval.Empty
# let rev_interval = Rev_int_interval.create 4 3;;
val rev_interval : Rev_int_interval.t = Rev_int_interval.Interval (4, 3)
```

Importantly, `Rev_int_interval.t` is a different type than `Int_interval.t`, even though its physical representation is the same. Indeed, the type system will prevent us from confusing them.

```
# Int_interval.contains rev_interval 3;;
Characters 22-34:
  Int_interval.contains rev_interval 3;;
                        ^^^^^^^^^^^^
Error: This expression has type Rev_int_interval.t
       but an expression was expected of type
         Int_interval.t = Make_interval(Int).t
```

This is important, because confusing the two kinds of intervals would be a semantic error, and it's an easy one to make. The ability of functors to mint new types is a useful trick that comes up a lot.

Making the functor abstract

There's a problem with `Make_interval`. The code we wrote depends on the invariant that the upper bound of an interval is greater than its lower bound, but that invariant can be violated. The invariant is enforced by the create function, but because `Interval.t` is not abstract, we can bypass the `create` function.

```
# Int_interval.create 4 3;; (* going through create *)
- : Int_interval.t = Int_interval.Empty
# Int_interval.Interval (4,3);; (* bypassing create *)
- : Int_interval.t = Int_interval.Interval (4, 3)
```

To make `Int_interval.t` abstract, we need to apply an interface to the output of the `Make_interval`. Here's an explicit interface that we can use for that purpose.

```
# module type Interval_intf = sig
   type t
   type endpoint
   val create : endpoint -> endpoint -> t
   val is_empty : t -> bool
   val contains : t -> endpoint -> bool
   val intersect : t -> t -> t
  end;;
```

This interface includes the type `endpoint` to represent the type of the endpoints of the interval. Given this interface, we can redo our definition of `Make_interval`. Notice that we added the type `endpoint` to the implementation of the module to make the implementation match `Interval_intf`.

```
# module Make_interval(Endpoint : Comparable) : Interval_intf = struct

    type endpoint = Endpoint.t
    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty

    ....

  end ;;
module Make_interval : functor (Endpoint : Comparable) -> Interval_intf
```

Sharing constraints

The resulting module is abstract, but unfortunately, it's too abstract. In particular, we haven't exposed the type `endpoint`, which means that we can't even construct an interval anymore.

```
# module Int_interval = Make_interval(Int);;
module Int_interval : Interval_intf
# Int_interval.create 3 4;;
Characters 20-21:
  Int_interval.create 3 4;;
                      ^
Error: This expression has type int but an expression was expected of type
         Int_interval.endpoint
```

To fix this, we need to expose the fact that `endpoint` is equal to `Int.t` (or more generally, `Endpoint.t`, where `Endpoint` is the argument to the functor). One way of doing this is through a *sharing constraint*, which allows you to tell the compiler to expose the fact that a given type is equal to some other type. The syntax for a sharing constraint on a module type is as follows.

```
S with type s = t
```

where `S` is a module type, `s` is a type inside of `S`, and `t` is a type defined outside of `S`. The result of this expression is a new signature that's been modified so that it exposes the fact that `s` is equal to `t`. We can use a sharing constraint to create a specialized version of `Interval_intf` for integer intervals.

```
# module type Int_interval_intf = Interval_intf with type endpoint = int;;
module type Int_interval_intf =
  sig
    type t
    type endpoint = int
    val create : endpoint -> endpoint -> t
    val is_empty : t -> bool
    val contains : t -> endpoint -> bool
    val intersect : t -> t -> t
  end
```

And we can also use it in the context of a functor, where the right-hand side of the sharing constraint is an element of the functor argument. Thus, we expose an equality between a type in the output of the functor (in this case, the type `endpoint`) and a type in its input (`Endpoint.t`).

```
# module Make_interval(Endpoint : Comparable)
      : Interval_intf with type endpoint = Endpoint.t = struct

    type endpoint = Endpoint.t
    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty

    ...

  end ;;
module Make_interval :
  functor (Endpoint : Comparable) ->
    sig
      type t
      type endpoint = Endpoint.t
      val create : endpoint -> endpoint -> t
      val is_empty : t -> bool
      val contains : t -> endpoint -> bool
      val intersect : t -> t -> t
    end
```

So now, the interface is as it was, except that `endpoint` is now known to be equal to `Endpoint.t`. As a result of that type equality, we can now do things like construct intervals again.

```
# let i = Int_interval.create 3 4;;
val i : Int_interval.t = <abstr>
```

```
# Int_interval.contains i 5;;
- : bool = false
```

Destructive substitution

Sharing constraints basically do the job, but they have some downsides. In particular, we've now been stuck with the useless type declaration of `endpoint` that clutters up both the interface and the implementation. A better solution would be to modify the `Interval_intf` signature by replacing `endpoint` with `Endpoint.t` everywhere it shows up, and deleting the definition of `endpoint` from the signature. We can do just this using what's called *destructive substitution*. Here's the basic syntax.

```
S with type s := t
```

The following shows how we could use this with `Make_interval`.

Here's an example of what we get if we use destructive substitution to specialize the `Interval_intf` interface to integer intervals.

```
# module type Int_interval_intf = Interval_intf with type endpoint := int;;
module type Int_interval_intf =
  sig
    type t
    val create : int -> int -> t
    val is_empty : t -> bool
    val contains : t -> int -> bool
    val intersect : t -> t -> t
  end
```

There's now no mention of n `endpoint`, all occurrences of that type having been replaced by `int`. As with sharing constraints, we can also use this in the context of a functor.

```
# module Make_interval(Endpoint : Comparable)
    : Interval_intf with type endpoint := Endpoint.t =
  struct

    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty

    ....

  end ;;
module Make_interval :
  functor (Endpoint : Comparable) ->
    sig
      type t
      val create : Endpoint.t -> Endpoint.t -> t
      val is_empty : t -> bool
      val contains : t -> Endpoint.t -> bool
      val intersect : t -> t -> t
    end
```

The interface is precisely what we want, and we didn't need to define the `endpoint` type alias in the body of the module. If we instantiate this module, we'll see that it works properly: we can construct new intervals, but `t` is abstract, and so we can't directly access the constructors and violate the invariants of the data structure.

```
# module Int_interval = Make_interval(Int);;
# Int_interval.create 3 4;;
- : Int_interval.t = <abstr>
# Int_interval.Interval (4,3);;
Characters 0-27:
  Int_interval.Interval (4,3);;
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: Unbound constructor Int_interval.Interval
```

Using multiple interfaces

Another feature that we might want for our interval module is the ability to serialize the type, in particular, by converting to s-expressions. If we simply invoke the `sexplib` macros by adding `with sexp` to the definition of `t`, though, we'll get an error:

```
# module Make_interval(Endpoint : Comparable)
    : Interval_intf with type endpoint := Endpoint.t = struct

    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty
    with sexp

    ....

  end ;;
Characters 120-123:
        type t = | Interval of Endpoint.t * Endpoint.t
                               ^^^^^^^^^^
Error: Unbound value Endpoint.t_of_sexp
```

The problem is that `with sexp` adds code for defining the s-expression converters, and that code assumes that `Endpoint` has the appropriate sexp-conversion functions for `Endpoint.t`. But all we know about `Endpoint` is that it satisfies the `Comparable` interface, which doesn't say anything about s-expressions.

Happily, Core comes with a built in interface for just this purpose called `Sexpable`, which is defined as follows:

```
module type Sexpable = sig
  type t = int
  val sexp_of_t : t -> Sexp.t
  val t_of_sexp : Sexp.t -> t
end
```

We can modify `Make_interval` to use the `Sexpable` interface, for both its input and its output. Note the use of destructive substitution to combine multiple signatures together. This is important because it stops the `type t`'s from the different signatures from interfering with each other.

Also note that we have been careful to override the sexp-converter here to ensure that the data structures invariants are still maintained when reading in from an s-expression.

```
# module type Interval_intf_with_sexp = sig
   type t
   include Interval_intf with type t := t
   include Sexpable      with type t := t
  end;;
# module Make_interval(Endpoint : sig
    type t
    include Comparable with type t := t
    include Sexpable   with type t := t
  end) : Interval_intf_with_sexp with type endpoint := Endpoint.t =
  struct

      type t = | Interval of Endpoint.t * Endpoint.t
               | Empty
      with sexp

      let create low high =
         ...

      (* put a wrapper round the auto-generated sexp_of_t to enforce
         the invariants of the data structure *)
      let t_of_sexp sexp =
        match t_of_sexp sexp with
        | Empty -> Empty
        | Interval (x,y) -> create x y

      ....

     end ;;
module Make_interval :
  functor
    (Endpoint : sig
           type t
           val compare : t -> t -> int
           val sexp_of_t : t -> Sexplib.Sexp.t
           val t_of_sexp : Sexplib.Sexp.t -> t
         end) ->
    sig
      type t
      val create : Endpoint.t -> Endpoint.t -> t
      val is_empty : t -> bool
      val contains : t -> Endpoint.t -> bool
      val intersect : t -> t -> t
      val sexp_of_t : t -> Sexplib.Sexp.t
      val t_of_sexp : Sexplib.Sexp.t -> t
    end
```

And now, we can use that sexp-converter in the ordinary way:

```
# module Int = Make_interval(Int) ;;
# Int_interval.sexp_of_t (Int_interval.create 3 4);;
- : Sexplib.Sexp.t = (Interval 3 4)
# Int_interval.sexp_of_t (Int_interval.create 4 3);;
- : Sexplib.Sexp.t = Empty
```

Extending modules

Another common use of functors is to generate type-specific functionality for a given module in a standardized way. Let's see how this works in the context of a functional queue, which is just a functional version of a FIFO (first-in, first-out) queue. Being functional, operations on the queue return new queues, rather than modifying the queues that were passed in.

Here's a reasonable mli

```
(* file: fqueue.mli *)

type 'a t

val empty : 'a t

val enqueue : 'a t -> 'a -> 'a t

(** [dequeue q] returns None if the [q] is empty *)
val dequeue : 'a t -> ('a * 'a t) option

val fold : 'a t -> init:'acc -> f:('acc -> 'a -> 'acc) -> 'acc
```

Now let's implement `Fqueue`. A standard trick is for the `Fqueue` to maintain an input and an output list, so that one can efficiently `enqueue` on the first list, and can efficiently dequeue from the out list. If you attempt to dequeue when the output list is empty, the input list is reversed and becomes the new output list. Here's an implementation that uses that trick.

```
(* file: fqueue.ml *)
open Core.Std

type 'a t = 'a list * 'a list

let empty = ([],[])

let enqueue (l1,l2) x = (x :: l1,l2)

let dequeue (in_list,out_list) =
  match out_list with
  | hd :: tl -> Some (hd, (in_list,tl))
  | [] ->
```

```
    match List.rev in_list with
    | [] -> None
    | hd::tl -> Some (hd, ([], tl))

let fold (in_list,out_list) ~init ~f =
  List.fold ~init:(List.fold ~init ~f out_list) ~f
    (List.rev in_list)
```

One problem with our `Fqueue` is that the interface is quite skeletal. There are lots of useful helper functions that one might want that aren't there. For example, for lists we have `List.iter` which runs a function on each node; and a `List.find` that finds the first element on the list that matches a given predicate. Such helper functions come up for pretty much every container type, and implementing them over and over is a bit of a dull and repetitive affair.

As it happens, many of these helper functions can be derived mechanically from just the fold function we already implemented. Rather than write all of these helper functions by hand for every new container type, we can instead use a functor that will let us add this functionality to any container that has a `fold` function.

We'll create a new module, `Foldable` that automates the process of adding helper functions to a fold-supporting container. As you can see, `Foldable` contains a module signature `S` which defines the signature that is required to support folding; and a functor `Extend` that allows one to extend any module that matches `Foldable.S`.

```
(* file: foldable.ml *)

open Core.Std

module type S = sig
  type 'a t
  val fold : 'a t -> init:'acc -> f:('acc -> 'a -> 'acc) -> 'acc
end

module type Extension = sig
  type 'a t
  val iter    : 'a t -> f:('a -> unit) -> unit
  val length  : 'a t -> int
  val count   : 'a t -> f:('a -> bool) -> int
  val for_all : 'a t -> f:('a -> bool) -> bool
  val exists  : 'a t -> f:('a -> bool) -> bool
end

(* For extending a Foldable module *)
module Extend(Arg : S)
  : Extension with type 'a t := 'a Arg.t =
struct
  open Arg

  let iter t ~f =
    fold t ~init:() ~f:(fun () a -> f a)
  let length t =
    fold t ~init:0  ~f:(fun acc _ -> acc + 1)

  let count t ~f =
    fold t ~init:0  ~f:(fun count x -> count + if f x then 1 else 0)

  exception Short_circuit

  let for_all c ~f =
    try iter c ~f:(fun x -> if not (f x) then raise Short_circuit); true
    with Short_circuit -> false

  let exists c ~f =
    try iter c ~f:(fun x -> if f x then raise Short_circuit); false
    with Short_circuit -> true
end
```

Now we can apply this to `Fqueue`. We can rewrite the interface of `Fqueue` as follows.

```
(* file: fqueue.mli *)
open Core.Std

type 'a t
val empty : 'a t
val enqueue : 'a t -> 'a -> 'a t
val dequeue : 'a t -> ('a * 'a t) option
val fold : 'a t -> init:'acc -> f:('acc -> 'a -> 'acc) -> 'acc

include Foldable.Extension with type 'a t := 'a t
```

In order to apply the functor, we'll put the definition of `Fqueue` in a sub-module called `T`, and then call `Foldable.Extend` on `T`.

```
open Core.Std

module T = struct
  type 'a t = 'a list * 'a list

  let empty = [],[]

  let enqueue (l1,l2) x =
    (x :: l1,l2)

  let rec dequeue (in_list,out_list) =
    match out_list with
    | hd :: tl -> Some (hd, (in_list,tl))
    | [] -> dequeue ([], List.rev in_list)

  let fold (in_list,out_list) ~init ~f =
    List.fold ~init:(List.fold ~init ~f out_list) ~f
      (List.rev in_list)
end
include T
include Foldable.Extend(T)
```

This pattern comes up quite a bit in Core, and is used to for a variety of purposes.

- Adding comparison-based data structures like maps and sets, based on the `Comparable` interface.
- Adding hash-based data structures like hash sets and hash heaps.
- Support for so-called monadic libraries, like the ones discussed in Chapter 7 and Chapter 16. Here, the functor is used to provide a collection of standard helper functions based on the core `bind` and `return` operators.

CHAPTER 9

Files, Modules and Programs

We've so far experienced OCaml largely through the toplevel. As you move from exercises to real-world programs, you'll need to leave the toplevel behind and start building programs from files. Files are more than just a convenient way to store and manage your code; in OCaml, they also act as boundaries that divide your program into conceptual units.

In this chapter, we'll show you how to build an OCaml program from a collection of files, as well as the basics of working with modules and module signatures.

Single File Programs

We'll start with an example: a utility that reads lines from `stdin`, computing a frequency count of the lines that have been read in. At the end, the 10 lines with the highest frequency counts are written out. Here's a simple implementation, which we'll save as the file `freq.ml`. Note that we're using several functions from the `List.Assoc` module, which provides utility functions for interacting with association lists, *i.e.*, lists of key/value pairs.

```
(* freq.ml: basic implementation *)

open Core.Std

(* build_counts recursively builds up a mapping from lines to
   number of occurrences of that line. *)
let rec build_counts counts =
  match In_channel.input_line stdin with
  | None -> counts (* EOF, so return the counts accumulated so far *)
  | Some line ->
    (* get the number of times this line has been seen before,
       inferring 0 if the line doesn't show up in [counts] *)
    let count =
      match List.Assoc.find counts line with
      | None -> 0
      | Some x -> x
    in
    (* increment the count for line by 1, and recurse *)
    build_counts (List.Assoc.add counts line (count + 1))

let () =
  (* Compute the line counts *)
  let counts = build_counts [] in
  (* Sort the line counts in descending order of frequency *)
  let sorted_counts = List.sort ~cmp:(fun (_,x) (_,y) -> descending x y) counts  in
  (* Print out the 10 highest frequency entries *)
  List.iter (List.take 10 sorted_counts) ~f:(fun (line,count) ->
    printf "%3d: %s\n" count line)
```

Where is the main function?

Unlike C, programs in OCaml do not have a unique `main` function. When an OCaml program is evaluated, all the statements in the implementation files are evaluated in order. These implementation files can contain arbitrary expressions, not just function definitions. In this example, the declaration starting with `let () =` plays the role of the `main` declaration, kicking off the processing. But really the entire file is evaluated at startup, and so in some sense the full codebase is one big `main` function.

If we weren't using Core or any other external libraries, we could build the executable like this:

```
ocamlc freq.ml -o freq
```

But in this case, this command will fail with the error `Unbound module Core`. We need a somewhat more complex invocation to get Core linked in:

```
ocamlfind ocamlc -linkpkg -thread -package core freq.ml -o freq
```

Here we're using `ocamlfind`, a tool which itself invokes other parts of the ocaml toolchain (in this case, `ocamlc`) with the appropriate flags to link in particular libraries and packages. Here, `-package core` is asking `ocamlfind` to link in the Core library, `-linkpkg` is required to do the final linking in of packages for building a runnable executable, and `-thread` turns on threading support, which is required for Core.

While this works well enough for a one-file project, more complicated builds will require a tool to orchestrate the build. One great tool for this task is `ocamlbuild`, which is shipped with the OCaml compiler. We'll talk more about `ocamlbuild` in Chapter 23, but for now, we'll just walk through the steps required for this simple application. First, create a `_tags` file, containing the following lines.

```
true:package(core),thread,annot,debugging
```

2013-02-05
19:46:47

The purpose of the `_tags` file is to specify which compilation options are required for which files. In this case, we're telling `ocamlbuild` to link in the `core` package and to turn on threading, output of annotation files, and debugging support for all files (the condition `true` causes the options to be applied to every file in the project.)

We can then invoke `ocamlbuild` to build the executable in question.

```
$ ocamlbuild -use-ocamlfind freq.byte
```

If we'd invoked `ocamlbuild` with a target of `freq.native` instead of `freq.byte`, we would have gotten native-code instead.

We can now run the our program from the command-line. The following line extracts strings from the `ocamlopt` executable, and then reports the most frequently occurring ones.

```
$ strings `which ocamlopt` | ./freq.byte
 13: movq
 10: cmpq
  8: ", &
  7: .globl
  6: addq
  6: leaq
  5: ", $
  5: .long
  5: .quad
  4: ", '
```

Byte-code vs native-code

OCaml ships with two compilers---the `ocamlc` byte-code compiler, and the `ocamlopt` native-code compiler. Programs compiled with `ocamlc` are interpreted by a virtual machine, while programs compiled with `ocamlopt` are compiled to native machine code to be run on a specific operating system and processor architecture.

Aside from performance, executables generated by the two compilers have nearly identical behavior. There are a few things to be aware of. First, the byte-code compiler can be used on more architectures, and has some better tool support; in particular, the OCaml debugger only works with byte-code. Also, the byte-code compiler compiles faster than the native code compiler. Also, in order to run a bytecode executable you typically need to have OCaml installed on the system in question. That's not strictly required, though, since you can a byte-code executable with an embedded runtime, using the `-custom` compiler flag.

As a general matter, production executables should usually be built using the native-code compiler, but it sometimes makes sense to use byte-code for development builds. And, of course, bytecode makes sense when targeting a platform not supported by the native code compiler.

Multi-file programs and modules

Source files in OCaml are tied into the module system, with each file compiling down into a module whose name is derived from the name of the file. We've encountered modules before, for example, when we used functions like `find` and `add` from the `List.Assoc` module. At it's simplest, you can think of a module as a collection of definitions that are stored within a namespace.

Let's consider how we can use modules to refactor the implementation of `freq.ml`. Remember that the variable `counts` contains an association list representing the counts of the lines seen so far. But updating an association list takes time linear in the length of the list, meaning that the time complexity of processing a file is quadratic in the number of distinct lines in the file.

We can fix this problem by replacing association lists with a more efficient data structure. To do that, we'll first factor out the key functionality into a separate module with an explicit interface. We can consider alternative (and more efficient) implementations once we have a clear interface to program against.

We'll start by creating a file, `counter.ml`, that contains the logic for maintaining the association list used to describe the counts. The key function, called `touch`, updates the association list with the information that a given line should be added to the frequency counts.

```
(* counter.ml: first version *)

open Core.Std

let touch t s =
  let count =
    match List.Assoc.find t s with
    | None -> 0
    | Some x -> x
  in
  List.Assoc.add t s (count + 1)
```

We can now rewrite `freq.ml` to use `Counter`. Note that the resulting code can still be built with `build.sh`, since `ocamlbuild` will discover dependencies and realize that `counter.ml` needs to be compiled.

```
(* freq.ml: using Counter *)

open Core.Std

let rec build_counts counts =
  match In_channel.input_line stdin with
  | None -> counts
  | Some line -> build_counts (Counter.touch counts line)

let () =
  let counts = build_counts [] in
  let sorted_counts = List.sort counts
    ~cmp:(fun (_,x) (_,y) -> Int.descending x y)
  in
  List.iter (List.take sorted_counts 10)
    ~f:(fun (line,count) -> printf "%3d: %s\n" count line)
```

Signatures and Abstract Types

While we've pushed some of the logic to the `Counter` module, the code in `freq.ml` can still depend on the details of the implementation of `Counter`. Indeed, if you look at the invocation of `build_counts`:

```
  let counts = build_counts [] in
```

you'll see that it depends on the fact that the empty set of frequency counts is represented as an empty list. We'd like to prevent this kind of dependency, so that we can change the implementation of `Counter` without needing to change client code like that in `freq.ml`.

The first step towards hiding the implementation details of `Counter` is to create an interface file, `counter.mli`, which controls how `counter` is accessed. Let's start by writing down a simple descriptive interface, *i.e.*, an interface that describes what's currently available in `Counter` without hiding anything. We'll use `val` declarations in the `mli`, which have the following syntax

```
val <identifier> : <type>
```

and are used to expose the existence of a given value in the module. Here's an interface that describes the current contents of `Counter`. We can save this as `counter.mli` and compile, and the program will build as before.

```
(* counter.mli: descriptive interface *)

val touch : (string * int) list -> string -> (string * int) list
```

To actually hide the fact that frequency counts are represented as association lists, we need to make the type of frequency counts *abstract*. A type is abstract if its name is exposed in the interface, but its definition is not. Here's an abstract interface for `Counter`:

```
(* counter.mli: abstract interface *)

open Core.Std

type t

val empty : t
val to_list : t -> (string * int) list
val touch : t -> string -> t
```

Note that we needed to add `empty` and `to_list` to `Counter`, since otherwise, there would be no way to create a `Counter.t` or get data out of one.

Here's a rewrite of `counter.ml` to match this signature.

```
(* counter.ml: implementation matching abstract interface *)

open Core.Std

type t = (string * int) list

let empty = []

let to_list x = x

let touch t s =
  let count =
    match List.Assoc.find t s with
    | None -> 0
    | Some x -> x
  in
  List.Assoc.add t s (count + 1)
```

If we now try to compile `freq.ml`, we'll get the following error:

```
File "freq.ml", line 11, characters 20-22:
Error: This expression has type 'a list
       but an expression was expected of type Counter.t
```

This is because `freq.ml` depends on the fact that frequency counts are represented as association lists, a fact that we've just hidden. We just need to fix the code to use `Counter.empty` instead of `[]` and `Counter.to_list` to get the association list out at the end for processing and printing.

Now we can turn to optimizing the implementation of `Counter`. Here's an alternate and far more efficient implementation, based on the `Map` datastructure in Core.

```
(* counter.ml: efficient version *)

open Core.Std

type t = string Int.Map.t

let empty = Map.empty

let touch t s =
  let count =
    match Map.find t s with
    | None -> 0
    | Some x -> x
  in
  Map.add t s (count + 1)
```

```
let to_list t = Map.to_alist t
```

More on modules and signatures

Concrete types in signatures

In our frequency-count example, the module `Counter` had an abstract type `Counter.t` for representing a collection of frequency counts. Sometimes, you'll want to make a type in your interface *concrete*, by including the type definition in the interface.

For example, imagine we wanted to add a function to `Counter` for returning the line with the median frequency count. If the number of lines is even, then there is no precise median, so the function would return the two lines before and after the median instead. We'll use a custom type to represent the fact that there are two possible return values. Here's a possible implementation.

```
type median = | Median of string
              | Before_and_after of string * string

let median t =
  let sorted_strings = List.sort (Map.to_alist t)
      ~cmp:(fun (_,x) (_,y) -> Int.descending x y)
  in
  let len = List.length sorted_strings in
  if len = 0 then failwith "median: empty frequency count";
  let nth n = fst (List.nth_exn sorted_strings n) in
  if len mod 2 = 1
  then Median (nth (len/2))
  else Before_and_after (nth (len/2), nth (len/2 + 1));;
```

Now, to expose this usefully in the interface, we need to expose both the function and the type `median` with its definition. We'd do that by adding these lines to the `counter.mli`:

```
type median = | Median of string
              | Before_and_after of string * string

val get_median : t -> median
```

The decision of whether a given type should be abstract or concrete is an important one. Abstract types give you more control over how values are created and accessed, and makes it easier to enforce invariants beyond the what's enforced by the type itself; concrete types let you expose more detail and structure to client code in a lightweight way. The right choice depends very much on the context.

The `include` directive

OCaml provides a number of tools for manipulating modules. One particularly useful one is the `include` directive, which is used to include the contents of one module into another.

One natural application of `include` is to create one module which is an extension of another one. For example, imagine you wanted to build an extended version of the `List` module, where you've added some functionality not present in the module as distributed in Core. We can do this easily using `include`:

```
(* ext_list.ml: an extended list module *)

open Core.Std

(* The new function we're going to add *)
let rec intersperse list el =
  match list with
  | [] | [ _ ]   -> list
  | x :: y :: tl -> x :: el :: intersperse (y::tl) el

(* The remainder of the list module *)
include List
```

Now, what about the interface of this new module? It turns out that include works on the signature language as well, so we can pull essentially the same trick to write an `mli` for this new module. The only trick is that we need to get our hands on the signature for the list module, which can be done using `module type of`.

```
(* ext_list.mli: an extended list module *)

open Core.Std

(* Include the interface of the list module from Core *)
include (module type of List)

(* Signature of function we're adding *)
val intersperse : 'a list -> 'a -> 'a list
```

And we can now use `Ext_list` as a replacement for `List`. If we want to use `Ext_list` in preference to `List` in our project, we can create a file of common definitions:

```
(* common.ml *)

module List = Ext_list
```

And if we then put `open Common` after `open Core.Std` at the top of each file in our project, then references to `List` will automatically go to `Ext_list` instead.

Modules within a file

Up until now, we've only considered modules that correspond to files, like `counter.ml`. But modules (and module signatures) can be nested inside other modules. As a simple example, consider a program that needs to deal with some class of identifier like a username. Rather than just keeping usernames as strings, you might want to mint an abstract type, so that the type-system will help you to not confuse usernames with other string data that is floating around your program.

Here's how you might create such a type, within a module:

```
open Core.Std

module Username : sig
  type t
  val of_string : string -> t
  val to_string : t -> string
end = struct
  type t = string
  let of_string x = x
  let to_string x = x
end
```

The basic structure of a module declaration like this is:

```
module <name> : <signature> = <implementation>
```

We could have written this slightly differently, by giving the signature its own top-level `module type` declaration, making it possible to in a lightweight way create multiple distinct types with the same underlying implementation.

```
module type ID = sig
  type t
  val of_string : string -> t
  val to_string : t -> string
end

module String_id = struct
  type t = string
  let of_string x = x
  let to_string x = x
end

module Username : ID = String_id
module Hostname : ID = String_id

(* Now the following buggy code won't compile *)
type session_info = { user: Username.t;
                      host: Hostname.t;
                      when_started: Time.t;
                    }
```

```
let sessions_have_same_user s1 s2 =
  s1.user = s2.user
```

We can also combine this with the use of the include directive to add some extra functionality to such a module. Thus, we could have rewritten the definition of `Hostname` above as follows to add a function `Hostname.mine` that returns the hostname of the present machine.

```
module Hostname : sig
  include ID
  val mine : unit -> t
end = struct
  include String_id
  let mine = Unix.gethostname
end
```

Opening modules

One useful primitive in OCaml's module language is the `open` directive. We've seen that already in the `open Core.Std` that has been at the top of our source files.

The basic purpose of `open` is to extend the namespaces that OCaml searches when trying to resolve an identifier. Roughly, if you open a module `M`, then every subsequent time you look for an identifier `foo`, the module system will look in `M` for a value named `foo`. This is true for all kinds of identifiers, including types, type constructors, values and modules.

`open` is essential when dealing with something like a standard library, but it's generally good style to keep opening of modules to a minimum. Opening a module is basically a tradeoff between terseness and explicitness - the more modules you open, the harder it is to look at an identifier and figure out where it's defined.

Here's some general advice on how to deal with opens.

- Opening modules at the top-level of a module should be done quite sparingly, and generally only with modules that have been specifically designed to be opened, like `Core.Std` or `Option.Monad_infix`.
- One alternative to local opens that makes your code terser without giving up on explicitness is to locally rebind the name of a module. So, instead of writing:

  ```
  let print_median m =
     match m with
     | Counter.Median string -> printf "True median:\n   %s\n"
     | Counter.Before_and_after of before * after ->
       printf "Before and after median:\n   %s\n   %s\n" before after
  ```

  ...you could write

  ```
  let print_median m =
  ```

```
let module C = Counter in
match m with
| C.Median string -> printf "True median:\n   %s\n"
| C.Before_and_after of before * after ->
  printf "Before and after median:\n   %s\n   %s\n" before after
```

  Because the module name `C` only exists for a short scope, it's easy to read and remember what `C` stands for. Rebinding modules to very short names at the top-level of your module is usually a mistake.

- If you do need to do an open, it's better to do a *local open*. There are two syntaxes for local opens. For example, you can write:

  ```
  let average x y =
    let open Int64 in
    x + y / of_int 2
  ```

  In the above, `of_int` and the infix operators are the ones from `Int64` module.

  There's another even more lightweight syntax for local opens, which is particularly useful for small expressions:

  ```
  let average x y =
    Int64.(x + y / of_int 2)
  ```

Common errors with modules

When OCaml compiles a program with an `ml` and an `mli`, it will complain if it detects a mismatch between the two. Here are some of the common errors you'll run into.

Type mismatches

The simplest kind of error is where the type specified in the signature does not match up with the type in the implementation of the module. As an example, if we replace the `val` declaration in `counter.mli` by swapping the types of the first two arguments:

```
val touch : string -> t -> t
```

and then try to compile `Counter` (by writing `ocamlbuild -use-ocamlfind counter.cmo`), we'll get the following error:

```
File "counter.ml", line 1, characters 0-1:
Error: The implementation counter.ml
       does not match the interface counter.cmi:
       Values do not match:
         val touch :
           ('a, int) Core.Std.Map.t -> 'a -> ('a, int) Core.Std.Map.t
       is not included in
         val touch : string -> t -> t
```

This error message is a bit intimidating at first, and it takes a bit of thought to see where the first type, which is the type of [touch] in the implementation, doesn't match the second one, which is the type of [touch] in the interface. You need to recognize that [t] is in fact a [Core.Std.Map.t], and the problem is that in the first type, the first argument is a map while the second is the key to that map, but the order is swapped in the second type.

Missing definitions

We might decide that we want a new function in `Counter` for pulling out the frequency count of a given string. We can update the `mli` by adding the following line.

```
val count : t -> string -> int
```

Now, if we try to compile without actually adding the implementation, we'll get this error:

```
File "counter.ml", line 1, characters 0-1:
Error: The implementation counter.ml
       does not match the interface counter.cmi:
       The field `count' is required but not provided
```

A missing type definition will lead to a similar error.

Type definition mismatches

Type definitions that show up in an `mli` need to match up with corresponding definitions in the `ml`. Consider again the example of the type `median`. The order of the declaration of variants matters to the OCaml compiler so, if the definition of `median` in the implementation lists those options in a different order:

```
type median = | Before_and_after of line * line
              | Median of line
```

that will lead to a compilation error:

```
File "counter.ml", line 1, characters 0-1:
Error: The implementation counter.ml
       does not match the interface counter.cmi:
       Type declarations do not match:
         type median = Before_and_after of string * string | Median of string
       is not included in
         type median = Median of string | Before_and_after of string * string
       Their first fields have different names, Before_and_after and Median.
```

Order is similarly important in other parts of the signature, including the order in which record fields are declared and the order of arguments (including labeled and optional arguments) to a function.

Cyclic dependencies

In most cases, OCaml doesn't allow circular dependencies, *i.e.*, a collection of definitions that all refer to each other. If you want to create such definitions, you typically have to mark them specially. For example, when defining a set of mutually recursive values, you need to define them using `let rec` rather than ordinary `let`.

The same is true at the module level. By default, circular dependencies between modules is not allowed, and indeed, circular dependencies among files is never allowed.

The simplest case of this is that a module can not directly refer to itself (although definitions within a module can refer to each other in the ordinary way). So, if we tried to add a reference to `Counter` from within `counter.ml`:

```
let singleton l = Counter.touch Counter.empty
```

then when we try to build, we'll get this error:

```
File "counter.ml", line 17, characters 18-31:
Error: Unbound module Counter
Command exited with code 2.
```

The problem manifests in a different way if we create circular references between files. We could create such a situation by adding a reference to Freq from `counter.ml`, *e.g.*, by adding the following line:

```
let build_counts = Freq.build_counts
```

In this case, `ocamlbuild` will notice the error and complain:

```
Circular dependencies: "freq.cmo" already seen in
  [ "counter.cmo"; "freq.cmo" ]
```

2013-02-05
19:46:47

CHAPTER 10

Imperative Programming

Most of the code shown so far in this book, and indeed, most OCaml code in general, is *pure*. Pure code works without mutating the program's internal state, performing I/O, reading the clock, or in any other way interacting with changeable parts of the world. Thus, a pure function behaves like a mathematical function, always returning the same results when given the same inputs. *Imperative* code, on the other hand, operates by side-effects that modify a program's internal state or interacts with the outside world, and so can have a different effect, and return different results, every time they're called.

Pure code is the default in OCaml, and for good reason --- it's generally easier to reason about, less error prone and more composeable. But imperative code is of fundamental importance to any practical programming language because real-world tasks require that you interact with the outside world, whether by receiving a network packet or by writing data to a file. Imperative programming can also be important for performance. While pure code is quite efficient in OCaml, there are many algorithms that can only be implemented efficiently using imperative techniques.

OCaml offers a happy compromise here, making it easy and natural to program in a pure style, but also providing great support for imperative programming where you need it. This chapter will walk you through OCaml's imperative features, and help you use them to their fullest.

A simple dictionary

We'll walk through the implementation of a simple imperative dictionary, *i.e.*, a mutable mapping from keys to values. Both Core and OCaml's standard library provide multiple data structures for implementing such dictionaries, and for most real world tasks, you should use one of those. But we'll walk through this nonetheless as a way of seeing OCaml's imperative constructs in action.

We'll implement our dictionary as a hash table, based on an *open hashing* scheme, which is to say it will be structured as an array of buckets, with each bucket containing a list of key/value pairs that have been hashed into that bucket. For simplicity's sake, we'll use a fixed-length bucket array, though growing the bucket array as more elements are added would be necessary for a practical implementation.

First, we'll write down the `mli` for our dictionary.

```
(* file: dictionary.mli *)
open Core.Std

type ('a, 'b) t

val create : unit -> ('a, 'b) t
val length : ('a, 'b) t -> int
val add    : ('a, 'b) t -> key:'a -> data:'b -> unit
val find   : ('a, 'b) t -> 'a -> 'b option
val iter   : ('a, 'b) t -> f:(key:'a -> data:'b -> unit) -> unit
val remove : ('a, 'b) t -> 'a -> unit
```

The type `('a, 'b) t` is used to represent a dictionary with keys of type `'a` and data of type `'b`. The `mli` also includes a collection of helper functions. Notice that a number of the functions, in particular, ones like `add` that modify the dictionary, return unit. This is typical of functions that act by side-effect.

The implementation is shown below. We'll go through the code bit by bit, explaining different imperative constructs as they show up.

First, we define the type of a dictionary as a record with two fields. The first field, `length` is declared as mutable; records are immutable by default, but individual fields are mutable when explicitly marked as such. The second field, `buckets`, is immutable, but contains an array, which is itself a mutable data structure, as we'll see.

```
(* file: dictionary.ml *)
open Core.Std

type ('a, 'b) t = { mutable length: int;
                    buckets: ('a * 'b) list array;
                  }
```

Next, we'll define the function for choosing a hash-bucket based on the key. We do this using `Hashtbl.hash` to compute a hash value for a key. `Hashtbl.hash` is a special polymorphic function (meaning it can take an input of any type) provided by the OCaml runtime that can compute a hash for almost any OCaml value. (While it can be applied to any type as far as the type-system is concerned, it won't always succeed. `Hashtbl.hash` will throw an exception if given a value it can't handle, like a function or a value from a C libraries that lives outside the OCaml heap.)

There's also code for `create`, which creates an empty dictionary, `length`, which grabs the length from the corresponding record field, and `find`, which looks for a matching key in the table using `List.find_map` on the corresponding bucket. In `find`, you'll notice that we make use of the `array.(index)` syntax for looking up a value in an array.

```
let num_buckets = 17
let hash_bucket key = (Hashtbl.hash key) mod num_buckets

let create () =
  { length = 0;
    buckets = Array.create ~len:num_buckets [];
  }

let length t = t.length

let find t key =
  List.find_map t.buckets.(hash_bucket key)
    ~f:(fun (key',data) -> if key' = key then Some data else None)
```

The `iter` function, shown below, is designed to walk over all the entries in the dictionary. In particular, `iter d ~f` will call `f` for each key/value pair in dictionary `d`.

The code for `iter` uses two forms of iteration: a `for` loop is used to iterate over the array of buckets; and within that loop, `List.iter` is used to walk over the list of values in a given bucket. `for` loops are syntactically convenient, and are more familiar for someone coming from an imperative language. But they're not of fundamental importance to the language: instead of using `for`, we could have implemented the outer loop using the `Array.iter`, which in turn could be implemented as a recursive function.

```
let iter t ~f =
  for i = 0 to Array.length t.buckets - 1 do
    List.iter t.buckets.(i) ~f:(fun (key, data) -> f ~key ~data)
  done
```

It's worth noting that in such an iteration function, the function `f` is expected to return unit, because it is supposed to operate by side effect rather than by returning a value.

Next, we have the code for adding and removing mappings from the dictionary. The code is made more complicated by the fact that we need to detect whether we are overwriting or removing an existing binding, so we can decide whether `t.length` needs to be changed. The helper function `bucket_has_key` is used to answer this question.

This code uses the `<-` operator for updating elements of an array (`array.(i) <- expr`) and for updating a record field (`record.field <- expression`). In addition, we see the use of the semicolon to sequence the imperative operations that are the last two lines of the function.

Note also that we do all of the side-effecting operations at the very end of each function. This is good practice because it minimizes the chance that such operations will be interrupted with an exception, leaving the data structure in an inconsistent state.

```
let bucket_has_key t i key =
  List.exists t.buckets.(i) ~f:(fun (key',_) -> key' = key)

let add t ~key ~data =
    let i = hash_bucket key in
    let replace = bucket_has_key t i key in
    let filtered_bucket =
      if replace then
        List.filter t.buckets.(i) ~f:(fun (key',_) -> key' <> key)
      else
        t.buckets.(i)
    in
    t.buckets.(i) <- (key, data) :: filtered_bucket;
    if not replace then t.length <- t.length + 1

  let remove t key =
    let i = hash_bucket key in
    if not (bucket_has_key t i key) then ()
    else (
      let filtered_bucket =
        List.filter t.buckets.(i) ~f:(fun (key',_) -> key' <> key)
      in
      t.buckets.(i) <- filtered_bucket;
      t.length <- t.length - 1
    )
```

Mutable data structures

We've already encountered two different forms of mutable data: records with mutable fields, and arrays. We'll discuss those, and a few other forms of mutable data, in more detail now.

Array-like data

OCaml supports a number of array-like data structures; *i.e.*, integer-indexed containers. The `array` type is one example, and the `Array` module comes with a variety of mutable operations on arrays, including `Array.set`, which modifies an individual element, and `Array.blit`, which efficiently copies a range of values in an array.

Arrays also come with a special syntax for getting an element from an array: `array.(index)`; and for setting an element: `array.(index) <- expr`. Literal arrays can be declared using [| and |] as delimiters. Thus, `[| 1; 2; 3 |]` is an integer array.

Strings are essentially byte-arrays, with some extra useful functions in the `String` module for dealing with textual data in this form. The main reason to use a `String.t` rather than a `Char.t array` (a `Char.t` is an 8-bit character) is that the former is considerably more space efficient; an array uses one word --- 8 bytes on a 64-bit machine --- to store a single entry, whereas strings use one byte per character.

Strings also come with their own syntax for getting and setting values: `string.[index]` and `string.[index] <- expr` respectively, and string literals are bounded by quotes.

A bigarray is a handle to a block of memory stored outside of the OCaml heap. These are mostly useful for interacting with C or Fortran libraries, and are discussed in Chapter 20. Bigarrays too have their own getting and setting syntax: `bigarray.{index}` and `bigarray.{index} <- expr`.

Mutable record and object fields and ref cells

As we've seen, records are immutable by default, but individual record fields can be declared as mutable. These mutable fields can be set using the `<-` operator, *i.e.*, `record.field <- expr`.

As we'll see in Chapter 11, fields of an object can similarly be declared as mutable, and can then be modified in much the same way as with records.

Ref Cells

Variables in OCaml are never mutable --- they can refer to mutable data, but what the variable points to can't be changed. Sometimes, though, you want to do exactly what you would do with a mutable variable in another language: define a single, mutable value. In OCaml this is typically achieved using a `ref`, which is essentially a container with a single, mutable value.

The standard library defines the following operators for working with refs.

- `ref expr` constructs a reference cell containing the value defined by the expression `expr`.
- `! refcell` returns the contents of the reference cell.
- `refcell := expr` replaces the contents of the reference cell.

You can see these in action below.

```
# let x = ref 1;;
val x : int ref = {contents = 1}
# !x;;
- : int = 1
# x := !x + 1;;
- : unit = ()
# !x;;
- : int = 2
```

There's nothing magic about a `ref`: it's really just a record. The `ref` type and its associated operations are defined as follows.

```
type 'a ref = { mutable contents : 'a }

let ref x = { contents = x }
let (!) r = r.contents
let (:=) r x = r.contents <- x
```

Foreign functions

Another source of imperative operations in OCaml is resources that come from interfacing with some external library through OCaml's foreign function interface (FFI). The FFI opens OCaml up to any imperative construct that is exported by a system call, a C library, or any other external resource that you connect to. Many of these come built in, like access to the `write` system call, or to the `clock`; while others come from user libraries, like LAPACK bindings.

More generally, when you wrap a library for use in OCaml, you'll often find yourself introducing new imperative operations to the language.

`for` and `while` loops

OCaml provides support for traditional imperative looping constructs, in particular, `for` and `while` loops, even though neither of them is strictly necessary. Anything you can do with such a loop you can also do with a recursive function, and you can also write higher-order functions like `Array.iter` that cover much of the same ground.

Nonetheless, explicit `for` and `while` loops are both more idiomatic for imperative programming and often more concise.

The `for` loop is the simpler of the two. Indeed, we've already seen the `for` loop in action --- the `iter` function in `Dictionary` is built using it. Here's a simple example of `for`.

```
# for i = 0 to 3 do Printf.printf "i = %d\n" i done;;
i = 0
i = 1
i = 2
i = 3
- : unit = ()
```

As you can see, the upper and lower bounds are inclusive. We can also use `downto` to iterate in the other direction.

```
# for i = 3 downto 0 do Printf.printf "i = %d\n" i done;;
i = 3
i = 2
i = 1
i = 0
- : unit = ()
```

A `while`-loop on the other hand, takes a condition and a body, and repeatedly runs the body until the condition is false. Here's a simple example of a function for reversing an array in-place.

```
# let rev_inplace ar =
    let i = ref 0 in
    let j = ref (Array.length ar - 1) in
    (* terminate when the upper and lower indices meet *)
    while !i < !j do
      (* swap the two elements *)
      let tmp = ar.(!i) in
      ar.(!i) <- ar.(!j);
      ar.(!j) <- tmp;
      (* bump the indices *)
      incr i;
      decr j
    done
  ;;
val rev_inplace : 'a array -> unit = <fun>
# let nums = [|1;2;3;4;5|];;
val nums : int array = [|1; 2; 3; 4; 5|]
# rev_inplace nums;;
- : unit = ()
# nums;;
- : int array = [|5; 4; 3; 2; 1|]
```

In the above, we used `incr` and `decr`, which are functions for incrementing and decrementing an `int ref` by one, respectively.

Doubly-linked lists

Another common imperative data structure is the doubly-linked list, which allows traversal in both directions, as well as constant-time deletion of any element. Doubly-linked lists are a cyclic data structure, meaning that it is possible to follow a nontrivial sequence of pointers from an element, through other elements, back to itself. In general, building cyclic data structures requires the use of side-effects (although in some limited cases, they can be constructed using `let rec`). This is done by constructing the data elements first, and then adding cycles using assignment afterwards.

Core defines a standard doubly-linked list, but let's define our own implementation for illustration. Here's the `mli` we'll start out with. Note that there are two types defined here: `'a t`, the type of a list, and `'a element`, the type of an individual element of a list. Elements act as pointers to the interior of a list, and allow us to navigate the list and give us a point at which to apply mutating operations.

```
(* file: dlist.mli *)
open Core.Std

type 'a t
type 'a element

(** Basic list operations  *)
val create   : unit -> 'a t
val is_empty : 'a t -> bool

(** navigating the [elements] of a list *)
val first : 'a t -> 'a element option
val next  : 'a element -> 'a element option
val prev  : 'a element -> 'a element option
val value : 'a element -> 'a

(** Simple iteration functions *)
val find : 'a t -> 'a -> 'a element option
val iter : 'a t -> f:('a -> unit) -> unit

(** mutators *)
val insert_first : 'a t -> 'a -> 'a element
val insert_after : 'a element -> 'a -> 'a element
val remove : 'a t -> 'a element -> unit
```

Now let's look at the implementation. We'll start by defining our two types. An `'a element` is a record containing the value to be stored in that node as well as optional (and mutable) fields pointing to the previous and next elements. These fields are optional so that at the beginning of the list, the `prev` field can be `None`, and at the end of the list, the `next` field can be `None`.

The type of the list itself, `'a t`, is an optional, mutable reference to an `element`. This reference is `None` if the list is empty, and `Some` otherwise.

```
open Core.Std

type 'a element =
  { value : 'a;
    mutable next : 'a element option;
    mutable previous : 'a element option
  }

type 'a t = 'a element option ref
```

Now we can define some functions using our list type. The following functions are all relatively straight-forward, largely following from our type definitions.

```
let create () = ref None
let is_empty l = !l = None

let value elt = elt.value

let first l = !l
let next elt = elt.next
let prev elt = elt.prev
```

Next, let's define the function that inserts a value into the list as a new first element. We define a new element `elt`, link it into the list, and set the list reference.

```
let insert_first l value =
  let elt = { previous = None; next = !l; value } in
  begin match !l with
  | Some old_first -> old_first.previous <- Some elt
  | None -> ()
  end;
```

```
    l := Some elt;
    elt
```

This example uses the sequencing operator `;` to separate the steps to be executed in order: first, create the new element `elt`; then set `old_first.previous` to point to it; then set the list `l` to refer to the `elt` element; then return the element `elt`.

In general, when a sequence expression `expr1; expr2` is evaluated, `expr1` is evaluated first, and then `expr2`. The expression `expr1` must have type `unit`, and the the value of `expr2` is returned as the value of the entire sequence. For example, the sequence `print_string "hello world"; 1 + 2` first prints the string `"hello world"`, then returns the integer `3`.

There are a few more things to note. First, `;` is a *separator* in OCaml, not a terminator as it is in C or Java. The compiler is somewhat relaxed about parsing a terminating semicolon, so it may work for you, but you should not rely on it.

```
# let i = print_string "Hello world\n"; 2; in i;;
Hello world
- : int = 2
```

Also note, the precedence of a `match` expression is very low, so to separate it from the following assignment `l := Some new_front`, we surround the match in a `begin ... end` bracketing (we could also use parentheses). If we did not, the final assignment would become part of the `None -> ...` case, which is not what we want.

To complete this initial part of the implementation, let's define a function for removal.

```
let remove l elt =
  let { previous = previous; next = next } = elt in
  (match previous with
   | Some p -> p.next <- next
   | None -> check_is_first_element l elt; l := next);
  (match next with
   | Some n -> n.previous <- previous;
   | None -> ());
  elt.previous <- None;
  elt.next <- None
```

The `remove` function unlinks the element from the list, the resets the `previous` and `next` links to `None`. For safety, we detect duplicate removals with a check. In the case where the previous element is `None`, the element must be first in the list. Here is the implementation of the `check_is_first_element` function.

```
let check_is_first_element l elt1 =
   match !l with
   | Some elt2 when elt1 == elt2 -> ()
   | _ -> raise (Invalid_argument "element has already been removed")
```

Iteration

When defining containers like lists, dictionaries, trees, etc. it is conventional to define some kind of iteration to allow the elements of the collection to be enumerated. When the containers are immutable, like `'a list`, this is normaly done with functions like `iter`, `map`, and `fold`. Each of these iteration functions takes a function that will be applied to each of the elements in order.

```
# List.iter  [1; 2; 3] ~f:(fun i -> Printf.printf "Element: %d\n" i);;
Element: 1
Element: 2
Element: 3
- : unit = ()
# List.map [1; 2; 3] ~f:((+) 10);;
- : int list = [11; 12; 13]
```

Defining this for doubly-linked lists is simple enough. The following function iterates through the list, applying the function `f` to each element in turn.

```
let iter l ~f =
  let rec loop = function
  | Some { value = v; next = next } -> f v; loop next
  | None -> ()
  in
  loop !l;;

# let l = create ();
  push_front l 1;
  push_front l 2;
  push_front l 3;
  iter l ~f:(Printf.printf "Item: %d\n");;
Item: 3
Item: 2
Item: 1
```

This style of iteration is concise and completely general when you want to iterate through all of the elements of the list. In some cases, you may wish to iterate through only part of the list, for example when searching for a particular element. In this case, you can use the functions `next` and `prev` to navigate through the list.

```
   (* Find the element containing x, using = for comparison *)
   let find l x : 'a element option =
      let rec search = function
      | None -> None
      | Some elt ->
           if value elt = x then
              Some elt
           else
              search (next elt)
      in
      search !l
```

Java-style iteration

One possible issue with navagation using the `next` and `prev` functions is that code is tied specifically to the doubly-linked list code we have just defined. It won't work for iteration through a hash-table, for example. Another technique used in other imperative languages like Java or C++ is to define a generic `iterator` type that can be used to enumerate and/or mutate the elements in multiple different kinds of containers. Here is the Java interface.

```
public iterface Iterator {
  public boolean hasNext();
  public Object next();
  public void remove();
};
```

At any time, a `Iterator` object refers (optionally) to some element of a container. The `hasNext()` method returns true if the iterator refers to an element; the method `next()` returns the element, and also advances to the next one; and `remove()` removes the last element returned by the iterator.

If we want to define a similar iterator concept in OCaml, we need to choose how to represent it. We *could* define a separate iterator type for each kind of container, but this would be inconvenient, since iterators have similar behavior for many different kinds of containers. To define a *generic* iterator, there are several reasonable choices: we can use first-class modules, or we can use objects. One of the simpler approaches is to use objects.

You can skip forward to the Objects chapter for more informatation about objects, but we'll be using basic objects, which are just collections of methods, similar to having a record of functions -- we could also implement the iterator as a record of functions, but the code would be somewhat more verbose.

First, we need to define a generic iterator type. For clarity, we'll use a more verbose type than in Java. We'll separate retrieving a value from advacing to the next element. The object type is specified like a record type, but using angle brackets `< ... >`.

```
type 'a iterator =
   < has_value : bool;
     value : 'a;
     next : unit;
     remove : unit;
     insert_after : 'a -> unit
   >
```

Each of the labeled parts `has_value`, `value`, etc. are object *methods*. This object type corresponds to an *interface* consisting of a set of methods.

Next, to define the iterator implementation, we implement each of the methods, bracketed by `object ... end`, declaring each method with the `method` keyword.

```
let iterator (list : 'a dlist) : 'a iterator =
  let current = ref !list in
  object
    method has_value = !current <> None
    method value =
      match !current with
      | Some { value = v } -> v
      | None -> raise (Invalid_argument "next")
    method next =
      match !current with
      | Some { next = next } -> current := next
      | None -> raise (Invalid_argument "next")
    method remove =
      match !current with
      | Some elt ->
           current := elt.next;
           remove list elt  (* This is the 'remove' function above *)
      | None -> raise (Invalid_argument "remove")
    method insert_after value =
       match !current with
       | Some elt -> ignore (insert_after elt value)  (* 'insert_after' above *)
       | None -> raise (Invalid_argument "insert_after")
  end
```

The reference cell `current` holds the current position in the list. The method `has_value` returns true if `current` refers to an element, `value` returns the element, and `next` advances the iterator. The method `remove` unlinks the `current` element by setting the previous element's `next` pointer, and the next's elements `previous` pointer, then advancing `current` to the next element. The following example illustrates the semantics.

```
# let () =
    Printf.printf "\nDList2\n";
    let l = create () in
    let _ = insert_first l 1 in
    let _ = insert_first l 2 in
    let _ = insert_first l 3 in

    let it = iterator l in
    while it#has_value do
      Printf.printf "Item: %d\n" it#value;
      it#next
    done;;
Item: 3
Item: 2
Item: 1
- : unit = ()
```

Note that the doubly-linked list is a *cyclic* data structure. Most notably, the builtin equality *does not work* in general with cyclic values.

```
# let l2 = create();
val l2 : '_a dlist
```

```
# insert_first l2 1; insert_first l2 3;;
- : unit = ()
# l == l2;;
- : bool = false
# l = l2;;
Out of memory during evaluation.
```

Hash tables with iterators

Let's return to the example of hash tables, but this time let's define an iterator-style interface. We'll use a similar `iterator` object type like we did for doubly-linked lists, but this time the iteration is over key/value pairs. The signature changes slightly, the main change being that the `find` function returns an iterator. This allows retrieval of the value associated with a key, and it also allows the entry to be deleted.

```
module Iterable_dictionary : sig
  type ('a, 'b) t

  val create : unit -> ('a, 'b) t
  val add : ('a, 'b) t -> key:'a -> data:'b -> unit
  val iterator : ('a, 'b) t -> ('a * 'b) iterator
  val find : ('a, 'b) t -> key:'a -> ('a * 'b) iterator
end
```

The implementation of `Iterable_dictionary` is similar to the original `Dictionary` using lists, except now we will use doubly-linked lists. The `create` function creates an array of doubly-linked lists. The `add` function first removes any existing entry, then add the new element to the front of the bucket.

```
module Iterable_dictionary = struct
  type ('a, 'b) t = ('a * 'b) DList.t array

  let num_buckets = 17
  let hash_bucket key = (Hashtbl.hash key) mod num_buckets

  let create () = Array.init num_buckets (fun _ -> DList.create ())

  let add table ~key ~data =
    let index = hash_bucket key in
    let it = DList.find table.(index) ~data:(key, data) in
    if it#has_value then it#remove;
    DList.insert_first table.(index) (key, value)

  ...
end
```

We can define iterators in the hash table as a pair of a bucket index and `DList` iterator into the bucket. To define this as an object, we'll introduce a few more object concepts, including mutable fields, private methods, and initializers. The function `make_iterator` `table index_ dlist_it_` returns an iterator for the bucket with index `index_` and list iterator `dlist_it_`.

```
let make_iterator table index_ dlist_it_ =
  object (self)
    val mutable index = index_
    val mutable dlist_it = dlist_it_
    method has_value = dlist_it#has_value
    method value = dlist_it#value
    method next =
       dlist_it#next;
       self#normalize
    method remove =
       dlist_it#remove;
       self#normalize
    method private normalize =
      while not dlist_it#has_value && index < num_buckets - 1 do
        index <- index + 1;
        dlist_it <- DList.iterator table.(index)
      done
    initializer self#normalize
  end
```

The iterator implementation relies on a "normal" form, where the list iterator *always* refers to an element. This is handled by the `normalize` method, which advances past empty buckets until either a non-empty bucket is found, or the end of the table is reached.

The `normalize` method is declared as `private`, so that it does not appear as part of the iterator type. The `has_value` and `value` methods delegate directly to the list iterator. The `next` and `remove` methods also delagate to the list iterator; however, since the iterator has been mutated, the `normalize` method is called to advance to the next element.

There are several more things to note. The syntax `object (self) ... end` means that the variable `self` refers to the object itself, allowing other method in the object to be called (like `self#normalize`). The fields `index` and `dlist_it` are declared as `val mutable`, which means that they can be modified by assignment using the `<-` syntax seen in the `normalize` method. Finally, the object also has an `initializer` expression, which is called when the object is first created, in this case normalizing the iterator.

Now that the iterator is defined, we can complete the `Iterable_dictionary` implementation.

```
let iterator table =
  make_iterator table 0 (DList.iterator table.(0))

let find table ~key =
  let index = hash_bucket key in
  let it = DList.iterator table.(index) in
  while it#has_value && fst it#value <> key do
    it#next
    done;
    if it#has_value then
       make_iterator table index it
    else
       make_iterator table num_buckets it
```

The `iterator` function returns in iterator that refers to the first element in the table (if the table is non-empty). The `find` function searches for an element in the table, returning an iterator referring to that value if found, or else the an iterator at the end of the table.

Iteration over the hash table is much the same as through a doubly-linked list. Note that the elements are reordered in the hash table.

```
# let () =
  let module IHM = Iterable_dictionary in
  let table = IHM.create () in
  IHM.add table ~key:"small" ~data:1.00;
  IHM.add table ~key:"medium" ~data:1.50;
  IHM.add table ~key:"large" ~data:2.25;
  IHM.add table ~key:"enormous" ~data:5.00;

  let it = IHM.iterator table in
  while it#has_value do
    let size, price = it#value in
    Printf.printf "Size %s is $%.02f\n" size price;
    it#next
  done;;
Size large is $2.25
Size medium is $1.50
Size small is $1.00
Size enormous is $5.00
- : unit = ()

# let () =
  let it = IHM.find table "enormous" in
  it#remove;

  let it = IHM.iterator table in
  while it#has_value do
    let size, price = it#value in
    Printf.printf "Size %s is $%.02f\n" size price;
    it#next
  done;;
Size large is $2.25
Size medium is $1.50
Size small is $1.00
- : unit = ()
```

Lazy computation

There are many instances where imperative programming is used to modify or improve the performance characteristics of a program, without otherwise changing the behavior. In other words, the program could be written without side-effects, but performance is improved by techniques like lazy computation, caching, memoization, etc.

One of the simplest of these is the builtin lazy computation. The keyword `lazy` can be used to prefix any expression, returning a value of type `'a Lazy.t`. The computation is delayed until forced with the `Lazy.force` function, and then saved thereafter.

```
# let v = lazy (print_string "performing lazy computation\n"; 1);;
val v : int lazy_t = <lazy>
# Lazy.force v;;
performing lazy computation
- : int = 1
# Lazy.force v;;
- : int = 1
```

The builtin `lazy` computation has a nice syntax, but the technique is pretty generic, and we can implement it with a mutable value.

```
module ImpLazy : sig
   type 'a t

   val create : (unit -> 'a) -> 'a t
   val force : 'a t -> 'a
end = struct
   type 'a delayed = Delayed of (unit -> 'a) | Value of 'a
   type 'a t = 'a delayed ref

   let create f = ref (Delayed f)
   let force v =
     match !v with
     | Delayed f ->
          let x = f () in
          v := Value x;
          x
     | Value x ->
          x
end;;
```

The `'a delayed` type contains a delayed value represented as a function, or else an actual value. The `ImpLazy.force` function forces the computation; if it is delayed, the function is evaluated, and the value is mutated to save the resulting value. Subsequent calls to `ImpLazy.force` will fall into the `Value` case, without needing to reevaluate the function. The main difference between our module `ImpLazy` and the builtin module `Lazy` is the nice syntax for the latter. Rather than writing `ImpLazy.create (fun () -> e)`, the builtin syntax is just `lazy e`.

Memoization

We can generalize lazy computations to function *memoization*, where we save the result of function applications to avoid their recomputation. One simple implementation is to use a hash table to save the values by side effect.

```
module Memo : sig
   type ('a, 'b) t

   val create : unit -> ('a, 'b) t
   val apply : ('a, 'b) t -> func:('a -> 'b) -> arg:'a -> 'b
end = struct
   type ('a, 'b) t = ('a, 'b) Dictionary.t

   let create = Dictionary.create
   let apply table ~func ~arg =
     match Dictionary.find table ~key:arg with
     | Some x -> x
     | None ->
         let x = func arg in
         Dictionary.add table ~key:arg ~data:x;
         x

end;;
```

Memoization is useful for *dynamic programming*, where problems are solved by breraking them down into simpler subproblems. If subproblems occur more than once, memoization can be used to avoid recomputing the subproblem. A canonical example of this is the Fibonacci sequence, which is defined by the following program, which produces the sequence *0, 1, 1, 2, 3, 5, 8, 13, 21, ...* starting from 0.

```
let rec fib i = if i <= 1 then i else fib (i - 1) + fib (i - 2);;
```

The complexity of this function is exponential *O(2^i)*, because for large inputs the function computes two similar-sized subproblems. To illustrate, let's time the computation using the `Sys.time` function to measure the wall clock.

```
# let time f x =
    let start = Sys.time () in
    let y = f x in
    Printf.printf "Time: %g sec\n" (Sys.time () -. start);
    y;;
val time : ('a -> 'b) -> 'a -> 'b = <fun>
# time fib 40;;
Time: 5.53724 sec
- : int = 102334155
```

Next, let's construct a memoized version of the function, where the recursive calls are made through a memo table. This makes a dramatic improvement in performance. Since the recursive calls are computed just once, the complexity is linear, and the computation is fast.

```
# let memo_fib =
    let memo = Memo.create () in
    let rec fib i =
      if i <= 1 then
         i
      else
         Memo.apply memo ~func:fib ~arg:(i - 1) +
         Memo.apply memo ~func:fib ~arg:(i - 2)
    in
    fib;;
val memo_fib : int -> int = <fun>
# time memo_fib 40;;
Time: 3.7e-05 sec
- : int = 102334155
```

Note that this use of memoization relies on side-effects to cache intermediate computations, but it doesn't change the values of the function. Its purpose is to improve performance.

Hash consing

"Hash consing" is a technique to share values that are structurally equal. The term comes from Lisp, where the technique is used to share s-expressions that are equal. In some situations, hash-consing can result in dramatic performance improvements in two ways. First, space can be reduced by using a single physical representation for values that are equal. Second, values can be compared for equality using the constant-time physical equality operator `==`.

One of the simplest ways to implement hash-consing is to use a hash-table to remember (memoize) values that have already been created. To illustrate, let's define a kind of numerical expression `Exp.t` consisting of integers, variables, addition, and multiplication (we can define more operators, but let's keep the example simple).

The type `Exp.t` is declared as `private`, meaning that pattern matching can be used on the expressions outside the module, but expressions can't be constructed without explicitly using the constructors `num`, `var`, `plus`, `times` provided by the `Exp` module. These functions enforce the hash-consing, ensuring that structurally equal expressions are mapped to physically equal representations.

```
module Exp : sig
  type t = private
  | Num of int
  | Var of string
  | Plus of t * t
  | Times of t * t

  val num : int -> t
  val var : string -> t
  val plus : t -> t -> t
  val times : t -> t -> t
end = struct
  type t =
  | Num of int
  | Var of string
  | Plus of t * t
  | Times of t * t

  let table = Dictionary.create ()
  let merge exp =
    match Dictionary.find table ~key:exp with
    | Some x -> x
    | None ->
         Dictionary.add table ~key:exp ~data:exp;
         exp

  let num i = merge (Num i)
  let var s = merge (Var s)
  let plus e1 e2 = merge (Plus (e1, e2))
  let times e1 e2 = merge (Times (e1, e2))
end;;
```

The implementation defines a hash table `table`, and a `merge` function that merges an expression into the table, returning the previous value if there was one, or inserting a new value if there is not. The constructors can rely on the fact that subexpressions have already been hash-consed, so they simply call the merge function to memoize the value.

Note that expressions that are structurally equal are now also physically equal.

```
# let e1 = Exp.times (Exp.num 10) (Exp.plus (Exp.var "x") (Exp.var "y"));;
val e1 : Exp.t = Exp.Times (Exp.Num 10, Exp.Plus (Exp.Var "x", Exp.Var "y"))
# let e2 = Exp.times (Exp.num 10) (Exp.plus (Exp.var "x") (Exp.var "y"));;
val e2 : Exp.t = Exp.Times (Exp.Num 10, Exp.Plus (Exp.Var "x", Exp.Var "y"))
# e1 == e2;;
- : bool = true
```

Expressions that are not equal are equal are not physically equal either, however common subexpressions are equal.

```
# let e3 = Exp.times (Exp.num 10) (Exp.plus (Exp.var "z") (Exp.var "y"));;
val e3 : Exp.t = Exp.Times (Exp.Num 10, Exp.Plus (Exp.Var "z", Exp.Var "y"))
# e1 == e3;;
- : bool = false
# let Exp.Times (a1, Exp.Plus (_, a2)) = e1;;
val a1 : Exp.t = Exp.Num 10
val a2 : Exp.t = Exp.Var "y"
# let Exp.Times (b1, Exp.Plus (_, b2)) = e3;;
val b1 : Exp.t = Exp.Num 10
val b2 : Exp.t = Exp.Var "y"
# a1 == b1;;
- : bool = true
# a2 == b2;;
- : bool = true
```

Weak hash consing

There is two issues with hash-consing as we have just defined it. A minor problem is that hashing is linear in the size of the expression. This can be fixed by storing the hash code in the expression itself, avoiding the recursive computation. If expressions are small, this won't be much of a benefit, but it can save time if large expressions are frequently constructed.

A more serious problem is that the hash table holds onto expressions *forever*, even if they are no longer used in the program. This can result in a space leak that cancels out any space saving we had in the first place, perhaps making it even worse.

To deal with this problem, we can use "weak" hash tables, implemented in the `Weak` module in the OCaml standard library. The main difference is that a weak table may drop values that are no longer being used elsewhere. Weak tables are tied into the garbage collector, which removes values that are no longer live.

We define the type `WExp.t` much as before, except including the hash code for the `Plus` and `Times` expressions.

The weak hash table requires that hash and equality functions be provided explicitly, so we construct a module `HashExp` that defines the `equal` and `hash` functions. Note that equality and hashing are both constant-time functions -- equality can rely on physical equality of subexpressions, and hashing can use the explcitly represented hash values.

The module `WeakHash` has the semantics of a set of elements. The `WeakHash.merge` function retrieves an element if it already exists, or adds it otherwise. The constructors are much as before.

```
module WExp : sig
  type t = private
  | Num of int
  | Var of string
  | Plus of int * t * t
  | Times of int * t * t

  val num : int -> t
  val var : string -> t
  val plus : t -> t -> t
  val times : t -> t -> t
end = struct
  type t =
  | Num of int
  | Var of string
  | Plus of int * t * t
  | Times of int * t * t

  module HashExp = struct
    type exp = t
    type t = exp
    let equal e1 e2 =
      match e1, e2 with
      | Num i1, Num i2 -> i1 = i2
      | Var v1, Var v2 -> v1 = v2
      | Plus (_, a1, a2), Plus (_, b1, b2)
      | Times (_, a1, a2), Times (_, b1, b2) ->
           a1 == b1 && a2 == b2
      | _ -> false
    let hash = function
    | Num i -> i lxor 0xabababab
    | Var v -> (Hashtbl.hash v) lxor 0xcdcdcdcdc
    | Plus (hash, _, _)
    | Times (hash, _, _) -> hash
  end

  module WeakHash = Weak.Make (HashExp);;

  let table = WeakHash.create 17
  let merge e = WeakHash.merge table e

  let num i = merge (Num i)
  let var s = merge (Var s)
  let plus e1 e2 =
     let hash = (HashExp.hash e1) lxor (HashExp.hash e2) lxor 0x12345678 in
     merge (Plus (hash, e1, e2))

  let times e1 e2 =
     let hash = (HashExp.hash e1) lxor (HashExp.hash e2) lxor 0xdeadbeef in
     merge (Times (hash, e1, e2))
end;;
```

Concurrency

Concurrency is another tool OCaml programmers can use to simplify programs in certain cases. In OCaml, threads do not introduce parallelism; only one thread may be running at a time. However, threads can be used to simplify control flow -- a program can devote some threads to reading input, others for performing work, others for producing output, etc.

The OCaml standard library supports threads, where individual threads of control can be created that run concurrently (but only one thread at a time), and context switches are *involuntary*, meaning that a thread may be preempted at any time.

When threads share imperative state, this gives rise to the standard synchronization issues, where multiple threads may be mutating shared state at the same time. To illustrate, let's write a program with two threads that increment a shared reference cell concurrently.

```
let value = ref 0

let loop () =
   for i = 1 to 3 do
```

```
      let i = !value in
      Printf.printf "i = %d\n" i;
      flush stdout;
      value := i + 1
    done

let thread1 = Thread.create loop ();;
let thread2 = Thread.create loop ();;

Thread.join thread1;;
Thread.join thread2;;

Printf.printf "value = %d\n" !value
```

The reference cell `value` holds a shared value that is incremented 10 times, in a loop, by the `loop` function. Each iteration of the loop prints the current value, then assigns the new value. We create two threads with the `Thread.create` expressions, then use `Thread.join` to block until the threads terminate.

The exact behavior of the program is nondeterminstic -- it depends on the relative sopeed of the two theads. One output is listed below.

```
i = 0
ii  ==  01

ii  ==  12

i = 2
value = 3
```

This represents a nearly perfect interleaving of the thread executions. Each thread fetches the value from the `value` reference cell, prints the value, then performs the assignment. Since both threads effectively run in lockstep, the final value in the `value` reference cell is the same as if there were just one thread running.

If this is not the behavior that was expected, one solution is to use a `Mutex` to ensure that the increment operation is atomic. We can allocate a lock with `Mutex.create`, then acquire the lock in the loop body.

```
let value = ref 0
let mutex = Mutex.create ()

let loop () =
   for i = 1 to 3 do
      Mutex.lock mutex;
      let i = !value in
      Printf.printf "i = %d\n" i;
      flush stdout;
      value := i + 1;
      Mutex.unlock mutex
   done
```

When we run this program, it produces a deterministic output.

```
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
value = 6
```

Dealing with concurrency

In general the interaction of concurrency with imperative programs causes problems with races. There are many techniques you can use to address the issue.

- Do not use assignment, mutable data structures, or perform input/output in threads.
- Use *cooperative* multitasking, where only one thread runs at a time, and context switches are *voluntary*. This is the dominant model in `Async`.
- Do not share mutable data between threads. In practice, this usually includes explicit communication channels between threads that otherwise have isolated state.
- Give in, and use threads, and the standard synchronization toolkit that comes with OCaml, including locks (`Mutex`), condition variables, etc.

Out of all of these choices, the simplest one is to use the `Async` model and cooperative multitasking. However, let's go ahead and work through some examples of using traditional concurrent programming using locks and other synchronization primitives to build a concurrency library.

Concurrent hash tables

Let's extend our hash table example to support concurrency. To begin, let's first give the signature of the module we will implement. The table has operations to add, remove, and find elements, and it also supports imperative iterators.

```
module Concurrent_dictionary : sig
  type ('a, 'b) t

  val create : unit -> ('a, 'b) t
  val add : ('a, 'b) t -> key:'a -> data:'b -> unit
  val find : ('a, 'b) t -> key:'a -> 'b option
  val remove : ('a, 'b) t -> key:'a -> unit
  val iterator : ('a, 'b) t -> ('a * 'b) iterator
end
```

We'll use the same basic construction that we used to implement the `Dictionary`, where a hash table contains an array of buckets. In addition we'll add locking to ensure that concurrent operations do not interfere. To reduce lock contention, we'll use an array of locks to partition the table into multiple parts. If operations are uniformly distributed, this should reduce lock contention.

```
type ('a, 'b) element = {
  key : 'a;
  mutable value : 'b
}
type ('a, 'b) t = {
  locks : Mutex.t array;
  mutable buckets : ('a, 'b) element list array
}

let num_locks = 32
let num_buckets = 256

let create () = {
  locks = Array.init num_locks (fun _ -> Mutex.create ());
  buckets = Array.create num_buckets []
}
```

Each `element` is a key/value pair, where the value is mutable so that the `add` function can mutate it in place. For this implementation, we'll use 32 locks, and start with 256 buckets.

Each bucket is an *association list*, meaning that it is list of key/value pairs that implement a dictionary. We can start the implementation by defining dictionary operations for association lists. The function `find_assoc` finds the value associated with a key, and `remove_assoc_exn` removes an association. The remove function raises an exception `Not_found` if the list does not contain the association. We'll use this to optimize removal.

```
let rec find_assoc key = function
| element :: tl ->
    if element.key = key then
       Some element
    else
       find_assoc key tl
| [] -> None

let rec remove_assoc_exn key = function
| element :: tl ->
    if element.key = key then
       tl
    else
       element :: remove_assoc_exn key tl
| [] -> raise Not_found
```

The locks are intended to partition the table into multiple sub-parts, where each lock provides synchronization for a contiguous range of buckets. For synchronization we define a function `synchronize` that takes a bucket index and a function, and evaluates the function with the bucket lock acquired, releasing the lock before returning.

```
let synchronize table index f =
  let lock = table.locks.(index * num_locks / num_buckets) in
  Mutex.lock lock;
  let result = f () in
  Mutex.unlock lock;
  result
```

Note that the `synchronize` function is *not* exception-safe, meaning that if evaluation of `f ()` raises an exception, the lock will not be released. An exception-safe version would catch all exceptions; when an exception is raised, the lock would be released, and the exception re-raised.

```
let synchronize_exn table index f =
  let lock = table.locks.(index * num_locks / num_buckets) in
  Mutex.lock lock;
  try let result = f () in Mutex.unlock lock; result with
    exn -> Mutex.unlock lock; raise exn
```

To add a new entry to the table, the `add` function acquires the bucket lock, then uses `find_assoc` to look for an existing association. If one is found, the `value` is updated in-place to the new value. Otherwise, a new entry is added to the beginning of the bucket.

```
let add table ~key ~data =
  let hash = Hashtbl.hash key in
  let index = hash mod num_buckets in
  let buckets = table.buckets in
  synchronize table index (fun () ->
        match find_assoc key buckets.(index) with
        | Some element ->
             element.value <- data
        | None ->
             buckets.(index) <- { key = key; value = data } :: buckets.(index))
```

Removing an element from the table is similar. If here is a previous entry in the table, the entry is removed. Otherwise, the `remove_assoc_exn` function raises `Not_found`, and we leave the bucket unchanged. The exception is an optimization to avoid copying the entire list in this case.

```
let remove table ~key =
  let hash = Hashtbl.hash key in
  let index = hash mod num_buckets in
  let buckets = table.buckets in
  synchronize table index (fun () ->
    try buckets.(index) <- remove_assoc_exn key buckets.(index) with
    | Not_found -> ())
```

The function to find an association in the table is similar -- we jsut find the entry in the table and return the value part. However, this particular implementation is somewhat more subtle, because it omits the synchronization step, examining the bucket *without* acquiring the lock.

```
let find table ~key =
  let hash = Hashtbl.hash key in
  let index = hash mod num_buckets in
  (* Unsynchronized *)
  match find_assoc key table.buckets.(index) with
  | Some element -> Some element.value
  | None -> None
```

From a performance perspective, this is clearly a win, because retrieving elements from the table has no locking at all. But why is it valid?

The reasoning has to do with two things: 1) the semantics we expect from the table, and 2) the OCaml memory model. Ideally, the semantics we would expect is *sequential semantics*, meaning that all memory operations are processed in *some* sequence that is compatible with the order in which they were performed in each thread. Thus, if some thread adds two entries for keys `K1` and `K2` in sequential order, then all other threads will see either, 1) neither entry, or 2) a entry for `K1`, or 3) a entry for both `K1` and `K2`, but it will *not* see an entry for `K2` without also having an entry for `K1`.

Unfortunately, for some processor architectures, primary memory does not have sequential semantics, due to caching and other effects. Fortunately for us, OCaml does provide sequential memory semantics due to its threading model where: 1) only one thread executes at a time, and 2) there is a memory barrier the prevents reordering of thread context switches and memort operations, and 3) the compiler does not reorder memory operations in ways that would violate sequential memory semantics.

Note, OCaml does not a *guarantee* this semantics. The OCaml implementation may change to support parallelism. If it does, the memory semantics will change accordingly. The simplest fix is just to synchronize the access. Performance of `find` operations will decrease somewhat due to contention.

```
let synchronized_find table ~key =
  let hash = Hashtbl.hash key in
  let index = hash mod num_buckets in
  synchronize table index (fun () ->
    (find_assoc key table.buckets.(index)).value)
```

For the final part of the implementation, let's define imperative iteration. The iterator object contains a bucket index, and the field `elements` refers to some suffix of the list stored in the bucket. The `value` method returns the current elements, and the `next` method advances the `elements` field. The method `normalize` is used to maintain the invariant that the `elements` field always refers to a value in the table unless the iterator has advanced past the final element. The `remove` method removes the current element from the bucket in which it is stored.

```
let rec remove_element_exn elements = function
| (hd :: tl) as elements' ->
     if elements' == elements then
```

```
        tl
     else
        hd :: remove_element_exn elements tl
| [] -> raise Not_found

let iterator table =
  let buckets = table.buckets in
  object (self)
    val mutable index = 0
    val mutable elements = buckets.(0)
    method has_value = elements <> []
    method value =
      match elements with
      | { key = key; value = value } :: _ -> key, value
      | [] -> raise (Invalid_argument "value")
    method next =
      elements <- List.tl elements;
      self#normalize
    method remove =
      synchronize table index (fun () ->
        try buckets.(index) <- remove_element_exn elements buckets.(index) with
          Not_found -> ());
      self#next
    method private normalize =
      while elements = [] && index < num_buckets do
        index <- index + 1;
    elements <- buckets.(index)
      done
    initializer self#normalize
  end
```

The iterator methods are all unsychronized except the method `remove`, which mutates the bucket. As a consequence, it means that hash operations that add and remove elements from the list can happen concurrently with iteration. Again, this is great from a performance perspective, but it means that iteration has non-sequential semantics. In particular, whenever iteration enters a new bucket, subsequent concurrent operations that add new elements or remove old ones from that bucket have *no effect* on the iteration. Iteration advances through that bucket as if it were unchanged.

One advantage of this relaxed iteration semantics is peformance, since iteration is largely unsynchronized. Another advantage is that deadlock is less likely. If we were to *lock* the bucket during iteration, then changes to that bucket would not be allowed during iteration (even by the iterating thread). We might allow lock recursion to allow mutations by the iterating thread, but in general the synchronization might involve multiple threads, resulting in deadlock. Lock-free iteration ensures that the `Concurrent_dictionary` will not be involved in a deadlock cycle.

Input and output (fix this chapter heading)

Input and output I/O is another kind of imperative operation, where the purpose is to either read input from a file, stream, or device, *consuming* the input by side-effect; or write output to a file, stream, or device, *modifying* the output by side-effect. There are several I/O libraries in OCaml. There is the basic builtin I/O library in the `Pervasives` module, and there are moe advanced I/O libraries in the `Unix` and `Async` modules. Let's look at the basic library -- the advanced libraries will be described elsewhere.

For basic I/O OCaml models input and output with *channels*. An `in_channel` is used for reading input, and and `out_channel` for producing output. Each OCaml process has three standard channels, similar to the three standard files in Unix.

- `stdin : in_channel`. The "standard input" channel. By default, input comes from the terminal, which handles keyboard input.
- `stdout : out_channel`. The "standard output" channel. By default, output written to `stdout` appears on the user terminal.
- `stderr : out_channel`. The "standard error" channel. This is similar to `stdout`, but it is intended for error messages.

The standard library has several functions for reading from and writing to the standard channels.

- `print_char : char -> unit` prints a single character to `stdout`.
- `print_string : string -> unit` prints a `string` to `stdout` as a sequence of characters, without quotes.
- `print_int : int -> unit` prints an integer to `stdout`.
- `print_float : float -> unit` prints a floating-point number to `stdout`.

Functions to write to `stderr` have similar names, using the prefix `prerr_` rather than `print_` (for example, `prerr_string`).

```
# let i = 1;;
val i : int = 1
# print_string "The value of i is "; print_int i; print_string ".\n";;
The value of i is 1.
- : unit = ()
```

Input is similar, but there are only a few functions.

- `read_line : unit -> string` reads a line of input from `stdin`.
- `read_int : unit -> int` reads a decimal integer from `stdin`.
- `read_float : unit -> float` reads a floating-point number from `stdin`.

These functions raise the exception `End_of_file` if the input channel is terminated before the input is read.

Here is a function to read a sequence of integers from `stdin`, sorting them and printing the result to `stdout`.

```
# let collate_input () =
   let rec read_input items =
      let item = try Some (read_int ()) with End_of_file -> None in
      match item with
       | Some i -> read_input (i :: items)
       | None -> items
   in
   let items = read_input [] in
   let sorted = List.sort (-) items in
   List.iter (fun i -> print_int i; print_char ' ') sorted;
   print_string "\n";;
val collate_input : unit -> unit = <fun>
#   collate_input ();;
8
56
2
34
-120
19
-120 2 8 19 34 56
- : unit = ()
```

Using exceptions to signal the end of input is a little awkward. The `read_input` loop above converts the exception into an `int option`, so that matching can be performed with `match` and the function is tail-recursive. The following function is *not* tail recursive, and should be avoided.

```
  (* AVOID -- non tail-recursive input reader *)
# let rec read_input items =
   try read_input (read_int () :: items) with
      End_of_file -> items;;
val read_input : int list -> int list = <fun>
# read_input [];;
34
45
56
-1
- : int list = [-1; 56; 45; 34]
```

Another way to address the input issue is to use iteration, rather than recursion. This requires collecting the input in a container than can be mutated by side-effect.

```
# let read_input () =
   let items = ref [] in
   try
      while true do
         items := read_int () :: !items
      done;
      []  (* not reached *)
   with End_of_file ->
      !items;;
val read_input : unit -> int list = <fun>
# read_input ();;
45
```

```
78
-345
- : int list = [-345; 78; 45]
```

In this loop, the value returns from a successful call to `read_int ()` is added to the `items` list by side-effect. When `read_int ()` reaches the end of input, it raises the `End_of_file` exception, terminating the loop. The `while` loop never terminates; the `[]` value afterward is only to satisfy the type checker.

Basic file input and output

It isn't necessary to perform all input through the standard channels. There are also functions to open files for reading and writing.

- `open_out : string -> out_channel` open a file for writing.
- `open_in : string -> in_channel` open a file for reading.
- `close_out : out_channel -> unit` close an output channel, flushing any pending output to the file.
- `close_in : in_channel -> unit` close an input channel.

The functions for input and output are similar to the `print_...` and `read_...` functions, but they use the prefix `output_` and `input_` and they take a channel as an argument. Let's write a function to print the contents of a file.

```
# let cat filename =
   let inx = open_in filename in
   try
      while true do
         print_string (input_line inx); print_char '\n'
      done
   with End_of_file ->
      close_in inx;;
val cat : string -> unit = <fun>
# let outf = open_out "file.txt";;
val outf : out_channel = <abstr>
# output_string outf "Hello world\n1\n2\n3\n";;
- : unit = ()
# close_out outf;;
- : unit = ()
# cat "file.txt";;
Hello world
1
2
3
- : unit = ()
```

It is important to close channels when you are finished with them, for two reasons. One reason is that the runtime system will usually have a limit on the number of channels that may be open simultaneously. To avoid problems, you should close channels you are not using. In addition, for efficiency, output is buffered and written to output channels in larger blocks. The output may not actually be written to the file unless the `out_channel` is closed, or with an explicit call to the `flush` function.

- `flush : out_channel -> unit`

```
# let outf = open_out "file.txt";;
val outf : out_channel = <abstr>
# output_string outf "Hello world\n1\n2\n3\n";;
- : unit = ()
# cat "file.txt";;
- : unit = ()
# flush outf;;
- : unit = ()
# cat "file.txt";;
Hello world
1
2
3
- : unit = ()
```

Formatted output with Printf

Output with the standard library functions like `output_char`, `output_int`, `output_string`, etc. is simple, but there is little flexibility, and it is often verbose. OCaml also supports *formatted* output using the `printf` function, which is modeled after `printf` in the C standard library. The `printf` function takes a *format string* that describe what to print and how to format it, as well as arguments to be printed.

```
printf format arg1 ... argN
```

The format string can contain character literals, which are printed without change, as well as conversion specifications, which specify how to print an value that is provided as an argument. A conversion specification begins with a percent (`%`) character, some flags, and a type specification. For example, `%d` is a conversion specification for printing an integer, `%s` prints a string, and `%f` prints a floating-point value.

```
# open Printf;;
# printf "int=%d string=%s float=%f\n" 10 "abc" 1e5;;
int=10 string=abc float=100000.000000
- : unit = ()
```

Conversions can also specify additional parameters for formatting the value. The general form has flags, width specified in number of characters, and decimal precision used for printing floating-point values. These parameters are optional.

```
% [flags] [width] [.precision] type
```

There are several flags of interest, the flag `'-'` left-justifies the output, and the flag `'0'` pads numerical values with leading zeroes. Let's write a program to print a price list in a tabular form.

```
# let print_prices prices =
    print_string "--------------------------------\n\
                  | Size   | Hexcode    | Price  |\n\
                  --------------------------------\n";
    List.iter (fun (size, code, price) ->
          printf "| %-6s | 0x%08x | %6.2f |\n" size code price) prices;
    print_string "--------------------------------\n";;
val print_prices : (string * int * float) list -> unit = <fun>
# print_prices
   [("small", 0x35, 1.02);
    ("medium", 0x7a1, 50.75);
    ("large", 0xbad, 400.8);
    ("vente", 0x11, 4136.);
    ("enormous", 100000, 100.)];;
--------------------------------
| Size   | Hexcode    | Price  |
--------------------------------
| small  | 0x00000035 |   1.02 |
| medium | 0x000007a1 |  50.75 |
| large  | 0x00000bad | 400.80 |
| vente  | 0x00000011 | 4136.00 |
| enormous | 0x000186a0 | 100.00 |
--------------------------------
- : unit = ()
```

The format specification `"| %-6s | 0x%08x | %6.2f |\n"` specifies that the first argument is a string that is to be left-justified and printed six characters wide; the second argument is an integer that should be printed in hexidecimal (format type `x`) eight characters wide with leading zeroes; and the third argument is a floating point value that should be printed right-justified, six characters wide, with two digits after the decimal point.

Note that the width specifies a *minimum* width. If the value requires more width to print completely the width is increased. The price `4136.00` and the size `enormous` both overflow the width, breaking the tabular form.

Strong typing and format strings

In OCaml, `printf` is type safe. The format is checked against the arguments at compile time, and rejected if the format string is malformed, or if the format does not match the arguments. In the following examples, `%v` is not a valid conversion specification, and floating-point values can't be printed with a decimal conversion specification.

```
# printf "Hello %v" 1;;
Characters 7-17:
  printf "Hello %v" 1;;
```

```
          ^^^^^^^^^^
Error: Bad conversion %v, at char number 6 in format string ``Hello %v''
# printf "Hello %d" 1.;;
Characters 18-20:
  printf "Hello %d" 1.;;
                    ^^
Error: This expression has type float but an expression was expected of type
         int
```

A consequence of strong typing is that the format string must ultimately be a string *literal*, known at compile time. It can't be computed by the program at runtime.

```
# let fmt = "%s";;
val fmt : string = "%s"
# printf fmt "Hello";;
Characters 7-10:
  printf fmt "Hello";;
         ^^^
Error: This expression has type string but an expression was expected of type
         ('a -> 'b, out_channel, unit) format =
           ('a -> 'b, out_channel, unit, unit, unit, unit) format6
```

Actually, this is not entirely true. The `printf` function takes a format string of type `('a, 'b, 'c) format` and it is only the type conversion from `string` to `format` where the string is required to be a literal.

```
# let fmt : ('a, 'b, 'c) format = "Size: %s.\n";;
val fmt : (string -> 'c, 'b, 'c) format = <abstr>
# printf fmt "small";;
Size: small
- : unit = ()
```

Output to channels, strings, and buffers

The function `printf` prints to the standard output channel `stdout`. There are alse several variations.

- `eprintf format arg1 ... argN`. Print to the standard error channel `stderr`.
- `fprintf channel format arg1 ... argN`. Print to the channel `channel`.
- `sprintf format arg1 ... argN`. Produce a string as output.
- `bprintf buffer format arg1 ... argN`. Print to a string buffer `buffer` of type `Buffer.t`.

```
# sprintf "The number is %10d." 123;;
- : string = "The number is        123."
# let buf = Buffer.create 32;;
val buf : Buffer.t = <abstr>
# bprintf buf "%s\n" "First line";;
- : unit = ()
# bprintf buf "%20s.\n" "Second line";;
```

```
- : unit = ()
# Buffer.contents buf;;
- : string = "First line\n         Second line.\n"
```

Formatted input with Scanf

The `scanf` function can be used for reading input, similar to `printf`. The general form takes a format string and a "receiver" function that is applied to the input. The format string contains literal characters, which must be read literally, and conversion specifications similar to `printf`. The result of the receiver function is returned as the result of `scanf`.

```
# scanf "%d" (fun i -> i);;
1871
- : int = 1871
```

For another example, consider the parsing a date string that might be in one of two forms, *month day, year* or *month/day/year*. The `scanf` functions raise the exception `Scan_failure` on error, so we can read a date string by trying both kinds of format.

```
# type month = String of string | Int of int;;
type month = String of string | Int of int
# let scan_date_string s =
    try
      sscanf s "%s %d, %d" (fun month day year ->
             year, String month, day)
    with Scan_failure _ | End_of_file ->
      sscanf s "%d/%d/%d" (fun month day year ->
             year, Int month, day);;
val scan_date_string : string -> int * month * int = <fun>
# scan_date_string "May 3, 1921";;
- : int * month * int = (1921, String "May", 3)
# scan_date_string "10/11/2001";;
- : int * month * int = (2001, Int 10, 11)
# scan_date_string "May 3";;
Exception:
Scanf.Scan_failure
 "scanf: bad input at char number 0: ``character 'M' is not a decimal digit''".
```

Summary

The OCaml language supports a fairly standard imperative programming model, with looping, assignment, mutable arrays, records, and objects. If desired, we can write programs that correspond directly to what we would have written in some other imperative language like C or Java. Of course, doing so is really not the best match -- if you want to write imperative programs, you should probably use an imperative programming language.

2013-02-05
19:46:47

However, there are times when imperative programming might provide efficiency (as with lazy evaluation, or memoization), or you might require techniques or data structures that are traditional imperative (like graphs represented with adjacency lists), and in these cases OCaml usually shines. Used with discretion, imperative programming can lead to smaller, simpler programs.

2013-02-05
19:46:47

CHAPTER 11

Object Oriented Programming

We've already seen several tools that OCaml provides for organizing programs, particularly first-class modules. In addition, OCaml also supports object-oriented programming. There are objects, classes, and their associated types. Objects are good for encapsulation and abstraction, and classes are good for code re-use.

What is Object-Oriented Programming?

Object-oriented programming (often shorted to OOP) is a programming style that encapsulates computation and data within logical *objects*. Each object contains some data stored in *fields*, and has *method* functions that can be invoked against the data within the object. The code definition behind an object is called a *class*, and objects are constructed from a class definition by calling a constructor with the data that the object will use to build itself.

There are four fundamental properties that differentiate OOP from other styles:

- *Abstraction*: the details of the implementation are hidden in the object, and the external interface is just the set of publically-accessible methods.
- *Dynamic lookup*: when a message is sent to an object, the method to be executed is determined by the implementation of the object, not by some static property of the program. In other words, different objects may react to the same message in different ways.
- *Subtyping*: if an object `a` has all the functionality of an object `b`, then we may use `a` in any context where `b` is expected.
- *Inheritance*: the definition of one kind of object can be re-used to produce a new kind of object. This new definition can override some behaviour, but also share code with its parent.

Almost every notable modern programming language has been influenced by OOP, and you'll have run across these terms if you've ever used C++, Java, C#, Ruby, Python or Javascript.

When to use objects

You might wonder when to use objects in OCaml, which has a multitude of alternative mechanisms to express the same concept. First-class modules are more expressive (a module can include types, while classes and objects cannot). Modules, functors, and algebraic data types also offer a wide range of ways to express program structure. In fact, many seasoned OCaml programmers rarely use classes and objects, if at all.

Modules already provide these features in some form, but the main focus of classes is on code re-use through inheritance and late binding of methods. This is a critical property of classes: the methods that implement an object are determined when the object is instantiated, a form of *dynamic* binding. In the meantime, while classes are being defined, it is possible (and necessary) to refer to methods without knowing statically how they will be implemented.

In contrast, modules use static (lexical) scoping. If you want to parameterize your module code so that some part of it can be implemented later, you would write a function or functor. This is more explicit, but often more verbose than overriding a method in a class.

In general, a rule of thumb is: use classes and objects in situations where dynamic binding is a big win, for example if you have many similar variations in the implementation of a concept. Two good examples are Xavier Leroy's Cryptokit (*http://gallium.inria.fr/~xleroy/software.html#cryptokit*), which provides a variety of cryptographic primitives that can be combined in building-block style, and the Camlimages (*http://cristal.inria.fr/camlimages/*) library which manipulates various graphical file formats.

In this chapter, we'll introduce you to the basics of object definition and use in OCaml, and then demonstrate their use with an example using Cryptokit. We'll return to the more advanced areas of object use later on in the book in Chapter 19.

OCaml objects

If you already know about object oriented programming in a language like Java or C++, the OCaml object system may come as a surprise. Foremost is the complete separation of subtyping and inheritance in OCaml. In a language like Java, a class name is also used as the type of objects created by instantiating it, and the subtyping rule corresponds to inheritance. For example. if we implement a class `Stack` in Java by inheriting from a class `Deque`, we would be allowed to pass a stack anywhere a deque is expected (this is a silly example of course, practitioners will point out that we shouldn't do it).

OCaml is entirely different. Classes are used to construct objects and support inheritance, including non-subtyping inheritance. Classes are not types. Instead, objects have *object types*, and if you want to use objects, you aren't required to use classes at all. Here's an example of a simple object.

```
# let p =
  object
    val mutable x = 0
    method get = x
    method set i = x <- i
  end;;
val p : < get : int; set : int -> unit > = <obj>
```

The object has an integer value `x`, a method `get` that returns x, and a method `set` that updates the value of `x`.

The object type is enclosed in angle brackets `< ... >`, containing just the types of the methods. Fields, like `x`, are not part of the public interface of an object. All interaction with an object is through its methods. The syntax for a method invocation (also called "sending a message" to the object) uses the `#` character.

```
# p#get;
- : int = 0
# p#set 17;;
- : unit = ()
# p#get;;
- : int = 17
```

Objects can also be constructed by functions. If we want to specify the initial value of the object, we can define a function that takes the value and returns an object.

```
# let make i =
  object
    val mutable x = i
    method get = x
    method set y = x <- y
  end;;
val make : 'a -> < get : 'a; set : 'a -> unit > = <fun>
# let p = make 5;;
val p : < get : int; set : int -> unit > = <obj>
# p#get;;
- : int = 5
```

Note that the types of the function `make` and the returned object now use the polymorphic type `'a`. When make is invoked on a concrete value `5`, we get the same object type as before, with type `int` for the value.

Object Polymorphism

Functions can also take object arguments. Let's construct a new object `average` that returns the average of any two objects with a `get` method.

```
# let average p1 p2 =
  object
    method get = (p1#get + p2#get) / 2
  end;;
val average :
  < get : int; .. > ->
  < get : int; .. > ->
  < get : int > = <fun>
```

There's some new syntax in the type that's been inferred for `average` here. The parameters have the object type `< get : int; .. >`. The `..` are ellipsis, standing for any other methods. The type `< get : int; .. >` specifies an object that must have at least a `get` method, and possibly some others as well.

We can use the `average` using the normal object invocation syntax:

```
# let p1 = make 5;;
# let p2 = make 15;;
# let a = average p1 p2;;
# a#get;;
- : int = 10
# p2#set 25;;
# a#get;;
- : int = 15
```

The potential extra parameters defined by the object are carefully tracked by the OCaml type checker. If we manually try and constrain the exact type `< get : int >` for an object with more methods, type inference will fail.

```
# let (p : < get : int >) = make 5;;
Error: This expression has type < get : int; set : int -> unit >
       but an expression was expected of type < get : int >
       The second object type has no method set
```

Elisions are polymorphic

The `..` in an object type is an elision, standing for "possibly more methods." It may not be apparent from the syntax, but an elided object type is actually polymorphic. If we try to write a type definition, we get an obscure error.

```
# type point = < get:int; .. >;;
```

```
Error: A type variable is unbound in this type declaration.
In type < get : int; .. > as 'a the variable 'a is unbound
```

A `..` in an object type is called a *row variable* and this typing scheme is called *row polymorphism*. Even though `..` doesn't look like a type variable, it actually is. The error message suggests a solution, which is to add the `as 'a` type constraint.

```
# type 'a point = < get:int; .. > as 'a;;
type 'a point = 'a constraint 'a = < get : int; .. >
```

In other words, the type `'a point` is equal to `'a`, where `'a = < get : int; .. >`. That may seem like an odd way to say it, and in fact, this type definition is not really an abbreviation because `'a` refers to the entire type.

An object of type `< get:int; .. >` can be any object with a method `get:int`, it doesn't matter how it is implemented. So far, we've constructed two objects with that type; the function `make` constructed one, and so did `average`. When the method `#get` is invoked, the actual method that is run is determined by the object.

```
# let print_point p = Printf.printf "Point: %d\n" p#get;;
val print_point : < get : int; .. > -> unit = <fun>
# print_point (make 5);;
Point: 5
# print_point (average (make 5) (make 15));;
Point: 10
```

Classes

Programming with objects directly is great for encapsulation, but one of the main goals of object-oriented programming is code re-use through inheritance. For inheritance, we need to introduce *classes*. In object-oriented programming, a class is a "recipe" for creating objects. The recipe can be changed by adding new methods and fields, or it can be changed by modifying existing methods.

In OCaml, class definitions must be defined as top-level statements in a module. A class is not an object, and a class definition is not an expression. The syntax for a class definition uses the keyword `class`.

```
# class point =
  object
    val mutable x = 0
    method get = x
    method set y = x <- y
  end;;
class point :
  object
    val mutable x : int
    method get : int
    method set : int -> unit
  end
```

The type `class point : ... end` is a *class type*. This particular type specifies that the `point` class defines a mutable field `x`, a method `get` that returns an `int`, and a method `set` with type `int -> unit`.

To produce an object, classes are instantiated with the keyword `new`.

```
# let p = new point;;
val p : point = <obj>
# p#get;;
- : int = 0
# p#set 5;;
- : unit = ()
# p#get;;
- : int = 5
```

Inheritance uses an existing class to define a new one. For example, the following class definition supports an addition method `moveby` that moves the point by a relative amount.

```
# class movable_point =
  object (self : 'self)
    inherit point
    method moveby dx = self#set (self#get + dx)
  end;;
class movable_point :
  object
    val mutable x : int
    method get : int
    method moveby : int -> unit
    method set : int -> unit
  end
```

This new `movable_point` class also makes use of the `(self : 'self)` binding after the `object` keyword. The variable `self` stands for the current object, allowing self-invocation, and the type variable `'self` stands for the type of the current object (which in general is a subtype of `movable_point`).

An Example: Cryptokit

Let's take a break from describing the object system with a more practical example that uses the OCaml cryptographic library.

Installing the Cryptokit library

The Cryptokit library can be installed via OPAM via `opam install cryptokit`. Once that's finished compiling and installing, you just need to `#require "cryptokit"` in your toplevel to load the library and make the modules available.

Let's build a command to mimic the `md5` command, which reads in an input file and returns a hexadecimal representation of its MD5 cryptographic hash. Cryptokit defines a number of different functions and collects them together under the `Cryptokit.hash` class type:

```
class type hash = object
  method add_byte : int -> unit
  method add_char : char -> unit
  method add_string : string -> unit
  method add_substring : string -> int -> int -> unit
  method hash_size : int
  method result : string
  method wipe : unit
end

val hash_string : hash -> string -> string
```

Hash objects are clearly imperative. Once instantiated, data is fed into them by the addition functions, and then the `result` is computed and then the contents erased via `wipe`. The `hash_string` convenience function applies the hash function fully to a string, and returns the result. The `md5` command is quite straight-forward now:

```
open Core.Std
open Cryptokit

let _ =
  In_channel.(input_all stdin) |!
  hash_string (Hash.md5 ()) |!
  transform_string (Hexa.encode ()) |!
  print_endline
```

After opening the right modules, we read in the entire standard input into an OCaml string. This is then passed onto the MD5 hash function, which returns a binary string. This binary is passed through the `Hexa` hexadecimal encoder, which returns an ASCII representation of the input. The output of this command will be the same as the `md5` command (or `md5sum` in some systems).

We can extend this simple example by selecting either the `md5` or `sha1` hash function at runtime depending on the name of our binary. `Sys.argv` is an array containing the arguments the command was invoked with, and the first entry is the name of the binary itself.

```
open Core.Std
open Cryptokit

let _ =
  let hash_fn =
    match Filename.basename Sys.argv.(0) with
    |"md5" -> Hash.md5 ()
    |"sha1" -> Hash.sha1 ()
    |_ -> Hash.md5 ()
  in
  In_channel.(input_all stdin) |!
  hash_string hash_fn |!
  transform_string (Hexa.encode ()) |!
  print_endline
```

Now let's try something more advanced. The `openssl` library is installed on most systems, and can be used to encrypt plaintext using several encryption strategies. At its simplest, it will take a secret phrase and derive an appropriate key and initialisation vector.

```
$ openssl enc -nosalt -aes-128-cbc -base64 -k "ocaml" -P
key=6217C07FF169F6AB2EB2731F855095F1
iv =8164D5477E66E6A9EC99A8D58ACAADAF
```

We've selected the `-nosalt` option here to make the output deterministic, and the `-P` option prints out the derived key and IV and exits. The algorithm used to derive these results is described in the `man EVP_BytesToKey` manual page (you may need to install the OpenSSL documentation packages on your system first). We can implement this derivation function using an imperative style:

```
let md5 s = hash_string (Hash.md5 ()) s

let evp_byte_to_key password tlen =
  let o = Hexa.encode () in
  let v = ref (md5 password) in
  o#put_string !v;
  while o#available_output/2 < tlen do
    let n = md5 (!v ^ password) in
    o#put_string n;
    v := n;
  done;
  String.uppercase o#get_string

let _ =
  let secret = "ocaml" in
  let key_len = 16 * 2 in
  let iv_len = 16 * 2 in
  let x = evp_byte_to_key secret (key_len+iv_len) in
  let key = String.sub x ~pos:0 ~len:key_len in
  let iv = String.sub x ~pos:key_len ~len:iv_len in
  Printf.printf "key=%s\niv =%s\n%!" key iv
```

The derivation algorithm takes an input password and desired total length (the addition of the key and IV length). It initialises a `Hexa.encode` transformer, which will accept arbitrary binary data and output a hexadecimal string (with two output bytes per input byte). A reference stores the last digest that's been calculated, and then the algorithm iterates until it has sufficient data to satisfy the required key length.

Notice how the encoder object is used as an accumulator, by using the `put_string` and `available_output` to keep track of progress. Objects don't *require* an imperative style though, and the same algorithm can be written more functionally:

```
let evp_byte_to_key password tlen =
  let rec aux acc v =
    match String.length acc < tlen with
    |true ->
      let v = md5 (v ^ password) in
      aux (acc^v) v
    |false -> acc
  in
  let v = md5 password in
  String.uppercase (transform_string (Hexa.encode ()) (aux v v))
```

In this version, we don't use any references, and instead a recursive function keeps track of the last digest in use and the accumulated result string. This version isn't quite as efficient as the previous one due to the careless use of string concatenation for the accumulator, but this can easily be fixed by using the `Buffer` module instead.

Class parameters and polymorphism

A class definition serves as the *constructor* for the class. In general, a class definition may have parameters that must be provided as arguments when the object is created with `new`.

Let's build an example of an imperative singly-linked list using object-oriented techniques. First, we'll want to define a class for a single element of the list. We'll call it a `node`, and it will hold a value of type `'a`. When defining the class, the type parameters are placed in square brackets before the class name in the class definition. We also need a parameter `x` for the initial value.

```
class ['a] node x =
object
  val mutable value : 'a = x
  val mutable next_node : 'a node option = None

  method get = value
  method set x = value <- x

  method next = next_node
  method set_next node = next_node <- node
end;;
```

2013-02-05
19:46:47

The `value` is the value stored in the node, and it can be retrieved and changed with the `get` and `set` methods. The `next_node` field is the link to the next element in the stack. Note that the type parameter `['a]` in the definition uses square brackets, but other uses of the type can omit them (or use parentheses if there is more than one type parameter).

The type annotations on the `val` declarations are used to constrain type inference. If we omit these annotations, the type inferred for the class will be "too polymorphic," `x` could have some type `'b` and `next_node` some type `'c option`.

```
  class ['a] node x =
  object
    val mutable value = x
    val mutable next_node = None

    method get = value
    method set x = value <- x

    method next = next_node
    method set_next node = next_node <- node
  end;;
Error: Some type variables are unbound in this type:
         class ['a] node :
           'b ->
           object
             val mutable next_node : 'c option
             val mutable value : 'b
             method get : 'b
             method next : 'c option
             method set : 'b -> unit
             method set_next : 'c option -> unit
           end
       The method get has type 'b where 'b is unbound
```

In general, we need to provide enough constraints so that the compiler will infer the correct type. We can add type constraints to the parameters, to the fields, and to the methods. It is a matter of preference how many constraints to add. You can add type constraints in all three places, but the extra text may not help clarity. A convenient middle ground is to annotate the fields and/or class parameters, and add constraints to methods only if necessary.

Next, we can define the list itself. We'll keep a field `head` the refers to the first element in the list, and `last` refers to the final element in the list. The method `insert` adds an element to the end of the list.

```
class ['a] slist =
object
   val mutable first : ('a) node option = None
   val mutable last : ('a) node option = None

   method is_empty = first = None
```

```
    method insert x =
       let new_node = Some (new node x) in
       match last with
          Some last_node ->
             last_node#set_next new_node;
             last <- new_node
        | None ->
             first <- new_node;
             last <- new_node
end;;
```

Object types

This definition of the class `slist` is not complete, we can construct lists, but we also need to add the ability to traverse the elements in the list. One common style for doing this is to define a class for an `iterator` object. An iterator provides a generic mechanism to inspect and traverse the elements of a collection. This pattern isn't restricted to lists, it can be used for many different kinds of collections.

There are two common styles for defining abstract interfaces like this. In Java, an iterator would normally be specified with an interface, which specifies a set of method types. In languages without interfaces, like C++, the specification would normally use *abstract* classes to specify the methods without implementing them (C++ uses the "= 0" definition to mean "not implemented").

```
// Java-style iterator, specified as an interface.
interface <T> iterator {
  T Get();
  boolean HasValue();
  void Next();
};

// Abstract class definition in C++.
template<typename T>
class Iterator {
 public:
  virtual ~Iterator() {}
  virtual T get() const = 0;
  virtual bool has_value() const = 0;
  virtual void next() = 0;
};
```

OCaml support both styles. In fact, OCaml is more flexible than these approaches because an object type can be implemented by any object with the appropriate methods, it does not have to be specified by the object's class *a priori*. We'll leave abstract classes for later. Let's demonstrate the technique using object types.

First, we'll define an object type `iterator` that specifies the methods in an iterator.

```
type 'a iterator = < get : 'a; has_value : bool; next : unit >;;`
```

Next, we'll define an actual iterator for the class `slist`. We can represent the position in the list with a field `current`, following links as we traverse the list.

```
class ['a] slist_iterator cur =
object
  val mutable current : 'a node option = cur

  method has_value = current <> None

  method get =
     match current with
        Some node -> node#get
      | None -> raise (Invalid_argument "no value")

  method next =
     match current with
        Some node -> current <- node#next
      | None -> raise (Invalid_argument "no value")
end;;
```

Finally, we add a method `iterator` to the slist class to produce an iterator. To do so, we construct an `slist_iterator` that refers to the first node in the list, but we want to return a value with the object type `iterator`. This requires an explicit coercion using the `:>` operator.

```
class ['a] slist = object
...
   method iterator = (new slist_iterator first :> 'a iterator)
end

# let l = new slist;;
# l.insert 5;;
# l.insert 4;;
# let it = l#iterator;;
# it#get;;
- : int = 5
# it#next;;
- : unit = ()
# it#get;;
- : int = 4
# it#next;;
- : unit = ()
# it#has_value;;
- : bool = false
```

We may also wish to define functional-style methods, `iter f` takes a function `f` and applies it to each of the elements of the list.

```
method iter f =
  let it = self#iterator in
  while it#has_value do
    f it#get
```

```
      it#next
    end
```

What about functional operations similar to `List.map` or `List.fold`? In general, these methods take a function that produces a value of some other type than the elements of the set. For example, the function `List.fold` has type `'a list -> ('b -> 'a -> 'b) -> 'b -> 'b`, where `'b` is an arbitrary type. To replicate this in the `slist` class, we need a method type `('b -> 'a -> 'b) -> 'b -> 'b`, where the method type is polymorphic over `'b`.

The solution is to use a type quantifier, as shown in the following example. The method type must be specified directly after the method name, which means that method parameters must be expressed using a `fun` or `function` expression.

```
method fold : 'b. ('b -> 'a -> 'b) -> 'b -> 'b =
   (fun f x ->
         let y = ref x in
         let it = self#iterator in
         while it#has_value do
            y := f !y it#get;
            it#next
         done;
         !y)
```

Immutable objects

Many people consider object-oriented programming to be intrinsically imperative, where an object is like a state machine. Sending a message to an object causes it to change state, possibily sending messages to other objects.

Indeed, in many programs, this makes sense, but it is by no means required. Let's define an object-oriented version of lists similar to the imperative list above. We'll implement it with a regular list type `'a list`, and insertion will be to the beginning of the list instead of to the end.

```
class ['a] flist =
object (self : 'self)
   val elements : 'a list = []

   method is_empty = elements = []

   method insert x : 'self = {< elements = x :: elements >}

   method iterator =
      (new flist_iterator elements :> 'a iterator)

   method iter (f : 'a -> unit) = List.iter f elements

   method fold : 'b. ('b -> 'a -> 'b) -> 'b -> 'b =
      (fun f x -> List.fold_left f x elements)
end;;
```

A key part of the implementation is the definition of the method `insert`. The expression `{< ... >}` produces a copy of the current object, with the same type, and the specified fields updated. In other words, the `new_fst new_x` method produces a copy of the object, with `x` replaced by `new_x`. The original object is not modified, and the value of `y` is also unaffected.

There are some restriction on the use of the expression `{< ... >}`. It can be used only within a method body, and only the values of fields may be updated. Method implementations are fixed at the time the object is created, they cannot be changed dynamically.

We use the same object type `iterator` for iterators, but implement it differently.

```
class ['a] flist_iterator l =
object
   val mutable elements : 'a list = l

   method has_value = l <> []

   method get =
      match l with
         h :: _ -> h
       | [] -> raise (Invalid_argument "list is empty")

   method next =
      match l with
         _ :: l -> elements <- l
       | [] -> raise (Invalid_argument "list is empty")
end;;
```

Class types

Once we have defined the list implementation, the next step is to wrap it in a module or `.ml` file and give it a type so that it can be used in the rest of our code. What is the type?

Before we begin, let's wrap up the implementation in an explicit module (we'll use explicit modules for illustration, but the process is similar when we want to define a `.mli` file). In keeping with the usual style for modules, we define a type `'a t` to represent the type of list values.

```
module SList = struct
   type 'a iterator = < get : 'a; has_value : bool; next : unit >
   type 'a t = < is_empty : bool; insert : 'a -> unit; iterator : 'a iterator >

   class ['a] node x = object ... end
   class ['a] slist_iterator cur = object ... end
   class ['a] slist = object ... end
   let make () = new slist
end;;
```

We have multiple choices in defining the module type, depending on how much of the implementation we want to expose. At one extreme, a maximally-abstract signature would completely hide the class definitions.

```
module AbstractSList : sig
   type 'a iterator = < get : 'a; has_value : bool; next : unit >
   type 'a t = < is_empty : bool; insert : 'a -> unit; iterator : 'a iterator >

   val make : unit -> 'a t
end = SList
```

The abstract signature is simple because we ignore the classes. But what if we want to include them in the signature, so that other modules can inherit from the class definitions? For this, we need to specify types for the classes, called *class types*. Class types do not appear in mainstream object-oriented programming languages, so you may not be familiar with them, but the concept is pretty simple. A class type specifies the type of each of the visible parts of the class, including both fields and methods. Just like for module types, you don't have to give a type for everything; anything you omit will be hidden.

```
module VisibleSList : sig
  type 'a iterator = < get : 'a; has_value : bool; next : unit >
  type 'a t = < is_empty : bool; insert : 'a -> unit; iterator : 'a iterator >

  class ['a] node : 'a ->
  object
     method get : 'a
     method set : 'a -> unit
     method next : 'a node option
     method set_next : 'a node option -> unit
  end

  class ['a] slist_iterator : 'a node option ->
  object
     method has_value : bool
     method get : 'a
     method next : unit
  end

  class ['a] slist :
  object
    val mutable first : 'a node option
    val mutable last : 'a node option
    method is_empty : bool
    method insert : 'a -> unit
    method iterator : 'a iterator
  end
```

```
    val make : unit -> 'a slist
  end = SList
```

In this signature, we've chosen to make nearly everything visible. The class type for `slist` specifies the types of the fields `first` and `last`, as well ad the types of each of the methods. We've also included a class type for `slist_iterator`, which is of somewhat more questionable value, since the type doesn't appear in the type for `slist` at all.

One more thing, in this example the function `make` has type `unit -> 'a slist`. But wait, we've stressed *classes are not types*, so what's up with that? In fact, what we've said is entirely true, classes and class names *are not* types. However, class names can be used to stand for types. When the compiler sees a class name in type position, it automatically constructs an object type from it by erasing all the fields and keeping only the method types. In this case, the type expression `'a slist` is exactly equivalent to `'a t`.

PART II

Practical Examples

2013-02-05
19:46:47

2013-02-05
19:46:47

CHAPTER 12

Data Serialization with S-Expressions

Data serialization, *i.e.* reading and writing program data to a sequence of bytes, is an important and common programming task. Sometimes you need to match someone else's data format (such as XML), and other times you just want to quickly dump some values to disk and read them back later. To this end, OCaml comes with several techniques for data serialization depending on what your problem is.

We'll start by introducing some features in Core that make it really easy to manipulate s-expressions and safe binary serialisers directly from OCaml types. After this section, we'll move onto interoperating with other third-party formats in Chapter 13 and Chapter 14.

S-expressions are nested paranthetical strings whose atomic values are strings. They were first popularized by the Lisp programming language in the 1960s, and have remained a simple way to encode data structures since then.

```
module Sexp : sig
  type t = Atom of string | List of t list
end
```

An s-expression is in essence a nested parenthetical list whose atomic values are strings. The `Sexp` module comes with functionality for parsing and printing s-expressions.

```
# let sexp =
    let a x = Sexp.Atom x and l x = Sexp.List x in
    l [a "this";l [a "is"; a "an"]; l [a "s"; a "expression"]];;
val sexp : Sexp.t = (this (is an) (s expression))
```

In addition, most of the base types in Core support conversion to and from s-expressions. For example, we can write:

```
# Int.sexp_of_t 3;;
- : Sexp.t = 3
# List.sexp_of_t;;
- : ('a -> Sexp.t) -> 'a List.t -> Sexp.t = <fun>
```

```
# List.sexp_of_t Int.sexp_of_t [1;2;3];;
- : Sexp.t = (1 2 3)
```

Notice that `List.sexp_of_t` is polymorphic, and takes as its first argument another conversion function to handle the elements of the list to be converted. Core uses this scheme more generally for defining sexp-converters for polymorphic types.

But what if you want a function to convert some brand new type to an s-expression? You can of course write it yourself:

```
# type t = { foo: int; bar: float };;
# let sexp_of_t t =
    let a x = Sexp.Atom x and l x = Sexp.List x in
    l [ l [a "foo"; Int.sexp_of_t t.foo  ];
        l [a "bar"; Float.sexp_of_t t.bar]; ]
  ;;
val sexp_of_t : t -> Core.Std.Sexp.t = <fun>
# sexp_of_t { foo = 3; bar = -5.5 };;
- : Core.Std.Sexp.t = ((foo 3) (bar -5.5))
```

This is somewhat tiresome to write, and it gets more so when you consider the parser, *i.e.*, `t_of_sexp`, which is considerably more complex. Writing this kind of parsing and printing code by hand is mechanical and error prone, not to mention a drag.

Given how mechanical the code is, you could imagine writing a program that inspected the type definition and auto-generated the conversion code for you. That is precisely where syntax extensions come in. Using Sexplib and adding `with sexp` as an annotation to our type definition, we get the functions we want for free.

```
# type t = { foo: int; bar: float } with sexp;;
type t = { foo : int; bar : float; }
val t_of_sexp__ : Sexplib.Sexp.t -> t = <fun>
val t_of_sexp : Sexplib.Sexp.t -> t = <fun>
val sexp_of_t : t -> Sexplib.Sexp.t = <fun>
# t_of_sexp (Sexp.of_string "((bar 35) (foo 3))");;
- : t = {foo = 3; bar = 35.}
```

(You can ignore `t_of_sexp__`, which is a helper function that is needed in very rare cases.)

The syntax-extensions in Core that we're going to discuss all have this same basic structure: they auto-generate code based on type definitions, implementing functionality that you could in theory have implemented by hand, but with far less programmer effort.

There are several syntax extensions distributed with Core, including:

- **Sexplib**: provides serialization for s-expressions.
- **Bin_prot**: provides serialization to an efficient binary format.
- **Fieldslib**: generates first-class values that represent fields of a record, as well as accessor functions and setters for mutable record fields.

- **Variantslib**: like Fieldslib for variants, producing first-class variants and other helper functions for interacting with variant types.
- **Pa_compare**: generates efficient, type-specialized comparison functions.
- **Pa_typehash**: generates a hash value for a type definition, *i.e.*, an integer that is highly unlikely to be the same for two distinct types.

We'll discuss each of these syntax extensions in detail, starting with Sexplib.

Sexplib

Formatting of s-expressions

Sexplib's format for s-expressions is pretty straightforward: an s-expression is written down as a nested parenthetical expression, with whitespace-separated strings as the atoms. Quotes are used for atoms that contain parenthesis or spaces themselves, backslash is the escape character, and semicolons are used to introduce comments. Thus, if you create the following file:

```
;; foo.scm

((foo 3.3) ;; Shall I compare thee  to a summer's dream?
 (bar "this is () an \" atom"))
```

we can load it up and print it back out again:

```
# Sexp.load_sexp "foo.scm";;
- : Sexp.t = ((foo 3.3) (bar "this is () an \" atom"))
```

Note that the comments were dropped from the file upon reading. This is expected, since there's no place in the `Sexp.t` type to store comments.

If we introduce an error into our s-expression, by, say, deleting the open-paren in front of `bar`, we'll get a parse error:

```
# Exn.handle_uncaught ~exit:false (fun () ->
    ignore (Sexp.load_sexp "foo.scm"));;
  Uncaught exception:

  (Sexplib.Sexp.Parse_error
   ((location parse) (err_msg "unexpected character: ')'") (text_line 4)
    (text_char 29) (global_offset 94) (buf_pos 94)))
```

(In the above, we use `Exn.handle_uncaught` to make sure that the exception gets printed out in full detail.)

Sexp converters

The most important functionality provided by Sexplib is the auto-generation of converters for new types. We've seen a bit of how this works already, but let's walk through a complete example. Here's the source for the beginning of a library for representing integer intervals.

```
(* file: int_interval.ml *)
(* Module for representing closed integer intervals *)

open Core.Std

(* Invariant: For any Range (x,y), y > x *)
type t = | Range of int * int
         | Empty
with sexp

let is_empty = function Empty -> true | Range _ -> false
let create x y = if x > y then Empty else Range (x,y)
let contains i x = match i with
   | Empty -> false
   | Range (low,high) -> x >= low && x <= high
```

We can now use this module as follows:

```
(* file: test_interval.ml *)

open Core.Std

let intervals =
  let module I = Int_interval in
  [ I.create 3 4;
    I.create 5 4; (* should be empty *)
    I.create 2 3;
    I.create 1 6;
  ]

let () =
  intervals
  |! List.sexp_of_t Int_interval.sexp_of_t
  |! Sexp.to_string_hum
  |! print_endline
```

But we're still missing something: we haven't created an `mli` for `Int_interval` yet. Note that we need to explicitly export the s-expression converters that were created within the ml. If we don't:

```
(* file: int_interval.mli *)
(* Module for representing closed integer intervals *)

type t
```

```
val is_empty : t -> bool
val create : int -> int -> t
val contains : t -> int -> bool
```

then we'll get the following error:

```
File "test_interval.ml", line 15, characters 20-42:
Error: Unbound value Int_interval.sexp_of_t
Command exited with code 2.
```

We could export the types by hand:

```
type t
val sexp_of_t : Sexp.t -> t
val t_of_sexp : t -> Sexp.t
```

But Sexplib has a shorthand for this as well, so that we can instead write simply:

```
type t with sexp
```

at which point `test_interval.ml` will compile again, and if we run it, we'll get the following output:

```
$ ./test_interval.native
((Range 3 4) Empty (Range 2 3) (Range 1 6))
```

Preserving invariants

One easy mistake to make when dealing with sexp converters is to ignore the fact that those converters can violate the invariants of your code. For example, the `Int_interval` module depends for the correctness of the `is_empty` check on the fact that for any value `Range (x,y)`, `y` is greater than or equal to `x`. The `create` function preserves this invariant, but the `t_of_sexp` function does not.

We can fix this problem by writing a custom sexp-converter, in this case, using the sexp-converter that we already have:

```
type t = | Range of int * int
         | Empty
with sexp

let create x y = if x > y then Empty else Range (x,y)

let t_of_sexp sexp =
  let t = t_of_sexp sexp in
  begin match t with
  | Range (x,y) when y < x ->
    of_sexp_error "Upper and lower bound of Range swapped" sexp
  | Empty | Range _ -> ()
  end;
  t
```

We call the function `of_sexp_error` to raise an exception because that improves the error reporting that Sexplib can provide when a conversion fails.

Getting good error messages

There are two steps to deserializing a type from an s-expression: first, converting the bytes in a file to an s-expression, and the second, converting that s-expression into the type in question. One problem with this is that it can be hard to localize errors to the right place using this scheme. Consider the following example:

```
(* file: read_foo.ml *)

open Core.Std

type t = { a: string; b: int; c: float option } with sexp

let run () =
  let t =
    Sexp.load_sexp "foo.scm"
    |! t_of_sexp
  in
  printf "b is: %d\n%!" t.b

let () =
  Exn.handle_uncaught ~exit:true run
```

If you were to run this on a malformatted file, say, this one:

```
;; foo.scm
((a not-an-integer)
 (b not-an-integer)
 (c ()))
```

you'll get the following error:

```
read_foo $ ./read_foo.native
Uncaught exception:

  (Sexplib.Conv.Of_sexp_error
   (Failure "int_of_sexp: (Failure int_of_string)") not-an-integer)
```

If all you have is the error message and the string, it's not terribly informative. In particular, you know that the parsing error-ed out on the atom "not-an-integer", but you don't know which one! In a large file, this kind of bad error message can be pure misery.

But there's hope! If we make small change to the `run` function as follows:

```
let run () =
  let t = Sexp.load_sexp_conv_exn "foo.scm" t_of_sexp in
  printf "b is: %d\n%!" t.b
```

and run it again, we'll get the following much more helpful error message:

```
read_foo $ ./read_foo.native
Uncaught exception:

  (Sexplib.Conv.Of_sexp_error
   (Sexplib.Sexp.Annotated.Conv_exn foo.scm:3:4
    (Failure "int_of_sexp: (Failure int_of_string)"))
   not-an-integer)
```

In the above error, "foo.scm:3:4" tells us that the error occurred on "foo.scm", line 3, character 4, which is a much better start for figuring out what has gone wrong.

Sexp-conversion directives

Sexplib supports a collection of directives for modifying the default behavior of the auto-generated sexp-converters. These directives allow you to customize the way in which types are represented as s-expressions without having to write a custom parser. We describe these directives below.

sexp-opaque

The most commonly used directive is `sexp_opaque`, whose purpose is to mark a given component of a type as being unconvertible. Anything marked with `sexp_opaque` will be presented as the atom `<opaque>` by the to-sexp converter, and will trigger an exception from the from-sexp converter. Note that the type of a component marked as opaque doesn't need to have a sexp-converter defined. Here, if we define a type without a sexp-converter, and then try to use it another type with a sexp-converter, we'll error out:

```
# type no_converter = int * int;;
type no_converter = int * int
# type t = { a: no_converter; b: string } with sexp;;
Characters 14-26:
  type t = { a: no_converter; b: string } with sexp;;
                ^^^^^^^^^^^^
Error: Unbound value no_converter_of_sexp
```

But with `sexp_opaque`, we won't:

```
# type t = { a: no_converter sexp_opaque; b: string } with sexp;;
type t = { a : no_converter Core.Std.sexp_opaque; b : string; }
val t_of_sexp__ : Sexplib.Sexp.t -> t = <fun>
val t_of_sexp : Sexplib.Sexp.t -> t = <fun>
val sexp_of_t : t -> Sexplib.Sexp.t = <fun>
```

And if we now convert a value of this type to an s-expression, we'll see the contents of field `a` marked as opaque:

```
# sexp_of_t { a = (3,4); b = "foo" };;
- : Sexp.t = ((a <opaque>) (b foo))
```

sexp_option

Another common directive is `sexp_opaque`, which is used to make an optional field in a record. Ordinary optional values are represented either as `()` for `None`, or as `(x)` for `Some x`. If you put an option in a record field, then the record field will always be required, and its value will be presented in the way an ordinary optional value would. For example:

```
# type t = { a: int option; b: string } with sexp;;
# sexp_of_t { a = None; b = "hello" };;
- : Sexp.t = ((a ()) (b hello))
# sexp_of_t { a = Some 3; b = "hello" };;
- : Sexp.t = ((a (3)) (b hello))
```

But what if we want a field to be optional, *i.e.*, we want to allow it to be omitted from the record entirely? In that case, we can mark it with `sexp_option`:

```
# type t = { a: int sexp_option; b: string } with sexp;;
# sexp_of_t { a = Some 3; b = "hello" };;
- : Sexp.t = ((a 3) (b hello))
# sexp_of_t { a = None; b = "hello" };;
- : Sexp.t = ((b hello))
```

sexp_list

One problem with the auto-generated sexp-converters is that they can have more parentheses than one would ideally like. Consider, for example, the following variant type:

```
# type compatible_versions = | Specific of string list
                             | All
  with sexp;;
# sexp_of_compatible_versions (Specific ["3.12.0"; "3.12.1"; "3.13.0"]);;
- : Sexp.t = (Specific (3.12.0 3.12.1 3.13.0))
```

You might prefer to make the syntax a bit less parenthesis-laden by dropping the parentheses around the list. `sexp_list` gives us this alternate syntax:

```
# type compatible_versions = | Specific of string sexp_list
                             | All
  with sexp;;
# sexp_of_compatible_versions (Specific ["3.12.0"; "3.12.1"; "3.13.0"]);;
- : Sexp.t = (Specific 3.12.0 3.12.1 3.13.0)
```

Bin_prot

S-expressions are a good serialization format when you need something machine-parseable as well as human readable and editable. But Sexplib's s-expressions are not particularly performant. There are a number of reasons for this. For one thing, s-expression serialization goes through an intermediate type, `Sexp.t`, which must be allocated and is then typically thrown away, putting non-trivial pressure on the GC. In addition, parsing and printing to strings in an ASCII format can be expensive for types like `int`s, `float`s and `Time.t`s where some real computation needs to be done to produce or parse the ASCII representation.

Bin_prot is a library designed to address these issues by providing fast serialization in a compact binary format. Kicking off the syntax extension is done by putting `with bin_io`. (This looks a bit unsightly in the top-level because of all the definitions that are generated. We'll elide those definitions here, but you can see it for yourself in the top-level.)

Here's a small complete example of a program that can read and write values using bin-io. Here, the serialization is of types that might be used as part of a message-queue, where each message has a topic, some content, and a source, which is in turn a hostname and a port.

```
(* file: message_example.ml *)

open Core.Std

(* The type of a message *)
module Message = struct
  module Source = struct
    type t = { hostname: string;
               port: int;
             }
    with bin_io
  end

  type t = { topic: string;
             content: string;
             source: Source.t;
           }
  with bin_io
end

(* Create the 1st-class module providing the binability of messages *)
let binable = (module Message : Binable.S with type t = Message.t)

(* Saves a message to an output channel.  The message is serialized to
   a bigstring before being written out to the channel.  Also, a
   binary encoding of an integer is written out to tell the reader how
   long of a message to expect.  *)
let save_message outc msg =
  let s = Binable.to_bigstring binable msg in
  let len = Bigstring.length s in
  Out_channel.output_binary_int outc len;
  Bigstring.really_output outc s

(* Loading the message is done by first reading in the length, and by
   then reading in the appropriate number of bytes into a Bigstring
   created for that purpose. *)
let load_message inc =
  match In_channel.input_binary_int inc with
  | None -> failwith "Couldn't load message: length missing from header"
  | Some len ->
    let buf = Bigstring.create len in
    Bigstring.really_input ~pos:0 ~len inc buf;
    Binable.of_bigstring binable buf

(* To generate some example messages *)
let example content =
  let source =
    { Message.Source.
      hostname = "ocaml.org"; port = 2322 }
  in
  { Message.
    topic = "rwo-example"; content; source; }

(* write out three messages... *)
let write_messages () =
  let outc = Out_channel.create "tmp.bin" in
  List.iter ~f:(save_message outc) [
    example "a wonderful";
    example "trio";
    example "of messages";
  ];
  Out_channel.close outc

(* ... and read them back in *)
let read_messages () =
  let inc = In_channel.create "tmp.bin" in
  for i = 1 to 3 do
    let msg = load_message inc in
    printf "msg %d: %s\n" i msg.Message.content
  done

let () =
  write_messages (); read_messages ()
```

Fieldslib

One common idiom when using records is to provide field accessor functions for a particular record.

```
type t = { topic: string;
           content: string;
           source: Source.t;
```

```
         }

let topic   t = t.topic
let content t = t.content
let source  t = t.source
```

Similarly, sometimes you simultaneously want an accessor to a field of a record and a textual representation of the name of that field. This might come up if you were validating a field and needed the string representation to generate an error message, or if you wanted to scaffold a form in a GUI automatically based on the fields of a record. Fieldslib provides a module `Field` for this purpose. Here's some code for creating `Field.t`'s for all the fields of our type `t`.

```
# module Fields = struct
    let topic =
      { Field.
        name   = "topic";
        setter = None;
        getter = (fun t -> t.topic);
        fset   = (fun t topic -> { t with topic });
      }
    let content =
      { Field.
        name   = "content";
        setter = None;
        getter = (fun t -> t.content);
        fset   = (fun t content -> { t with content });
      }
    let source =
      { Field.
        name   = "source";
        setter = None;
        getter = (fun t -> t.source);
        fset   = (fun t source -> { t with source });
      }
  end ;;
module Fields :
  sig
    val topic : (t, string list) Core.Std.Field.t
    val content : (t, string) Core.Std.Field.t
    val source : (t, Source.t) Core.Std.Field.t
  end
```

2013-02-05
19:46:47

CHAPTER 13

Handling JSON data

Data serialization, *i.e.* reading and writing program data to a sequence of bytes, is an important and common programming task. Sometimes you need to match someone else's data format (such as XML), and other times you just want to quickly dump some values to disk and read them back later. To this end, OCaml comes with several techniques for data serialization depending on what your problem is.

We'll start off by looking at JSON and XML first, as they are very common third-party data formats. After that, we'll introduce some syntax extensions in Core that make it really easy to manipulate s-expressions and safe binary serialisers directly from OCaml types.

JSON is a lightweight data-interchange format often used in web services and browsers. It is described in RFC4627 (*http://www.ietf.org/rfc/rfc4627.txt*), and is designed to be easy to parse and generate. You'll run into JSON very often when working with modern APIs, and so we'll cover several different ways to manipulate it in this chapter. Along the way we'll introduce new libraries and syntax extensions which make the job easier.

JSON consists of just two basic structures: an unordered collection of key/value pairs, and an ordered list of values. Values can be strings, booleans, floats, integers or null. Let's see what an example JSON record for a book description looks like:

```
{
  "title": "Real World OCaml",
  "tags" : [ "functional programming", "ocaml", "algorithms" ],
  "pages": 450,
  "authors": [
    { "name": "Jason Hickey", "affiliation": "Google" },
    { "name": "Anil Madhavapeddy", "affiliation": "Cambridge"},
    { "name": "Yaron Minsky", "affiliation": "Jane Street"}
  ],
  "is_online": true
}
```

JSON values usually start with an object at the top level that contains a set of key/value pairs. The keys must be strings, but values can be any JSON type. In the example above, `tags` is a string list, while the `authors` field contains a list of records. Unlike OCaml lists, JSON lists can contain completely different JSON types within them.

This free-form nature of JSON types is both a blessing and a curse. It's very easy to generate JSON values, but code parsing them also has to cope with handling subtle variations in how values are represented. For example, what if the `pages` value above is actually represented as a string value of `"450"` instead of an integer?

Our first task is to parse the JSON into a more structured OCaml type so that we can use static typing more effectively. When manipulating JSON in Python or Ruby, you might write unit tests to check that you have handled unusual inputs. The OCaml model prefers compile-time static checking as well as unit tests. For example, using pattern matching can warn you if you've not checked that a value can be `Null` as well as contain an actual value.

Installing the Yojson library

There are several JSON libraries available for OCaml. For this chapter, we've picked the `Yojson` (*http://mjambon.com/yojson.html*) library. The easiest way to install it is by using the OPAM package manager (see Chapter 23 for installation instructions). Once OPAM is working, you will need to run `opam install yojson`.

Parsing standard JSON with Yojson

The JSON specification has very few data types, and Yojson implements these in the `Yojson.Basic` module. The `json` type shown below is sufficient to express any valid JSON structure. Note that some of the types are recursive, so that fields can contain references to more JSON fields, and that it also specifically includes a `Null` variant for empty fields.

```
type json = [
  | `Assoc of (string * json) list
  | `Bool of bool
  | `Float of float
  | `Int of int
  | `List of json list
  | `Null
  | `String of string ]
```

Let's parse the earlier JSON string example into this type now. The first stop is the `Yojson.Basic` documentation, where we find these helpful functions:

```
val from_string : ?buf:Bi_outbuf.t -> ?fname:string -> ?lnum:int -> string -> json
(* Read a JSON value from a string.
  [buf] : use this buffer at will during parsing instead of creating a new one.
  [fname] : data file name to be used in error messages. It does not have to be a real file.
  [lnum] : number of the first line of input. Default is 1.
```

```
val from_channel : ?buf:Bi_outbuf.t -> ?fname:string -> ?lnum:int -> in_channel -> json
(* Read a JSON value from a channel. See [from_string] for the meaning of the
   optional arguments. *)

val from_file : ?buf:Bi_outbuf.t -> ?fname:string -> ?lnum:int -> string -> json
(* Read a JSON value from a file. See [from_string] for the meaning of the optional
   arguments. *)
```

When first reading these interfaces, you can generally ignore the optional arguments (which have the question marks in the type signature), as they will be filled in with sensible values. The signature for these values with the optional elements removed makes their purpose quite clear:

```
val from_string : string -> json
val from_file : string -> json
val from_channel : in_channel -> json
```

The `in_channel` constructor is from the original OCaml standard library, and its use is considered deprecated when using the Core standard library. This leaves us with two ways of parsing the JSON: either from a string buffer, or from a file on a filesystem. The next example shows both in action, assuming the JSON record is stored in a file called *book.json*:

```
open Core.Std

let _ =
  (* Read JSON file into an OCaml string *)
  let buf = In_channel.read_all "book.json" in

  (* Use the string JSON constructor *)
  let json1 = Yojson.Basic.from_string buf in

  (* Use the file JSON constructor *)
  let json2 = Yojson.Basic.from_file "book.json" in

  (* Test that the two values are the same *)
  print_endline (if json1 = json2 then "OK" else "FAIL")
  print_endline (if phys_equal json1 json2 then "FAIL" else "OK")
```

The `from_file` function accepts an input filename and takes care of opening and closing it for you. It's far more common to use `from_string` to construct a JSON value from an OCaml `string`. These strings can come from a network connection (we'll see more of this in Chapter 16) or even a database. Finally, the example checks that the two input mechanisms actually resulted in the same OCaml data structure.

The difference between = and ==, and `phys_equal` in Core

If you come from a C/C++ background, you will probably reflexively use `==` to test two values for equality. In OCaml, `==` tests for *physical* equality, and `=` tests for *structural* equality.

The == physical equality test will match if two data structures have precisely the same pointer in memory. Two data structures that have identical contents, but are constructed separately, will not match using this operator. In the JSON example, the `json1` and `json2` values are not identical and so would fail the physical equality test.

The = structural equality operator recursively inspects each field in the two values and tests them individually for equality. In the JSON parsing example, every field will be traversed and checked, and they will check out as equal. Crucially, if your data structure is cyclical (that is, a value recursively points back to another field within the same structure), the = operator will never terminate, and your program will hang! In this situation, you must use the physical equality operator, or write a custom comparison function that breaks the recursion.

It's quite easy to mix up the use of = and ==, so Core disables the == operator and provides `phys_equal` instead. You'll see a type error if you use == anywhere:

```
# 1 == 2;;
Error: This expression has type int but an expression was expected of type
         [ `Consider_using_phys_equal ]
# phys_equal 1 2;;
- : bool = false
```

If you feel like hanging your OCaml interpreter, you can verify what happens with recursive values for yourself:

```
# type t1 = { foo1:int; bar1:t2 } and t2 = { foo2:int; bar2:t1 } ;;
type t1 = { foo1 : int; bar1 : t2; }
and t2 = { foo2 : int; bar2 : t1; }
# let rec v1 = { foo1=1; bar1=v2 } and v2 = { foo2=2; bar2=v1 };;
<lots of text>
# v1 == v1;;
- : bool = true
# phys_equal v1 v1;;
- : bool = true
# v1 = v1 ;;
<press ^Z and kill the process now>
```

Selecting values from JSON structures

Now that we've figured out how to parse the example JSON, lets see how we can manipulate it from OCaml code with a more complete example.

```
open Core.Std
open Async.Std
open Printf

let _ =
  (* Read the JSON file *)
  let json = Yojson.Basic.from_file "book.json" in

  (* Locally open the JSON manipulation functions *)
```

```
let open Yojson.Basic.Util in
let title = json |! member "title" |! to_string in
let tags = json |! member "tags" |! to_list |! filter_string in
let pages = json |! member "pages" |! to_int in
let is_online = json |! member "is_online" |! to_bool_option in
let is_translated = json |! member "is_translated" |! to_bool_option in
let authors = json |! member "authors" |! to_list in
let names = List.map authors ~f:(fun json -> member "name" json |! to_string) in

(* Print the results of the parsing *)
printf "Title: %s (%d)\n" title pages;
printf "Authors: %s\n" (String.concat ~sep:", " names);
printf "Tags: %s\n" (String.concat ~sep:", " tags);
let string_of_bool_option =
  function
  | None -> "<none>"
  | Some true -> "yes"
  | Some false -> "no" in
printf "Online: %s\n" (string_of_bool_option is_online);
printf "Translated: %s\n" (string_of_bool_option is_translated)
```

This introduces the `Yojson.Basic.Util` module, which contains *combinator* functions for JSON manipulation. Combinators are a style of function that can be chained together using the `|!` pipe operator to select and convert values out of the JSON structure. Let's examine some of them in more detail:

- For the `title` string, the `member` combinator extracts the key from the array, and casts it to an OCaml string. An exception is raised if the JSON value is not a string.
- The `tags` field is similar to `title`, but are passed through the `to_list` combinator since they are a JSON list. The `filter_string` combinator folds all of the strings in the JSON list into an OCaml list (any non-strings also in there are simply ignored).
- The `is_online` and `is_translated` fields are optional in our JSON schema, and no error is raised if they are not present in the JSON array. The resulting OCaml type is a `string option` to reflect this. In our example, only `is_online` is present and `is_translated` will be `None`.

In the last part of the example, we simply print the parsed fields since they are just normal OCaml values. This technique of using chained parsing functions is very powerful in combination with the OCaml type system. Many errors that don't make sense at runtime (for example, mixing up lists and objects) will be caught statically via a type error.

Constructing JSON values

To build and print JSON values, you can just construct values of type `json` and call the `to_string` function. There are also pretty-printing functions that lay out the output in a more human-readable style:

```
# let x = `Assoc [ ("key", `String "value") ] ;;
val x : [> `Assoc of (string * [> `String of string ]) list ] =
  `Assoc [("key", `String "value")]

# Yojson.Basic.pretty_to_string x ;;
- : string = "{ \"key\": \"value\" }"

# Yojson.Basic.pretty_to_channel stdout x ;;
{ "key": "value" }
- : unit = ()
```

In the example above, although the value that `x` has is compatible with the type `json`, it's not explicitly defined as such. The type inference engine will figure out a type that is based on how the value `x` is used, and in this case only the `Assoc` and `String` variants are present. One difficulty you will encounter is that type errors involving polymorphic variants can be quite verbose if you make a mistake in your code. For example, suppose you build an `Assoc` and include a single value instead of a list of keys:

```
# let x = `Assoc ("key", `String "value");;
val x : [> `Assoc of string * [> `String of string ] ] =
  `Assoc ("key", `String "value")

# Yojson.Basic.pretty_to_string x;;
Error: This expression has type
         [> `Assoc of string * [> `String of string ] ]
       but an expression was expected of type Yojson.Basic.json
       Types for tag `Assoc are incompatible
```

The type error above isn't *wrong* as such, but can be inconvenient to wade through for larger values. An easy way to narrow down this sort of type error is to add explicit type annotations as a compiler hint about your intentions:

```
# let (x:Yojson.Basic.json) = `Assoc ("key", `String "value");;
Error: This expression has type 'a * 'b
       but an expression was expected of type
         (string * Yojson.Basic.json) list
```

In this case, we've marked the `x` as being of type `Yojson.Basic.json`, and the compiler immediately spots that the argument to the `Assoc` variant has the incorrect type. This illustrates the strengths and drawbacks of using polymorphic variants: they make it possible to easily subtype across module boundaries (the `Basic` and `Safe` in Yojson's case), but the error messages can be more confusing. However, a bit of careful manual type annotation is all it takes to make tracking down such issues much easier.

avsm: segway into memory representation of polyvariants here?

Using non-standard JSON extensions

The standard JSON types are *really* basic, and OCaml types are far more expressive. Yojson supports an extended JSON format for those times when you're not interoperating with external systems and just want a convenient human-readable local format. The `Yojson.Safe.json` type is a superset of the `Basic` polymorphic variant, and looks like this:

```
type json = [
  | `Assoc of (string * json) list
  | `Bool of bool
  | `Float of float
  | `Floatlit of string
  | `Int of int
  | `Intlit of string
  | `List of json list
  | `Null
  | `String of string
  | `Stringlit of string
  | `Tuple of json list
  | `Variant of string * json option ]
```

You should immediately be able to spot a benefit of using polymorphic variants here. A standard JSON type such as a `String` will type-check against both the `Basic` module and also the non-standard `Safe` module. However, if you use extension values such as `Tuple` with the `Basic` module, they will not be a valid sub-type and the compiler will complain.

The extensions includes with Yojson include:

- The `lit` suffix denotes that the value is stored as a JSON string. For example, a `Floatlit` will be stored as `"1.234"` instead of `1.234`.
- The `Tuple` type is stored as `("abc", 123)` instead of a list.
- The `Variant` type encodes OCaml variants more explicitly, as `<"Foo">` or `<"Bar": 123>` for a variant with parameters.

The only purpose of these extensions is to make the data representation more expressive without having to refer to the original OCaml types. You can always cast a `Safe.json` to a `Basic.json` type by using the `to_basic` function as follows:

```
val to_basic : json -> Yojson.Basic.json
(** Tuples are converted to JSON arrays, Variants are converted to JSON strings
or arrays of a string (constructor) and a json value (argument). Long integers
are converted to JSON strings.  Examples:

`Tuple [ `Int 1; `Float 2.3 ]   ->    `List [ `Int 1; `Float 2.3 ]
`Variant ("A", None)            ->    `String "A"
`Variant ("B", Some x)          ->    `List [ `String "B", x ]
`Intlit "12345678901234567890"  ->    `String "12345678901234567890"
 *)
```

Automatically mapping JSON to OCaml types

The combinators described earlier make it fairly easy to extract fields from JSON records, but the process is still pretty manual. We'll talk about how to do larger-scale JSON parsing now, using a domain-specific language known as ATD (*http://oss.wink.com/atdgen/*).

The idea behind ATD is to specify the format of the JSON in a separate file, and then run a compiler (`atdgen`) that outputs OCaml code to construct and parse JSON values. This means that you don't need to write any OCaml parsing code at all, as it will all be auto-generated for you.

Let's go straight into looking at an example of how this works, by using a small portion of the Github API. Github is a popular code hosting and sharing website that provides a JSON-based web API (*http://developer.github.com*). The ATD code fragment below describes the Github authorization API. It is based on a pseudo-standard web protocol known as OAuth, and is used to authorized users to access Github services.

```
type scope = [
    User <json name="user">
  | Public_repo <json name="public_repo">
  | Repo <json name="repo">
  | Repo_status <json name="repo_status">
  | Delete_repo <json name="delete_repo">
  | Gist <json name="gist">
]

type app = {
  name: string;
  url: string;
}  <ocaml field_prefix="app_">

type authorization_request = {
  scopes: scope list;
  note: string;
} <ocaml field_prefix="auth_req_">

type authorization_response = {
  scopes: scope list;
  token: string;
  app: app;
  url: string;
  id: int;
  ?note: string option;
  ?note_url: string option;
}
```

ATD is (deliberately) similar to OCaml type definitions. Each field can include extra annotations to customise the parsing code for a particular backend. For example, the Github `scope` field above is defined as a variant type, but with the actual JSON values being defined explicitly (as lower-case versions).

The ATD spec can be compiled to a number of OCaml targets. Let's run the compiler twice, to generate some OCaml type definitions, and a JSON serialiser.

```
$ atdgen -t github.atd
$ atdgen -j github.atd
```

This will generate some new files in your current directory. `Github_t.ml` and `Github_t.mli` will contain an OCaml module with types defines that correspond to the ATD file. It looks like this:

```
type scope = [
  | `User | `Public_repo | `Repo | `Repo_status
  | `Delete_repo | `Gist
]

type app = {
  app_name (*atd name *): string;
  app_url (*atd url *): string
}

type authorization_request = {
  auth_req_scopes (*atd scopes *): scope list;
  auth_req_note (*atd note *): string
}

type authorization_response = {
  scopes: scope list;
  token: string;
  app: app;
  url: string;
  id: int;
  note: string option;
  note_url: string option
}
```

There is an obvious correspondence to the ATD definition. Note in particular that field names in separate OCaml records cannot shadow each other, and so we specifically prefix every field with a prefix to distinguish it from other records. For example, `<ocaml field_prefix="auth_req_">` in the ATD spec prefixes every field name in the generated `authorization_request` record with `auth_req`.

The `Github_t` module only contains the type definitions, while `Github_j` has a concrete serialization module to and from JSON. You can read the `github_j.mli` to see the full interface, but the important functions for most uses are the conversion functions to and from a string. For our example above, this looks like:

```
val string_of_authorization_response :
  ?len:int -> authorization_response -> string
  (** Serialize a value of type {!authorization_response}
      into a JSON string.
      @param len specifies the initial length
```

```
                    of the buffer used internally.
                    Default: 1024. *)

val authorization_response_of_string :
  string -> authorization_response
```

This is pretty convenient! We've written a single ATD file, and all the OCaml boilerplate to convert between JSON and a strongly typed record has been generated for us. You can control various aspects of the serializer by passing flags to `atdgen`. The important ones for JSON are:

- `-j-std`: work in standard JSON mode, and never print non-standard JSON extensions.
- `-j-custom-fields FUNCTION`: call a custom function for every unknown field encountered, instead of raising a parsing exception.
- `-j-defaults`: force the output a JSON value even if the specification defines it as the default value for that field.

The full ATD specification is quite sophisticated (and well documented online at its homepage). The ATD compiler can also target formats other than JSON, and also outputs code for other languages such as Java if you need more interoperability. There are also several similar projects you can investigate which automate the code generation process: Piqi (*http://piqi.org*) uses the Google protobuf format, and Thrift (*http://thrift.apache.org*) supports a huge variety of other programming languages.

We'll also return to the Github example here later in the book when discussing the Async networking library, and you can find the full ATD specification for Github in the `ocaml-github` (*http://github.com/avsm/ocaml-github*) repository.

CHAPTER 14

XML Streams and Trees

XML is a markup language designed to store tree-structured data in a format that is (somewhat) human- and machine-readable. Like JSON, it is a textual format commonly used in web technologies, with a complete specification (*http://www.w3.org/TR/REC-xml/*) available online. A complete description is beyond the scope of this book, but we'll explain how to manipulate it now.

Obtaining and installing XMLM

The remainder of this section uses the freely available XMLM library. It's easiest to obtain it via OPAM (see Chapter 23 for installation instructions). You need to run `opam install xmlm` once OPAM is installed. The library documentation is also readable online (*http://erratique.ch/software/xmlm/doc/Xmlm*).

Since XML is such a common web format, we've taken our example document from the DuckDuckGo (*http://duckduckgo.com*) search engine. This is a smaller search engine than the usual suspects, but has the advantage of a freely available API that doesn't require you to register before using it. We'll talk more about how to use the live API later in Chapter 16, but for now here's what a shortened XML search response from DuckDuckGo looks like:

```
<DuckDuckGoResponse version="1.0">
<Heading>DuckDuckGo</Heading>
<AbstractText>DuckDuckGo is an Internet search engine.</AbstractText>
<AbstractURL>https://en.wikipedia.org/wiki/DuckDuckGo</AbstractURL>
<AbstractSource>Wikipedia</AbstractSource>
<Results>
<Result>
  <Text>Official site</Text>
  <FirstURL>https://duckduckgo.com/</FirstURL>
  </Result>
</Results>
<RelatedTopics>
```

```
 <RelatedTopic>
   <Text>Companies based in Pennsylvania</Text>
   <FirstURL>
     http://duckduckgo.com/c/Companies_based_in_Pennsylvania
   </FirstURL>
 </RelatedTopic>
 <RelatedTopic>
   <Text>Internet search engines</Text>
   <FirstURL>
     http://duckduckgo.com/c/Internet_search_engines
   </FirstURL>
 </RelatedTopic>
</RelatedTopics>
</DuckDuckGoResponse>
```

The XML document is structured as a series of `<tags>` that are closed by an end `</tag>`. The opening tags have an optional set of key/value attributes and usually contain text data or further tags within them. If the XML document is large, we don't want to read the whole thing into memory before processing it. Luckily we don't have to, as there are two parsing strategies for XML: a low-level *streaming* API that parses a document incrementally, and a simpler but more inefficient tree API. We'll start with the streaming API first, as the tree API is built on top of it.

Stream parsing XML

The XMLM documentation is a good place to read about the overall layout of the library. It tells us that:

> A well-formed sequence of `signal`s represents an XML document tree traversal in depth-first order. Input pulls a well-formed sequence of `signal`s from a data source and output pushes a well-formed sequence of `signal`s to a data destination. Functions are provided to easily transform sequences of `signal`s to/from arborescent data structures.

The type of a `signal` reveals the basic structure of the streaming API in XMLM:

```
type signal = [
  | `Data of string
  | `Dtd of dtd
  | `El_end
  | `El_start of tag
]
```

XMLM outputs an ordered sequence of these signals to your code as it parses the document. The first `signal` when inputting an XML document is always a `Dtd`. The DTD (or *document type description*) optionally defines which tags are allowed within the XML document. Some XML parsers can validate a document against a DTD, but XMLM is a *non-validating* parser that reads the DTD if present, but disregards its contents. The `El_start` and `El_end` signals indicate the opening and closing of tags, and `Data` passes the free-form information contained between tags.

Let's take a shot at handling signals by writing the XML identity function that parses some XML and outputs it again. There is no explicit buffering required since this uses the XMLM streaming API.

```
let xml_id i o =
  let rec pull i o depth =
    Xmlm.output o (Xmlm.peek i);
    match Xmlm.input i with
     | `El_start _ -> pull i o (depth + 1)
     | `El_end -> if depth > 1 then pull i o (depth - 1)
     | `Data _ -> pull i o depth
     | `Dtd _ -> assert false
  in
  Xmlm.output o (Xmlm.input i); (* `Dtd *)
  pull i o 0;
  if not (Xmlm.eoi i) then invalid_arg "document not well-formed"

let _ =
  let i = Xmlm.make_input (`Channel (open_in "ddg.xml")) in
  let o = Xmlm.make_output (`Channel stdout) in
  xml_id i o
```

Let's start at the bottom, where we open up input and output channels to pass to `Xmlm` parser. The `input` and `output` constructor functions use a polymorphic variant to define the mechanism that the library should use to read the document. `Channel` is the simplest, but there are several others available.

```
type source = [
  | `Channel of in_channel
  | `Fun of unit -> int
  | `String of int * string
]
```

The `Fun` channel returns one character at a time as an integer, and `String` starts parsing an OCaml string from the given integer offset. Both of these are will normally be used in preference to `Channel`, which uses an interface that is deprecated in Core.

The `xml_id` function begins by reading one signal, which will always be a `Dtd`. The recursive `pull` function is then invoked to iterate over the remaining signals. This uses `Xmlm.peek` to inspect the current input signal and immediately output it. The rest of the function is not strictly necessary, but it tracks that all of the tags that have been started via the `El_start` signal are also closed by a corresponding `El_end` signal. Once the `pull` function has finished due to the opening tag being closed, the `Xmlm.eoi` function verifies that the "end of input" has been reached.

Tree parsing XML

Signals enforce a very iterative style of parsing XML, as your program has to deal with signals arriving serially. It's often more convenient to deal with complete XML documents directly in-memory as an OCaml data structure. You can convert a signal stream into an OCaml structure by defining the following data type and helper functions:

```
type tree =
  | Element of Xmlm.tag * tree list
  | Data of string

let in_tree i =
  let el tag children = Element (tag, children) in
  let data d = Data d in
  Xmlm.input_doc_tree ~el ~data i

let out_tree o t =
  let frag = function
  | Element (tag, childs) -> `El (tag, childs)
  | Data d -> `Data d
  in
  Xmlm.output_doc_tree frag o t
```

The type `tree` can be pattern-matched and traversed like a normal OCaml data structure. Let's see how this works by extracting out all the "Related Topics" in the example document. First, we'll need a few helper combinator functions to filter through tags and trees, with the following signature:

```
(* Extract a textual name from an XML tag.
   Discards the namespace information. *)
val name : Xmlm.tag -> string

(* Filter out the contents of a tag [n] from a tagset,
   and return the concatenated contents of all of them *)
val filter_tag : string -> tree list -> tree list

(* Given a list of [trees], concatenate all of the data contents
   into a string, and discard any sub-tags within it *)
val concat_data : tree list -> string
```

The implementation of these signatures fold over the `tree` structure to filter the tags which match the desired tag name. A similar version that matches on tag attributes is left as an exercise for you to try.

```
let name ((_,n),_) = n

let filter_tag n =
  List.fold_left ~init:[] ~f:(fun acc ->
    function
    |Element (tag, ts) when name tag = n ->
      ts @ acc
    |_ -> acc
  )

let concat_data =
  List.fold_left ~init:"" ~f:(fun acc ->
    function
    |Data s -> acc ^ s
    |_ -> acc
  )
```

(*avsm*: have we explained `fold_left` before this section or does it need a full intro?)

Notice the use of a *guard pattern* in the `filter_tag` pattern match. This looks for an `Element` tag that matches the name parameter, and concatenates the results with the accumulator list.

Once we have these helper functions, the selection of all the `<Text>` tags is a matter of chaining the combinators together to peform the selection over the `tree` data structure.

```
let topics trees =
  filter_tag "DuckDuckGoResponse" trees |!
  filter_tag "RelatedTopics" |!
  filter_tag "RelatedTopic" |!
  filter_tag "Text" |!
  List.iter ~f:(fun x -> concat_data [x] |! print_endline)

let _ =
  let i = Xmlm.make_input (`Channel (open_in "ddg.xml")) in
  let (_,it) = in_tree i in
  topics [it]
```

The `filter_tag` combinator accepts a `tree list` parameter and outputs a `tree list`. This lets us easily chain together the results of one filter to another, and hence select hierarchical XML tags very easily. When we get to the `<Text>` tag, we iterate over all the results and print each one individually.

Constructing XML documents using syntax extensions

In the earlier JSON chapter, we explained how to construct records by creating the records directly. You can do exactly the same thing for XML, but there is also a more automated method available by using OCaml's facility for syntax extensions.

The OCaml distribution provides the `camlp4` tool for this purpose, which you can view as a type-safe preprocessor. Camlp4 operates by loading in a set of syntax extension modules that transform the Abstract Syntax Tree (AST) of OCaml, usually by adding nodes that generate code. We'll talk about how to build your own syntax extensions later in the book, but for now we'll describe how to *use* several syntax extensions that make it easier to manipulate external data formats such as XML.

We'll use the Atom 1.0 syndication format as our example here. Atom feeds allow web-based programs (such as browsers) to poll a website for updates. The website owner publishes a feed of content in a standardized XML format via HTTP. This feed is then parsed by clients and compared against previously downloaded versions to determine which contents are available.

Here's an example of an Atom feed:

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
 <title>Example Feed</title>
 <subtitle>A subtitle.</subtitle>
 <link href="http://example.org/feed/" rel="self" />
 <link href="http://example.org/" />
 <id>urn:uuid:60a76c80-d399-11d9-b91C-0003939e0af6</id>
 <updated>2003-12-13T18:30:02Z</updated>

 <entry>
  <title>Atom-Powered Robots Run Amok</title>
  <link href="http://example.org/2003/12/13/atom03" />
  <link rel="alternate" type="text/html" href="http://example.org/2003/12/13/atom03.html"/>
  <link rel="edit" href="http://example.org/2003/12/13/atom03/edit"/>
  <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
  <updated>2003-12-13T18:30:02Z</updated>
  <summary>Some text.</summary>
  <author>
    <name>John Doe</name>
    <email>johndoe@example.com</email>
  </author>
 </entry>
</feed>
```

We want to build this by minimising the amount of repetitive XML generation code. The "Caml on the Web" (COW) library provides a syntax extension that is useful here.

Installing Caml on the Web (COW)

The COW library and syntax extension can be installed via OPAM by `opam install cow`. There are two OCamlfind packages installed: the library is called `cow` and the syntax extension is activated with the `cow.syntax` package.

One caveat to bear in mind is that COW isn't fully compatible with Core yet, and so you must use the syntax extension before opening the Core modules. (*avsm*: we can fix this easily, but note is here as a warning to reviewers).

Let's start to build up an Atom specification using Cow. First, the `<author>` tag can be represented with the following type:

```
type author = {
  name: string;
  uri: string option;
  email: string option;
} with xml
```

This is a standard record type definition with the addition of `with xml` at the end. This uses a syntax extension to signify that we wish to generate boilerplate code for handling this record as an XML document.

Invoking `camlp4` syntax extensions

The OCaml compiler can call `camlp4` automatically during a compilation to preprocess the source files. This is specified via the `-pp` flag to the compiler. You don't normally need to specify this flag yourself. Use the `ocamlfind` utility instead to generate the right command-line flags for you. Here's a small shell script which preprocesses a source file with the COW syntax extension:

```
#!/bin/sh -x

file=$1
lib=cow.syntax
bin=ocamlfind
args=`$bin query -predicates syntax,preprocessor -r -format '-I %d %a' $lib`
camlp4o -printer o $args $file
```

You can supply `ocamlfind` with a number of different predicates to define the type of build you are running (preprocessing, compilation or linking). The final part of the script invokes the `camlp4o` binary on your ML source file and outputs the transformed source code to your terminal.

Let's see the OCaml code that has been generated for our `author` record after it has been preprocessed:

```
type author = {
  name: string;
  uri: string option;
  email: string option;
}

let rec xml_of_author author : Cow.Xml.t =
  List.flatten
    [ (match match author.email with
             | None -> []
             | Some var1 -> [ `Data var1 ]
       with
       | [] -> []
       | _ ->
         [ `El (((("", "email"), []) : Cow.Xml.tag),
           (match author.email with
            | None -> []
            | Some var1 -> [ `Data var1 ])) ]);
      (match match author.uri with
        | None -> [] | Some var2 -> [ `Data var2 ]
       with
       | [] -> []
       | _ ->
```

```
         [ `El (((("", "uri"), []) : Cow.Xml.tag),
           (match author.uri with
             | None -> []
             | Some var2 -> [ `Data var2 ])) ]);
      (match [ `Data author.name ] with
       | [] -> []
       | _ ->
         [ `El (((("", "name"), []) : Cow.Xml.tag),
         [ `Data author.name ]) ]) ]
```

Notice that the `with xml` clause has been replaced with a new `xml_of_author` function that has been generated for you. It accepts an `author` value and returns an `Xml.t` value. The generated code isn't really meant to be human-readable, but you don't normally see it when using the syntax extension (we've only dumped it out here to illustrate how `camlp4` works).

If we run `xml_of_author` and convert the result to a human-readable string, our complete example looks like:

```
type author = {
  name: string;
  uri: string option;
  email: string option;
} with xml

let anil = {
  name = "Anil Madhavapeddy";
  uri = Some "http://anil.recoil.org";
  email = Some "anil@recoil.org"
}

let _ = print_endline (Cow.Xml.to_string (xml_of_author anil))
```

This will generate the following XML output on the terminal when you execute it:

```
<?xml version="1.0" encoding="UTF-8"?>
<email>anil@recoil.org</email>
<uri>http://anil.recoil.org</uri>
<name>Anil Madhavapeddy</name>
```

This is convenient, but just one small portion of Atom. How do we express the full Atom scheme from earlier? The answer is with just a few more records that match the Atom XML schema.

```
type author = {
  name: string;
  uri: string option;
  email: string option;
} with xml

type date =
  int * int * int * int * int (* year, month, date, hour, minute *)
```

2013-02-05
19:46:47

```
with xml

let xml_of_date (year,month,day,hour,min) =
  let d = Printf.sprintf "%.4d-%.2d-%.2dT%.2d:%.2d:00Z" year month day hour min in
  <:xml< $str:d$ >>

type meta = {
  id: string;
  title: string;
  subtitle: string option;
  author: author option;
  rights: string option;
  updated: date;
} with xml
```

We've now filled in more of the Atom schema with these records. The first problem we run into is that occasionally there is a mismatch between the syntax extension's idea of what the auto-generated XML should look like, and the reality of the protocol you are mapping to.

The Atom date field is a good example. We define it as a tuple of integers, but the format mandated by the specification is actually a free-form text format and not XML. However, because the syntax extension generates normal OCaml functions, we can just override the `xml_of_date` function with a custom one which returns the correct XML fragment. Any references further down the module will just use our overridden version and ignore the auto-generated one.

There's another interesting bit of new syntax in the `xml_of_date` function known as a *quotation*. OCaml not only allows code to be generated during pre-processing, but also to override the core language grammar with new constructs. The most common way of doing this is by embedding the custom grammars inside `<:foo< ... >>` tags, where `foo` represents the particular grammar being used. In the case of COW, this lets you generate XMLM-compatible OCaml values just by typing in XML tags.

TODO antiquotations.

TODO finish the atom example.

Working with XHTML

TODO use Cow.Html to generate a more complete Atom feed.

2013-02-05
19:46:47

2013-02-05
19:46:47

CHAPTER 15

Input and Output

2013-02-05
19:46:47

CHAPTER 16

Concurrent Programming with Async

When you start building OCaml code that interfaces with external systems, you'll soon need to handle concurrent operations. Consider the case of a web server sending a large file to many clients, or a GUI waiting for a mouse clicks. These applications often need to block while waiting for input for a particular task, and process something else during that time. Meanwhile, when new data does appear, the blocked task needs to be resumed as quickly as possible.

Efficiency really matters here, as busy servers can often handle tens of thousands of simultaneous connections. An equally important concern is readable source code, where the control flow of the program is obvious at a glance.

You've probably used preemptive system threads before in some programming languages such as Java or C#. In this model, each task is usually given an operating system thread of its own. Other languages such as Javascript are single-threaded, and applications must register function callbacks to be triggered upon external events (such as a timeout or browser click).

Both of these mechanisms have tradeoffs. Preemptive threads require their own memory stacks and can be memory hungry. The operating system can also arbitrarily interleave the execution of threads, and so they require careful locking around shared data structures.

Event-driven systems usually only execute a single task at a time and require less locking. However, the program structure can often descend into a maze of event callbacks for even a simple operation that blocks a few times. Code readability matters, and so we'd like to avoid such spaghetti control flow.

The `Async` OCaml library offers a hybrid model that lets you write event-driven code that can block *without* the complexity of preemptive threading. Let's dive straight into an example to see what this looks like, and then explain some of the new concepts. We're going to search for definitions of English terms using the DuckDuckGo search engine.

Example: searching definitions with DuckDuckGo

DuckDuckGo is a search engine with a freely available search interface. A DuckDuckGo search is executed by making an HTTP request to `api.duckduckgo.com`. The result comes back in either JSON or XML format, depending on what was requested in the original query string. Let's write some functions that construct the right URI and can parse the resulting JSON.

Before we can make the HTTP calls, we need a couple of helper functions with the following signature.

```
(* Generate a DuckDuckGo API search URI for [query] *)
val make_ddg_uri : query:string -> Uri.t

(* Extract the Definition field from the DuckDuckGo search
   response, or return [None] if it doesn't exist *)
val get_definition_from_json: string -> string option
```

This code uses a couple of new libraries we haven't seen before. You will need to OPAM install `uri` and `yojson` (refer to Appendix if you need help). Let's see how to implement them first.

URI handling

You're hopefully familiar with HTTP URLs, which identify endpoints across the World Wide Web. These are actually part of a more general family known as Uniform Resource Identifiers (URIs). The full URI specification is defined in RFC3986 (*http://tools.ietf.org/html/rfc3986*) (and is rather complicated!). Luckily, the `ocaml-uri` library provides a strongly-typed interface which takes care of much of the hassle.

```
(* Generate a DuckDuckGo search URI from a query string *)
let make_ddg_uri =
  let base_uri = "http://api.duckduckgo.com/?format=json" in
  let uri = Uri.of_string base_uri in
  fun ~query ->
    Uri.add_query_param uri ("q", [query])
```

A `Uri.t` is constructed from the `Uri.of_string` function, and a query parameter `q` is added with the desired search query. The library takes care of encoding the URI correctly when outputting it in the network protocol.

Note that the URI manipulation functions are all *pure* functions which return a new URI value, and never modify the input. This makes it easier to pass around URI values through your application stack without fear of modification.

Parsing JSON strings

The HTTP response from DuckDuckGo is in JSON, a common (and thankfully simple) format that is specified in RFC4627 (*http://www.ietf.org/rfc/rfc4627.txt*). There are quite a few JSON parsers available for OCaml, and we've picked `Yojson` (*http://mjambon.com/yojson.html*) for this example.

There are a few non-standard extensions to JSON, so Yojson exposes them as the `Basic` and `Safe` sub-modules. It doesn't really matter which one we pick for this simple example, so we'll go with `Safe`.

The input `string` is parsed using `Yojson.Safe.from_string` into an OCaml data type. The JSON values are represented using polymorphic variants, and can thus be pattern matched more easily once they have been parsed by Yojson.

```
type json = [
  | `Assoc of (string * json) list
  | `Bool of bool
  | `Float of float
  | `Int of int
  | `List of json list
  | `Null
  | `String of string
  | `Tuple of json list
]
```

We're expecting the DuckDuckGo response to be a record, with an optional `Description` field being one of the keys in the record. The `get_definition_from_json` does a pattern match on this, and returns an optional string if a definition is found within the result.

```
(* Extract the Definition field from the DuckDuckGo search
   response, or return [None] if it doesn't exist *)
let get_definition_from_json (json:string) =
  match Yojson.Safe.from_string json with
  |`Assoc kv_list ->
      let open Option in
      List.Assoc.find kv_list "Definition" >>|
      Yojson.Safe.to_string
  |_ -> None
```

Notice that we use options here instead of throwing exceptions on an error. When the `Option` module is opened, it provides a `map` operator (`>>|`) which calls the bound closure if the value exists. If no result is found, then the `Yojson.Safe.to_string` conversion function is simply ignored, and a `None` returned.

Executing an HTTP client query

Now that we've written those utility functions, let's look at the Async code that performs the actual search:

```
(* Execute the DuckDuckGo search *)
(* TODO: This client API is being simplified in Cohttp *)
let do_ddg_query query =
  Cohttp_async.Client.call `GET (make_ddg_uri ~query)
  >>= function
  | Some (res, Some body) ->
      let buf = Buffer.create 128 in
      Pipe.iter_without_pushback body ~f:(Buffer.add_string buf)
      >>| fun () ->
      get_definition_from_json (Buffer.contents buf) |!
      Option.value ~default:"???"
  | Some (_, None) | None ->
      failwith "no body in response"
```

For this code, you'll need to OPAM install the `cohttp` library. The `Cohttp_async.Client` module executes the HTTP call, and returns a status and response body wrapped. This whole result is wrapped in a type you haven't seen before: `Async.Deferred.t`.

The `Deferred.t` represents a *future* value whose result is not available yet. You can "wait" for the result by binding a callback using the `>>=` operator (which is imported when you open `Async.Std`). This is the same monad pattern available in other Core libraries such as `Option`, but instead of operating on optional values, we are now mapping over future values. We'll come back to monads later in this chapter. (*avsm*: TODO xref)

The `ddg_query` function invokes the HTTP client call, and returns a tuple containing the response codes and headers, and a `string Pipe.Reader`. Pipes in Async are often used to transmit large amounts of data between two processes or concurrent threads. The `Cohttp` library creates a `Pipe.Writer` which it outputs the HTTP body into, and provides your application with the `Reader` end.

In this case, the HTTP body probably isn't very large, so we just iterate over the Pipe's contents until we have the full HTTP body in a `Buffer.t`. Once the full body has been retrieved into our buffer, the next callback passes it through the JSON parser and returns a human-readable string of the search description that DuckDuckGo gave us.

```
(* Run a single search *)
let run_one_search =
  do_ddg_query "Camel" >>| prerr_endline

(* Start the Async scheduler *)
let _ = Scheduler.go ()
```

Let's actually use the search function to run a real query now. The fragment above spawns a single search, and then fires up the Async scheduler. The scheduler is where all the work happens, and must be started in every application that uses Async. Without it, logging won't be output, nor will blocked functions ever wake up. When the scheduler is active, it is waiting for incoming I/O events and waking up function callbacks that were sleeping on that particular file descriptor or timeout.

A single connection isn't that interesting from a concurrency perspective. Luckily, Async makes it very easy to run multiple parallel searches:

```
(* Run many searches in parallel *)
let run_many_searches =
  let searches = ["Duck"; "Sheep"; "Cow"; "Llama"; "Camel"] in
  Deferred.List.map ~how:`Parallel searches ~f:do_ddg_query >>|
  List.iter ~f:print_endline
```

The `Deferred.List` module lets you specify exactly how to map over a collection of futures. The searches will be executed simultaneously, and the map thread will complete once all of the sub-threads are complete. If you replace the `Parallel` parameter with `Serial`, the map will wait for each search to fully complete before issuing the next one.

Terminating Async applications

When you run the search example, you'll notice that the application doesn't terminate even when all of the searches are complete. The Async scheduler doesn't terminate by default, and so most applications will listen for a signal to exit or simply use `CTRL-C` to interrupt it from a console.

Another alternative is to run an Async function in a separate system thread. You can do this by wrapping the function in the `Async.Thread_safe.block_on_async_exn`. The `utop` top-level does this automatically for you if you attempt to evaluate an Async function interactively.

Manipulating Async threads

Now that we've seen the search example above, let's examine how Async works in more detail.

Async threads are co-operative and never preempt each other, and the library internally converts blocking code into a single event loop. The threads are normal OCaml heap-allocated values (without any runtime magic!) and are therefore very fast to allocate. Concurrency is mostly limited only by your available main memory, or operating system limits on non-memory resources such as file descriptors.

Lets begin by constructing a simple thread. Async follows the Core convention and provides an `Async.Std` that provides threaded variants of many standard library functions. The examples throughout this chapter assume that `Async.Std` is open in your environment.

```
# require "async.unix" ;;
# open Async.Std ;;
# return 5 ;;
- : int Deferred.t = <abstr>
```

The basic type of an Async thread is a `Deferred.t`, which can be constructed by the `return` function. The type parameter (in this case `int`) represents the ultimate type of the thread once it has completed in the future. This return value cannot be used directly while it is wrapped in a `Deferred.t` as it may not be available yet. Instead, we `bind` a function closure that is called once the value is eventually ready.

```
# let x = return 5 ;;
val x : int Deferred.t = <abstr>
# let y = Deferred.bind x (fun a -> return (string_of_int a)) ;;
val y : string Deferred.t = <abstr>
```

Here, we've bound a function to `x` that will convert the `int` to a `string`. Notice that while both `x` and `y` share a common `Deferred.t` type, their type variables differ and so they cannot be interchangably used except in polymorphic functions. This is useful when refactoring large codebases, as you can tell if any function will block simply by the presence of an `Deferred.t` in the signature.

Let's examine the function signatures of `bind` and `return` more closely.

```
# return ;;
- : 'a -> 'a Deferred.t = <fun>
# Deferred.bind ;;
- : 'a Deferred.t -> ('a -> 'b Deferred.t) -> 'b Deferred.t = <fun>
```

`return`, `bind` and the `Deferred.t` type all contain polymorphic type variables (the `'a`) which represent the type of the thread, and are inferred based on how they are used in your code. The `'a` type of the argument passed to the `bind` callback *must* be the same as the `'a Deferred.t` of the input thread, preventing runtime mismatches between thread callbacks. Both `bind` and `return` form a design pattern in functional programming known as *monads*, and you will run across this signature in many applications beyond just threads.

_(avsm: do we talk about Monads earlier in the Core chapter? I presume we do, since the Option monad is very useful)

Binding callbacks is to deferred values is the most common way to compose blocking operations, and inline operators are provided to make it easier to use. In the fragment below, we see `>>=` and `>>|` used in similar ways to convert an integer into a string:

```
# let x = return 5 ;;
val x : int Deferred.t = <abstr>
# x >>= fun y -> return (string_of_int y) ;;
val - : string Deferred.t = <abstr>
# x >>| string_of_int ;;
val - : string Deferred.t = <abstr>
```

The `>>=` operator is exactly the same as `bind` and unpacks the integer future into the `y` variable. The subsequent closure receives the unpacked integer and builds a new string future. It can be a little verbose to keep calling `bind` and `return`, and so the `>>|` operator maps a non-Async function across a future value. In the second example, the future value of `x` is mapped to `string_of_int` directly, and the result is a `string` future.

Async threads can be evaluated from the toplevel by wrapping them in `Thread_safe.block_on_async_exn`, which spawns a system thread that waits until a result is available. The `utop` top-level automatically detects `Deferred.t` types that are entered interactively and wraps them in this function for you automatically.

```
# let fn () = return 5 >>| string_of_int ;;
val fn : unit -> string Deferred.t = <abstr>
# Thread_safe.block_on_async_exn fn ;;
- : string = "5"
# fn () ;;
- : string = "5"
```

In the second evaluation of `fn`, the top-level detected the return type of a future and evaluated the result into a concrete string.

(*avsm*: this utop feature not actually implemented yet for Async, but works for Lwt)

Timing and Thread Composition

Our examples so far have been with static threads, and now we'll look at how to coordinate multiple threads and timeouts. Let's write a program that spawns two threads, each of which sleep for some random time and return either "Heads" or "Tails", and the quickest thread returns its value.

```
# let flip () =
  let span = Time.Span.of_sec 3.0 in
  let span_heads = Time.Span.randomize span ~percent:0.75 in
  let span_tails = Time.Span.randomize span ~percent:0.75 in
  let coin_heads =
    Clock.after span_heads
    >>| fun () ->
    "Heads!", span_heads, span_tails
  in
  let coin_tails =
    Clock.after span_tails
    >>| fun () ->
    "Tails!", span_heads, span_tails
  in
  Deferred.any [coin_heads; coin_tails] ;;
val flip : unit -> (string * Time.Span.t * Time.Span.t) Deferred.t = <fun>
```

This introduces a couple of new time-related Async functions. The `Time` module contains functions to express both absolute and relative temporal relationships. In our coin flipping example, we create a relative time span of 3 seconds, and then permute it randomly twice by 75%. We then create two threads, `coin_heads` and `coin_tails` which return after their respective intervals. Finally, `Deferred.any` waits for the first thread which completes and returns its value, ignoring the remaining undetermined threads.

Both of the threads encode the time intervals in their return value so that you can can easily verify the calculations (you could also simply print the time spans to the console as they are calculated and simplify the return types). You can see this by executing the `flip` function at the toplevel a few times.

```
# Thread_safe.block_on_async_exn flip ;;
# - : string * Time.Span.t * Time.Span.t = ("Heads!", 2.86113s, 3.64635s)
# Thread_safe.block_on_async_exn flip ;;
# - : string * Time.Span.t * Time.Span.t = ("Tails!", 4.44979s, 2.14977s)
```

The `Deferred` module has a number of other ways to select between multiple threads, such as:

Function	# Threads	Behaviour
both	2	Combines both threads into a tuple and returns both values.
any	list	Returns the first thread that becomes determined.
all	list	Waits for all threads to complete and returns their values.
all_unit	list	Waits for all `unit` threads to complete and returns `unit`.
peek	1	Inspects a single thread to see if it is determined yet.

Try modifying the `Deferred.any` in the above example to use some of the other thread joining functions above, such as `Deferred.both`.

Cancellation

A simple TCP Echo Server

Onto an HTTP Server

Binding to the Github API

Show how we can use a monadic style to bind to the Github API and make simple JSON requests/responses.

A Note on Portability

Explain libev and why its needed here.

2013-02-05
19:46:47

PART III

Advanced Topics

2013-02-05
19:46:47

CHAPTER 17

First class modules

_(jyh: I'm going to start some new text on FCM. We might want another chapter, but let's see how it goes. I've kept Ron's original text below.)

OCaml provides several mechanisms for organizing your programs, including modules and functors, files and compilation units, and classes and objects. Files and compilation units (`.ml` and `.mli` files) are really just a simplified module system. Classes and objects are a different form of organization altogether (as we'll see in Chapter 11. Yet, in each of these cases, there is a clear separation between types and values -- values cannot contain types, and types cannot contain values. And since modules can contain types, modules can't be values.

(yminsky: Instead of saying that ml and mli files are a simplified module system, maybe say that they "provide a simple way of creating modules and interfaces", or some such? It's not like there's a simplified module system floating around)

(yminsky: consider dropping "Yet" in the above.)

Next, we'll relax this restriction with *first-class modules*. "First-class" means that modules can be passed around as ordinary values that can be created from and converted back to regular modules. This is a relatively recent addition to the OCaml language, and while it might seem trivial to say, it has profound consequences on the language. First-class modules are strictly more expressive than any other organization mechanism, including classes and objects. Once you use first-class modules, you'll never want to go back.

(yminsky: I wouldn't say they're strictly more expressive. For example, they don't give you a way of expressing sub typing relationships effectively, which objects do.)

This is not say that first-class modules should be used indiscriminately. When you pass modules as values, the reason is to support dynamic behavior, and this can have a negative impact on understandability. As we proceed, we'll compare first-class modules to other techniques, and suggest alternatives when it seems appropriate.

_(jyh: Original text You can think of OCaml as being broken up into two sub-language: a core language that is concerned with values and types, and a module language that is concerned with modules and module signatures. These sub-languages are stratified, in that modules can contain types and values, but ordinary values can't contain modules or module types. That means you can't do things like define a variable whose definition is a module, or a function that takes a module as an argument.

OCaml provides a way around this stratification in the form of *first-class modules*. First-class modules are ordinary values that can be created from and converted back to regular modules. As we'll see, letting modules into the core language makes it possible to use more flexible and dynamic module-oriented designs.)_

Another trivial example

Much as we did with functors, we'll start out with an utterly trivial example, to allow us to show the basic mechanics of first class modules with a minimum of fuss.

A first-class module is created by packaging up a module with a signature that it satisfies. The following defines a simple signature and a module that matches it.

```
# module type X_int = sig val x : int end;;
module type X_int = sig val x : int end
# module Three : X_int = struct let x = 3 end;;
module Three : X_int
# Three.x;;
- : int = 3
```

We can then create a first-class module using the `module` keyword.

```
# let three = (module Three : X_int);;
val three : (module X_int) = <module>
```

Note that the type of the first-class module, `(module X_int)`, is based on the name of the signature that we used in constructing it.

To get at the contents of `three`, we need to unpack it into a module again, which we can do using the `val` keyword.

```
# module New_three = (val three : X_int) ;;
module New_three : X_int
# New_three.x;;
- : int = 3
```

Using these conversions as building blocks, we can create tools for working with first-class modules in a natural way. The following shows the definition of two function, `to_int`, which converts a `(module X_int)` into an `int`. And `plus`, which adds two `(module X_int)`s.

```
# let to_int m =
    let module M = (val m : X_int) in
    M.x
```

```
  ;;
val to_int : (module X_int) -> int = <fun>
# let plus m1 m2 =
    (module struct
       let x = to_int m1 + to_int m2
     end : X_int)
  ;;
val plus : (module X_int) -> (module X_int) -> (module X_int) = <fun>
```

With these functions in hand, we can start operating on our `(module X_int)`'s in a more natural style, taking full advantage of the concision and simplicity of the core language.

```
# let six = plus three three;;
val six : (module X_int) = <module>
# to_int (List.fold ~init:six ~f:plus [three;three]);;
- : int = 12
```

Of course, all we've really done with this example is come up with a more cumbersome way of working with integers. Let's see what happens when with work with more complex abstract types.

Standard vs. first-class modules

(yminsky: I'm not in solve with the example. It feels in some sense too artificial, and that aside, when you get to the end of the example, you haven't really gotten any juice of first-class modules)

(yminsky: using "standard" in quotes seems a little awkward. Maybe just drop the quotes, and talk about standard or ordinary modules directly?)

Let's compare the style of "standard" modules to first-class modules, using a simple library of abstract geometric shapes. In a "standard" module definition, we would define the shapes using abstract data types, where there is a type `t` that defines the actual representation, and the module would include functions that operate on the values of type `t`. In the following code, the module type `Shape` defines the type of generic shape, and the modules `Rectangle` and `Line` implement some concrete shapes.

```
module type Shape = sig
  type t
  val area : t -> int
  val position : t -> int * int
end

module Rectangle = struct
   type t = { width : int; height : int; x : int; y : int }
   let make ~x ~y ~width ~height =
      { width = width; height = height; x = x; y = y }
   let area { width = width; height = height } = width * height
   let position { x = x; y = y } = (x, y)
end
```

```
module Line = struct
   type t = { dx : int; dy : int; x : int; y : int }
   let make ~x ~y ~dx ~dy = { dx = dx; dy = dy; x = x; y = y }
   let area _ = 0
   let position { x = x; y = y } = (x, y)
end
```

Next, if we want to define a generic shape that is either a rectangle or a line, we would probably use a variant type. The following module `Shapes` is entirely boilerplate. We define the variant type, then functions to perform a dynamic dispatch based on the type of object.

```
module Shapes = struct
   type t = [ `Rect of Rectangle.t | `Line of Line.t ]
   let make_rectangle = Rectangle.make
   let make_line = Line.make
   let area = function
      `Rect r -> Rectangle.area r
    | `Line l -> Line.area l
   let position = function
      `Rect r -> Rectangle.position r
    | `Line l -> Line.position l
end;;
```

In fact, confronted with this boilerplate, we would probably choose not use modules at all, but simply define a single module with a variant type and the code for all of the shapes. This isn't to say that separate code for separate shapes is bad, it just means that the language doesn't support it well (at least with standard modules).

With first-class modules, the situation changes, but we have to dispense with the representation type altogether. For immutable shapes, the implementation is now trivial.

```
# module type Shape = sig
    val area : int
    val position : int * int
  end;;
module type Shape = sig val area : int val position : int * int end
# let make_rectangle ~x ~y ~width ~height =
   let module Rectangle = struct
      let area = width * height
      let position = (x, y)
   end in
   (module Rectangle : Shape);;
val make_rectangle :
  x:int -> y:int -> width:int -> height:int -> (module Shape) = <fun>
# let make_line ~x ~y ~dx ~dy =
   let module Line = struct
      let area = 0
      let position = (x, y)
   end in
   (module Line : Shape);;
val make_line : x:int -> y:int -> dx:'a -> dy:'b -> (module Shape) = <fun>
```

For mutable shapes, it isn't much different, but we have to include the state as values in the module implementations. For this, we'll define a representation type `t` in the module implementation, and for rectangles, a value `rect` of that type. The code for lines is similar.

```
# module type Shape = sig
     val area : unit -> int
     val position : unit -> int * int
     val moveby : dx:int -> dy:int -> unit
     val enlargeby : size:int -> unit
  end;;
module type Shape = ...
# let make_rectangle ~x ~y ~width ~height =
    let module Rectangle = struct
      type t = { mutable x : int; mutable y : int;
                 mutable width : int; mutable height : int }
      let rect = { x = x; y = y; width = width; height = height }
      let area () = rect.width * rect.height
      let position () = (rect.x, rect.y)
      let moveby ~dx ~dy =
         rect.x <- rect.x + dx;
         rect.y <- rect.y + dy
      let enlargeby ~size =
         rect.width <- rect.width * size;
         rect.height <- rect.height * size
    end in
    (module Rectangle : Shape);;
val make_rectangle :
  x:int -> y:int -> width:int -> height:int -> (module Shape) = <fun>
```

A more complete example -- containers

So far, we haven't done anything that really needs modules. The type `Shape` could just as well be specified as a record type `type shape = { area : int; position : int * int; ... }`.

To explore the topic more fully, let's implement a system of dynamic containers. OCaml already provides a set of standard containers like `List`, `Set`, `Hashtbl`, etc., but these types have to be selected statically. If a function expects a value of type `Set.Make(Element Type).t`, then you have to pass it a set of exactly that type. What we would like is a kind of container where the container implementation is chosen by the caller. We define an abstract *interface*, as a module type, then define one or more concrete module implementations.

Let's start by defining an abstract container interface. It contains some elements of type `elt`, and functions to examine and iterate through the contents. For convenience, we also define a normal type `'a container` to represent containers with elements of type `'a`.

```
module type Container = sig
```

```
   type elt
   val empty : unit -> bool
   val iter : (elt -> unit) -> unit
   val fold : ('a -> elt -> 'a) -> 'a -> 'a
end;;

type 'a container = (module Container with type elt = 'a)
```

Imperative containers

For imperative containers, will also want functions to mutate the contents by adding or removing elements. For example, a stack can be implemented as a module `Stack` that includes all the functions in the generic `Container` module, as well as functions to push and pop elements.

```
module type Stack = sig
   include Container
   val push : elt -> unit
   val pop : unit -> elt
end;;

type 'a stack = (module Stack with type elt = 'a)
```

Now that the types are defined, the next step is to define a concrete container implementation. For this simple example, we'll use a list to represent a stack. The function `make_list_stack` constructs module implementation using a `let module` construction, then returns the result.

```
# let make_list_stack (type element) () : element stack =
    let module ListStack = struct
      type elt = element
      let contents = ref []
      let empty () = !contents = []
      let iter f = List.iter f !contents
      let fold f x = List.fold_left f x !contents
      let push x = contents := x :: !contents
      let pop () =
         match !contents with
            x :: rest -> contents := rest; x
          | [] -> raise (Invalid_argument "stack is empty")
    end in
    (module ListStack : Stack with type elt = element);;
val make_list_stack : unit -> 'a stack = <fun>
```

Note the use of the explicit type parameter `element`. This is required because the use of a type variable in the module definition (like `type elt = 'a`) would be rejected by the compiler. The construction and use of the stack is straightforward.

```
# let demo (s : int stack) =
    let module S = (val s) in
    S.push 5;
    S.push 17;
    S.iter (fun i -> Printf.printf "Element: %d\n" i);;
val demo : int stack -> unit = <fun>
# demo (make_list_stack ());;
Element: 17
Element: 5
- : unit = ()
```

The `demo` function is entirely oblivious to the implementation of the stack. Instead of passing a module implementation based on lists, we could pass a different implementation based on arrays.

We could go on to define other containers, sets, dictionaries, queues, etc. but the implementations would be similar to what we have seen. Instead, let's look at functional data structures, which require a little more work to express.

Pure functional containers

Imperative data structures have simpler types that functional ones because the return type of imperative functions is just `unit`. When we look at pure functional data structures, we immediately run into a problem with type recursion.

```
# module type Container = sig
    type elt
    val empty : bool
    val iter : (elt -> unit) -> unit
    val fold : ('a -> elt -> 'a) -> 'a -> 'a
    val add : elt -> (module Container)
  end;;
Characters 160-178:
     val add : elt -> (module Container)
                      ^^^^^^^^^^^^^^^^^^
Error: Unbound module type Container
```

The problem here is that module type definitions are not recursive -- we can't use the type being defined in its own definition.

Recursive modules provide a solution, but it requires a "trick", where we define a module that is equal to itself. This module contains only type definitions, and the only purpose of the outer recursive module is to allow the recursion in the definition. While we're at it, let's include a `map` function with the usual semantics.

```
module rec Container : sig
   module type T = sig
      type elt
      val empty : bool
      val iter : (elt -> unit) -> unit
      val fold : ('a -> elt -> 'a) -> 'a -> 'a
      val map : (elt -> 'a) -> 'a Container.t
      val add : elt -> elt Container.t
   end
```

```
    type 'a t = (module Container.T with type elt = 'a)
end = Container;;
```

There are several ways to write this model, but this definition is convenient because it defines both a module type `Container.T` and a value type `'a Container.t`. The outer recursive module `Container` allows the module type `T` to refer to the value type `t` and *vice versa*. Note that the module `Container` is defined as itself (as `Container`).

With this first technicality out of the way, the next one is how to construct values of type `Container.t`. In the imperative version of the stack, we used a function `make_list_stack`. We want to do the same here, but the function definition must be both recursive and polymorphic.

```
# let make_stack () =
    let rec make : 'a. 'a list -> 'a Container.t = fun
      (type element) (contents : element list) ->
      let module NewList = struct
         type elt = element
         let empty = contents = []
         let iter f = List.iter f contents
         let fold f x = List.fold_left f x contents
         let map f = make (List.map f contents)
         let add x = make (x :: contents)
      end in
      (module NewList : Container.T with type elt = element)
   in
   make [];;
val make_stack : unit -> 'a Container.t = <fun>
```

The recursion here is particularly important. The functions `map` and `add` return new collections, so they call the function `make` recursively. The explicit polymorphic type `make : 'a. 'a list -> 'a Container.t` means that the function `make` is properly polymorphic, so that the `map` function is polymorphic.

Now that the construction is done, the usage is similar to the imperative case, except that now the data structure is functional.

```
# let demo (s : int Container.t) =
    let module S = (val s) in
    let module S = (val (S.add 5)) in
    let module S = (val (S.add 17)) in
    S.iter (fun i -> Printf.printf "Int Element: %d\n" i);
    let s = S.map (fun i -> float_of_int i +. 0.1) in
    let module S = (val s) in
    S.iter (fun x -> Printf.printf "Float Element: %f\n" x);
    s;;
val demo : int Container.t -> float Container.t = <fun>
# demo (make_stack ());;
Int Element: 17
Int Element: 5
Float Element: 17.100000
Float Element: 5.100000
- : unit = ()
```

The syntactic load here is pretty high, requiring a `let module` expression to name every intermediate value. First-class modules are fairly new to the language, and this is likely to change, but in the meantime the syntactic load can be pretty daunting.

Let's look a some other more typical examples, where dynamic module selection is more localized.

_(jyh: This is a rough draft, I'm not sure about the ordering and the topics, yet. Switching back to Ron's text now.)

Dynamically choosing a module

Perhaps the simplest thing you can do with first-class modules that you can't do without them is to pick the implementation of a module at runtime.

Consider an application that does I/O multiplexing using a system call like `select` to determine which file descriptors are ready to use. There are in fact multiple APIs you might want to use, including `select` itself, `epoll`, and `libev`, where different multiplexers make somewhat different performance and portability trade-offs. You could support all of these in one application by defining a single module, let's call it `Mutliplexer`, whose implementation is chosen at run-time based on an environment variable.

To do this, you'd first need an interface `S` that all of the different multiplexer implementations would need to match, and then an implementation of each multiplexer.

```
(* file: multiplexer.ml *)

(* An interface the OS-specific functionality *)
module type S = sig ... end

(* The implementations of each individual multiplexer *)
module Select : S = struct ... end
module Epoll  : S = struct ... end
module Libev  : S = struct ... end
```

We can choose the first-class module that we want based on looking up an environment variable.

```
let multiplexer =
  match Sys.getenv "MULTIPLEXER" with
  | None
  | Some "select" -> (module Select : S)
  | Some "epoll"  -> (module Epoll : S)
  | Some "libev"  -> (module Libev : S)
  | Some other -> failwithf "Unknown multiplexer: %s" other ()
```

Finally, we can convert the resulting first-class module back to an ordinary module, and then include that so it becomes part of the body of our module.

```
(* The final, dynamically chosen, implementation *)
include (val multiplexer : S)
```

Example: A service bundle

This section describes the design of a library for bundling together multiple services, where a service is a piece of code that exports a query interface. A service bundle combines together multiple individual services under a single query interface that works by dispatching incoming queries to the appropriate underlying service.

The following is a first attempt at an interface for our `Service` module, which contains both a module type `S`, which is the interface that a service should meet, as well as a `Bundle` module which is for combining multiple services.

```
(* file: service.mli *)

open Core.Std

(** The module type for a service. *)
module type S = sig
  type t
  val name           : string
  val create         : unit -> t
  val handle_request : t -> Sexp.t -> Sexp.t Or_error.t
end

(** Bundles multiple services together *)
module Bundle : sig
  type t
  val create : (module S) list -> t
  val handle_request : t -> Sexp.t -> Sexp.t Or_error.t
  val service_names  : t -> string list
end
```

Here, a service has a state, represented by the type `t`, a name by which the service can be referenced, a function `create` for instantiating a service, and a function by which a service can actually handle a request. Here, requests and responses are delivered as s-expressions. At the `Bundle` level, the s-expression of a request is expected to be formatted as follows:

```
(<service-name> <body>)
```

where `<service_name>` is the service that should handle the request, and `<body>` is the body of the request.

Now let's look at how to implement `Service`. The core datastructure of `Bundle` is a hashtable of request handlers, one per service. Each request handler is a function of type `(Sexp.t -> Sexp.t Or_error.t)`. These request handlers really stand in for the underlying service, with the particular state of the service in question being hidden inside of the request handler.

The first part of `service.ml` is just the preliminaries: the definition of the module type `S`, and the definition of the type `Bundle.t`.

```
(* file: service.ml *)

open Core.Std

module type S = sig
  type t
  val name           : string
  val create         : unit -> t
  val handle_request : t -> Sexp.t -> Sexp.t Or_error.t
end

module Bundle = struct
  type t = { handlers: (Sexp.t -> Sexp.t Or_error.t) String.Table.t; }
```

The next thing we need is a function for creating a `Bundle.t`. This `create` function builds a table to hold the request handlers, and then iterates through the services, unpacking each module, constructing the request handler, and then putting that request handler in the table.

```
  (** Creates a handler given a list of services *)
  let create services =
    let handlers = String.Table.create () in
    List.iter services ~f:(fun service_m ->
      let module Service = (val service_m : S) in
      let service = Service.create () in
      if Hashtbl.mem handlers Service.name then
        failwith ("Attempt to register duplicate handler for "^Service.name);
      Hashtbl.replace handlers ~key:Service.name
        ~data:(fun sexp -> Service.handle_request service sexp)
    );
    {handlers}
```

Note that the `Service.t` that is created is referenced by the corresponding request handler, so that it is effectively hidden behind the function in the `handlers` table.

Now we can write the function for the bundle to handle requests. The handler will examine the s-expression to determine the body of the query and the name of the service to dispatch to. It then looks up the handler calls it to generate the response.

```
  let handle_request t sexp =
    match sexp with
    | Sexp.List [Sexp.Atom name;query] ->
```

```
      begin match Hashtbl.find t.handlers name with
      | None -> Or_error.error_string ("Unknown service: "^name)
      | Some handler ->
        try handler query
        with exn -> Error (Error.of_exn exn)
      end
    | _ -> Or_error.error_string "Malformed query"
```

Last of all, we define a function for looking up the names of the available services.

```
  let service_names t = Hashtbl.keys t.handlers

end
```

To see this system in action, we need to define some services, create the corresponding bundle, and then hook that bundle up to some kind of client. For simplicity, we'll build a simple command-line interface. There are two functions below: `handle_one`, which handles a single interaction; and `handle_loop`, which creates the bundle and then runs `handle_one` in a loop.

```
(* file: service_client.ml *)

open Core.Std

(** Handles a single request coming from stdin *)
let handle_one bundle =
  printf ">>> %!"; (* prompt *)
  match In_channel.input_line stdin with
  | None -> `Stop (* terminate on end-of-stream, so Ctrl-D will exit *)
  | Some line ->
    let line = String.strip line in (* drop leading and trailing whitespace *)
    if line = "" then `Continue
    else match Or_error.try_with (fun () -> Sexp.of_string line) with
    | Error err ->
      eprintf "Couldn't parse query: %s\n%!" (Error.to_string_hum err);
      `Continue
    | Ok query_sexp ->
      let resp = Service.Bundle.handle_request bundle query_sexp in
      Sexp.output_hum stdout (<:sexp_of<Sexp.t Or_error.t>> resp);
      Out_channel.newline stdout;
      `Continue

let handle_loop services =
  let bundle = Service.Bundle.create services in
  let rec loop () =
    match handle_one bundle with
    | `Stop -> ()
    | `Continue -> loop ()
  in
  loop ()
```

Now we'll create a couple of toy services. One service is a counter that can be updated by query; and the other service lists a directory. The last line then kicks off the shell with the services we've defined.

```
module Counter : Service.S = struct
  type t = int ref

  let name = "update-counter"
  let create () = ref 0

  let handle_request t sexp =
    match Or_error.try_with (fun () -> int_of_sexp sexp) with
    | Error _ as err -> err
    | Ok x ->
      t := !t + x;
      Ok (sexp_of_int !t)
end

module List_dir : Service.S = struct
  type t = unit

  let name = "ls"
  let create () = ()

  let handle_request () sexp =
    match Or_error.try_with (fun () -> string_of_sexp sexp) with
    | Error _ as err -> err
    | Ok dir -> Ok (Array.sexp_of_t String.sexp_of_t (Sys.readdir dir))
end

let () =
  handle_loop [(module List_dir : Service.S); (module Counter : Service.S)]
```

And now we can go ahead and start up the client.

```
$ ./service_client.byte
>>> (update-counter 1)
(Ok 1)
>>> (update-counter 10)
(Ok 11)
>>> (ls .)
(Ok
 (_build _tags service.ml service.mli service.mli~ service.ml~
  service_client.byte service_client.ml service_client.ml~))
>>>
```

Now, let's consider what happens to the design when we want to make the interface of a service a bit more realistic. In particular, right now services are created without any configuration. Let's add a config type to each service, and change the interface of `Bundle` so that services can be registered along with their configs. At the same time, we'll change the `Bundle` API to allow services to be changed dynamically, rather than just added at creation time.

2013-02-05
19:46:47

CHAPTER 18

Understanding the runtime system

Much of the static type information contained within an OCaml program is checked and discarded at compilation time, leaving a much simpler *runtime* representation for values. Understanding this difference is important for writing efficient programs, and also for interfacing with C libraries that work directly with the runtime system.

Why do OCaml types disappear at runtime?

The OCaml compiler runs through several phases of during the compilation process. After syntax checking, the next stage is *type checking*. In a validly typed program, a function cannot be applied with an unexpected type. For example, the `print_endline` function must receive a single `string` argument, and an `int` will result in a type error.

Since OCaml verifies these properties at compile time, it doesn't need to keep track of as much information at runtime. Thus, later stages of the compiler can discard and simplify the type declarations to a much more minimal subset that's actually required to distinguish polymorphic values at runtime. This is a major performance win versus something like a Java or .NET method call, where the runtime must look up the concrete instance of the object and dispatch the method call. Those languages amortize some of the cost via "Just-in-Time" dynamic patching, but OCaml prefers runtime simplicity instead.

Let's start by explaining the memory layout, and then move onto the details of how C bindings work.

The garbage collector

A running OCaml program uses blocks of memory (i.e. contiguous sequences of words in RAM) to represent many of the values that it deals with such as tuples, records, closures or arrays. An OCaml program implicitly allocates a block of memory when such a value is created.

```
# let x = { foo = 13; bar = 14 } ;;
```

An expression such as the record above requires a new block of memory with two words of available space. One word holds the `foo` field and the second word holds the `bar` field. The OCaml compiler translates such an expression into an explicit allocation for the block from OCaml's runtime system: a C library that provides a collection of routines that can be called by running OCaml programs. The runtime system manages a *heap*, which a collection of memory regions it obtains from the operating system using *malloc(3)*. The OCaml runtime uses these memory regions to hold *heap blocks*, which it then fills up in response to allocation requests by the OCaml program.

When there is'nt enough memory available to satisfy an allocation request from the allocated heap blocks, the runtime system invokes the *garbage collector* (or GC). An OCaml program does not explicitly free a heap block when it is done with it, and the GC must determine which heap blocks are "alive" and which heap blocks are *dead*, i.e. no longer in use. Dead blocks are collected and their memory made available for re-use by the application.

The garbage collector does not keep constant track of blocks as they are allocated and used. Instead, it regularly scans blocks by starting from a set of *roots*, which are values that the application always has access to (such as the stack). The GC maintains a directed graph in which heap blocks are nodes, and there is an edge from heap block `b1` to heap block `b2` if some field of `b1` points to `b2`. All blocks reachable from the roots by following edges in the graph must be retained, and unreachable blocks can be reused.

With the typical OCaml programming style, many small blocks are frequently allocated, used for a short period of time, and then never used again. OCaml takes advantage of this fact to improve the performance of allocation and collection by using a *generational* garbage collector. This means that it has different memory regions to hold blocks based on how long the blocks have been alive. OCaml's heap is split in two; there is a small, fixed-size *minor heap* used for initially allocating most blocks, and a large, variable-sized *major heap* for holding blocks that have been alive longer or are larger than 4KB. A typical functional programming style means that young blocks tend to die young, and old blocks tend to stay around for longer than young ones (this is referred to as the *generational hypothesis*). To reflect this, OCaml uses different memory layouts and garbage collection algorithms for the major and minor heaps.

The fast minor heap

The minor heap is one contiguous chunk of memory containing a sequence of heap blocks that have been allocated. If there is space, allocating a new block is a fast constant-time operation in which the pointer to the end of the heap is incremented by the desired size. To garbage collect the minor heap, OCaml uses *copying collection* to copy all live blocks in the minor heap to the major heap. This only takes work proportional to the number of live blocks in the minor heap, which is typically small according to the generational hypothesis.

One complexity of generational collection is that in order to know which blocks in the minor heap are live, the collector must know which minor-heap blocks are directly pointed to by major-heap blocks. To do this, OCaml maintains a set of such intergenerational pointers, and, through cooperation with the compiler, uses a write barrier to update this set whenever a major-heap block is modified to point at a minor-heap block.

The long-lived major heap

The major heap consists of a number of chunks of memory, each containing live blocks interspersed with regions of free memory. The runtime system maintains a free list data structure that indexes all the free memory, and this list is used to satisfy allocation requests. OCaml uses mark and sweep garbage collection for the major heap. The *mark* phase to traverses the block graph and marks all live blocks by setting a bit in the color tag of the block header. (*avsm*: we only explain the color tag in the next section, so rephrase or xref).

The *sweep* phase sequentially scans all heap memory and identifies dead blocks that weren't marked earlier. The *compact* phase relocates live blocks to eliminate the gaps of free memory between them and ensure memory does not fragment.

A garbage collection must *stop the world* (that is, halt the application) in order to ensure that blocks can be safely moved. The mark and sweep phases run incrementally over slices of memory, and are broken up into a number of steps that are interspersed with the running OCaml program. Only a compaction touches all the memory in one go, and is a relatively rare operation.

The `Gc` module lets you control all these parameters from your application, and we will discuss garbage collection tuning in (*avsm*: crossref).

The representation of values

Every OCaml *value* is a single word that is either an integer or a pointer. If the lowest bit of the word is non-zero, the value is an unboxed integer. Several OCaml types map onto this integer representation, including `bool`, `int`, the empty list, `unit`, and variants without constructors. Integers are the only unboxed runtime values in OCaml, and are the cheapest values to allocate.

If the lowest bit of the `value` is zero, then the value is a pointer. A pointer value is stored unmodified, since pointers are guaranteed to be word-aligned and the bottom bits are always zero. If the pointer is inside an area managed by the OCaml runtime, it is assumed to point to an OCaml *block*. If it points outside the OCaml runtime area, it is is treated as an opaque C pointer to some other system resource.

Blocks and values

An OCaml *block* is the basic unit of allocation on the heap. A block consists of a one-word header (either 32- or 64-bits) followed by variable-length data, which is either opaque bytes or *fields*. The collector never inspects opaque bytes, but fields are valid OCaml values. The runtime always inspects fields, and follows them as part of the garbage collection process described earlier. Every block header has a multipurpose tag byte that defines whether to interprete the subsequent data as opaque or OCaml fields.

(*avsm*: pointers to blocks actually point 4/8 bytes into it, for some efficiency reason that I cannot recall right now).

```
+-----------------------+-------+----------+----------+----------+----
| size of block in words |  col  | tag byte | value[0] | value[1] | ...
+-----------------------+-------+----------+----------+----------+----
 <-either 22 or 54 bits-> <2 bit> <--8 bit-->
```

The size field records the length of the block in memory words. Note that it is limited to 22-bits on 32-bit platforms, which is the reason why OCaml strings are limited to 16MB on that architecture. If you need bigger strings, either switch to a 64-bit host, or use the `Bigarray` module (*avsm*: xref). The 2-bit color field is used by the garbage collector to keep track of its status, and is not exposed directly to OCaml programs.

Tag Color	Block Status
blue	on the free list and not currently in use
white	not reached yet, but possibly reachable
gray	reachable, but its fields have not been scanned
black	reachable, and its fields have been scanned

A block's tag byte is multi-purpose, and indicates whether the data array represents opaque bytes or fields. If a block's tag is greater than or equal to `No_scan_tag` (251), then the block's data are all opaque bytes, and are not scanned by the collector. The most common such block is the `string` type, which we describe more below.

(*avsm*: too much info here) If the header is zero, then the object has been forwarded as part of minor collection, and the first field points to the new location. Also, if the block is on the `oldify_todo_list`, part of the minor gc, then the second field points to the next entry on the oldify_todo_list.

The exact representation of values inside a block depends on their OCaml type. They are summarised in the table below, and then we'll examine some of them in greater detail.

OCaml Value	Representation
any `int` or `char`	directly as a value, shifted left by 1 bit, with the least significant bit set to 1

OCaml Value	Representation
`unit`, `[]`, `false`	as OCaml `int` 0.
`true`	as OCaml `int` 1.
`Foo | Bar`	as ascending OCaml `ints`, starting from 0.
`Foo | Bar of int`	variants with parameters are boxed, while entries with no parameters are unboxed (see below).
polymorphic variants	variable space usage depending on the number of parameters (see below).
floating point number	as a block with a single field containing the double-precision float.
string	word-aligned byte arrays that are also directly compatible with C strings.
`[1; 2; 3]`	as `1::2::3::[]` where `[]` is an int, and `h::t` a block with tag 0 and two parameters.
tuples, records and arrays	an array of values. Arrays can be variable size, but structs and tuples are fixed size.
records or arrays, all float	special tag for unboxed arrays of floats. Doesn't apply to tuples.

Integers, characters and other basic types

Many basic types are stored directly as unboxed values at runtime. The native `int` type is the most obvious, although it drops a single bit of precision due to the tag bit described earlier. Other atomic types such as the `unit` and empty list `[]` value are stored as constant integers. Boolean values have a value of `0` and `1` for `true` and `false` respectively.

Why are OCaml integers missing a bit?

Since the lowest bit of an OCaml value is reserved, native OCaml integers have a maximum allowable length of 31- or 63-bits, depending on the host architecture. The rationale for reserving the lowest bit is for efficiency. Pointers always point to word-aligned addresses, and so their lower bits are normally zero. By setting the lower bit to a non-zero value for integers, the garbage collector can simply iterate over every header tag to distinguish integers from pointers. This reduces the garbage collection overhead on the overall program.

(*avsm*: explain that integer manipulation is almost as fast due to isa quirks)

Tuples, records and arrays

```
+---------+----------+----------- - - - -
| header  | value[0] | value[1] | ....
+---------+----------+----------+- - - - -
```

Tuples, records and arrays are all represented identically at runtime, with a block with tag `0`. Tuples and records have constant sizes determined at compile-time, whereas arrays can be of variable length. While arrays are restricted to containing a single type of element in the OCaml type system, this is not required by the memory representation.

You can check the difference between a block and a direct integer yourself using the `Obj` module, which exposes the internal representation of values to OCaml code.

```
# Obj.is_block (Obj.repr (1,2,3)) ;;
- : bool = true
# Obj.is_block (Obj.repr 1) ;;
- : bool = false
```

The `Obj.repr` function retrieves the runtime representation of any OCaml value. `Obj.is_block` checks the bottom bit to determine if the value is a block header or an unboxed integer.

Floating point numbers and arrays

Floating point numbers in OCaml are always stored as full double-precision values. Individual floating point values are stored as a block with a single field that contains the number. This block has the `Double_tag` set which signals to the collector that the floating point value is not to be scanned.

```
# Obj.tag (Obj.repr 1.0) = Obj.double_tag ;;
- : int = 253
# Obj.double_tag ;;
- : int = 253
```

Since each floating-point value is boxed in a separate memory block, it can be inefficient to handle large arrays of floats in comparison to unboxed integers. OCaml therefore special-cases records or arrays that contain *only* `float` types. These are stored in a block that contains the floats packed directly in the data section, with the `Double_array_tag` set to signal to the collector that the contents are not OCaml values.

```
+---------+----------+----------- - - - -
| header  | float[0] | float[1] | ....
+---------+----------+----------+- - - - -
```

You can test this for yourself using the `Obj.tag` function to check that the allocated block has the expected runtime tag, and `Obj.double_field` to retrieve a float from within the block.

```
# open Obj ;;
# tag (repr [| 1.0; 2.0; 3.0 |]) ;;
- : int = 254
# tag (repr (1.0, 2.0, 3.0) ) ;;
- : int = 0
# double_field (repr [| 1.1; 2.2; 3.3 |] ) 1 ;;
- : float = 2.2
```

```
# Obj.double_field (Obj.repr 1.234) 0;;
- : float = 1.234
```

Notice that float tuples are *not* optimized in the same way as float records or arrays, and so they have the usual tuple tag value of `0`. Only records and arrays can have the array optimization, and only if every single field is a float.

Variants and lists

Basic variant types with no extra parameters for any of their branches are simply stored as an OCaml integer, starting with `0` for the first option and in ascending order.

```
# open Obj ;;
# type t = Apple | Orange | Pear ;;
type t = Apple | Orange | Pear
# ((magic (repr Apple)) : int) ;;
- : int = 0
# ((magic (repr Pear)) : int) ;;
- : int = 2
# is_block (repr Apple) ;;
- : bool = false
```

`Obj.magic` unsafely forces a type cast between any two OCaml types; in this example the `int` type hint retrieves the runtime integer value. The `Obj.is_block` confirms that the value isn't a more complex block, but just an OCaml `int`.

Variants that have parameters arguments are a little more complex. They are stored as blocks, with the value *tags* ascending from 0 (counting from leftmost variants with parameters). The parameters are stored as words in the block.

```
# type t = Apple | Orange of int | Pear of string | Kiwi ;;
type t = Apple | Orange of int | Pear of string | Kiwi
# is_block (repr (Orange 1234)) ;;
- : bool = true
# tag (repr (Orange 1234)) ;;
- : int = 0
# tag (repr (Pear "xyz")) ;;
- : int = 1
# (magic (field (repr (Orange 1234)) 0) : int) ;;
- : int = 1234
(magic (field (repr (Pear "xyz")) 0) : string) ;;
- : string = "xyz"
```

In the above example, the `Apple` and `Kiwi` values are still stored as normal OCaml integers with values `0` and `1` respectively. The `Orange` and `Pear` values both have parameters, and are stored as blocks whose tags ascend from `0` (and so `Pear` has a tag of `1`, as the use of `Obj.tag` verifies). Finally, the parameters are fields which contain OCaml values within the block, and `Obj.field` can be used to retrieve them.

Lists are stored with a representation that is exactly the same as if the list was written as a variant type with `Head` and `Cons`. The empty list `[]` is an integer `0`, and subsequent blocks have tag `0` and two parameters: a block with the current value, and a pointer to the rest of the list.

`Obj` module considered harmful

The `Obj` module is an undocumented module that exposes the internals of the OCaml compiler and runtime. It is very useful for examining and understanding how your code will behave at runtime, but should *never* be used for production code unless you understand the implications. The module bypasses the OCaml type system, making memory corruption and segmentation faults possible.

Some theorem provers such as Coq do output code which uses `Obj` internally, but the external module signatures never expose it. Unless you too have a machine proof of correctness to accompany your use of `Obj`, stay away from it except for debugging!

Due to this encoding, there is a limit around 240 variants with parameters that applies to each type definition, but the only limit on the number of variants without parameters is the size of the native integer (either 31- or 63-bits). This limit arises because of the size of the tag byte, and that some of the high numbered tags are reserved.

Polymorphic variants

Polymorphic variants are more flexible than normal variants when writing code, but can be less efficient at runtime. This is because there isn't as much static compile-time information available to optimise their memory layout. This isn't always the case, however. A polymorphic variant without any parameters is stored as an unboxed integer and so only takes up one word of memory. Unlike normal variants, the integer value is determined by apply a hash function to the *name* of the variant. The hash function isn't exposed directly by the compiler, but the `type_conv` library from Core provides an alternative implementation.

```
# #require "type_conv" ;;
# Pa_type_conv.hash_variant "Foo" ;;
- : int = 3505894
# (Obj.magic (Obj.repr `Foo) : int) ;;
- : int = 3505894
```

The hash function is designed to give the same results on 32-bit and 64-bit architectures, so the memory representation is stable across different CPUs and host types.

Polymorphic variants use more memory space when parameters are included in the datatype constructors. Normal variants use the tag byte to encode the variant value, but this byte is insufficient to encode the hashed value for polymoprhic variants. Therefore, they must allocate a new block (with tag `0`) and store the value in there instead. This means that polymorphic variants with constructors use one word of memory more than normal variant constructors.

Another inefficiency is when a polymorphic variant constructor has more than one parameter. Normal variants hold parameters as a single flat block with multiple fields for each entry, but polymorphic variants must adopt a more flexible uniform memory representation since they may be re-used in a different context. They allocate a tuple block for the parameters that is pointed to from the argument field of the variant. Thus, there are three additional words for such variants, along with an extra memory indirection due to the tuple.

String values

Strings are standard OCaml blocks with the header size defining the size of the string in machine words. The `String_tag` (252) is higher than the `No_scan_tag`, indicating that the contents of the block are opaque to the collector. The block contents are the contents of the string, with padding bytes to align the block on a word boundary.

```
+---------------+----------------+--------+-----------+
| header        | 'a' 'b' 'c' 'd' 'e' 'f' | '\O' '\1' |
+---------------+----------------+--------+-----------+
                L data                    L padding
```

On a 32-bit machine, the padding is calculated based on the modulo of the string length and word size to ensure the result is word-aligned. A 64-bit machine extends the potential padding up to 7 bytes instead of 3.

String length mod 4	Padding
0	`00 00 00 03`
1	`00 00 02`
2	`00 01`
3	`00`

This string representation is a clever way to ensure that the string contents are always zero-terminated by the padding word, and still compute its length efficiently without scanning the whole string. The following formula is used:

```
number_of_words_in_block * sizeof(word) - last_byte_of_block - 1
```

The guaranteed NULL-termination comes in handy when passing a string to C, but is not relied upon to compute the length from OCaml code. Thus, OCaml strings can contain null bytes at any point within the string, but care should be taken that any C library functions can also cope with this.

Custom heap blocks

OCaml supports *custom* heap blocks via a `Custom_tag` that let the runtime perform user-defined operations over OCaml values. A custom block lives in the OCaml heap like an ordinary block and can be of whatever size the user desires. The `Custom_tag` (255) is higher than `No_scan_tag` and so cannot contain any OCaml values.

The first word of the data within the custom block is a C pointer to a `struct` of custom operations. The custom block cannot have pointers to OCaml blocks and is opaque to the garbage collector.

```
struct custom_operations {
  char *identifier;
  void (*finalize)(value v);
  int (*compare)(value v1, value v2);
  intnat (*hash)(value v);
  void (*serialize)(value v,
                    /*out*/ uintnat * wsize_32 /*size in bytes*/,
                    /*out*/ uintnat * wsize_64 /*size in bytes*/);
  uintnat (*deserialize)(void * dst);
  int (*compare_ext)(value v1, value v2);
};
```

The custom operations specify how the runtime should perform polymorphic comparison, hashing and binary marshalling. They also optionally contain a finalizer, which the runtime will call just before the block is garbage collected. This finalizer has nothing to do with ordinary OCaml finalizers, as created by `Gc.finalise`. (*avsm*: xref to GC module explanation)

When a custom block is allocated, you can also specify the proportion of "extra-heap resources" consumed by the block, which will affect the garbage collector's decision as to how much work to do in the next major slice. (*avsm*: elaborate on this or move to the C interface section)

Interfacing with C

Now that you understand the runtime structure of the garbage collector, interfacing with C libraries is actually pretty simple. OCaml defines an `external` keyword that maps OCaml functions to a C symbol. That C function will be passed the arguments with the C `value` type which corresponds to the memory layout for OCaml values described earlier.

Getting started with a "Hello World" C binding

Let's define a simple "Hello World" C binding to see how this works. First create a `hello.ml` that contains the external declaration:

```
external hello_world: unit -> unit = "caml_hello_world"
let _ = hello_world ()
```

If you try to compile this module now, you should receive a linker error:

```
$ ocamlopt -o hello hello.ml
Undefined symbols for architecture x86_64:
  "_caml_hello_world", referenced from:
      .L100 in hello.o
      _camlHello in hello.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
File "caml_startup", line 1:
Error: Error during linking
```

This is the system linker telling you that there is a missing `caml_hello_world` symbol that must be provided before a binary can be linked. Now create a file called `hello_stubs.c` which contains the C function.

```
#include <stdio.h>
#include <caml/mlvalues.h>

CAMLprim value
caml_hello_world(value v_unit)
{
  printf("Hello OCaml World!\n");
  return Val_unit;
}
```

Now attempt to recompile the `hello` binary with the C file also included in the compiler invocation, and it should succeed:

```
$ ocamlopt -o hello hello.ml hello_stubs.c
$ ./hello
Hello OCaml World!
```

The compiler uses the file extensions to determine how to compile each file. In the case of the `.c` extension, it passes it to the system C compiler and appends an include directory containing the OCaml runtime header files that define conversion functions to-and-from OCaml values.

The `mlvalues.h` header is the basic header that all C bindings need. Locate it in your system by using `ocamlc -where` to find your system OCaml installation. It defines a few important typedefs early on that should be familiar after the earlier explanations:

```
typedef intnat value;

#define Is_long(x)   (((x) & 1) != 0)
#define Is_block(x)  (((x) & 1) == 0)

#define Val_unit Val_int(0)
```

The `value` typedef is a word that can either be an integer if `Is_long` is true, or a heap block if `Is_block` is true. Our C function definition of `caml_hello_world` accepts a single parameter, and returns a `value`. In our simple example, all the types of parameters and returns are `unit`, and so we use the `Val_unit` macro to construct the return value.

You must be *very* careful that the value you return from the C function corresponds exactly to the memory representation of the types you declared earlier in the `external` declaration of the ML file, or else heap carnage and corruption will ensure.

Activating the debug runtime

Despite your best efforts, it is easy to introduce a bug into C bindings that cause heap invariants to be violated. OCaml includes a variant of the runtime library that is compiled with debugging symbols, and includes regular memory integrity checks upon every garbage collection. Running these often will abort the program near the point of corruption and helps track it down quickly.

To use this, just recompile with `-runtime-variant d` set:

```
$ ocamlopt -runtime-variant d -verbose -o hello hello.ml hello_stubs.c
$ ./hello
### OCaml runtime: debug mode ###
Initial minor heap size: 2048k bytes
Initial major heap size: 992k bytes
Initial space overhead: 80%
Initial max overhead: 500%
Initial heap increment: 992k bytes
Initial allocation policy: 0
Hello OCaml World!
```

2013-02-05
19:46:47

CHAPTER 19

Object Subtyping and Inheritance

Subtyping

Subtyping is a central concept in object-oriented programming. It governs when an object with one type *A* can be used in an expression that expects an object of another type *B*. When this is true, we say that *A* is a *subtype* of *B*. Actually, more concretely, subtyping determines when the coercion operator `e :> t` can be applied. This coercion works only if the expression `e` has some type `s` and `s` is a subtype of `t`.

To explore this, let's define some simple classes for geometric shapes. The generic type `shape` has a method to compute the area, and a `square` is a specific kind of shape.

```
type shape = < area : float >;;

class square w =
object (self : 'self)
  method area = self#width *. self#width
  method width = w
end;;
```

A `square` has a method `area` just like a `shape`, and an additional method `width`. Still, we expect a `square` to be a `shape`, and it is. The coercion `:>` must be explicit.

```
# let new_square x : shape = new square x;;
Characters 27-39:
  let new_square x : shape = new square x;;
                             ^^^^^^^^^^^^
Error: This expression has type square but an expression was expected of type shape
The second object type has no method width
# let new_square x : shape = (new square x :> shape);;
val new_square : float -> shape = <fun>
```

What are the rules for subtyping? In general, object subtyping has two general forms, called *width* and *depth* subtyping. Width subtyping means that an object type *A* is a a subtype of *B*, if *A* has all of the methods of *B*, and possibly more. A `square` is a subtype of `shape` because it implements all of the methods of `shape` (the `area` method).

The subtyping rules are purely technical, they have no relation to object semantics. We can define a class `rectangle` that has all of the methods of a `square`, so it is a subtype of square and can be used wherever a `square` is expected.

```
# class rectangle h w =
  object (self : 'self)
     inherit square w
     method area = self#width *. self#height
     method height = h
  end;;
# let square_rectangle h w : square = (new rectangle h w :> square);;
val square_rectangle : float -> float -> square = <fun>
```

This may seem absurd, but this concept is expressible in all object-oriented languages. The contradiction is semantic -- we know that in the real world, not all rectangles are squares; but in the programming world, rectangles have all of the features of squares (according to our definition), so they can be used just like squares. Suffice it to say that it is usually better to avoid such apparent contradictions.

Next, let's take a seemingly tiny step forward, and start building collections of shapes. It is easy enough to define a `slist` of squares.

```
# let squares =
     let l = SList.make () in
     l#insert (new square 1.0);
     l#insert (new square 2.0);
     l;;
val squares : square slist = <obj>
```

We can also define a function to calculate the total area of a list of shapes. There is no reason to restrict this to squares, it should work for any list of shapes with type `shape slist`. The problem is that doing so raises some serious typing questions -- can a `square slist` be passed to a function that expects a `shape slist`? If we try it, the compiler produces a verbose error message.

```
# let total_area (l : shape slist) : float =
     let total = ref 0.0 in
     let it = l#iterator in
     while it#has_value do
        total := !total +. it#get#area;
        it#next
     done;
     !total;;
val total_area : shape slist -> float = <fun>
# total_area squares;;
Characters 11-18:
  total_area squares;;
             ^^^^^^^
Error: This expression has type
         square slist =
           < insert : square -> unit; is_empty : bool;
```

```
      iterator : square iterator >
but an expression was expected of type
  shape slist =
    < insert : shape -> unit; is_empty : bool;
      iterator : shape iterator >
Type square = < area : float; width : float >
is not compatible with type shape = < area : float >
The second object type has no method width
```

It might seem tempting to give up at this point, especially because the subtyping is not even true -- the type `square slist` is not a subtype of `shape slist`. The problem is with the `insert` method. For `shape slist`, the `insert` method takes an arbitrary `shape` and inserts it into the list. So if we could coerce a `square slist` to a `shape slist`, then it would be possible to insert an arbitrary shape into the list, which would be an error.

Using more precise types to address subtyping problems

Still, the `total_area` function should be fine, in principle. It doesn't call `insert`, so it isn't making that error. To make it work, we need to use a more precise type that indicates we are not going to be mutating the list. We define a type `readonly_shape_slist` and confirm that we can coerce the list of squares.

```
# type readonly_shape_slist = < iterator : shape iterator >;;
type readonly_shape_slist = < iterator : shape iterator >
# (squares :> readonly_shape_slist);;
- : readonly_shape_slist = <obj>
# let total_area (l : readonly_shape_slist) : float = ...;;
val total_area : readonly_shape_slist -> float = <fun>
#   total_area (squares :> readonly_shape_slist);;
- : float = 5.
```

Why does this work, why is a `square slist` a subtype of `readonly_shape_slist`. The reasoning is in two steps. First, the easy part is width subtyping: we can drop the other methods to see that `square slist` is a subtype of `< iterator : square iterator >`. The next step is to use *depth* subtyping, which, in its general form, says that an object type `< m : t1 >` is a subtype of a type `< m : t2>` iff `t1` is a subtype of `t2`. In other words, instead of reasoning about the number of methods in a type (the width), the number of methods is fixed, and we look within the method types themselves (the "depth").

In this particular case, depth subtyping on the `iterator` method requires that `square iterator` be a subtype of `shape iterator`. Expanding the type definition for the type `iterator`, we again invoke depth subtyping, and we need to show that the type `< get : square >` is a subtype of `<get : shape >`, which follows because `square` is a subtype of `shape`.

This reasoning may seem fairly long and complicated, but it should be pointed out that this typing *works*, and in the end the type annotations are fairly minor. In most typed object-oriented languages, the coercion would simply not be possible. For example, in C++, a STL type `slist<T>` is invariant in `T`, it is simply not possible to use `slist<square>` where `slist<shape>` is expected (at least safely). The situation is similar in Java, although Java supports has an escape hatch that allows the program to fall back to dynamic typing. The situation in OCaml is much better; it works, it is statically checked, and the annotations are pretty simple.

Using elided types to address subtyping problems

Before we move to the next topic, there is one more thing to address. The typing we gave above, using `readonly_shape_slist`, requires that the caller perform an explicit coercion before calling the `total_area` function. We would like to give a better type that avoids the coercion.

A solution is to use an elided type. Instead of `shape`, we can use the elided type `< area : float; .. >`. In fact, once we do this, it also becomes possible to use the `slist` type.

```
# let total_area (l : < area : float; .. > slist) : float = ...;;
val total_area : < area : float; .. > slist -> float = <fun>
# total_area squares;;
- : float = 5.
```

This works, and it removes the need for explicit coercions. This type is still fairly simple, but it does have the drawback that the programmer needs to remember that the types `< area : float; ..>` and `shape` are related.

OCaml supports an abbreviation in this case, but it works only for classes, not object types. The type expression `# classname` is an abbreviation for an elided type containing all of the methods in the named class, and more. Since `shape` is an object type, we can't write `#shape`. However, if a class definition is available, this abbreviation can be useful. The following definition is exactly equivalent to the preceeding.

```
# class cshape = object method area = 0.0 end;;
class cshape : object method area : float end
# let total_area (l : #cshape list) : float = ...;;
val total_area : #cshape slist -> float = <fun>
# total_area squares;;
- : float = 5.
```

Narrowing

Narrowing, also called *down casting*, is the ability to coerce an object to one of its subtypes. For example, if we have a list of shapes `shape slist`, we might know (for some reason) what the actual type of each shape is. Perhaps we know that all objects in the list have type `square`. In this case, *narrowing* would allow the re-casting of the object from type `shape` to type `square`. Many languages support narrowing through dynamic type checking. For example, in Java, a coercion `(Square) x` is allowed if the value `x` has type `Square` or one of its subtypes; otherwise the coercion throws an exception.

Narrowing is *not permitted* in OCaml. Period.

Why? There are two reasonable explanations, one based on a design principle, and another technical (the technical reason is simple: it is hard to implement).

The design argument is this: narrowing violates abstraction. In fact, with a structural typing system like in OCaml, narrowing would essentially provide the ability to enumerate the methods in an object. To check whether an object `obj` has some method `foo : int`, one would attempt a coercion `(obj :> < foo : int >)`.

More commonly, narrowing leads to poor object-oriented style. Consider the following Java code, which returns the name of a shape object.

```
String GetShapeName(Shape s) {
  if (s instanceof Square) {
    return "Square";
  } else if (s instanceof Circle) {
    return "Circle";
  } else {
    return "Other";
  }
}
```

Most programmers would consider this code to be "wrong." Instead of performing a case analysis on the type of object, it would be better to define a method to return the name of the shape. Instead of calling `GetShapeName(s)`, we should call `s.Name()` instead.

However, the situation is not always so obvious. The following code checks whether an array of shapes looks like a "barbell," composed to two `Circle` objects separated by a `Line`, where the circles have the same radius.

```
boolean IsBarBell(Shape[] s) {
  return s.length == 3 && (s[0] instanceof Circle) &&
    (s[1] instanceof Line) && (s[2] instanceof Circle) &&
    ((Circle) s[0]).radius() == ((Circle) s[2]).radius();
}
```

In this case, it is much less clear how to augment the `Shape` class to support this kind of pattern analysis. It is also not obvious that object-oriented programming is well-suited for this situation. Pattern matching seems like a better fit.

```
let is_bar_bell = function
 | [Circle r1; Line _; Circle r2] when r1 == r2 -> true
 | _ -> false;;
```

Regardless, there is a solution if you find yourself in this situation, which is to augment the classes with variants. You can define a method `variant` that injects the actual object into a variant type.

```
type shape = < variant : repr; area : float>
and circle = < variant : repr; area : float; radius : float >
and line = < variant : repr; area : float; length : float >
and repr =
 | Circle of circle
 | Line of line;;

let is_bar_bell = function
 | [s1; s2; s3] ->
   (match s1#variant, s2#variant, s3#variant with
     | Circle c1, Line _, Circle c2 when c1#radius == c2#radius -> true
     | _ -> false)
 | _ -> false;;
```

This pattern works, but it has drawbacks. In particular, the recursive type definition should make it clear that this pattern is essentially equivalent to using variants, and that objects do not provide much value here.

Binary methods

A *binary method* is a method that takes an object of `self` type. One common example is defining a method for equality.

```
# class square w =
  object (self : 'self)
    method width = w
    method area = self#width * self#width
    method equals (other : 'self) = other#width = self#width
  end;;
class square : int ->
  object ('a)
    method area : int
    method equals : 'a -> bool
    method width : int
  end
# class rectangle w h =
  object (self : 'self)
    method width = w
    method height = h
    method area = self#width * self#height
    method equals (other : 'self) = other#width = self#width && other#height = self#height
  end;;
...
# (new square 5)#equals (new square 5);;
- : bool = true
# (new rectangle 5 6)#equals (new rectangle 5 7);;
- : bool = false
```

This works, but there is a problem lurking here. The method `equals` takes an object of the exact type `square` or `rectangle`. Because of this, we can't define a common base class `shape` that also includes an equality method.

```
# type shape = < equals : shape -> bool; area : int >;;
# let sq = new square 5;;
# (sq :> shape);;
Characters 0-13:
  (sq :> shape);;
  ^^^^^^^^^^^^^
Error: Type square = < area : int; equals : square -> bool; width : int >
       is not a subtype of shape = < area : int; equals : shape -> bool >
Type shape = < area : int; equals : shape -> bool > is not a subtype of
  square = < area : int; equals : square -> bool; width : int >
```

The problem is that a `square` expects to be compared with a `square`, not an arbitrary shape; similarly for `rectangle`.

This problem is fundamental. Many languages solve it either with narrowing (with dynamic type checking), or by method overloading. Since OCaml has neither of these, what can we do?

One proposal we could consider is, since the problematic method is equality, why not just drop it from the base type `shape` and use polymorphic equality instead? Unfortunately, the builtin equality has very poor behavior when applied to objects.

```
# (object method area = 5 end) = (object method area = 5 end);;
- : bool = false
```

The problem here is that the builtin polymorphic equality compares the method implementations, not their return values. The method implementations (the function values that implement the methods) are different, so the equality comparison is false. There are other reasons not to use the builtin polymorphic equality, but these false negatives are a showstopper.

If we want to define equality for shapes in general, the remaining solution is to use the same approach as we described for narrowing. That is, introduce a *representation* type implemented using variants, and implement the comparison based on the representation type.

```
type shape_repr =
 | Square of int
 | Circle of int
 | Rectangle of int * int;;

type shape = < repr : shape_repr; equals : shape -> bool; area : int >;;

class square w =
object (self : 'self)
  method width = w
  method area = self#width * self#width
  method repr = Square self#width
  method equals (other : shape) = self#repr = other#repr
end;;
```

The binary method `equals` is now implemented in terms of the concrete type `shape_repr`. In fact, the objects are now isomorphic to the `shape_repr` type. When using this pattern, you will not be able to hide the `repr` method, but you can hide the type definition using the module system.

```
module Shapes : sig
  type shape_repr
  type shape = < repr : shape_repr; equals : shape -> bool; area -> int >

  class square : int ->
    object
      method width : int
      method area : int
      method repr : shape_repr
      method equals : shape -> bool
    end
end = struct
  type shape_repr = Square of int | Circle of int | Rectangle of int * int
  ...
end;;
```

Private methods

Methods can be declared *private*, which means that they may be called by subclasses, but they are not visible otherwise (similar to a *protected* method in C++).

To illustrate, let's build a class `vector` that contains an array of integers, resizing the storage array on demand. The field `values` contains the actual values, and the `get`, `set`, and `length` methods implement the array access. For clarity, the resizing operation is implemented as a private method `ensure_capacity` that resizes the array if necessary.

```
# class vector =
  object (self : 'self)
     val mutable values : int array = [||]

     method get i = values.(i)
     method set i x =
        self#ensure_capacity i;
        values.(i) <- x
     method length = Array.length values

     method private ensure_capacity i =
        if self#length <= i then
           let new_values = Array.create (i + 1) 0 in
           Array.blit values 0 new_values 0 (Array.length values);
           values <- new_values
  end;;
# let v = new vector;;
# v#set 5 2;;
# v#get 5;;
- 2 : int
# v#ensure_capacity 10;;
Characters 0-1:
  v#ensure_capacity 10;;
  ^
Error: This expression has type vector
       It has no method ensure_capacity
```

To be precise, the method `ensure_capacity` is part of the class type, but it is not part of the object type. This means the object `v` has no method `ensure_capacity`. However, it is available to subclasses. We can extend the class, for example, to include a method `swap` that swaps two elements.

```
# class swappable_vector =
  object (self : 'self)
     inherit vector

     method swap i j =
        self#ensure_capacity (max i j);
        let tmp = values.(i) in
        values.(i) <- values.(j);
        values.(j) <- tmp
  end;;
```

Yet another reason for private methods is to factor the implementation and support recursion. Moving along with this example, let's build a binary heap, which is a binary tree in heap order: where the label of parent elements is smaller than the labels of its children. One efficient implementation is to use an array to represent the values, where the root is at index 0, and the children of a parent node at index `i` are at indexes `2 * i` and `2 * i + 1`. To insert a node into the tree, we add it as a leaf, and then recursively move it up the tree until we restore heap order.

```
class binary_heap =
object (self : 'self)
   val values = new swappable_vector

   method min =
      if values#length = 0 then
         raise (Invalid_argument "heap is empty");
      values#get 0

   method add x =
      let pos = values#length in
      values#set pos x;
      self#move_up pos

   method private move_up i =
      if i > 0 then
         let parent = (i - 1) / 2 in
            if values#get i < values#get parent then begin
               values#swap i parent;
               self#move_up parent
            end
end;;
```

The method `move_up` implements the process of restoring heap order as a recursive method (though it would be straightforward avoid the recursion and use iteration here).

The key property of private methods is that they are visible to subclasses, but not anywhere else. If you want the stronger guarantee that a method is *really* private, not even accessible in subclasses, you can use an explicit typing that omits the method. In the following code, the `move_up` method is explicitly omitted from the object type, and it can't be invoked in subclasses.

```
# class binary_heap :
  object
    method min : int
    method add : int -> unit
  end =
  object (self : 'self) {
    ...
    method private move_up i = ...
  end;;
```

Virtual classes and methods

A *virtual* class is a class where some methods or fields are declared, but not implemented. This should not be confused with the word "virtual" as it is used in C++. In C++, a "virtual" method uses dynamic dispatch, regular non-virtual methods use static dispatched. In OCaml, *all* methods use dynamic dispatch, but the keyword *virtual* means the method or field is not implemented.

In the previous section, we defined a class `swappable_vector` that inherits from `array_vector` and adds a `swap` method. In fact, the `swap` method could be defined for any object with `get` and `set` methods; it doesn't have to be the specific class `array_vector`.

One way to do this is to declare the `swappable_vector` abstractly, declaring the methods `get` and `set`, but leaving the implementation for later. However, the `swap` method can be defined immediately.

```
class virtual abstract_swappable_vector =
object (self : 'self)
   method virtual get : int -> int
   method virtual set : int -> int -> unit
   method swap i j =
      let tmp = self#get i in
      self#set i (self#get j);
      self#set j tmp
end;;
```

At some future time, we may settle on a concrete implementation for the vector. We can inherit from the `abstract_swappable_bvector` to get the `swap` method "for free." Here's one implementation using arrays.

```
class array_vector =
object (self : 'self)
   inherit abstract_swappable_vector

   val mutable values = [||]
   method get i = values.(i)
   method set i x =
      self#ensure_capacity i;
      values.(i) <- x
   method length = Array.length values

   method private ensure_capacity i =
      if self#length <= i then
         let new_values = Array.create (i + 1) 0 in
            Array.blit values 0 new_values 0 (Array.length values);
            values <- new_values
end
```

Here's a different implementation using `HashTbl`.

```
class hash_vector =
object (self : 'self)
   inherit abstract_swappable_vector

   val table = Hashtbl.create 19

   method get i =
      try Hashtbl.find table i with
         Not_found -> 0

   method set = Hashtbl.add table
end;;
```

One way to view a `virtual` class is that it is like a functor, where the "inputs" are the declared, but not defined, virtual methods and fields. The functor application is implemented through inheritance, when virtual methods are given concrete implementations.

We've been mentioning that fields can be virtual too. Here is another implementation of the swapper, this time with direct access to the array of values.

```
class virtual abstract_swappable_array_vector =
object (self : 'self)
   val mutable virtual values : int array
   method private virtual ensure_capacity : int -> unit

   method swap i j =
      self#ensure_capacity (max i j);
      let tmp = values.(i) in
      values.(i) <- values.(j);
      values.(j) <- tmp
end;;
```

This level of dependency on the implementation details is possible, but it is hard to justify the use of a virtual class -- why not just define the `swap` method as part of the concrete class? Virtual classes are better suited for situations where there are multiple (useful) implementations of the virtual parts. In most cases, this will be public virtual methods.

Multiple inheritance

When a class inherits from more than one superclass, it is using *multiple inheritance*. Multiple inheritance extends the variety of ways in which classes can be combined, and it can be quite useful, particularly with virtual classes. However, it can be tricky to use, particularly when the inheritance hierarchy is a graph rather than a tree, so it should be used with care.

How names are resolved

The main "trickiness" of multiple inheritance is due to naming -- what happens when a method or field with some name is defined in more than one class?

If there is one thing to remember about inheritance in OCaml, it is this: inheritance is like textual inclusion. If there is more than one definition for a name, the last definition wins. Let's look at some artificial, but illustrative, examples.

First, let's consider what happens when we define a method more than once. In the following example, the method `get` is defined twice; the second definition "wins," meaning that it overrides the first one.

```
# class m1 =
object (self : 'self)
   method get = 1
   method f = self#get
   method get = 2
end;;
class m1 : object method f : int method get : int end
# (new m1)#f;;
- : int = 2
```

Fields have similar behavior, though the compiler produces a warning message about the override.

```
# class m2 =
# class m2 =
  object (self : 'self)
     val x = 1
     method f = x
     val x = 2
  end;;
Characters 69-74:
     val x = 2
         ^^^^^
Warning 13: the instance variable x is overridden.
The behaviour changed in ocaml 3.10 (previous behaviour was hiding.)
class m2 : object val x : int method f : int end
# (new m2)#f;;
- : int = 2
```

Of course, it is unlikely that you will define two methods or two fields of the same name in the same class. However, the rules for inheritance follow the same pattern: the last definition wins. In the following definition, the `inherit` declaration comes last, so the method definition `method get = 2` overrides the previous definition, always returning 2.

```
# class m4 = object method get = 2 end;;
# class m5 =
  object
    val mutable x = 1
    method get = x
    method set x' = x <- x'
    inherit m4
  end;;
class m5 : object val mutable x : int method get : int method set : int -> unit end
# let x = new m5;;
val x : m5 = <obj>
# x#set 5;;
- : unit = ()
# x#get;;
- : int = 2
```

To reiterate, to understand what inheritance means, replace each `inherit` directive with its definition, and take the last definition of each method or field. This holds even for private methods. However, it does *not* hold for private methods that are "really" private, meaning that they have been hidden by a type constraint. In the following definitions, there are three definitions of the private method `g`. However, the definition of `g` in `m8` is not overridden, because it is not part of the class type for `m8`.

```
# class m6 =
  object (self : 'self)
     method f1 = self#g
     method private g = 1
  end;;
class m6 : object method f1 : int method private g : int end
# class m7 =
  object (self : 'self)
     method f2 = self#g
     method private g = 2
  end;;
class m7 : object method f2 : int method private g : int end
# class m8 : object method f3 : int end =
  object (self : 'self)
     method f3 = self#g
     method private g = 3
  end;;
class m8 : object method f3 : int end
# class m9 =
  object (self : 'self)
     inherit m6
     inherit m7
     inherit m8
  end;;
# class m9 :
  object
    method f1 : int
    method f2 : int
    method f3 : int
    method private g : int
  end
# let x = new m9;;
val x : m9 = <obj>
# x#f1;;
- : int = 2
# x#f3;;
- : int = 3
```

Mixins

When should you use multiple inheritance? If you ask multiple people, you're likely to get multiple (perhaps heated) answers. Some will argue that multiple inheritance is overly complicated; others will argue that inheritance is problematic in general, and one should use object composition instead. But regardless of who you talk to, you will rarely hear that multiple inheritance is great and you should use it widely.

In any case, if you're programming with objects, there's one general pattern for multiple inheritance that is both useful and reasonably simple, the *mixin* pattern. Generically, a *mixin* is just a virtual class that implements a feature based on another one. If you have a class that implements methods *A*, and you have a mixin *M* that provides methods *B* from *A*, then you can inherit from *M* -- "mixing" it in -- to get features *B*.

That's too abstract, so let's give an example based on collections. In Section XXX:Objecttypes, we introduced the *iterator* pattern, where an *iterator* object is used to enumerate the elements of a collection. Lots of containers can have iterators, singly-linked lists, dictionaries, vectors, etc.

```
type 'a iterator = < get : 'a; has_value : bool; next : unit >;;
class ['a] slist : object ... method iterator : 'a iterator end;;
class ['a] vector : object ... method iterator : 'a iterator end;;
class ['a] deque : object ... method iterator : 'a iterator end;;
class ['a, 'b] map : object ... method iterator : 'b iterator end;;
...
```

The collections are different is some ways, but they share a common pattern for iteration that we can re-use. For a simple example, let's define a mixin that implements an arithmetic sum for a collection of integers.

```
# class virtual int_sum_mixin =
  object (self : 'self)
     method virtual iterator : int iterator
     method sum =
        let it = self#iterator in
        let total = ref 0 in
        while it#has_value do
           total := !total + it#get;
           it#next
        done;
        !total
  end;;
# class int_slist =
  object
     inherit [int] slist
     inherit int_sum_mixin
  end;;
# let l = new int_slist;;
val l : int_slist = <obj>
# l#insert 5;;
# l#insert 12;;
# l#sum;;
- : int = 17
# class int_deque =
  object
     inherit [int] deque
     inherit int_sum_mixin
  end;;
```

In this particular case, the mixin works only for a collection of integers, so we can't add the mixin to the polymorphic class definition `['a] slist` itself. However, the result of using the mixin is that the integer collection has a method `sum`, and it is done with very little of the fuss we would need if we used object composition instead.

The mixin pattern isn't limited to non-polymorphic classes, of course. We can use it to implement generic features as well. The following mixin defines functional-style iteration in terms of the imperative iterator pattern.

```
class virtual ['a] fold_mixin =
object (self : 'self)
   method virtual iterator : 'a iterator
   method fold : 'b. ('b -> 'a -> 'b) -> 'b -> 'b =
      (fun f x ->
            let y = ref x in
            let it = self#iterator in
            while it#has_value do
               y := f !y it#get;
               it#next
            done;
            !y)
end;;

class ['a] slist_with_fold =
object
   inherit ['a] slist
   inherit ['a] fold_mixin
end;;
```

CHAPTER 20

Managing external memory with Bigarrays

Bigarrays for external memory blocks

An OCaml bigarray is a useful custom block provided as standard to manipulate memory blocks outside the OCaml heap. It has `Custom_tag` in the header, and the first word points to the `custom_operations` struct for bigarrays. Following this is a `caml_ba_array` struct.

```
struct caml_ba_array {
  void * data;                /* Pointer to raw data */
  intnat num_dims;            /* Number of dimensions */
  intnat flags;               /* Kind of element array + memory layout + allocation status */
  struct caml_ba_proxy * proxy; /* The proxy for sub-arrays, or NULL */
  intnat dim[]  /*[num_dims]*/; /* Size in each dimension */
};
```

The `data` is usually a pointer to a `malloc`'ed chunk of memory, which the custom finalizer operation `free`'s when the block is free. The `flags` field encodes three values, located in the bits as specified by three masks:

```
CAML_BA_KIND_MASK = 0xFF     /* Mask for kind in flags field */
CAML_BA_LAYOUT_MASK = 0x100  /* Mask for layout in flags field */
CAML_BA_MANAGED_MASK = 0x600 /* Mask for "managed" bits in flags field */
```

The `CAML_BA_KIND_MASK` bits hold a value of the `caml_ba_kind` enum that identifies the kind of value in the bigarray `data`.

```
enum caml_ba_kind {
  CAML_BA_FLOAT32,           /* Single-precision floats */
  CAML_BA_FLOAT64,           /* Double-precision floats */
  CAML_BA_SINT8,             /* Signed 8-bit integers */
  CAML_BA_UINT8,             /* Unsigned 8-bit integers */
  CAML_BA_SINT16,            /* Signed 16-bit integers */
  CAML_BA_UINT16,             /* Unsigned 16-bit integers */
  CAML_BA_INT32,              /* Signed 32-bit integers */
  CAML_BA_INT64,              /* Signed 64-bit integers */
  CAML_BA_CAML_INT,           /* OCaml-style integers (signed 31 or 63 bits) */
  CAML_BA_NATIVE_INT,         /* Platform-native long integers (32 or 64 bits) */
  CAML_BA_COMPLEX32,          /* Single-precision complex */
  CAML_BA_COMPLEX64,          /* Double-precision complex */
}
```

The `CAML_BA_LAYOUT_MASK` bit says whether multi-dimensional arrays are layed out C or Fortran style.

```
enum caml_ba_layout {
  CAML_BA_C_LAYOUT = 0,           /* Row major, indices start at 0 */
  CAML_BA_FORTRAN_LAYOUT = 0x100, /* Column major, indices start at 1 */
};
```

The `CAML_BA_MANAGED_MASK` bits hold a value of the `caml_ba_managed` enum that identifies whether OCaml is responsible for freeing the `data` or some other code is.

```
enum caml_ba_managed {
  CAML_BA_EXTERNAL = 0,        /* Data is not allocated by OCaml */
  CAML_BA_MANAGED = 0x200,     /* Data is allocated by OCaml */
  CAML_BA_MAPPED_FILE = 0x400, /* Data is a memory mapped file */
};
```

CHAPTER 21

Inside the Runtime

(*avsm*: this chapter is still being chopped and changed)

Runtime Memory Management

The OCaml runtime divides the address space into memory pages of 4KB each (this is configurable by recompiling the runtime). At any given time, every page that is in use is used for a single purpose: major heap, minor heap, static data or code. The runtime guarantees this by always allocating slightly more memory than requested so that that it can choose align the memory it will actually use at the beginning of a 4KB page.

The runtime maintains a *page table* that allows it to determine the status of any virtual memory address in the operating system process. The status defines whether that address is a page in use by the OCaml runtime, and if so, which of the four purposes it is being used for.

Since the virtual memory space can be very large and sparsely used (especially on a 64-bit CPU), the page table is implemented as a hash table in which keys are page-aligned addresses and values are a single byte. The hash table is represented as an array of words, with each word being a key-value pair. The key-value pair is the bitwise `or` of the virtual address of the start of the page (which has zeros for its lower 12-bits due to being aligned to 4KB), and the lower 8 bits are used for the value. To look up an address, one masks out the lower 12-bits of the memory address, compute a multiplicative hash to get a table index, and then compares against the address (i.e. the key) at that index. Linear probing is used to resolve collisions.

The byte value stored is a bitwise `or` of the following status bits:

Page table status	Value	Meaning
`In_heap`	1	in the major heap
`In_young`	2	in the minor heap
`In_static_data`	4	in the statically allocated data segment

Page table status	Value	Meaning
`In_code_area`	8	in the statically allocated code segment

The page table starts with a size aiming to be between 25% and 50% full of entries, and is automatically doubled in size if it becomes half full. It is never shrunk.

Allocating on the minor heap

The minor heap is a contiguous chunk of virtual memory. Its size is set on program startup and decided by the `OCAMLRUNPARAM` environment variable (*avsm*: xref), and then only changed later by calls to `Gc.set`. The default size is 256k.

The range of memory usable for allocation goes from the `caml_young_start` to `caml_young_end` C variables managed by the runtime.

```
                    <---- size ---->
     base --- start ---------------- end
              limit      ptr <------
                              blocks
```

In a fresh minor heap, the `limit` will equal the `start`, and the current `ptr` will equal the `end`. As blocks are allocated, `caml_young_ptr` will decrease until it reaches `caml_young_limit`, at which point a minor garbage collection is triggered. To allocate a block in the minor heap, we decrement `caml_young_ptr` by the size of the block (including the header), and then set the the header to a valid value. If there isn't enough space left for the block without decrementing past the `limit`, a minor collection is triggered.

To force a minor gc to occur, one can set the `caml_young_limit` to equal `caml_young_end`, which causes signal handlers to be run and to "urge" the runtime (*avsm*: elaborate on this urging business, and how to set young from within OCaml via `Gc.??`).

Allocating on the major heap

The major heap is a singly linked list of contiguous memory chunks, sorted in increasing order of virtual address. Each chunk is a single memory chunk allocated via *malloc(3)* and consists of a header and a data area which contains OCaml blocks. A pointer to a heap chunk points to the start of the data area, and access to the header is done by a negative offset from this pointer. A chunk header has:

- the address of the memory that the chunk is in, as allocated by *malloc(3)*. It is needed when the chunk is freed.
- the size in bytes of the data area

- an allocation size in bytes, used during heap compaction to merge small blocks to defragment the heap.
- a link to the next heap chunk in the list.

The chunk's data area always starts on a page boundary, and its size is a multiple of the page size (4KB). It contains a contiguous sequence of heap blocks. These can be as small as one or two 4KB pages, but are usually allocated in 1MB chunks (or 512KB on 32-bit architectures). You can modify these defaults by editing `Heap_chunk_def` in `byterun/config.h` and recompiling the runtime. (*avsm*: talk about modifying the defaults in a separate callout, as there are quite a few variables which can be tweaked)

Allocating a block on the major heap first checks the free list of blocks (see below). If there isn't enough room on the free list, the runtime expands the major heap with a fresh block that will be large enough. That block is then added to the free list, and the free list is checked again (and this time will definitely succeed).

The major heap free list

The free space in the major heap's chunks is organized as a singly linked list of OCaml blocks, ordered by increasing virtual address. The runtime has a pointer to the first block in the free list. A free list block is at least two words: a header followed by a pointer to the next free-list block. The header specifies the length of the block just as with a normal block. (*avsm*: I'm not sure that this is quite true. It seems from `freelist.c` that the freelist blocks are normal OCaml blocks, with the first data entry being the next pointer. when detached, they become normal ocaml blocks)

As soon as the runtime finds a free block that is larger than the request, there are three possibilities:

- If the free block is exactly the right size, it is unlinked from the free list and returned as the requested block.
- If the free block is one word too large, it is unlinked from the free list, and the first word is given a special header recognizable to the collector as an unused word, while the rest of the block is returned as the requested block.
- If the free block is two or more words larger than the requested block, it remains in the free list, with its length shortened, and the end of the free block is returned for the requested block. Since the allocated block is right-justified within the free block, the linking of the free list doesn't need to be changed at all as the block that remains in the free list is the original one.

Memory allocation strategies

Allocating a new block in the major heap always looks in the free list. There are two allocation policies: first fit and next fit (the default).

Next-fit allocation

Next-fit allocation keeps a pointer to the block in the free list that was most recently used to satisfy a request. When a new request comes in, the allocator searches from the next block until the end of the free list, and then from the beginning of the free list up to that block.

First-fit allocation

First-fit allocation focusses on reducing memory fragmentation, at the expense of slower block allocation. For some workloads, the reduction in the frequency in heap compaction will outweigh the extra allocation cost. (*avsm*: example?)

The runtime maintains an ordered array of freelist chunks, called the `flp` array. Imagine a function mapping a block's index in the free list to its size. The flp array pointers are to the high points of this graph. That is, if you walk the free list between `flp[i]` and `flp[i+1]`, you will come across blocks that have sizes at most the size of `flp[i]`. Furthermore this sequence of smaller-than-flp[i] blocks cannot be extended, which is equivalent to saying `size(flp[i+1]) > size(flp[i])`.

When allocating, we first check the flp-array. If `flp[i]` is not big enough for our new block, then we may as well skip to `flp[i+1]`, because everything in the free list before then will also be too small.

If there's nothing big enough in the `flp` array, we extend it by walking the free list starting at the *last* pointer in the `flp`-array, say `flp[N]`. We extend the `flp` array along the way, so that at each block, if this block is bigger than the current last thing in `flp` (which is equivalent to saying this is the biggest block we've ever seen, since the blocks pointed to by the $_{\mathrm{flp}}$ array are increasing in size), we add it to the end of $_{\mathrm{flp}}$. We stop this walk when we come across a block big enough to house our desired new block.

There's also the case when the `flp` array has its ceiling size of `FLP_MAX` (default 100). Then we just start at the end of the `flp` array and walk until we find something big enough. This is known in the as a slow first-fit search, since this linear walk may take a long time.

If we did manage to find something suitable in the `flp` array, say at index `i`, we need to update `flp`. This update is rather complex, and the reason why first-fit allocation is slower than next-fit. We walk through the free list between `flp[i-1]` and `flp[i]` and record every high point we come across. Say we find `j` such points. We move the upper portion of `flp` (from `flp[i+1]` to the end) to the right by `j` places and insert each new high point into the array. There is a further corner case when adding in `j` new high points would make `flp` bigger than `FLP_MAX`.

(*avsm*: this really needs a diagram)

Which allocation policy should I use?

(*avsm*: 0 is the next-fit policy, which is quite fast but can result in fragmentation. 1 is the first-fit policy, which can be slower in some cases but can be better for programs with fragmentation problems.)

Inter-generational pointers

Most incremental generational garbage collectors have to keep careful track of values pointing from old generations to younger ones. The OCaml runtime is no exception, and maintains a set of addresses in the major heap that may point into the minor heap. These addresses are *not* OCaml pointers, and just literal memory addresses. The runtime ensures that it never relocates values in the major heap unless this "remembered" set is empty. The set is maintained as a dynamically resized array of pointers, which is itself maintained via a collection of pointers known as the `caml_ref_table`.

```
struct caml_ref_table {
  value **base;
  value **end;
  value **threshold;
  value **ptr;
  value **limit;
  asize_t size;
  asize_t reserve;
};
```

The relationships of the pointers are as follows:

```
                 limit
   base <= ptr     threshold         end
     |----------------|----------------|
           size            reserve
```

An address is added to `caml_ref_table` when all of these conditions are satisfied:

- a field in a block in the major heap is mutated
- the field previously did not point to the minor heap
- the field is being changed to point into the minor heap

In that case the entry is added at `caml_ref_table.ptr`, which is then incremented. If `ptr` is already at `limit`, the table is doubled in size before adding the address.

The same address can occur in `caml_ref_table` multiple times if a block field is mutated repeatedly and alternated between pointing at the minor heap and the major heap. The field in `caml_ref_table` also may not always point into the minor heap (if it was changed after being added), since fields are never removed. The entire table is cleared as part of the minor collection process.

The write barrier

The write barrier is one of the reasons why using immutable data structures can sometimes be faster than mutable records. The OCaml compiler keeps track of any mutable types and adds a call to `caml_modify` before making the change. The `caml_modify` checks that the remembered set is consistent, which, although reasonably efficient, can be slower than simply allocating a fresh value on the fast minor heap.

Let's see this for ourselves with a simple test program:

```
type t1 = { mutable iters1: int; mutable count1: float }
type t2 = { iters2: int; count2: float }

let rec test_mutable t1 =
  match t1.iters1 with
  |0 -> ()
  |n ->
    t1.iters1 <- t1.iters1 - 1;
    t1.count1 <- t1.count1 +. 1.0;
    test_mutable t1

let rec test_immutable t2 =
  match t2.iters2 with
  |0 -> ()
  |n ->
    let iters2 = n - 1 in
    let count2 = t2.count2 +. 1.0 in
    test_immutable { iters2; count2 }

open Printf
let time name fn arg =
  Gc.compact ();
  let w1 = Gc.((stat ()).minor_collections) in
  let t1 = Unix.gettimeofday () in
  fn arg;
  let w2 = Gc.((stat ()).minor_collections) in
  let t2 = Unix.gettimeofday () in
  printf "%s: %.4fs (%d minor collections)\n" name (t2 -. t1) (w2 - w1)

let _ =
  let iters = 1000000000 in
  time "mutable" test_mutable { iters1=iters; count1=0.0 };
  time "immutable" test_immutable { iters2=iters; count2=0.0 }
```

This program defines a type `t1` that is mutable, and `t2` that is immutable. The main loop iterates over both fields and runs a simple counter. It measures two things: the wallclock time that all the iterations take, and the number of minor garbage collections that occurred during the test. The results should look something like this:

```
mutable: 8.6923s (7629 minor collections)
immutable: 2.6186s (19073 minor collections)
```

Notice the space/time tradeoff here. The mutable version runs almost 4 times slower than the immutable one, but has significantly fewer garbage collection cycles. Minor collections in OCaml are very fast, and so it is often acceptable to use immutable data structures in preference to the more conventional mutable versions. On the other hand, if you only rarely mutable a value, it can be faster to take the write barrier hit and not allocate at all.

(*avsm*: it would be really nice to use a benchmark suite here and shorten the example. Investigate the options and edit this section)

(*avsm*: need to mention when a value is allocated directly into the major heap somewhere)

How garbage collection works

Collecting the minor heap

For those familiar with garbage collection terminology, here is OCaml's minor colection in one sentence. OCaml's minor collection uses copying collection with forwarding pointers, and does a depth-first traversal of the block graph using a stack represented as a linked list threaded through blocks that need to be scanned.

The goal of minor collection is to empty the minor heap by moving to the major heap every block in the minor heap that might be used in the future, and updating each pointer to a moved block to the new version of the block. A block is *live* if is reachable by starting at some *root* pointer into a block in the minor heap,a nd then following pointers in blocks. There are many different kinds of roots:

- OCaml stack(s)
- C stack(s), identified by `BeginRoots` or `CAMLparam` in C code (*avsm*: xref C bindings chapter)
- Global roots
- Finalized values (*avsm*: ?)
- Intergenerational pointers in the `caml_ref_table` (*avsm*: xref above?)

Moving a block between heaps is traditionally called *forwarding*. The OCaml runtime code uses that term as well as the term *oldify*, which is useful to understand when profiling hotspots in your code. The minor collector first visits all roots and forwards them if they point to a block in the minor heap. When a block is forwarded, the collector sets the tag of the original block to a special `Forward_tag` (250), and the first field of the original block to point to the new block. Then, if the collector ever encounters a pointer to the original block again, it can simply update the pointer directly into the forwarded block.

Because a forwarded block might itself contain pointers, it must at some point be scanned to see if those pointers point to blocks in the minor heap, so that those blocks can also be forwarded. The collector maintains a linked list (called the `oldify_todo_list`) of forwarded objects that it still needs to scan. That linked list looks like:

```
             oldify_todo_list
                |
                |
                v
minor heap  | 0 | v | f1 ... |      | 0 | v | f1 ... |
                  |                     ^ |
                  |      +--------------+ |      ---------- ...
                  v      |                v      |
major heap    | h | f0 | ^ | ... |    | h | f0 | ^
```

Each value on the `oldify_todo_list` is marked as forwarded, and the first word points to the new block in the major heap. That new version contains the actual value header, the real first field of the value, and a link (pointer) to the next value on the oldify_todo_list, or NULL at the end of the list. Clearly this approach won't work if an value has only one field, since there will be no second field to store the link in. Values with exactly one field are never put on the `oldify_todo_list`; instead, the collector immediately traverses them, essentially making a tail call in the depth-first search.

Values that are known from the tag in their header to not contain pointers are simply forwarded and completely copied, and never placed on the `oldify_todo_list`. These tags are all greater than `No_scan_tag` and include strings and float arrays.

(*avsm*: note from sweeks to investigate: There is a hack for objects whose tag is `Forward_tag` that does some kind of path compression, or at least removal of one link, but I'm not sure what's going on.)

(*avsm*: I dont think we've introduced weak references yet, so this needs rearranging) At the end of the depth-first search in minor collection, the collector scans the `weak-ref` table, and clears any weak references that are still pointing into the minor heap. The collector then empties the `weak-ref` table and the minor heap.

Collecting the major heap

The major heap collections operates incrementally, as the amount of memory being tracked is a lot larger than the minor heap. The major collector can be in any of a number of phases:

- `Phase_idle`
- `Phase_mark`
    - `Subphase_main`: main marking phase
    - `Subphase_weak1`: clear weak pointers
    - `Subphase_weak2`: remove dead weak arrays, observe finalized values

- —`Subphase_final`: initialise for the sweep phase
- `Phase_sweep`

Marking the major heap

Marking maintains an array of gray blocks, `gray_vals`. It uses as them as a stack, pushing on a white block that is then colored gray, and popping off a gray block when it is scanned and colored black. The `gray_vals` array is allocated via *malloc(3)*, and there is a pointer, `gray_vals_cur`, to the next open spot in the array.

The `gray_vals` array initially has 2048 elements. `gray_vals_cur` starts at `gray_vals`, and increases until it reachs `gray_vals_end`, at which point the `gray_vals` array is doubled, as long as its size (in bytes) is less than 1/2^10th of the heap size (`caml_stat_heap_size`). When the gray vals is of its maximum allowed size, it isn't grown any further, and the heap is marked as impure (`heap_is_pure=0`), and last half of `gray_vals` is ignored (by setting `gray_vals_cur` back to the middle of the `gray_vals` array.

If the marking is able to complete using just the gray list, it will. Otherwise, once the gray list is emptied, the mark phase will observe that the heap is impure and initiate a backup approach to marking. In this approach it marks the heap as pure and then walks through the entire heap block by block, in increasing order of memory address. If it finds a gray block, it adds it to the gray list and does a DFS marking using the gray list as a stack in the usual way. Once the scan of the complete heap is finished, the mark phase checks again whether the heap has again become impure, and if so initiates another scan. These full-heap scans will continue until a successful scan completes without overflowing the gray list.

(*avsm*: I need to clarify this more, possibly a diagram too. It's not really clear what the implications of an impure heap are atm)

Sweeping unused blocks from the major heap

Compaction and defragmenting the major heap

2013-02-05
19:46:47

2013-02-05
19:46:47

CHAPTER 22

Performance Tuning and Profiling

Byte code Profiling

ocamlcp and call trace information

Native Code Profiling

gdb

requires shinwell's patch in ocaml trunk via opam

perf

requires fabrice's frame pointer patch

dtrace

requires my dtrace/instruments patch for libasmrun

2013-02-05
19:46:47

CHAPTER 23

Packaging and Build Systems

The OCaml toolchain is structured much like a C compiler, with several tools that generate intermediate files and finally link against a runtime. The final outputs don't have to be just executables. Many people embed OCaml code as object files that are called from other applications, or even compile it to Javascript and other esoteric targets. Let's start by covering some of the standard OCaml tools, and then move on to some of the higher level methods for packaging and publishing your code online.

The OCaml toolchain

There are two distinct compilers for OCaml code included in the standard distribution. The first outputs bytecode that is interpreted at runtime, and the second generates fast, efficient native code directly. Both of these share the front-end type-checking logic, and only diverge when it comes to code generation.

The `ocamlc` bytecode compiler

The simplest code generator is the `ocamlc` compiler, which outputs bytecode that is interpreted via the `ocamlrun` runtime. The OCaml bytecode virtual machine is a stack machine (much like the Java Virtual Machine), with the exception of a single register that stores the most recent result. This provides a simple runtime model that is easy to implement or embed within other systems, but executes rather slowly due to being interpreted.

Here are some of the intermediate files generated by `ocamlc`:

Extension	Purpose
.ml	Source files for compilation unit module implementations.
.mli	Source files for compilation unit module interfaces. If missing, generated from the `.ml` file.
.cmi	Compiled module interface from a corresponding `.mli` source file.
.cmo	Compiled bytecode object file of the module implementation.

Extension	Purpose
.cma	Library of bytecode object files packed into a single file.
.o	C source files are compiled into native object files by the system `cc`.

To obtain a bytecode executable, you need to compile a set of `cmo` object files, and then link them into an executable

The `ocamlopt` native code compiler

Extension	Purpose
.cmi	Compiled module interface from a corresponding `.mli` source file. (*avsm*: this is not compatible with the ocamlc version iirc)
.o	Compiled native object file of the module implementation.
.cmx	Contains extra information for linking and cross-module optimization of the object file.
.cmxa/.a	Library of `cmx` and `o` units, stored in the `cmxa` and `a` files respectively.

The `ocaml` top-level loop

The Findlib compiler frontend

Packaging applications with OASIS

ocamlbuild

Distributing applications with OPAM

CHAPTER 24

Parsing with OCamllex and OCamlyacc

2013-02-05
19:46:47

APPENDIX
Installation

The easiest way to install OCaml is via the binary packages available in many operating systems. For day-to-day code development, it is easier to use a source-based manager that lets you recompile individual libraries easily.

For the purposes of this book, we'll use the OPAM source manager. There are other alternatives available such as GODI and ODB, but not covered here. Let's get started with OPAM now, as you will need it to run the examples in the rest of the book. OPAM manages multiple simultaneous OCaml compiler and library installations, tracks library versions across upgrades, and recompiles dependencies automatically if they get out of date.

OPAM Base Installation

To install OPAM, you will need a working OCaml installation to bootstrap the package manager. Once installed, all of the OPAM state is held in the `$HOME/.opam` directory, and you can reinitialise it by deleting this directory and starting over.

OCamlfind and OPAM

OPAM maintains multiple compiler and library installations, but this can clash with a global installation of the `ocamlfind` tool. Uninstall any existing copies of `ocamlfind` before installing OPAM, and use the OPAM version instead.

MacOS X

The easiest way to install OCaml on MacOS X is via Homebrew (*http://github.com/mxcl/homebrew*).

```
$ brew install ocaml
$ brew install pcre
$ brew install opam
```

The Perl-compatible Regular Expression library (PCRE) is used by the Core suite. It's a commonly used library that you may already have installed for some other reason.

Another popular package manager is MacPorts (*http://macports.org*):

```
$ port install ocaml
$ port install ocaml-pcre
$ port install opam
```

Linux

On Debian Linux, you should install OCaml via binary packages, and then install the latest OPAM release from source.

```
$ sudo apt-get install build-essential m4 ocaml ocaml-native-compilers camlp4-extra
$ sudo apt-get install git libpcre3-dev curl
$ tar -zxvf opam-<version>.tar.gz
$ cd opam-<version>.tar.gz
$ ./configure && make && sudo make install
```

Note to reviewers: The OPAM instructions will be simplified when integrated upstream into Debian and Fedora. Until then, the source installation instructions above may require a few more packages, so please leave a comment with any amended instructions.

Fedora/RHEL: TODO

Windows

Windows is not currently supported, although it is being worked on. Until that's ready, we recommend using a virtual machine running Debian Linux on your local machine.

Using the OPAM top-level

All the OPAM state is held in the `.opam` directory in your home directory, including compiler installations. On Linux and MacOS X, this will be the `~/.opam` directory. You shouldn't switch to an admin user to install packages; if you run into problems, just delete the whole `.opam` directory and follow the installations instructions from the `opam init` stage again.

```
$ opam init
$ opam switch 4.00.1+short-types
$ opam install utop async core_extended
$ eval `opam config env`
```

This will initialise OPAM with a default package set, and install some packages to augment the OCaml distribution.

The `utop` package is an interactive command-line interface to OCaml that has tab-completion, persistent history and integration with Emacs so that you can run it within your editing environment. We use `utop` instead of the more spartan default OCaml top-level throughout the book.

The other two packages are `async` and `core_extended`, which are libraries from the Jane Street Core standard library. These will take a few minutes to compile on a modern machine (particularly `core_extended`) as they contain a large number of modules to help with day-to-day programming. We use Core throughout the examples in this book.

OPAM figures out the minimal set of dependencies required for thse packages and installs them too. The `eval` command sets your `PATH` variable to point to the current active compiler, and you should add this to your local shell `.profile` to run every time you open a new command shell. It won't affect any other commands, as OPAM makes sure to preserve any custom extensions you may already have to your `PATH` variable.

Switching compiler versions

The default compiler installed by OPAM uses the system OCaml installation. You can use `opam switch` to swap between different compiler versions, or experiment with a different set of libraries or new compiler versions. For instance, one of the alternate compilers is patched to simplify the types that are output in the top-level. You can switch it to by:

```
$ opam switch list
$ opam switch 4.00.1+short-types
$ eval `opam config env`
$ opam install utop async
```

The new compiler will be compiled and installed into `~/.opam/4.00.1+short-types` and the libraries will be tracked separately from your system installation. You can have any number of compilers installed simultaneously, but only one can be active at any time. You can find all the available compilers by running `opam switch list`.

Editing Environment

Command Line

The `utop` tool provides a convenient interactive top-level, with full command history, command macros and module name completion. An `.ocamlinit` file in your home directory will initialise `utop` with common libraries and syntax extensions open, e.g.:

```
#use "topfind"
#camlp4o
#thread
#require "core.top";;
```

```
#require "async";;
open Core.Std
open Async.Std
```

TODO: the `.ocamlinit` handling in OPAM is being finalised and is tracked in issue 185 (*https://github.com/OCamlPro/opam/issues/185*).

Editors

Emacs users have tuareg and Typerex (*http://www.typerex.org/*).

Vim users can use the built-in style, and ocaml-annot (*http://github.com/avsm/ocaml-annot*) may also be useful.

Eclipse plugins: which one is maintained?

Developing with OPAM

Package listings are obtained by adding *remotes* that provide package descriptions, installation instructions and URLs.