CHAPTER 5

Function-Building Functions

This chapter builds on the idea of first-class functions by explaining how and why one builds functions on the fly. It explores various ways to facilitate function "composition"—snapping together functions like Lego blocks to build richer functionality from parts.

The Essence of Functional Composition

Recall that the function `invoker` from Chapter 4 built a function taking an object as its first argument and attempted to call a method on it. If you'll recall, `invoker` returned `undefined` if the method was not available on the target object. This can be used as a way to compose multiple invokers together to form polymorphic functions, or functions that exhibit different behaviors based on their argument(s). To do this, I'll need a way to take one or more functions and keep trying to invoke each in turn, until a non-undefined value is returned. This function, `dispatch`, is defined imperatively as follows:

```
function dispatch(/* funs */) {
  var funs = _.toArray(arguments);
  var size = funs.length;

  return function(target /*, args */) {
    var ret = undefined;
    var args = _.rest(arguments);

    for (var funIndex = 0; funIndex < size; funIndex++) {
      var fun = funs[funIndex];
      ret = fun.apply(fun, construct(target, args));

      if (existy(ret)) return ret;
    }

    return ret;
  };
}
```

This is a lot of code to perform a simple task.[1]

To be clear, what you want to do is return a function that loops through an array of functions, calls each with an object, and returns the first actual value it finds (i.e., "existy"). However, despite its seeming complexity, `dispatch` fulfills the definition of polymorphic JavaScript functions. It does so in a way that simplifies the task of delegating to concrete method behaviors. For example, in the implementation of Underscore, you'll very often see the following pattern repeated in many different functions:

1. Make sure the target exists.
2. Check if there is a native version and use it if so.
3. If not, then do some specific tasks implementing the behavior:
   - Do type-specific tasks, if applicable.
   - Do argument-specific tasks, if applicable.
   - Do argument count–specific tasks, if applicable.

In code-speak, this same pattern is expressed in the implementation of Underscore's `_.map` function:

```
_.map = _.collect = function(obj, iterator, context) {
  var results = [];
  if (obj == null) return results;
  if (nativeMap && obj.map === nativeMap) return obj.map(iterator, context);
  each(obj, function(value, index, list) {
    results[results.length] = iterator.call(context, value, index, list);
  });
  return results;
};
```

The use of `dispatch` can work to simplify some of this code and allow easier extensibility. Imagine you're tasked with writing a function to generate the string representation for both array and string types. Using `dispatch` leads to an elegant implementation:

```
var str = dispatch(invoker('toString', Array.prototype.toString),
                   invoker('toString', String.prototype.toString));


str("a");
//=> "a"

str(_.range(10));
//=> "0,1,2,3,4,5,6,7,8,9"
```

1. Recall that the `construct` function was defined all the way back in Chapter 2.

That is, by coupling `invoker` with `dispatch`, I can delegate down to concrete implementations like `Array.prototype.toString` rather than using a single function that groups type and existence checks via `if-then-else`.[2]

Of course, the operation of `dispatch` is not dependent on the use of `invoker`, but instead adheres to a certain contract. That is, it will keep trying to execute functions until it runs out or one returns an existy value. I can tap into this contract by supplying a function that adheres to the contract at hand, as in `stringReverse`:

```
function stringReverse(s) {
  if (!_.isString(s)) return undefined;
  return s.split('').reverse().join("");
}

stringReverse("abc");
//=> "cba"

stringReverse(1);
//=> undefined
```

Now `stringReverse` can be composed with the `Array#reverse` method to define a new, polymorphic function, `rev`:

```
var rev = dispatch(invoker('reverse', Array.prototype.reverse), stringReverse);

rev([1,2,3]);
//=> [3, 2, 1]

rev("abc");
//=> "cba"
```

In addition, we can exploit the contract of `dispatch` to compose a terminating function that provides some default behavior by always returning an existy value or one that always throws an exception. As a nice bonus, a function created by `dispatch` can *also* be an argument to `dispatch` for maximum flexibility:

```
var sillyReverse = dispatch(rev, always(42));

sillyReverse([1,2,3]);
//=> [3, 2, 1]

sillyReverse("abc");
//=> "cba"

sillyReverse(100000);
//=> 42
```

2. Using `Array.prototype.toString` directly.

A more interesting pattern that `dispatch` eliminates is the switch statement manual dispatch, which looks like the following:

```
function performCommandHardcoded(command) {
  var result;

  switch (command.type)
  {
  case 'notify':
    result = notify(command.message);
    break;
  case 'join':
    result = changeView(command.target);
    break;
  default:
    alert(command.type);
  }

  return result;
}
```

The `switch` statement in the `performCommandHardcoded` function looks at a field on a `command` object and dispatches to relevant code depending on the command string:

```
performCommandHardcoded({type: 'notify', message: 'hi!'});
// does the nofity action

performCommandHardcoded({type: 'join', target: 'waiting-room'});
// does the changeView action

performCommandHardcoded({type: 'wat'});
// pops up an alert box
```

I can eliminate this pattern nicely using dispatch in the following way:

```
function isa(type, action) {
  return function(obj) {
    if (type === obj.type)
      return action(obj);
  }
}

var performCommand = dispatch(
  isa('notify', function(obj) { return notify(obj.message) }),
  isa('join',   function(obj) { return changeView(obj.target) }),
  function(obj) { alert(obj.type) });
```

The preceding code starts with an `isa` function that takes a `type` string and an `action` function and returns a new function. The returned function will call the `action` function only if the `type` string and the `obj.type` field match; otherwise, it returns `undefined`.

It's the return of `undefined` that signals to `dispatch` to try the next dispatch sub-function.[3]

To extend the `performCommandHardcoded` function, you would need to go in and changed the actual `switch` statement itself. However, you can extend the `performCommand` function with new behavior by simply wrapping it in another dispatch function:

```
var performAdminCommand = dispatch(
  isa('kill', function(obj) { return shutdown(obj.hostname) }),
  performCommand);
```

The newly created `performAdminCommand` states that it first tries to dispatch on the `kill` command, and if that fails then it tries the commands handled by `performCommand`:

```
performAdminCommand({type: 'kill', hostname: 'localhost'});
// does the shutdown action

performAdminCommand({type: 'flail'});
// alert box pops up

performAdminCommand({type: 'join', target: 'foo'});
// does the changeView action
```

You can also restrict the behavior by overriding commands earlier in the dispatch chain:

```
var performTrialUserCommand = dispatch(
  isa('join', function(obj) { alert("Cannot join until approved") }),
  performCommand);
```

Running through a couple of examples shows the new behavior:

```
performTrialUserCommand({type: 'join', target: 'foo'});
// alert box denial pops up

performTrialUserCommand({type: 'notify', message: 'Hi new user'});
// does the notify action
```

This is the essence of functional composition: using existing parts in well-known ways to build up new behaviors that can later serve as behaviors themselves. In the remainder of this chapter, I will discuss other ways to compose functions to create new behavior, starting with the notion of currying.

Mutation Is a Low-Level Operation

You've already been exposed to examples of functions implemented in an imperative fashion, and you will continue to see more as the book progresses. While often it's ideal to write code in a functional way, there are times when the primitives of a library, for

3. Some languages provide this kind of dispatch automatically as multimethods, i.e., function behavior determined by the result of a list of predicates or an arbitrary function.

the sake of speed or expediency, should be implemented using imperative techniques. Functions are quanta of abstraction, and the most important part of any given function is that it adheres to its contract and fulfills a requirement. No one cares if a variable was mutated deep within the confines of a function and never escaped. Mutation is sometimes necessary, but I view it as a low-level operation—one that should be kept out of sight and out of mind.

This book is not about spewing dogma regarding the virtues of functional programming. I think there are many functional techniques that offer ways to rein in the complexities of software development, but I realize that at times, there are better ways to implement any given individual part (Figure 5-1).

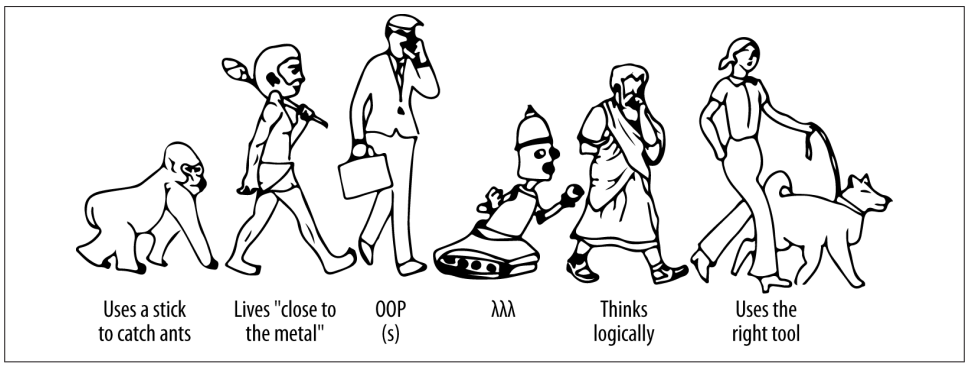

Figure 5-1. An "evolved" programmer knows when to use the right tool

Whenever you're building an application, it's always wise to explore the parameters of your personal execution needs to determine if a given implementation technique is appropriate. This book, while about functional programming, advocates above all else a full understanding of your problem and solution spaces to come to an understanding of your best-possible solution. I will discuss this theme throughout Chapter 7, but for now I present a recipe for delicious curry![4]

Currying

You've already seen an example of a curried function (namely, `invoker`). A curried function is one that returns a new function for every logical argument that it takes. In

4. The term "currying" has nothing at all to do with the delicious foodstuff. Instead, it's named after the mathematician Haskell Curry, who rediscovered a technique devised by another mathematician named Moses Schönfinkel. While Haskell Curry certainly contributed heaps to computer science, I think we've missed a fun opportunity to have a useful programming technique called schönfinkeling.

the case of `invoker`, you can imagine it operating in a slightly different (and more naive) way, as shown here:

```
function rightAwayInvoker() {
  var args = _.toArray(arguments);
  var method = args.shift();
  var target = args.shift();

  return method.apply(target, args);
}

rightAwayInvoker(Array.prototype.reverse, [1,2,3])
//=> [3, 2, 1]
```

That is, the function `rightAwayInvoker` does not return a function that then awaits a target object, but instead calls the method on the target taken as its second argument. The `invoker` function, on the other hand, is curried, meaning that the invocation of the method on a given target is deferred until its logical number of arguments (i.e., two) is exhausted. You can see this in action via the following:

```
invoker('reverse', Array.prototype.reverse)([1,2,3]);
//=> [3, 2, 1]
```

The double parentheses give away what's happening here (i.e., the function returned from `invoker` bound to the execution of `reverse` is immediately called with the array `[1,2,3]`).

Recall the idea that it's useful to return functions (closures) that are "configured" with certain behaviors based on the context in which they were created. This same idea can be extended to curried functions as well. That is, for every logical parameter, a curried function will keep returning a gradually more configured function until all parameters have been filled (Figure 5-2).

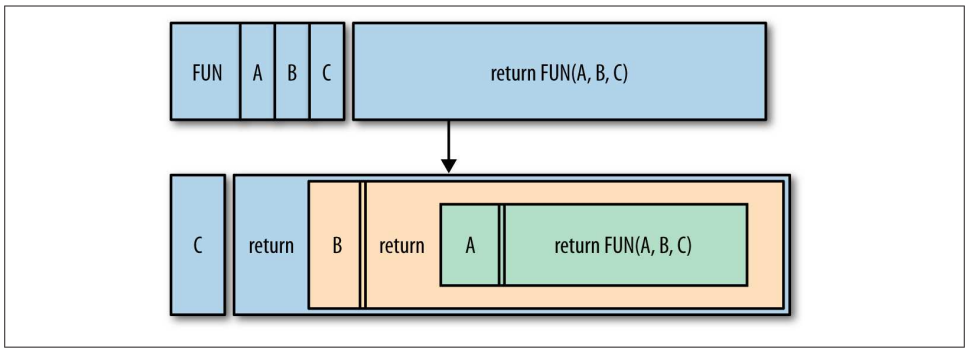

Figure 5-2. An illustration of currying

The idea of currying is simple, but there is one consideration that must be addressed. That is, if a curried function returns one function per parameter, then which parameter does the "uncurrying" start with, and with which does it end?

To Curry Right, or To Curry Left

The direction that you curry doesn't really matter, but the choice will have some implications on your API. For the purposes of this book (and my preference in general), I will curry starting at the rightmost argument and move left. In a language like JavaScript that allows you to pass any number of arguments, right-to-left currying allows you to fix the optional arguments to certain values.

To illustrate what I mean by the difference in argument direction, observe the following two functions:

```
function leftCurryDiv(n) {
  return function(d) {
    return n/d;
  };
}

function rightCurryDiv(d) {
  return function(n) {
    return n/d;
  };
}
```

The use of a division operation to illustrate currying works nicely because the result changes if the arguments are switched (i.e., it's not associative). Using the `leftCurryDiv` function, observe how the curried parameters produce a result:

```
var divide10By = leftCurryDiv(10);
```

The function produced on the initial call, with `10` named `divide10By`, produces a function that, for all intents and purposes, contains a body pertaining to `10 / ?`, where `?` is the curried rightmost parameter awaiting a value on the next call:

```
divide10By(2);
//=> 5
```

The second invocation of the curried function (named `divide10By`) now executes the fully populated body `10 / 2`, resulting in the value `5`. However, if the `rightCurryDiv` function is used, the behavior changes:

```
var divideBy10 = rightCurryDiv(10);
```

Now the body of the curried function named `divideBy10` is instead `? / 10`, awaiting the leftmost argument before executing:

```
divideBy10(2);
//=> 0.2
```

As I mentioned, I will begin currying from the rightmost argument, so my calculations will operate as shown in Figure 5-2.

Another reason for currying from the right is that partial application handles working from the left (partial application will be discussed in greater depth in the next section). Between partial application and currying, I have both directions covered, allowing the full range of parameter specialization. Having said all of that, I'll presently implement a few functions both manually curried (as in `leftCurryDiv` and `rightCurryDiv`) and with an auto-currying function or two that I'll also implement.

Automatically Currying Parameters

The functions `over10` and `divideBy10` were both curried by hand. That is, I explicitly wrote the functions to return the right number of functions corresponding to the number of function parameters. Likewise, for the purposes of illustration, my function `rightCurryDiv` returned a function corresponding to a division function taking two logical arguments. However, there is value in a simple higher-order function that takes a function and returns a function "pinned" to receive only one argument; I'll call this function `curry` and implement it as follows:

```
function curry(fun) {
  return function(arg) {
    return fun(arg);
  };
}
```

The operation of `curry` can be summarized as follows:

- Takes a function
- Returns a function expecting one parameter

This seems like a fairly useless function, no? Why not simply use the function directly instead? In many functional programming languages, there are few compelling reasons to provide an unadorned delegation like `curry` provides, but in JavaScript the story is slightly different. Very often in JavaScript, functions will take some number of expected arguments and an additional number of "specialization" arguments. For example, the JavaScript function `parseInt` takes a string and returns its equivalent integer:

```
parseInt('11');
//=> 11
```

Additionally, `parseInt` will accept a second argument that defines the radix to use when parsing (i.e., the number base):

```
parseInt('11', 2);
//=> 3
```

The preceding call, given a radix value of 2, means that the number is parsed as a binary (base-2) number. Complications arise using `parseInt` in a first-class way because of that optional second argument, as shown here:

```
['11','11','11','11'].map(parseInt)
//=> [11, NaN, 3, 4]
```

The problem here is that in some versions of JavaScript, the function given to `Array#map` will be invoked with each element of the array, the index of the element, plus the array itself.[5] So as you might have guessed, the radix argument for `parseInt` starts with `0` and then becomes `1`, `2`, and then `3`. Ouch! Thankfully, using `curry`, you can force `parseInt` to receive only one argument on each call:

```
['11','11','11','11'].map(curry(parseInt));
//=> [11, 11, 11, 11]
```

I could have just as easily written a function that takes an arbitrary number of arguments and figures out how to curry the remaining arguments, but I like to be explicit when currying. The reason is that the use of a function like `curry` allows me to explicitly control the behavior of the function being called by fixing (or ignoring) the optional right-leaning arguments used for specialization.

Take, for example, the act of currying two function parameters using a `curry2` function, defined as such:

```
function curry2(fun) {
  return function(secondArg) {
    return function(firstArg) {
      return fun(firstArg, secondArg);
    };
  };
}
```

The `curry2` function takes a function and curries it up to two parameters deep. Using it to implement a version of the previously defined `divideBy10` function is shown here:

```
function div(n, d) { return n / d }

var div10 = curry2(div)(10);

div10(50);
//=> 5
```

Just like `rightCurryDiv`, the `div10` function awaits its first argument with a logical body corresponding to `? / 10`. And just for the sake of completion, `curry2` can also be used to fix the behavior of `parseInt` so that it handles only binary numbers when parsing:

5. The Underscore `map` function is subject to this problem as well.

```
var parseBinaryString = curry2(parseInt)(2);

parseBinaryString("111");
//=> 7

parseBinaryString("10");
//=> 2
```

Currying is a useful technique for specifying the specialized behavior of JavaScript functions and for "composing" new functions from existing functions, as I'll show next.

Building new functions using currying

I showed a way to use `curry2` to build a simple `div10` function that expects a numerator in a division operator, but that's not the full extent of its usefulness. In fact, in exactly the same way that closures are used to customize function behavior based on captured variables, currying can do the same via fulfilled function parameters. For example, Underscore provides a `_.countBy` function that, given an array, returns an object keying the count of its items tagged with some piece of data. Observe the operation of `_.countBy`:

```
var plays = [{artist: "Burial", track: "Archangel"},
             {artist: "Ben Frost", track: "Stomp"},
             {artist: "Ben Frost", track: "Stomp"},
             {artist: "Burial", track: "Archangel"},
             {artist: "Emeralds", track: "Snores"},
             {artist: "Burial", track: "Archangel"}];

_.countBy(plays, function(song) {
  return [song.artist, song.track].join(" - ");
});

//=> {"Ben Frost - Stomp": 2,
//    "Burial - Archangel": 3,
//    "Emeralds - Snores": 1}
```

The fact that `_.countBy` takes an arbitrary function as its second argument should provide a hint about how you might use `curry2` to build customized functionality. That is, you can curry a useful function with `_.countBy` to implement custom counting functions. In the case of my artist counting activity, I might create a function named `songCount` as follows:

```
function songToString(song) {
  return [song.artist, song.track].join(" - ");
}

var songCount = curry2(_.countBy)(songToString);

songCount(plays);
//=> {"Ben Frost - Stomp": 2,
```

```
//    "Burial - Archangel": 3,
//    "Emeralds - Snores": 1}
```

The use of currying in this way forms a virtual sentence, effectively stating "to implement songCount, countBy songToString." You often see currying in the wild used to build fluent functional interfaces. In this book you'll see the same.

Currying three parameters to implement HTML hex color builders

Using the same pattern of implementation as `curry2`, I can also define a function that curries up to three parameters:

```
function curry3(fun) {
  return function(last) {
    return function(middle) {
      return function(first) {
        return fun(first, middle, last);
      };
    };
  };
};
```

I can use `curry3` in various interesting ways, including using Underscore's `_.uniq` function to build an array of all of the unique songs played:

```
var songsPlayed = curry3(_.uniq)(false)(songToString);

songsPlayed(plays);

//=> [{artist: "Burial", track: "Archangel"},
//    {artist: "Ben Frost", track: "Stomp"},
//    {artist: "Emeralds", track: "Snores"}]
```

By spacing out the call to `curry3` and aligning it with the direct call of `_.uniq`, you might see the relationship between the two more clearly:

```
       _.uniq(plays, false,  songToString);

curry3(_.uniq)     (false) (songToString);
```

In my own adventures, I've used `curry3` as a way to generate HTML hexadecimal values with specific hues. I start with a function `rgbToHexString`, defined as follows:

```
function toHex(n) {
  var hex = n.toString(16);
  return (hex.length < 2) ? [0, hex].join(''): hex;
}

function rgbToHexString(r, g, b) {
  return ["#", toHex(r), toHex(g), toHex(b)].join('');
}
```

```
rgbToHexString(255, 255, 255);
//=> "#ffffff"
```

This function can then be curried to varying depths to achieve specific colors or hues:

```
var blueGreenish = curry3(rgbToHexString)(255)(200);

blueGreenish(0);
//=> "#00c8ff"
```

And that is that.

Currying for Fluent APIs

A tangential benefit of currying is that it very often lead to fluent functional APIs. In the Haskell programming language, functions are curried by default, so libraries naturally take advantage of that fact. In JavaScript, however, functional APIs must be designed to take advantage of currying and must be documented to show how. However, a general-purpose rule when determining if currying is an appropriate tool for any given circumstance is this: does the API utilize higher-order functions? If the answer is yes, then curried functions, at least to one parameter, are appropriate. Take, for example, the `checker` function built in Chapter 4. It indeed accepts a function as an argument used to check the validity of a value. Using curried functions to build a fluent `checker` call is as simple as this:

```
var greaterThan = curry2(function (lhs, rhs) { return lhs > rhs });
var lessThan    = curry2(function (lhs, rhs) { return lhs < rhs });
```

By currying two functions that calculate greater-than and less-than, the curried version can be used directly where `validator` expects a predicate:

```
var withinRange = checker(
  validator("arg must be greater than 10", greaterThan(10)),
  validator("arg must be less than 20", lessThan(20)));
```

This use of curried functions is much easier on the eyes than directly using the anonymous versions of the greater-than and less-than calculations. Of course, the `withinRange` checker works as you might expect:

```
withinRange(15);
//=> []

withinRange(1);
//=> ["arg must be greater than 10"]

withinRange(100);
//=> ["arg must be less than 20"]
```

So as you might agree, the use of curried functions can provide tangible benefits in creating fluent interfaces. The closer your code gets to looking like a description of the

activity that it's performing, the better. I will strive to achieve this condition throughout the course of this book.

The Disadvantages of Currying in JavaScript

While it's nice to provide both `curry2` and `curry3`, perhaps it would be better to provide a function named `curryAll` that curries at an arbitrary depth. In fact, creating such a function is possible, but in my experience it's not very practical. In a programming language like Haskell or Shen, where functions are curried automatically, APIs are built to take advantage of arbitrarily curried functions. That JavaScript allows a variable number of arguments to functions actively works against currying in general and is often confusing. In fact, the Underscore library offers a plethora of different function behaviors based on the type and count of the arguments provided to many of its functions, so currying, while not impossible, must be applied with careful attention.

The use of `curry2` and `curry3` is occasionally useful, and in the presence of an API designed for currying, they can be an elegant approach to functional composition. However, I find it much more common to partially apply functions at arbitrary depths than to curry them, which is what I will discuss next.

Partial Application

You'll recall that I stated, in effect, that a curried function is one that returns a progressively more specific function for each of its given arguments until it runs out of parameters. A partially applied function, on the other hand, is a function that is "partially" executed and is ready for immediate execution given the remainder of its expected arguments, as shown in Figure 5-3.[6]

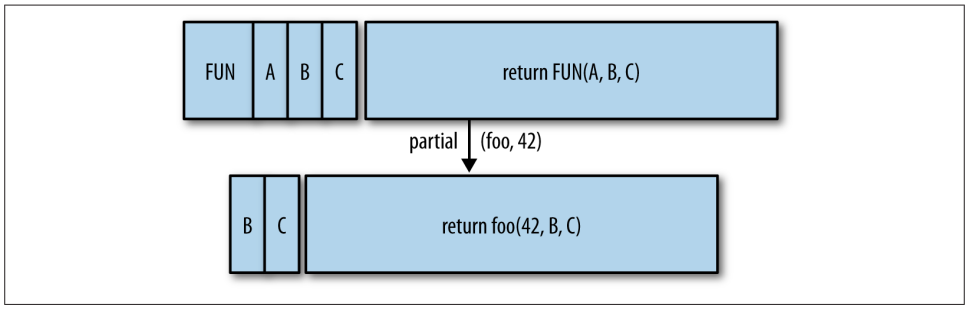

Figure 5-3. An illustration of partial application

6. More recent versions of JavaScript provide a method `Function.bind` that performs partial application (Herman 2012).

Textual descriptions and pictures are nice, but the best way to understand partial application is to see it in action. Imagine a different implementation of `over10`, as shown here:

```
function divPart(n) {
  return function(d) {
    return n / d;
  };
}

var over10Part = divPart(10);
over10Part(2);
//=> 5
```

The implementation of `over10Part` looks almost exactly like the implementation of `leftCurryDiv`, and that fact highlights the relationship between currying and partial application. At the moment that a curried function will accept only one more argument before executing, it is effectively the same as a partially applied function expecting one more argument. However, partial application doesn't necessarily deal with one argument at a time, but instead deals with some number of partially applied arguments stored for later execution, given the remaining arguments.

The relationship between currying and partial application is shown in Figure 5-4.

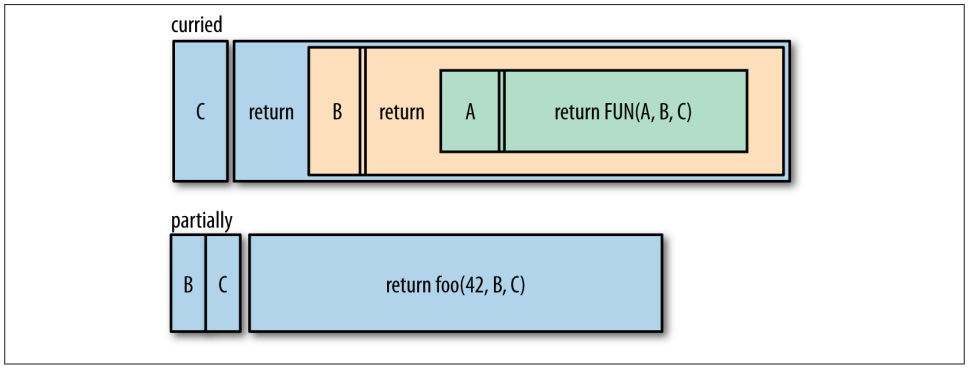

Figure 5-4. The relationship between currying and partial application; the curried function needs three cascading calls (e.g. curried(3)(2)(1)) before FUN runs, whereas the partially applied function is ready to rock, needing only one call of two args (e.g., partially(2, 3))

While currying and partial application are related, they are used quite differently. Never mind that my `curry2` and `curry3` functions work from right to left in the parameter list, although that fact alone would be enough to motivate different API shapes and usage patterns. The main difference with partial application is that it's less confusing in the face of the `varargs` function. JavaScript functions utilizing `varargs` usually directly bind

the first few arguments and reserve the final arguments as optional or behavior specializing. In other words, JavaScript APIs, while allowing any functionality, usually concretely specify a known set of parameters, leading to concrete and default behavior. The use of partial application can take advantage of this, as I'll show next.

Partially Applying One and Two Known Arguments

Like currying, a discussion of partial application is best started simply. A function that partially applies its first argument is written as follows:[7]

```
function partial1(fun, arg1) {
  return function(/* args */) {
    var args = construct(arg1, arguments);
    return fun.apply(fun, args);
  };
}
```

Observe that the function returned from `partial1` captures the argument `arg1` from the original call and puts it at the front of the arglist of the executing call. You can see this operation in action in the following:

```
var over10Part1 = partial1(div, 10);

over10Part1(5);
//=> 2
```

So again, I've re-created the operation of the `over10` function by composing a function from another function and a "configuration" argument.[8] A function to partially apply up to two arguments is implemented similarly:

```
function partial2(fun, arg1, arg2) {
  return function(/* args */) {
    var args = cat([arg1, arg2], arguments);
    return fun.apply(fun, args);
  };
}

var div10By2 = partial2(div, 10, 2)

div10By2()
//=> 5
```

7. You can also use the native `bind` method (if it's available) to implement `partial1` by replacing its body with `return fun.bind(undefined, arg1);`.

8. Likewise, `over10` can be implemented via native `bind`, when available, as `var over10 = div.bind(undefined, 10)`; you can also use the native `bind` method, if it's available, to implement `partial2` by replacing its body with `return fun.bind(undefined, arg1, arg2)`.

Partially applying one or two arguments is typically what you'll see in practice, but it would be useful to instead capture an arbitrary number of arguments for later execution, as I'll explain presently.

Partially Applying an Arbitrary Number of Arguments

Unlike currying, which is complicated by `varargs` in JavaScript, partial application of an arbitrary number of arguments is a legitimate composition strategy. Thankfully, the implementation of a function `partial` is not significantly more complex than either `partial1` nor `partial2`. In fact, the same basic implementation premise applies:

```
function partial(fun /*, pargs */) {
  var pargs = _.rest(arguments);

  return function(/* arguments */) {
    var args = cat(pargs, _.toArray(arguments));
    return fun.apply(fun, args);
  };
}
```

As you might have noticed, the principle is the same: `partial` captures some number of arguments and returns a function that uses them as the prefix arguments for its final call.[9] In action, `partial` works exactly as you might expect:

```
var over10Partial = partial(div, 10);
over10Partial(2);
//=> 5
```

While the presence of varargs in JavaScript does not completely defeat the usefulness of partial application, it can still complicate matters, as shown below:[10]

```
var div10By2By4By5000Partial = partial(div, 10, 2, 4, 5000);
div10By2By4By5000Partial();
//=> 5
```

While you might be aware that a number that you're attempting to partially apply expects a fixed number of arguments, the fact that it will accept any number can at times cause confusion. In fact, the partially applied `div` function is just called one time with the arguments `10` and `2`, and the remaining arguments are simply ignored. Adding partial

9. JavaScript's native bind, when available, allows you to partially apply a function up to any number of arguments. To achieve the same effect as the body of `partial`, you can perform the following: `fun.bind.apply(fun, construct(undefined, args))`.

10. Underscore also has a `partial` function that works just like the one in this chapter. However, Underscore's very nature is that the default argument ordering is not amenable to its use. Where `partial` really shines is in creating new functions from existing functions. Having the collection first, as is the prevailing case in Underscore, eliminates the power potential to specialize higher-order functions by partially applying a modifier function in the first argument position.

application as a level of misdirection only exacerbates the confusion. The good news is that I've rarely run into this problem in practice.

Partial Application in Action: Preconditions

Recall the `validator` function from Chapter 4:

```
validator("arg must be a map", aMap)(42);
//=> false
```

The `validator` higher-order function takes a validation predicate and returns an array of the errors encountered along the way. If the error array is empty, then there were no reported errors. `validator` can also be used for more general purposes, such as the manual validation of arguments to functions:

```
var zero = validator("cannot be zero", function(n) { return 0 === n });
var number = validator("arg must be a number", _.isNumber);

function sqr(n) {
  if (!number(n)) throw new Error(number.message);
  if (zero(n))    throw new Error(zero.message);

  return n * n;
}
```

Calls to the `sqr` function are checked as such:

```
sqr(10);
//=> 100

sqr(0);
// Error: cannot be zero

sqr('');
// Error: arg must be a number
```

This is fairly nice to my eyes, but it can be even better using partial application. While there is certainly a class of errors that fall within the purview of essential data-check failures, there is another set of errors that do not. That is, there is a class of errors that pertains to the guarantees of a computation. In the latter case, you would say that there are two types of guarantees:

Preconditions
: Guarantees on the caller of a function

Postconditions
: Guarantees on the result of a function call, assuming the preconditions were met

In English, the relationship between pre- and postconditions is described as follows: given that you've provided a function data that it can handle, it will ensure that the return meets a specific criteria.

I showed one function—`sqr`—that had two preconditions pertaining to the "numberness" and "zeroness" of its lone argument. We could check these conditions every single time, and that might be fine, but really they refer to a guarantee of `sqr` relative to the context of a running application. Therefore, I can use a new function, `condition1`, and partial application to attach the preconditions separately from essential calculations:

```
function condition1(/* validators */) {
  var validators = _.toArray(arguments);

  return function(fun, arg) {
    var errors = mapcat(function(isValid) {
      return isValid(arg) ? [] : [isValid.message];
    }, validators);

    if (!_.isEmpty(errors))
      throw new Error(errors.join(", "));

    return fun(arg);
  };
}
```

You'll notice that the function returned from `condition1` is meant to take only a single argument. This is done primarily for illustrative purposes, as the `vararg` version is a bit more complicated and obfuscates the point I'm trying to make.[11] The point is that the function returned by `condition1` takes a function and a set of functions, each created with `validator`, and either builds an `Error` or returns the value of the execution of `fun`. This is a very simple but powerful pattern, used as shown here:

```
var sqrPre = condition1(
  validator("arg must not be zero", complement(zero)),
  validator("arg must be a number", _.isNumber));
```

This is a very fluent validation API, as far as JavaScript goes. Very often you'll find that, through function composition, your code becomes more declarative (i.e., it says what it's supposed to do rather than how). A run-through of the operation of `sqrPre` bears out the operation of `condition1`:

```
sqrPre(_.identity, 10);
//=> 10

sqrPre(_.identity, '');
// Error: arg must be a number

sqrPre(_.identity, 0);
// Error: arg must not be zero
```

11. I leave this as an exercise for the reader.

Recalling the definition of `sqr`, with its built-in error handling, you might have guessed how we can use `sqrPre` to check its arguments. If not, then imagine an "unsafe" version of `sqr` defined as follows:

```
function uncheckedSqr(n) { return n * n };

uncheckedSqr('');
//=> 0
```

Clearly, the square of the empty string shouldn't be 0, even if it can be explained by JavaScript's foibles. Thankfully, I've been building a set of tools, realized in the creation of `validator`, `partial1`, `condition1`, and `sqrPre`, to solve this particular problem, shown here:[12]

```
var checkedSqr = partial1(sqrPre, uncheckedSqr);
```

The creation of the new function `checkedSqr` was fully formed through the creation of functions, function-creating functions, and their interplay to build functions anew:

```
checkedSqr(10);
//=> 100

checkedSqr('');
// Error: arg must be a number

checkedSqr(0);
// Error: arg must not be zero
```

As shown in the preceding code, the new `checkedSqr` works exactly like `sqr`, except that by separating the validity checks from the main calculation, I've achieved an ideal level of flexibility. That is, I can now turn off condition checking altogether by not applying conditions to functions at all, or even mix in additional checks at a later time:

```
var sillySquare = partial1(
  condition1(validator("should be even", isEven)),
  checkedSqr);
```

Because the result of `condition1` is a function expecting another function to delegate to, the use of `partial1` combines the two:

```
sillySquare(10);
//=> 100

sillySquare(11);
// Error: should be even

sillySquare('');
// Error: arg must be a number
```

12. I could have used `partial` instead of `partial1` in this example, but sometimes I like more explicitness in my code.

```
sillySquare(0);
// Error: arg must not be zero
```

Now obviously you wouldn't want to constrain the squaring of numbers to such a silly degree, but I hope the point is clear. The functions that compose other functions should themselves compose. Before moving on to the next section, it's worth taking a step back and seeing how to re-implement the command object (from Chapter 4) creation logic with validation:

```
var validateCommand = condition1(
  validator("arg must be a map", _.isObject),
  validator("arg must have the correct keys", hasKeys('msg', 'type')));

var createCommand = partial(validateCommand, _.identity);
```

Why use the `_.identity` function as the logic part of the `createCommand` function? In JavaScript, much of the safety that we achieve is built via discipline and careful thinking. In the case of `createCommand`, the intention is to provide a common gateway function used for creating and validating command objects, as shown below:

```
createCommand({});
// Error: arg must have right keys

createCommand(21);
// Error: arg must be a map, arg must have right keys

createCommand({msg: "", type: ""});
//=> {msg: "", type: ""}
```

However, using functional composition allows you to later build on top of the existing creation abstraction in order to customize the actual building logic or the validation itself. If you wanted to build a derived command type that required the existence of another key, then you would further compose with the following:

```
var createLaunchCommand = partial1(
  condition1(
    validator("arg must have the count down", hasKeys('countDown'))),
  createCommand);
```

And as you might expect, `createLaunchCommand` works as follows:

```
createCommand({msg: "", type: ""});
// Error: arg must have the count down

createCommand({msg: "", type: "", countDown: 10});
//=> {msg: "", type: "", countDown: 10}
```

Whether you use currying or partial application to build functions, there is a common limitation on both: they only compose based on the specialization of one or more of their arguments. However, it's conceivable that you might want to compose functions based on the relationships between their arguments and their return values. In the next

section, I will talk about a `compose` function that allows the end-to-end stitching of functions.

Stitching Functions End-to-End with Compose

An idealized (i.e., not one that you're likely to see in production) functional program is a pipeline of functions fed a piece of data in one end and emitting a whole new piece of data at the other. In fact, JavaScript programmers do this all the time. Observe:

```
!_.isString(name)
```

The pipeline in play here is built from the function `_.isString` and the `!` operator, where:

- `_.isString` expects an object and returns a Boolean value
- `!` expects a Boolean value (in principle) and returns a Boolean

Functional composition takes advantage of this type of data chain by building new functions from multiple functions and their data transformations along the way:

```
function isntString(str) {
  return !_.isString(str);
}

isntString(1);
//=> true
```

But this same function can be built from function composition, using the Underscore function `_.compose` as follows:

```
var isntString = _.compose(function(x) { return !x }, _.isString);

isntString([]);
//=> true
```

The `_.compose` function works from right to left in the way that the resulting function executes. That is, the result of the rightmost functions are fed into the functions to their left, one by one. Using selective spacing, you can see how this maps to the original:

```
                                    !       _.isString("a");

_.compose(function(str) { return !str }, _.isString)("a");
```

In fact, the `!` operator is useful enough to encapsulate it into its own function:

```
function not(x) { return !x }
```

The `not` function then composes as you'd expect:

```
var isntString = _.compose(not, _.isString);
```

Using composition this way effectively turns a string into a Boolean value without explicitly changing either one—a worthy result indeed. This model for composition can form the basis for entire function suites where primitive data transformers are plugged together like Lego blocks to build other functionality.

A function that I've already defined, `mapcat`, can be defined using `_.compose` in the following way:[13]

```
var composedMapcat = _.compose(splat(cat), _.map);

composedMapcat([[1,2],[3,4],[5]], _.identity);
//=> [1, 2, 3, 4, 5]
```

There are infinite ways to compose functions to form further functionality, one of which I'll show presently.

Pre- and Postconditions Using Composition

If you recall from the previous section, I mentioned that preconditions define the constraints under which a function's operation will produce a value adhering to a different set of constraints. These production constraints are called postconditions. Using `condition1` and `partial`, I was able to build a function (`checkedSqr`) that checked the input arguments to `uncheckedSqr` for conformance to its preconditions. However, if I want to define the postconditions of the act of squaring, then I need to define them using `condition1` as such:

```
var sqrPost = condition1(
  validator("result should be a number", _.isNumber),
  validator("result should not be zero", complement(zero)),
  validator("result should be positive", greaterThan(0)));
```

I can run through each error case manually using the following:

```
sqrPost(_.identity, 0);
// Error: result should not be zero, result should be positive

sqrPost(_.identity, -1);
// Error: result should be positive

sqrPost(_.identity, '');
// Error: result should be a number, result should be positive

sqrPost(_.identity, 100);
//=> 100
```

13. I defined `splat` way back in Chapter 1.

But the question arises: how can I glue the postcondition `check` function onto the existing `uncheckedSqr` and `sqrPre`? The answer, of course, is to use `_.compose` for the glue: [14]

```
var megaCheckedSqr = _.compose(partial(sqrPost, _.identity), checkedSqr);
```

And its use is exactly the same as `checkedSqr`:

```
megaCheckedSqr(10);
//=> 100

megaCheckedSqr(0);
// Error: arg must not be zero
```

Except:

```
megaCheckedSqr(NaN);
// Error: result should be positive
```

Of course, if the function ever throws a postcondition error, then that means that either my preconditions are under-specified, my postconditions are over-specified, or my internal logic is busted. As the provider of a function, a post-condition failure is *always* my fault.

Summary

In this chapter, I worked through the idea that new functions can be built from existing functions, be they generic or special-purpose. The first phase of composition is done manually by just calling one function after another, then wrapping the calls in another function. However, using specialized composition functions was often easier to read and reason about.

The first composition function covered was `_.curry`, which took a function and some number of arguments and returned a function with the rightmost arguments fixed to those given. Because of the nature of JavaScript, which allows a variable number of arguments, a few static currying functions—`curry` and `curry2`—were used to create functions of known parameter sizes to a known number of curried arguments. In addition to introducing currying, I implemented a few interesting functions using the technique.

The second composition function covered was `partial`, which took a function and some number of arguments and returned a function that fixed the leftmost arguments to those given. Partial application via `partial`, `partial1`, and `partial2` proved a much more broadly applicable technique than currying.

14. Another option is to rewrite `condition1` to work with an intermediate object type named `Either` that holds *either* the resulting value or an error string.

The final composition function covered was `_.compose`, which took some number of functions and strung them end to end from the rightmost to the leftmost. The `_.compose` higher-order function was used to build on the lessons learned from Chapter 4's implementation of `checker` to provide a pre- and postcondition function "decorator," using a surprisingly small amount of code.

The next chapter is again a transition chapter covering a topic not very prevalent in JavaScript, though more so in functional programming in general: recursion.