Functional abstraction


Language level: Intermediate Student With Lambda

Topics:

- Functions are first class values
- Contracts and types
- Anonymous functions
- Syntax & semantics
- Abstracting from examples
- Higher-order functions
What is abstraction?

Abstraction is the process of finding similarities or common aspects, and forgetting unimportant differences.

Example: writing a function.

The differences in parameter values are forgotten, and the similarity is captured in the function body.

We have seen many similarities between functions, and captured them in design recipes.

But some similarities still elude us.
Eating apples

(define (eat-apples lst)
  (cond [(empty? lst) empty]
        [(not (symbol= (first lst) 'apple))
         (cons (first lst) (eat-apples (rest lst)))]
        [else (eat-apples (rest lst))])))
Keeping odd numbers

(define (keep-odds lst)
  (cond [(empty? lst) empty]
        [(odd? (first lst))
         (cons (first lst) (keep-odds (rest lst)))]
        [else (keep-odds (rest lst))])))
Abstracting from these examples

*Key Point:* What these two functions have in common is their *general structure*.

Where they differ is in the specific predicate used to decide whether an item is removed from the answer or not.

We could write one function to do both these tasks if we could supply, as an argument to that function, the predicate to be used.

The Intermediate language permits this.
Functions as first-class values

In the Intermediate language, functions are values. In fact, functions are first-class values.

Functions have the same status as the other values we’ve seen. They can be:

1. *consumed* as function arguments
2. *produced* as function results
3. *bound* to identifiers
4. *put in structures and lists*
Functions as first-class values has historically been missing from languages that are not primarily functional.

The utility of functions-as-values is now widely recognized, and they are at least partially supported in many languages that are not primarily functional, including C++, C#, Java, Go, JavaScript, Python, and Ruby.

Functions-as-values provides a clean way to think about the concepts and issues involved in abstraction.

You can then worry about how to implement a high-level design in a given programming language.
Consuming functions

\[(\text{define} \ (\text{foo} \ f \ x \ y) \ (f \ x \ y))\]

\[(\text{foo} \ + \ 2 \ 3) \Rightarrow 5\]

\[(\text{foo} \ \ast \ 2 \ 3) \Rightarrow 6\]

The expression \((\text{foo} \ + \ 2 \ 3)\) is not only passing in 2 and 3 as arguments to \text{foo}, it is also passing in the function \(+\) as an argument.
my-filter

(define (my-filter pred? lst)
  (cond [(empty? lst) empty]
        [(pred? (first lst))
         (cons (first lst) (my-filter pred? (rest lst)))]
        [else (my-filter pred? (rest lst))])))

If pred? is true then cons it to the answer.
Otherwise skip that element and apply pred? to the rest of the list.
I.e. keep the items in the list where pred? produces true.
Tracing my-filter

\[(\text{my-filter odd? (list 5 6 7)})\]
\[\Rightarrow (\text{cons 5 (my-filter odd? (list 6 7))})\]
\[\Rightarrow (\text{cons 5 (my-filter odd? (list 7))})\]
\[\Rightarrow (\text{cons 5 (cons 7 (my-filter odd? empty))})\]
\[\Rightarrow (\text{cons 5 (cons 7 empty))}\]

my-filter is an **abstract list function** which handles the general operation of removing items from lists.
Using my-filter

(define (keep-odds lst) (my-filter odd? lst))

(define (not-symbol-apple? item) (not (symbol = ? item 'apple)))
(define (eat-apples lst) (my-filter not-symbol-apple? lst))

The function filter, which behaves identically to our my-filter, is built into Intermediate Student and full Racket.

filter and other abstract list functions provided in Racket are used to apply common patterns of structural recursion.

We’ll discuss how to write contracts for them shortly.
Advantages of functional abstraction

Functional abstraction is the process of creating abstract functions such as filter.

More specifically functional abstraction is the process of combining two or more related functions into a single definition.

It reduces code size.

It avoids cut-and-paste.

Bugs can be fixed in one place instead of many.

Improving one functional abstraction improves many applications.
Producing functions

We saw in lecture module 09 how local could be used to create functions during a computation, to be used in evaluating the body of the local.

*Key Point*: But now, because functions are values, the body of the local can produce such a function as a value.

Though it is not apparent at first, this is enormously useful.

We illustrate with a very small example.
(define (make-adder n)
  (local
    [(define (f m) (+ n m))]
    f))

What is (make-adder 3)?

We can answer this question with a trace.
(make-adder 3) ⇒
(local [(define (f m) (+ 3 m))] f) ⇒
(define (f_42 m) (+ 3 m)) f_42

(make-adder 3) is the renamed function f_42, which is a function that adds 3 to its argument.

We can apply this function immediately, or we can use it in another expression, or we can put it in a data structure.
Here’s what happens if we apply it immediately.

\[
((\text{make-adder } 3) \ 4) \Rightarrow \\
((\text{local } [(\text{define } (f \ m) \ (+ \ 3 \ m)) \ f]) \ 4) \Rightarrow \\
(\text{define } (f_{42} \ m) \ (+ \ 3 \ m)) \ (f_{42} \ 4) \Rightarrow \\
(+ \ 3 \ 4) \Rightarrow 7
\]
A note on scope:

\[
\text{(define (add3 m)}
\begin{align*}
&\text{(+ 3 m))} \\
\text{(define (make-adder n)}
\begin{align*}
&\text{(local [(define (f m) (\quad (\quad + n m))]} \\
&\quad f)}
\end{align*}
\end{align*}
\]

In \text{add3} the parameter \text{m} is of no consequence after \text{add3} is applied. Once \text{add3} produces its value, \text{m} can be safely forgotten.

However, our earlier trace of \text{make-adder} shows that after it is applied the parameter \text{n} does have a consequence. It is embedded into the result, \text{f}, where it is “remembered” and used again, potentially many times.
Binding functions to identifiers

The result of `make-adder` can be bound to an identifier and then used repeatedly.

```scheme
(define add2 (make-adder 2))
(define add3 (make-adder 3))
```

```scheme
(add2 3) ⇒ 5
(add3 10) ⇒ 13
(add3 13) ⇒ 16
```
How does this work?

\[
\text{(define add2 (make-adder 2)) } \Rightarrow \\
\text{(define add2 (local [(define (f m) (+ 2 m))] f)) } \Rightarrow \\
\text{(define (f m) (+ 2 m)) } \; \text{rename and lift out f} \\
\text{(define add2 f_43)}
\]

\[
\text{(add2 3) } \Rightarrow \\
\text{(f_43 3) } \Rightarrow \\
\text{(+ 2 3) } \Rightarrow \\
5
\]
Putting functions in lists

Recall our code in lecture module 08 for evaluating alternate arithmetic expressions such as ’(+(∗ 3 4) 2).

;; eval: AltAExp → Num
(define (eval aax)
  (cond [(number? aax) aax]
        [else (my-apply (first aax) (rest aax))])))
;; my-apply: Sym AltAExpList → Num

(define (my-apply f aaxl)
  (cond
   [(and (empty? aaxl) (symbol=? f '∗)) 1]
   [(and (empty? aaxl) (symbol=? f '＋)) 0]
   [(symbol=? f '∗)
    (∗ (eval (first aaxl)) (my-apply f (rest aaxl)))]
   [(symbol=? f '＋)
    (+ (eval (first aaxl)) (my-apply f (rest aaxl)))]))

Note the similar-looking code.
Much of the code is concerned with translating the symbol ‘+’ into the function +, and the same for ‘∗’ and ∗.

If we want to add more functions to the evaluator, we have to write more code which is very similar to what we’ve already written.

We can use an association list to store the above correspondence, and use the function lookup-al we saw in lecture module 06 to look up symbols.
(define trans-table (list (list ' + + )
 (list ' ∗ ∗ )))

Now (lookup-al ' + trans-table) produces the function +.

((lookup-al ' + trans-table) 3 4 5) ⇒ 12
;; newapply: Sym AltAExpList → Num
(define (newapply f aaxl)
  (cond
   [(and (empty? aaxl) (symbol = ? f '∗)) 1]
   [(and (empty? aaxl) (symbol = ? f '＋)) 0]
   [else ((lookup-al f trans-table)
     (eval (first aaxl)) (newapply f (rest aaxl))))])))

We can simplify this even further, because in Intermediate Student, + and ∗ allow zero arguments.

(+) ⇒ 0 and (∗) ⇒ 1
(define (newapply f aaxl)
  (local [(define op (lookup-al f trans-table))]
    (cond [(empty? aaxl) (op)]
          [else (op (eval (first aaxl))
                    (newapply f (rest aaxl)))])))

Now, to add a new binary function (that is also defined for 0 arguments), we need only add one line to trans-table.