Today’s topics
- Types of objects and procedures
- Procedural abstractions
- Capturing patterns across procedures – Higher Order Procedures

Types

\( (+ 5 10) \Rightarrow 15 \)

\( (+ \text{"hi"} 5) \)
- The object ‘hi’, passed as the first argument to integer-add, is not the correct type
  - Addition is not defined for strings

Types – simple data
- We want to collect a taxonomy of expression types:
  - Simple Data
    - Number
    - Integer
    - Real
    - Rational
    - String
    - Boolean
    - Names (symbols)
- We will use this for notational purposes, to reason about our code. Scheme checks types of arguments for built-in procedures, but not for user-defined ones.

Types – compound data
- Pair\(<A,B>\>
  - A compound data structure formed by a cons pair, in which the first element is of type \(A\), and the second of type \(B\): e.g. \((\text{cons} 1 2)\) has type \(\text{Pair}<\text{number}, \text{number}>\)
- List\(<A>\)=\(\text{Pair}<A, \text{List}<A>\text{ or nil}>\)
  - A compound data structure that is recursively defined as a pair, whose first element is of type \(A\), and whose second element is either a list of type \(A\) or the empty list.
    - E.g. \((\text{list} 1 2 3)\) has type \(\text{List}<\text{number}>\); while \((\text{list} 1 \text{"string"} 3)\) has type \(\text{List}<\text{number or string}>\)

Examples

25 ; Number
3.45 ; Number
"this is a string" ; String
(> a b) ; Boolean
(cons 1 3) ; Pair<Number, Number>
(list 1 2 3) ; List<Number>
(cons "foo" (cons "bar" nil)) ; List<String>

Types – procedures
- Because procedures operate on objects and return values, we can define their types as well.
- We will denote a procedures type by indicating the types of each of its arguments, and the type of the returned value, plus the symbol \(\Rightarrow\) to indicate that the arguments are mapped to the return value.
  - E.g. number \(\Rightarrow\) number specifies a procedure that takes a number as input, and returns a number as value
Types

• \((+ 5 10) \Rightarrow 15\)
• \((+ "\text{hi}" 5)\)
• Addition is not defined for strings
• The type of the integer-add procedure is \(\text{number, number} \Rightarrow \text{number}\)

Type examples

• expression: evaluates to a value of type:
  15 \(\text{number}\)
  "hi" \(\text{string}\)
  \(\text{square}\) \(\text{number} \Rightarrow \text{number}\)
  \(>\) \(\text{number, number} \Rightarrow \text{boolean}\)
  \((> 5 4) \Rightarrow \#t\)

• The type of a procedure is a contract:
  • If the operands have the specified types, the procedure will result in a value of the specified type
  • Otherwise, its behavior is undefined
  • May be an error, maybe random behavior

Types, precisely

• A type describes a set of scheme values
  • \(\text{number} \Rightarrow \text{number}\) describes the set:
    • all procedures, whose result is a number, which require one argument that must be a number

• Every scheme value has a type
  • Some values can be described by multiple types
  • If so, choose the type which describes the largest set
  • Special form keywords like \texttt{define} do not name values
  • Therefore special form keywords have no type

Your turn

• The following expressions evaluate to values of what type?
  \((\lambda (a b c) (\text{if} (> a 0) (+ b c) (- b c)))\)
  \(\text{number, number, number} \Rightarrow \text{number}\)
  \((\lambda (p) (\text{if} p "\text{hi}" "\text{bye}"))\)
  \(\text{Boolean} \Rightarrow \text{string}\)
  \((* 3.14 (* 2 5))\)
  \(\text{number}\)

End of part 1

• type: a set of values
  • every value has a type
  • procedure types (types which include \(\Rightarrow\)) indicate
    • number of arguments required
    • type of each argument
    • type of result of the procedure
• Types: a mathematical theory for reasoning efficiently about programs
  • useful for preventing certain common types of errors
  • basis for many analysis and optimization algorithms
What is procedure abstraction?

Capture a common pattern
(* 2 2)
(* 57 57)
(* k k)

(lambdas (x) (* x x))

Formal parameter for pattern
Actual pattern

Give it a name (define square (lambdas (x) (* x x)))

Note the type: number → number

Other common patterns

• 1 + 2 + … + 100 = (100 * 101)/2
• 1 + 4 + 9 + … + 100² = (100 * 101 * 201)/6
• 1 + 1/3² + 1/5² + … + 1/101² = π²/6

(define (sum-integers a b)
  (if (> a b)
    0
    (+ a (sum-integers (+ 1 a) b))))

(define (sum-squares a b)
  (if (> a b)
    0
    (+ (square a)
      (sum-squares (+ 1 a) b))))

(define (pi-sum a b)
  (if (> a b)
    0
    (+ (/ 1 (square a))
      (pi-sum (+ a 2) b))))

A quick sidebar

(define (sum-integers a b)
  (if (> a b)
    0
    (+ a (sum-integers (+ 1 a) b))))

This is the same as:

(define sum-integers
  (lambdas (a b)
    (if (> a b)
      0
      (+ a (sum-integers (+ 1 a) b)))))

Other common patterns

• 1 + 2 + … + 100 = (100 * 101)/2
• 1 + 4 + 9 + … + 100² = (100 * 101 * 201)/6
• 1 + 1/3² + 1/5² + … + 1/101² = π²/6

(define (sum-integers a b)
  (if (> a b)
    0
    (+ (sum-integers (+ 1 a) b)))))

(define (sum-squares a b)
  (if (> a b)
    0
    (+ (square a)
      (sum-squares (+ 1 a) b)))))

(define (pi-sum a b)
  (if (> a b)
    0
    (+ (/ 1 (square a))
      (pi-sum (+ a 2) b)))))

Let's check this new procedure out!

(define (sum term a next b)
  (if (> a b)
    0
    (+ (term a)
      (sum term (next a) next b))))

What is the type of this procedure?

(procedure number number number number number number number number number → number)

Higher order procedures

• A higher order procedure:
takes a procedure as an argument or returns one as a value

(define (sum-integers1 a b)
  (sum (lambda (x) a) (lambda (x) (+ x 1)) b))

(define (sum-squares1 a b)
  (sum square a (lambda (x) (+ x 1)) b))

(define (pi-sum1 a b)
  (sum (lambda x (/ 1 (square x))) a
       (lambda x (+ x 2)) b))

6.001 SICP 18/41
Be careful

\[
(\text{define } (\text{sum } \text{term } a \ \text{next } b) \\
(\text{if } (> a b) \\
0 \\
( (+ (\text{term } a) \\
(\text{sum } \text{term } (\text{next } a) \ \text{next } b))))))
\]

\[
(\text{define } (\text{sum-integers } a \ b) \\
(\text{if } (> a b) \\
0 \\
( (+ (\text{sum-integers } (+ 1 a) \ b) ))))
\]

Let's check to be sure

\[
(\text{sum-integers } 1 \ 4 \\
(\text{sum } (\lambda (x) x) \ 1 \ (\lambda (x) (+ x 1)) \ 4) \\
(\text{if } (> 1 4) \\
0 \\
(+ (\lambda (x) x) 1) \\
(\text{sum } (\lambda (x) x) \\
(\lambda (x) (+ x 1)) \\
4)))
\]

\[
(+ 1 \ (\text{sum } (\lambda (x) x) \ 2 \ (\lambda (x) (+ x 1)) \ 4)) \\
(+ 1 \ (+ 2 \ (\text{sum } (\lambda (x) x) \ 3 \ (\lambda (x) (+ x 1)) \ 4)))
\]

Computing derivatives

\[
f : x \rightarrow x^2 \\
Df : x \rightarrow 2x
\]

\[
f : x \rightarrow x^3 \\
Df : x \rightarrow 3x^2
\]

We can easily write \( f \) in either case:

\[
(\text{define } f (\lambda (x) (* x x x)))
\]

But what is \( D \)??

\[
\begin{align*}
\text{maps a function (or procedure) to a different function} \\
\text{here is a good approximation:}
\end{align*}
\]

\[
Df (x) \approx \frac{f(x + \epsilon) - f(x)}{\epsilon}
\]

Using “deriv”

\[
(\text{define deriv} \\
(\lambda (f) \\
(\lambda (x) (/ (- (f (+ x epsilon)) (f x)) epsilon))) ساعت)\\n(\text{define square} (\lambda (y) (* y y)) )
\]

\[
(\text{define deriv} \\
(\lambda (f) \\
(\lambda (x) (/ (- (f (+ x epsilon)) (f x)) epsilon))) ساعت)
\]

\[
5
\]

\[
(\text{use } (\lambda (y) (* y y)) (+ 5 epsilon))\\n(\text{use } (\lambda (y) (* y y)) 5)
\]

10.001
deriv composes

\[ \text{deriv} \text{ composes} \]
\[
\begin{align*}
\text{(define (cube x) \(* x x x\))} \\
\text{(define epsilon 0.001)} \\
\text{(define (deriv f)} \\
\text{\( (\lambda (x) \frac{(- (f (+ x \text{epsilon}))}{\text{epsilon}}) \))} \\
\text{(deriv cube) 3))} \\
\text{result: 27.009000999996147} \\
\text{(deriv (deriv cube)) 3) \rightarrow 18.006000004788802
\end{align*}
\]

Common Pattern #1: Transforming a List

\[ \text{Common Pattern #1: Transforming a List} \]
\[
\begin{align*}
\text{(define (square-list lst)} \\
\text{\( (\text{if (null? lst)} \text{nil)} \text{\=(cons (square (first lst)) (square-list (rest lst)))}) \))} \\
\text{(define (double-list lst)} \\
\text{\( (\text{if (null? lst)} \text{nil)} \text{\=(cons (* 2 (first lst)) (double-list (rest lst)))}) \))} \\
\text{(define (MAP proc lst)} \\
\text{\( (\text{if (null? lst)} \text{nil)} \text{\=(cons (proc (first lst)) (map proc (rest lst))))}) \)) \\
\text{(define (square-list lst)} \\
\text{\( (\text{map square lst}) \))} \\
\text{(define (double-list lst)} \\
\text{\( (\text{map (lambda (x) \(* 2 x\)) lst}) \))}
\end{align*}
\]
Transforms a list to a list, replacing each element by the procedure applied to that value

Common Pattern #2: Filtering a List

\[ \text{Common Pattern #2: Filtering a List} \]
\[
\begin{align*}
\text{(define (filter pred lst)} \\
\text{\( (\text{cond ((null? lst) nil)} \text{\=(cons (first lst) (filter pred (rest lst))))}) \)) \\
\text{(define (filter even? (list 1 2 3 4 5 6)) \text{result: (2 4 6})}
\end{align*}
\]
Using common patterns over data structures

• We can more compactly capture our earlier ideas about common patterns using these general procedures.
• Suppose we want to compute a particular kind of summation:

\[
\sum_{i=0}^{n} f(a+i\delta) = f(a) + f(a+\delta) + f(a+2\delta) + \ldots + f(a+n\delta)
\]

Using common patterns over data structures

\[ \text{Using common patterns over data structures} \]
\[
\begin{align*}
\text{(define (generate-interval a b)} \\
\text{\( (\text{if (> a b)} \text{nil)} \text{\=(cons a (generate-interval (+ 1 a) b)))}) \)) \\
\text{(define (sum f start inc terms)} \\
\text{\( (\text{fold-right + 0)} \text{\=(cons f (fold-right + 0 \((\text{map (lambda (x) \((+ start (* x inc)))}) \text{(generate-interval 0 terms))}))}) \))}
\end{align*}
\]
Integration as a procedure

Integration under a curve is given roughly by
\[ \int_a^b f(x) \, dx \approx f(a) + f(a + dx) + f(a + 2dx) + \ldots + f(b) \]

```
(define (integral f a b n)
  (let ((delta (/ (- b a) n)))
    (* (sum f a delta n) delta)))
```

```
(define atan (lambda (a)
  (integral (lambda (x) (/ 1 (+ 1 (square x)))) 0 a)))
```

Finding fixed points of functions

Square root of \( x \) is defined by \( \sqrt{x} = x/\sqrt{x} \)

Think of as a transformation \( f : y \rightarrow \frac{1}{y} \) then if we can find a \( y = \sqrt{x} \), then \( f(y) = y \), and such a \( y \) is called a fixed point of \( f \).

- Here’s a common way of finding fixed points
  - Given a guess \( x_1 \), let new guess by \( f(x_1) \)
  - Keep computing \( f \) of last guess, till close enough

```
(define (close? u v)   (< (abs (- u v)) 0.0001))
(define (fixed-point f i-guess)
  ;; (number->number, number) -> number
  (define (try g)
    ;; number -> number
    (if (close? (f g) g)
        (f g)
        (try (f g))));
  (try i-guess))
```

```
(fixed-point (lambda (x) (+ 1 (/ 1 x))) 1)  \rightarrow  1.6180
```

or \( x = 1 + 1/x \) when \( x = (1 + \sqrt{5})/2 \)

```
(define (sqrt x)
  (fixed-point
   (lambda (y) (/ x y))
   1))
```

Unfortunately if we try \( \sqrt{2} \), this oscillates between 1, 2, 1, 2, ...

Using fixed points

```
(fixed-point (lambda (x) (+ 1 (/ 1 x))) 1)  \rightarrow  1.6180
or x = 1 + 1/x when x = (1 +\sqrt{5})/2
```

```
(define (sqrt x)
  (fixed-point
   (lambda (y) (/ x y))
   1))
```

Unfortunately if we try \( \sqrt{2} \), this oscillates between 1, 2, 1, 2, ...

... which gives us a clean version of sqrt

```
(define (sqrt x)
  (fixed-point
   (average-damp
    (lambda (y) (/ x y)))
   1))
```

Compare this to Heron’s algorithm in textbook – same process, but ideas intertwined with code

```
(define (cbrt x)
  (fixed-point
   (average-damp
    (lambda (y) (/ x (square y))))
   1))
```

So damp out the oscillation

```
(define average-damp
  (lambda (f)
    (lambda (x)
      (average x (f x))))))
```

Check out the type:

```
((average-damp square) 10)
```

```
((lambda (x) (average x (square x))) 10)
```

```
*average 10 (square 10))
```

Higher order procedures

- A higher order procedure:
  takes a procedure as an argument or returns one as a value

```
(define hopl (lambda (f x) (+ 2 (f (+ x 1))))
  (hopl square 3)
  (+ 2 (square (+ 3 1))
  (+ 2 (square 4)
  (+ 2 (* 4 4))
  (+ 2 16)
  18
  (hopl (lambda (x) (* x x)) 3)
  ...
  18
```

... which gives us a clean version of sqrt
Type of hop1

\[
\text{hop1} = \lambda f \, x. (+ 2 \, (f \, (+ \, x \, 1)))
\]

- \((\text{number} \rightarrow \text{number}), \text{number} \rightarrow \text{number}\)
- 1st arg must be a procedure
- 2nd arg must be a number
- result is a number

A more interesting higher-order procedure

\[
\text{compose} = \lambda f \, g. \lambda x. (f \, (g \, x))
\]

\[
(\text{compose} \, \text{square} \, \text{double} \, 3)
\]

\[
(\text{square} \, (\text{double} \, 3))
\]

\[
(\text{square} \, 6)
\]

\[
(* \, 6 \, 6)
\]

36

What is the type of compose? Is it:

\((\text{number} \rightarrow \text{number}), \text{number} \rightarrow \text{number} \rightarrow \text{number}\)

No! Nothing in compose requires a number

Compose works on other types too

\[
\text{compose} = \lambda f \, g. \lambda x. (f \, (g \, x))
\]

\[
((\text{compose} \, \lambda \, p. \, \text{if} \, p \, \text{"hi"} \, \text{"bye")})\, \lambda \, x. \, (\lt \, x \, 0))\]

\[-5 \] ➞ “bye”

result: a string

Will any call to compose work? No!

\[
(((\text{compose} \, \lt \, \text{square}) \, 5) \] wrong number of args to \lt

\[
\lt: \text{number}, \text{number} \rightarrow \text{boolean}
\]

\[
((\text{compose} \, \text{square} \, \text{double}) \, \text{"hi")})
\]

wrong type of arg to double

\[
\text{double: number} \rightarrow \text{number}
\]

Type of compose

\[
\text{compose} = \lambda f \, g. \lambda x. (f \, (g \, x))
\]

- Use type variables.
- \(\text{compose:} \, (A \rightarrow B), \, (C \rightarrow A) \rightarrow (C \rightarrow B)\)

- Meaning of type variables:
  - All places where a given type variable appears must match when you fill in the actual operand types

- The constraints are:
  - \(F\) and \(G\) must be functions of one argument
  - the argument type of \(G\) matches the type of \(X, C\)
  - the argument type of \(F\) matches the result type of \(G, A\)
  - the result type of \(\text{compose}\) is the function that maps the type of \(X, C, \) to the result type of \(F, B: C \rightarrow B\)

Higher order procedures

- Procedures may be passed in as arguments
- Procedures may be returned as values
- Procedures may be used as parts of data structures
  - E.g., \((\text{cons} \, \lambda \, x. \, x)\) is perfectly legit
- Procedures are first class objects in Scheme!!