Types, precisely

• A type describes a set of scheme values
  • number 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 define do not name values
  • therefore special form keywords have no type

Procedure Types

• procedure types (types which include ) 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

Example Abstraction using Higher-order procedures

Basic idea: use procedural abstraction to capture regular patterns:

\[
\begin{align*}
&(* 2 8) & (* 3 8) \\
&(* 2 29) & (* 3 29) \\
&(* 2 55) & (* 3 55) \\
&(double 8) & (triple 8) \\
&(double 29) & (triple 29) \\
&(double 55) & (triple 55) \\
\end{align*}
\]

We use Generic Type Variables when any type is possible:

The correct type of identity is:

\[
A \rightarrow A
\]
Example Abstraction using Higher-order procedures

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

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

\[
\text{(define double \( \text{make-mult 2} \))}
\]

\[
\text{(define triple \( \text{make-mult 3} \))}
\]

\[
\text{(define make-mult}
\]
\[
\quad \lambda (n)
\]
\[
\quad \lambda (x) (* n x))
\]

Example Abstraction using Higher-order procedures

Define a procedure \textit{swap} that takes a single argument whose type is a function that takes two arguments, and returns a function that acts as the input function but with its argument positions swapped.

E.g., \((- 5 4) \& 1\), \((\text{swap } -) 5 4 ) \& -1\).

\[
\text{(define swap \( \lambda (f) \))}
\]
\[
\quad \lambda (x y)
\]
\[
\quad f y x)
\]

Define a composing function

Write a procedure \textit{composef} that takes two arguments (e.g. \(f\) and \(g\)), whose type is a function, and returns a function that first applies \(g\) to the input, then \(f\) to the result – i.e. \((\text{composef } f \ g)\ x\) should be the same as \(f\ (g\ x)\).

\[
\text{(define composef}
\]
\[
\quad \lambda (f g)
\]
\[
\quad \lambda (x)
\]
\[
\quad (f (g x)))
\]

\[
\text{(composef double cube) 3}\]

\[
(((\lambda (f g) (\lambda (x) (f (g x)))))\ double cube) 3\]

;; if double and cube were primitive:

\[
\text{((\lambda (x) (double (cube x))) 3)}
\]

\[
\text{(double (cube 3))}
\]

\[
54
\]

composef example

\[
\text{(define composef}
\]
\[
\quad \lambda (f g)
\]
\[
\quad \lambda (x)
\]
\[
\quad (f (g x)))
\]

\[
\text{(composef double cube) 3}\]

\[
(((\lambda (f g) (\lambda (x) (f (g x)))))\ double cube) 3\]

;; now expand double and cube:

\[
\text{(((\lambda (f g) (\lambda (x) (f (g x))))\ double cube) 3)}
\]

;; now turn the crank! apply the composef lambda to the double and cube lambdas:

\[
\text{((\lambda (y) (double (cube y))) 3)}
\]

;; continue on next slide

composef example

\[
\text{((\lambda (y) (double (cube y))) 3)}
\]

;; note how confusing it is to have many different lambdas all with the same parameter name "\(x\)"

;; we can use the "lambda parameter renaming trick", and rename some of the \(x\)'s to unused names when we copy in \textit{double} and \textit{cube}:

\[
\text{(((\lambda (f g) (\lambda (x) (f (g x))))\ double cube) 3)}
\]

;; now turn the crank! apply the composef lambda to the double and cube lambdas:

;; here's the body of composef lambda: \((\lambda (f g) (\lambda (x)) \text{ if } (g x)\))

;; \(f\) and \(g\) are replaced with the \textit{double} and \textit{cube} definitions, so

\[
\text{((\lambda (y) (\lambda (z) (* z z))) 3)}
\]

;; now compute the composef lambda:

\[
\text{((\lambda (y) (\lambda (z) (* z z))) 3)}
\]

\[
\text{((\lambda (y) (\lambda (z) (* z z))) 27)}
\]

\[
54
\]

;; continue on next slide
Type of composef

(define composef (lambda (f g)
    (lambda (x) (f (g x)))))

(define myfunc (composef square double))
(myfunc 3) ==> 36

The value of a lambda expression is a procedure
The body of composef is a lambda expression
Therefore, the result value of composef is a procedure

Type of composef

The result value of composef is a procedure
composef: ?_ → (?→?)
It has two arguments, both procedures
composef: (?→?), (?→?) → (?→?)
The first argument is the function f(), the second g(), and composef computes f(g(x)), so we realize that the output type of f is the output type of the composed function
composef: (?→C), (?→?) → (?→?)
Similarly the input is the same as the input of g
composef: (?→C), (A→?) → (A→C)
and the output of g goes into f!
composef: (B→C), (A→B) → (A→C) ;; done!

Remember...

• calendar...