We have an intuitive idea of data type in a programming language, whether it is Scheme, or some other language—even though every piece of data in a computer is a bit pattern, we distinguish between integers, booleans, vectors, lists, trees, and so on. But in Scheme, when we evaluate \((+ \ x \ y)\), how do we know that \(x\) is not a procedure, or a list, or some peculiar data type that cannot be added? There is no telling what such an addition would do to the evolution of a computation.

In the Scheme language, data is typed dynamically. This means all the data is explicitly tagged with its type, and the underlying addition procedure checks the tags before each addition. While the programmer is not obligated to annotate a program with types for each of its constituent parts, this dynamic type checking has the obvious potential to slow down computation, because the integrity of data is constantly being checked before primitive operations.

As an alternative, programming language designers have looked as well at compile-time type checking. In other words, let the programmer provide type annotations to a program, which is checked by the compiler for type integrity. If a program passes the examination of a compiler, it is then guaranteed to run without a type error, like adding an integer to a boolean value.

Such an alternative presents many problems, among them the following. How cumbersome is it for the programmer to provide all this type information in a program? Could the compiler instead infer reasonable type information? Finally, what are the rules that say what is a type-safe program, and what isn’t? If the compiler is required to ensure that compiled programs must run without type errors, then we expect the compiler to be conservative—it may reject programs that would run without error. In fact, this is necessary, since it can be formally proved (using the Halting Problem) that any algorithm in a compiler cannot accept exactly the type-safe programs, and reject the type-unsafe programs.

In this handout, we will sketch a version of a fragment of Scheme augmented with a reasonable version of type annotations. We will then give a compiler that accepts properly annotated programs. The limitations of this approach will be appreciated quickly, and in following handouts, we will look at a Scheme compiler that does a similar form of type inference.

Here is an example of the kind of language we imagine:

\begin{verbatim}
(define (x int) 5)

(define (square (int int))
         (lambda ((n int)) (* n n)))

(define (factorial (int int))
         (lambda ((n int)) (if (zero? n) 1 (* n (fact (- n 1))))))

(define (map ((* (int bool) (listof int)) (listof bool)))
         (lambda ((f (int bool)) (lst (listof int)))
                  (if (null? lst) '() (cons (f (car lst)) (map f (cdr lst))))))
\end{verbatim}
These Scheme procedures are annotated versions of old friends:

(define x 5)
(define square (lambda (n) (* n n)))
(define factorial
  (lambda (n) (if (zero? n) 1 (* n (factorial (- n 1)))))
)
(define map
  (lambda (f lst)
    (if (null? lst) '() (cons (f (car lst)) (map f (cdr lst))))
  ))

In every definition, we write down the type of the defined value. Value x has type int (integer); values square and factorial have type (int int) (a function from integers to integers, usually written in the mathematical notation int → int; value map has type ((* (int bool) (listof int)) (listof bool)), usually written ((int → bool)×listof int)→listof bool.

In every lambda expression, we write down the type of each parameter. Thus the body of factorial begins (lambda ((n int)) ...), specifying that the argument must be an integer.

Notice that typing information can be both cumbersome and redundant: in defining the identity function (lambda (x) x), what is the type of x, and does it make a difference? And once we have written (lambda ((x int)) x), this procedure can only be used with an integer argument. Ridiculous—and all this to please the type checker!

1 What is a type?

Before we get too deep into the story, we need to say explicitly what our types are. Notice that this is an inductive definition!

- int (integers) and bool (booleans) are types; these “simplest” types we will often call base types.

- If α is a type, so is a type we write as (listof α), which means a list of values of type α.

- If α and β are types, so is α → β, which we will write as (α β) in Scheme notation. A Scheme expression of type (α β) is a procedure whose input is a value of type α, and whose output must be of type β.

- If α and β are types, so is α × β, which we will write as (× α β) in Scheme notation. A value of this type is a pair of values, the first with type α, and the second with type β.

- That’s all. There are no other ways to be a type.

Here are some examples of Scheme expressions with their types:

- 5 has type int and #t has type bool.

- 1+ has type (int int) and + and - have type ((× int int) int). (The multiplication procedure * also has this type—but do not confuse the dual uses of *.)

- (cons 3 (cons 2 (cons 0 '()))) has type (listof int); (cons + (cons - (cons * '()))) has type (listof ((× int int) int)); and (cons (cons 320 '()) '()) has type (listof (listof int)).

We are delaying the answer to the question “what are the types of cons, car, and cdr?” because they make the story a little more complicated.
2 How do we decide the type of a Scheme expression?

We compute the type of a Scheme expression in a syntax-directed way, very similar to the way the compiler and evaluator work. Like the compiler, all the syntax is evaluated once, and evaluation produces a type as a value. Procedure eval now returns the type of an expression; the environment is now an association of variables to types.

We represent the inference rules for types below, where the constraints above the line represent preconditions for the type computation below the line.

\[(\text{variable}) \quad \frac{(\text{lookup-variable-value } \, x \, \text{env}) = \tau}{(\text{eval } \, x \, \text{env}) = \tau}\]

\[(\lambda) \quad \frac{(\text{eval } \, E \, (\text{extend-environment } \, x \, \, t_1 \, \text{env})) = \tau_2}{(\text{eval } \, (\text{lambda } ((x \, t_1)) \, E) \, \text{env}) = \tau_1(\, t_2)}\]

\[(\text{apply}) \quad \frac{(\text{eval } \, E \, \text{env}) = \tau_1(\, t_2) \quad (\text{eval } \, F \, \text{env}) = \tau_1}{(\text{eval } \, (E \, F) \, \text{env}) = \tau_2}\]

\[(\text{if}) \quad \frac{(\text{eval } \, P \, \text{env}) = \text{bool} \quad (\text{eval } \, E \, \text{env}) = \tau \quad (\text{eval } \, F \, \text{env}) = \tau}{(\text{eval } \, (\text{if } \, P \, E \, F) \, \text{env}) = \tau}\]

Note the use of quoted metavariables.

These are not all the inference rules we will need, but they are enough to get us started.

The code that follows is very similar to that for the metacircular evaluator.

(define (eval exp env)
  (cond ((self-evaluating? exp) (constant-type exp))
        ((variable? exp) (lookup-variable-value exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
          (make-procedure (lambda-parameters exp)
                          (lambda-body exp)
                          env))
        ((application? exp)
          (apply (eval (operator exp) env)
                 (list-of-values (operands exp) env)))
        (else
          (error "Unknown expression type -- EVAL" exp))))
  ;Value: eval

(define (try exp) (eval exp the-global-environment))
  ;Value: try
2.1 Environments

Here is the code for hacking environments, lifted directly from the code for the metacircular evaluator in the textbook:

```
(define (enclosing-environment env) (cdr env))
;Value: enclosing-environment

(define (first-frame env) (car env))
;Value: first-frame

(define the-empty-environment '())
;Value: the-empty-environment

(define (make-frame variables values) (cons variables values))
;Value: make-frame

(define (frame-variables frame) (car frame))
;Value: frame-variables

(define (frame-values frame) (cdr frame))
;Value: frame-values

(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))
;Value: add-binding-to-frame!

(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
    (cons (make-frame vars vals) base-env)
    (if (< (length vars) (length vals))
      (error "Too many arguments supplied" vars vals)
      (error "Too few arguments supplied" vars vals))))
;Value: extend-environment

(define the-global-environment
  (extend-environment
    '(true false < + * 1+ zero? car cdr cons null?)
    '(bool bool ((* int int) bool) ((* int int) int) ((* int int) int)
      ((* int int) int) (int int) (int bool)
      (listof ?t) ?t) ((listof ?t) (listof ?t))
    (listof ?t) (listof ?t)) (listof ?t) bool)
;Value: the-global-environment
```

Notice that the global environment now stores the types of primitive procedures, rather than their implementations in the underlying Scheme system.
2.2 Constants and variables

Self-evaluating expressions are easy:

```
(define (self-evaluating? exp)
  (or (number? exp) (eq? exp '#t) (eq? exp '#f)))
;Value: self-evaluating?
```

```
(define (constant-type exp)
  (cond ((number? exp) 'int)
        ((or (eq? exp '#t) (eq? exp '#f)) 'bool)
        (else (error "Unknown constant" exp))))
;Value: constant-type
```

```
(constant-type 3)
;Value: int
```

```
(constant-type '#t)
;Value: bool
```

The types of variables are looked up in the environment, just as before:

```
(define (variable? exp) (symbol? exp))
;Value: variable?
```

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
              (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env))
               (scan (frame-variables frame)
                     (frame-values frame)))
          (env-loop env))
  ;Value: lookup-variable-value

(lookup-variable-value '< the-global-environment)
;Value: ((* int int) bool)
```
2.3 Definitions

Definitions are a little tricky because they can be recursive (like factorial). The beginning starts off innocuously enough:

```scheme
(define definition? (begins? 'define))
;Value: definition?

(define (definition-variable exp) (caadr exp))
;Value: definition-variable

(define (definition-type exp) (cadr (cadr exp)))
;Value: definition-type

(define (definition-value exp)
  (if (null? (cdddr exp))
      (caddr exp)
      (cons 'begin (cddr exp))))
;Value: definition-value

(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
              (add-binding-to-frame! var val frame))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals))))))
    (scan (frame-variables frame) (frame-values frame))))
;Value: define-variable!
```

Now comes the stuff specific to computing types. First, we write a procedure to detect if a variable occurs unbound in an expression. For example, in `(define (n int) (1+ n))` the occurrence of `n` in the definition body is unbound. Try typing `(define n (1+ n))` into the Scheme evaluator and see what happens! Since the Scheme evaluator complains, we will also want our type-checker to complain with an error message.

```scheme
(define (unbound? var exp)
  (cond ((null? exp) #f)
        ((lambda? exp) #f)
        ((list? exp)
          (or (unbound? var (car exp))
              (unbound? var (cdr exp))))
        (else (eq? var exp))))
;Value: unbound?
```
For a definition to type properly, we add the asserted type of the defined name to the environment, type the body of the definition, and verify that the computed type of the body equals the declared type of the defined name:

```
(define (eval-definition exp env)
  (if (unbound? (definition-variable exp) (definition-value exp))
      (error "Unbound variable:" exp)
    (let ((type (eval (definition-value exp)
                     (extend-environment (list (definition-variable exp))
                     (list (definition-type exp))
                     env)))
      (if (equal? type (definition-type exp))
        (begin
          (define-variable! (definition-variable exp)
            type
            env)
        'ok)
      (error "Bad type of definition:" type (definition-type exp))))))

;Value: eval-definition

(eval '(define (square (int int))
       (lambda ((n int)) (* n n))) the-global-environment)
;Value: ok

(lookup-variable-value 'square the-global-environment)
;Value: (int int)
```

### 2.4 Conditionals

Here is how to type conditionals:

```
(define (if? exp) ((begins? 'if) exp))
;Value: if?

(define (if-predicate exp) (cadr exp))
;Value: if-predicate

(define (if-consequent exp) (caddr exp))
;Value: if-consequent

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
    (cadddr exp)
    'false))
;Value: if-alternative

(define (bool? type) (eq? type 'bool))
;Value: bool?

(define (eval-if exp env)
  (let ((type-pred (eval (if-predicate exp) env))
        (type-then (eval (if-consequent exp) env))
        (type-else (eval (if-alternative exp) env)))
    (if (bool? type-pred)
      (if (equal? type-then type-else)
```

7
type-then
  (error "Branches of if have different types:
           type-pred type-then type-else))
  (error "Predicate not of type bool" type-pred)))

;Value: eval-if

Procedure eval-if just checks that the predicate is of type bool, and the consequent and alternative have the same type.

2.5 Procedures and lambda

(define (lambda? exp) ((begins? 'lambda) exp))
;Value: lambda?

(define (lambda-parameters exp) (cadr exp))
;Value: lambda-parameters

(define (lambda-body exp)
  (if (null? (cdddr exp))
    (caddr exp)
    (cons 'begin (cddr exp))))
;Value: lambda-body

(define (make-procedure parameters body env)
  (let ((vars (map car parameters))
        (domain (map cadr parameters)))
    (let ((range (eval body (extend-environment vars domain env))))
      (list (if (null? (cdr domain)) (car domain) (cons '* domain))
            range))))
;Value: make-procedure

A lambda expression is evaluated by evaluating the body in a new environment where the parameters have been bound to the declared types. Notice the hack where the domain of (lambda ((x int) (y int)) (+ x y)) is (* int int), while the domain of (lambda ((x int)) (1+ x)) is int—thus the list (int int) in the former example needs a * consed in front, and in the latter examples, the list (int) needs to be reduced to int.

2.6 Procedure application

(define (application? exp) (pair? exp))
;Value: application?

(define (operator exp) (car exp))
;Value: operator

(define (operands exp) (cdr exp))
;Value: operands

(define (function? type)
  (if (or ((begins? '* type) ((begins? 'listof) type))
          #f
          (pair? type)))
;Value: function?

(define (domain type) (car type))

8
(define (range type) (cadr type))

(define (list-of-values operands env)
  (if (null? (cdr operands))
      (eval (car operands) env)
      (cons '* (map (lambda (e) (eval e env)) operands))))

(define (apply procedure arguments)
  (if (function? procedure)
      (if (equal? (domain procedure) arguments)
          (range procedure)
          (error "Type mismatch" procedure arguments))
      (error "No function in function position!")))

Notice that an application is typed by typing the operator and operands, and checking that the types of the operands
match the “domain” part of the type of the function.

Here are some examples of its use:

(try '(lambda ((f (int (int int))))
      (lambda ((a int) (b int))
        ((f a) b)))
  ;Value: ((int (int int)) ((* int int) int))

(try '
     (lambda ((fact (int int)))
       (lambda ((n int))
         (if (zero? n) 1 (* n (fact (- n 1)))))
     )
  ;Value: ((int int) (int int))

(try '
     (lambda ((fib (int int)))
       (lambda ((n int))
         (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
  ;Value: ((int int) (int int))

9
### 2.7 Driver loop

```
(define input-prompt ";;; M-Eval input:"
(define output-prompt ";;; M-Eval value:"
(define (prompt-for-input string) (newline) (newline) (display string) (newline))
(define (announce-output string) (newline) (display string) (newline))
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (display output))
  (driver-loop))
```

### 3 On the horizon: type inference and polymorphism

What have we accomplished so far? We have invented a clunky version of Scheme—granted, without the bells and whistles in a full system—where we have to put type declarations everywhere. This is both mechanical and boring—let's get the compiler to do this for us! We thus come across the problem of static type inference, where the compiler must infer the type of every subexpression (save the free variables!)

But before we leap forward to implement this, we need to stop and consider something else. In the implementation above, we did not include the primitives cons, car and cdr—and for good reason: what are their types supposed to be? And what is the type of ()?

Suppose if \(x\) has type \(\alpha\), and \(y\) had type \(\beta\), then \(\text{cons } x \ y\) had type \((\ast \ \alpha \ \beta)\). Suppose also that () had type \(\nu\). Then \(\text{cons } \text{cons } \text{cons } \text{cons } ()\) would have type \((\ast \ \text{int} \ \ast \ \text{int} \ \ast \ \text{int} \ \nu)\). This is a mess—how could we type the length function on lists if every list of integers had a different type?

Instead, suppose lists are homogeneous—every element in a list must have the same type. Then \(\text{cons } 3 \ \text{cons } 2 \ \text{cons } 0 \ ()\) could have type \((\text{listof int})\), and \(\text{cons}\) has type \((\ast \ \text{int} \ \text{listof int}) \ \text{listof int})\). But then what about \(\text{cons } \#f \ \text{cons } ()\), of type \((\text{listof bool})\)—\(\text{cons}\) must also have type \((\ast \ \text{bool} \ \text{listof bool}) \ \text{listof bool})\). A mind-blowing thought: listof is a function on types! Could we invent a “higher-order” programming language so that we write programs for the types too? Indeed we can!

In the sequel, we will follow a limited version of this tack: \(\text{cons}\) will get the polymorphic (“many forms”) type \((\forall t \ \ast \ t \ \text{listof } t))\), where \(\forall t\) is a type variable that may be replaced with the types we have already seen, like \(\text{int}\), \(\text{bool}\), and so on. The constant () will have type \((\forall t \ \text{listof } t))\). When we need to infer the type of \(\text{cons } 320 \ ()\), the constraint that 320 is of type int will set \(t\) in the type of that instance of cons to \((\ast \ \text{int} \ \text{listof int}) \ \text{listof int})\).

Similarly, when we type \((\lambda x \ x)\), it will get type \((\forall t \ ((? t) (? t)))\)—it is the identity function on any argument of type \(? t\), so we do not need separate identity functions for integers, booleans, lists of lists of integers, ...

How is all this going to work? We need to introduce a special kind of constraint solver called a unifier, generalizing the matching procedure we saw in our discussion of symbolic algebra. Type inference will then be a problem of generating a set of type constraints to give to the unification engine. The engine produces the types from the constraints. But that’s all we have time for, for now!

TO BE CONTINUED... SOME TIME IN THE FUTURE...