One-Slide Summary

- In lazy evaluation, expressions are not evaluated until their values are needed. We can use lazy evaluation to program with infinite data structures, such as a list of all natural numbers.
- A type is a (possibly infinite) set of values.
- Each type supports a set of valid operations.
- Types can be latent or manifest, static or dynamic, strong or weak.
- We can change the Charme interpreter to support manifest (program visible) types.

Outline

- Administration
- Lazy Evaluation Recap
- Quiz Results
- Types
- Type Taxonomy
- Static Charme
  - Charme with Manifest Types

The Textbook

- I get the sense that some of the students are attempting to read the book on-line. I would encourage everyone to read it on paper. It is pretty well established that people read faster and understand better on paper than on the screen.
  - David Evans, Course Book Author

Problem Set 8

- Understand and modify a dynamic web application
- Already posted

Problem Set 9

- Team requests and ideas due Friday April 16th (email me before midnight)

Lazy Evaluation Recap

- Don’t evaluate expressions until their value is really needed
  - We might save work this way, since sometimes we don’t need the value of an expression
  - We might change the meaning of some expressions, since the order of evaluation matters
- Change the Evaluation rule for Application
- Use thunks to delay evaluations
Lazy Application

```python
def evalApplication(expr, env):
    ops = map (
        lambda sexpr: Thunk(sexpr, env), expr[1:]
    )
    return mapply(forceeval(expr[0], env), ops)
```

Lazy Application

```python
def evalApplication(expr, env):
    # make Thunk object for each operand expression
    ops = map (
        lambda sexpr: Thunk(sexpr, env), expr[1:]
    )
    return mapply(forceeval(expr[0], env), ops)
```

Lazy Data Structures

```scheme
(define cons
    (lambda (a b)
        (lambda (p)
            (if p a b))))
(define car
    (lambda (p) (p #t)))
(define cdr
    (lambda (p) (p #f)))
```

Note: for PS7, you are defining these as primitives, which would not evaluate lazily.

Lazy Data Structures

```scheme
(define cons
    (lambda (a b)
        (lambda (p)
            (if p a b))))
(define car
    (lambda (p) (p #t)))
(define cdr
    (lambda (p) (p #f)))
```

Using Lazy Pairs

```scheme
(define cons
    (lambda (a b)
        (lambda (p)
            (if p a b))))
(define car
    (lambda (p) (p #t)))
(define cdr
    (lambda (p) (p #f)))
```

LazyCharme> (define mypair (cons 3 error))
LazyCharme> mypair
<Procedure ['p'] / ['if', 'p', 'a', 'b']>
LazyCharme> (car mypair)
3
LazyCharme> (cdr mypair)
Error: Undefined name: error

Using Lazy Pairs

```scheme
(define cons
    (lambda (a b)
        (lambda (p)
            (if p a b))))
(define car
    (lambda (p) (p #t)))
(define cdr
    (lambda (p) (p #f)))
```

LazyCharme> (define mypair (cons 3 error))
LazyCharme> mypair
<Procedure ['p'] / ['if', 'p', 'a', 'b']>
LazyCharme> (car mypair)
3
LazyCharme> (cdr mypair)
Error: Undefined name: error

Infinite Lists

```scheme
(define ints-from
    (lambda (n)
       (cons n (ints-from (+ n 1)))))
```

LazyCharme> (define allnaturals (ints-from 0))
LazyCharme> (car allnaturals)
0
LazyCharme> (car (cdr allnaturals))
1
LazyCharme> (car (cdr (cdr (cdr (cdr allnaturals)))))
4

Infinite Lists

```scheme
(define ints-from
    (lambda (n)
       (cons n (ints-from (+ n 1)))))
```

LazyCharme> (define allnaturals (ints-from 0))
LazyCharme> (car allnaturals)
0
LazyCharme> (car (cdr allnaturals))
1
LazyCharme> (car (cdr (cdr (cdr (cdr allnaturals)))))
4

Infinite Fibonacci Sequence

```scheme
(define fibo-gen
    (lambda (a b)
        (cons a (fibo-gen b (+ a b)))))
(define fibos (fibo-gen 0 1))
(define get-nth
    (lambda (lst n)
        (if (= n 0) (car lst)
            (get-nth (cdr lst) (- n 1)))))
(define fibo
    (lambda (n) (get-nth fibos n)))
```

Infinite Fibonacci Sequence

```scheme
(define fibo-gen
    (lambda (a b)
        (cons a (fibo-gen b (+ a b)))))
(define fibos (fibo-gen 0 1))
(define get-nth
    (lambda (lst n)
        (if (= n 0) (car lst)
            (get-nth (cdr lst) (- n 1)))))
(define fibo
    (lambda (n) (get-nth fibos n)))
```

Alternate Implementation

```scheme
(define merge-lists
    (lambda (lst1 lst2 proc)
        (if (null? lst1) null
            (if (null? lst2) null
                (cons (proc (car lst1) (car lst2))
                    (merge-lists (cdr lst1) (cdr lst2) proc))))))
(define fiboms
    ;; merge-list variant
    (cons 0 (cons 1 (merge-lists fiboms (cdr fiboms) +))))
```

Alternate Implementation

```scheme
(define merge-lists
    (lambda (lst1 lst2 proc)
        (if (null? lst1) null
            (if (null? lst2) null
                (cons (proc (car lst1) (car lst2))
                    (merge-lists (cdr lst1) (cdr lst2) proc))))))
(define fiboms
    ;; merge-list variant
    (cons 0 (cons 1 (merge-lists fiboms (cdr fiboms) +))))
```

Come back and understand this slide to study for the exams.
Liberal Arts Trivia: Cognitive Science

- This philosophy of mind dominated for the first half of the 20th century. It developed as a reaction to the inadequacies of introspectionism. In it, all things which organisms do - including acting, thinking and feeling - should be regarded as actions or reactions, usually to the environment. It holds that there are no philosophical differences between publicly observable processes (actions) and privately observable processes (thinking and feeling).
- Bonus: B.F. Who?

Liberal Arts Trivia: Civil Rights

- The landmark 1967 Supreme Court case Loving v. Virginia declared Virginia's anti-miscegenation statue, the “Racial Integrity Act of 1924”, unconstitutional. This effectively ended laws preventing what?

Types

- Numbers
- Strings
- Colors
- Beatle’s Songs that don’t end on the Tonic
- Lists of lists of lists of anything

- A **Type** is a (possibly infinite) set of values
- You can do some things with some types, but not others
  - Each Type has associated valid operations

Why have types?

- Detecting programming errors: (usually) better to notice error than report incorrect result
- Make programs easier to read, understand and maintain: thinking about types can help understand code
- Verification: types make it easier to prove properties about programs
- Security: can use types to constrain the behavior of programs

Types of Types

Does regular Scheme have types?

> (car 3)
**car**: expects argument of type <pair>; given 3

> (+ (cons 1 2))
**+**: expects argument of type <number>; given (1 . 2)

Yes, without types (car 3) would produce some silly result. Because of types, it produces a type error.
Does Python Have Types?

```python
>>> 3 + "hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Type Taxonomy

- **Latent vs. Manifest**
  - Are types visible in the program text?
- **Static vs. dynamic checking**
  - Do you have to run the program to know if it has type errors?
- **Weak vs. Strong checking**
  - How strict are the rules for using types?
    - (e.g., does the predicate for an if need to be a Boolean?)
  - Continuum (just matter of degree)

Scheme/Python/Charme

- Latent or Manifest?
  - All have **latent** types (none visible in code)
- Static or Dynamic?
  - All are **dynamic** (checked when expression is evaluated)
- Weak or Strong?
  - Which is the strictest?
  - You tell me!

Scheme/Python/Charme → Java/StaticCharme

- Scheme, Python, and Charme have **latent**, Dynamically checked types
  - Don’t see explicit types when you look at code
  - Checked when an expression is evaluated
- Java, StaticCharme have **Manifest, statically checked** types
  - Type declarations must be included in code
  - Types are checked statically before running the program (Java: not all types checked statically)

Strict Typing

Scheme> (+ 1 #t)

+: expects type <number> as 2nd argument, given: #t; other arguments were: 1

Python>>> 1 + True
2

Charme> (+ 1 #t)
2

Java Example

```java
class Test {
    int tester (String s)
    {
        int x;
        x = s;
        return "okay";
    }
}
```

The place x holds an integer

The parameter must be a String
Java Example

```java
class Test {
    int tester (String s)
    {
        int x;
        x = s;
        return "okay";
    }
}
```

The result is an integer

The parameter must be a String

What do we need to do to change our Charme interpreter to provide manifest types?

javac compiles (and type checks) the program. It does not execute it.

Truthiness!

Liberal Arts Trivia: Media Studies

• This technique in film editing combines a series of short shots into a sequence of condensed narrative. It is usually used to advance the story as a whole and often to suggest the passage of time.

Liberal Arts Trivia: European History

• This relatively slender, sharply pointed sword was popular in Europe in the 16th and 17th centuries. It is mainly used for thrusting attacks. It is characterized by a complex hilt, which protects the hand wielding it. The etymology may derive the word from the Greek ραπίζειν "to strike."

Liberal Arts Trivia: United Kingdom History

• This United Kingdom Tory prime minister was most famous for his military work during the Peninsular Campaign and the Napoleonic Wars. He was nicknamed the “Iron Duke” because of the iron shutters he had fixed to his windows to stop pro-reform mobs from breaking them - as an MP he was opposed to reform. It is unclear whether the well-known beef tenderloin, pate and puff pastry dish is named after him.
Programming Language Design Space

Expressiveness

- Scheme
- Python
- Charme
- LazyCharme
- StaticCharme
- Java

“Truthiness”

Types of Types

- Charme
- StaticCharme

Latent ➔ Manifest
change grammar, represent types

Dynamically Checked ➔ Statically Checked
typecheck expressions before eval

Manifest Types

Need to change the grammar rules to include types in definitions and parameter lists

Definition ::= (define Name : Type Expression)
Parameters ::= ε | Parameter Parameters
Parameter ::= Name : Type
Type ::= ??

Types in Charme

CType ::= CPrimitiveType | CProcedureType | CProductType
CPrimitiveType ::= Number | Boolean
CProcedureType ::= (CProductType -> Type)
CProductType ::= (CTypeList)
CTypeList ::= CType CTypeList

Representing Types

CType ::= CPrimitiveType | CProcedureType | CProductType
CPrimitiveType ::= Number | Boolean
CProcedureType ::= (CProductType -> Type)
CProductType ::= (CTypeList)
CTypeList ::= CType CTypeList

Changed parameters grammar rule:
Parameter ::= Name : Type

Number
+ 
((Number Number) -> Number)
(+ 3 3)

Number
(lamba \(x\):Number \(y\):Number \(x > y\))
((Number Number) -> Boolean)
```
class CType:  
    @staticmethod  
    def fromString(s):  
        # create type from string  
        tparse = parse(s)  
        return CType.fromParsed(tparse[0])  
    
    @staticmethod  
    def fromParsed(typ):  
        ... # create type from parsed type  

    # These methods are overridden by subclasses  
    def isPrimitiveType(self): return False  
    def isProcedureType(self): return False  
    def isProductType(self): return False  
    def isError(self): return False  

CPrimitiveType  

class CPrimitiveType(CType):  
    def __init__(self, s):  
        self._name = s  
    
    def __str__(self): return self._name  
    def isPrimitiveType(self): return True  
    def matches(self, other):  
        return other.isPrimitiveType() \\ 
                        and self._name == other._name  

CProcedureType  

class CProcedureType(CType):  
    def __init__(self, args, rettype):  
        self._args = args  
        self._rettype = rettype  
    
    def __str__(self):  
        return "(" + str(self._args) + " -> " + str(self._rettype) + ")"  
    def isProcedureType(self): return True  
    def getReturnType(self): return self._rettype  
    def getParameters(self): return self._args  
    def matches(self, other):  
        return other.isProcedureType() \\ 
                        and self.getParameters().matches(other.getParameters()) \\ 
                        and self.getReturnType().matches(other.getReturnType())  

CProductType  

class CProductType(CType):  
    def __init__(self, types): self._types = types  
    
    def __str__(self): ...  
    def isProductType(self): return True  
    def matches(self, other):  
```
**CProductType**

```python
class CProductType(CType):
    def __init__(self, types): self._types = types
    def __str__(self): ...
    def isProductType(self): return True
    def matches(self, other):
        if other.isProductType():
            st = self._types
            ot = other._types
            if len(st) == len(ot):
                for i in range(0, len(st)):
                    if not st[i].matches(ot[i]): return False
                # reached end of loop ==> all matched
                return True
            return False
```

**Homework**

- Show up to lecture on Monday
- Problem Set 7 due
- Problem Set 9 Team Requests