Programming Languages

Streams Wrapup, Memoization, Type Systems, and Some Monty Python

Quick Review of Constructing Streams

- Usually two ways to construct a stream.
- Method 1: Use a function that takes a(n) argument(s) from which the next element of the stream can be constructed.

```
(define (integers-from n)
  (stream-cons n (integers-from (+ n 1))))
(define ints-from-2 (integers-from 2))
```

- When you use this technique, your code usually looks a lot like you have infinite recursion.
- Often the code is very clear (easy to see how it works).

Quick Review of Constructing Streams

- Usually two ways to construct a stream.
- Method 2: Construct the stream directly by defining it in terms of a modified version of another stream or itself.

```
(define ints-from-2-alt
  (stream-cons 2
    (stream-map (lambda (x) (+ x 1))
                ints-from-2-alt)))
```

- This technique is fine, but can be harder to figure out how it works.

Quick Review of Constructing Streams

- Usually two ways to construct a stream.
- Method 2: Construct the stream directly by defining it in terms of a modified version of another stream or itself.

```
(define ints-from-2-alt-alt
  (stream-cons 2
    (stream-map2 +
                 infinite-ones
                 ints-from-2-alt-alt)))
```

Fibonacci

- Method 1:

```
(define (make-fib-stream a b)
  (stream-cons a (make-fib-stream b (+ a b))))

(define fibs1 (make-fib-stream 0 1))
```

Fibonacci

- Method 2:

```
(define fibs
  (stream-cons 0
    (stream-cons 1
      (stream-map2 + (stream-cdr fibs) fibs))))
```

Sieve of Eratosthenes

- Start with an infinite stream of integers, starting from 2.
- Remove all the integers divisible by 2.
- Remove all the integers divisible by 3.
- Remove all the integers divisible by 5…etc

Sieve of Eratosthenes

```
(define (not-divisible-by s div)
  (stream-filter
     (lambda (x) (> (remainder x div) 0)) s))

(define (sieve s)
  (stream-cons
   (stream-car s)
   (sieve (not-divisible-by s (stream-car s)))))

(define primes (sieve ints-from-2))
```

Stream wrapup

- Streams are an implementation of the **Iterator** abstraction.
- An Iterator is something that lets the programmer traverse data in a ordered, linear fashion.
- You've seen C++ iterators that let you iterate over vectors.
  - There are also C++ iterators that let you iterate over sets, the entries in maps, and lots of other data structures.

Stream wrapup

- Racket's streams obey the same semantics as C++ iterators.

	Racket Stream	C++ iterators
Get the current element	**stream-car**	***it**
Advance to the next element	**stream-cdr**	**it++**

- You can easily create infinite iterators in C++, just like you can create infinite streams in Racket.
- The concept of an iterator doesn't distinguish between **iterating over a pre-existing data structure** and **iterating over something that's being generated on the fly**.

Stream wrapup

- What to take away from all this:
- Most modern languages have one or more data types that encapsulate this iteration concept.
  - Iterators: C++, Java
  - Streams: Racket, Scheme, and most functional languages
  - Sequences: Python
  - Functions: Almost any language
- Can "fake" an iterator with a functions:

```
int nextInt()
{
  static int i = 0;
  i++;
  return i;
}
```

```
int nextInt(int old)
{
  return old + 1;
}
```

Stream wrapup

- Python's built-in iterators are called sequences.

```
for x in range(0, 100**100):
  print(x)
```

  - This code would never run if Python actually computed a list containing 100^{100} integers before starting to print them.
  - Instead, **`range`** returns an iterator over the numbers that doesn't generate the next integer until it's needed.

- Python actually has the advantage here over Racket, because Racket could never generate a stream of 100^{100} integers.
- Why not?

And Now For Something Completely Different (But Kind of Related)

Fibonacci

```
(define (make-fib-stream a b)
  (stream-cons a (make-fib-stream b (+ a b))))
(define fibs1 (make-fib-stream 0 1))
```

- More efficient (but less clear?) than

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (#t (+ (fib (- n 1)) (fib (- n 2))))))
```

- How to get the best of both worlds?

Memoization

- If a function has no side effects and doesn't read mutable memory, no point in computing it twice for the same arguments
  - Can keep a *cache* of previous results
  - Net win if (1) maintaining cache is cheaper than recomputing and (2) cached results are reused

- Similar to how we implemented promises, but the function takes arguments so there are multiple "previous results"

- For recursive functions, this *memoization* can lead to *exponentially* faster programs
  - Related to algorithmic technique of dynamic programming

```
(define fast-fib
 (let ((cache '()))
  (define (lookup-in-cache cache n)
   (cond ((null? cache) #f)
         ((= (caar cache) n) (cadar cache))
         (#t (lookup-in-cache (cdr cache) n))))

 (lambda (n)
  (if (or (= n 0) (= n 1)) n
      (let ((check-cache (lookup-in-cache cache n)))
       (cond ((not check-cache)
              (let ((answer (+ (fast-fib (- n 1))
                               (fast-fib (- n 2)))))
               (set! cache (cons (list n answer) cache))
                    answer))
             (#t check-cache)))))))
```

Memoization in other languages

- Code for memoization is often easier with an explicit hashtable data structure:

```
int fib(int n) {
  static map<int, int> cache;
  if (n < 2) return n;
  if (cache.count(n) == 0) {
    int ans = fib(n-1) + fib(n-2);
    cache[n] = ans;
    return ans;
  } else return cache[n];
}
```

Memoization wrapup

- Memoization is related to streams in that streams also remember their previously-computed values.
  - Remember how promises save their results and return them instead of re-computing?
- But memoization is more flexible because it works with any function.

- Memoization is a classic example of the time-space trade-off in CS:
  - With memoization, we use more space, but use less time.

And Now For Something Completely Different (It's Really Different This Time!)

Static vs. dynamic typing

- A big, juicy, essential, topic about how to think about PLs
  - Conversation usually overrun with half-informed opinions ☹
  - Will consider reasonable arguments “for” and “against” last

- First, a review!

Static vs Dynamic Typing

- A PL uses **static typing** when most type-checking is done at compile-time. (e.g., C, C++, Java)
  - (or for an interpreter, before the program begins running)
- A PL uses **dynamic typing** when most type-checking is done at run-time. (e.g., Python, Racket)

- Languages that are *usually* compiled often use static typing.
- Languages that are *usually* interpreted often use dynamic typing.

Static vs Dynamic Typing

- Static/dynamic typing has NOTHING to do with static/dynamic scoping!
  - The names are similar because "static" often refers to compile-time (or before the program starts running) and dynamic often refers to run-time (while the program is running).

Static checking

- *Static checking* is anything done to reject a program *after* it (successfully) parses but *before* it runs

- **What static checking is performed is part of the PL definition**
  - A "helpful tool" (like an IDE) can do more if it wants

- Most common way to define a PL's static checking is via a *type system*
  - *Approach* is to give each variable, expression, etc. a type
  - *Purposes* include preventing misuse of primitives (e.g., `4/"hi"`) and avoiding dynamic checking (dynamic means at run-time)

- Dynamically typed PLs (e.g., Python, Racket) do much less static checking than statically typed PLs (e.g., C++, Java)

Example: C++, what types prevent

In C++, type-checking ensures a program (when run) will **never**:

- Use a primitive operation on a value of the wrong type
  - Use arithmetic on a non-number
  - Let you call f(x) if f takes an int argument and x is a string.
  - Let you say if (whatever) if "whatever" cannot be casted to a boolean.
- Use a variable that is not in the environment

These two features are “standard” for type systems

Example: C++, what types don't prevent

In C++, type-checking does **not** prevent any of these errors

- Instead, detected at run-time

- Calling functions such that exceptions occur, e.g., dereferencing a null pointer.
- An array-bounds error
- Division-by-zero

And in general no type system prevents logic / algorithmic errors:

- Reversing the branches of a conditional
- Calling `f` instead of `g`

The purpose is to prevent something

Have discussed facts about *what* the C++ type system does and does not prevent

- Without discussing *how* (e.g., one type for each variable) though you know how types in C++ work

Part of language design is deciding what is checked and how

- Hard part is making sure the type system does it correctly

- Take away:
  - Static checking = checks done before the program is run (often relating to data types).
  - Dynamic checking = checks done while running.

A question of eagerness

"Catching a bug before it matters"
is in inherent tension with
"Don't report a bug that might not matter"

Static checking / dynamic checking are two points on a continuum

Silly example: Suppose we just want to prevent evaluating `3 / 0`

- Keystroke time: disallow it in the editor
- Compile time: disallow it if seen in code
- Link time: disallow it if seen in code that may be called to evaluate **`main`**
- Run time: disallow it right when we get to the division
- Later: Instead of doing the division, return **`+inf`** instead

Now consider costs and benefits

Having carefully stated facts about static typing,

we can *now* consider arguments about whether it is better or worse than dynamic typing.

An Argument With Myself

Claim 1a: Dynamic is more convenient

Dynamic typing lets you build a heterogeneous list or return a "number or a string" without getting in your way

```
(define (f y)
  (if (> y 0) (+ y y) "hi"))

(let ((ans (f x)))
  (if (number? ans) (number->string ans) ans))
```

```
C++: can't do this easily; what would the
  return type of f be?
```

Claim 1b: Static is more convenient

Can assume data has the expected type without cluttering code with dynamic checks or having errors far from the logical mistake

```
(define (cube x)
  (if (not (number? x))
      (error "bad arguments")
      (* x x x)))

(cube 7)
```

```
int cube(int x) {
  return x * x * x;
}
```

Claim 2a: Static prevents useful programs

Any sound static type system forbids programs that do nothing wrong, forcing the programmer to code around the limitation

```
def f(lst):
  lst.append(1)
  lst.append(True)
```

```
C++: nope nope nope
```

Claim 2b: Dynamic allows non-meaningful programs

```
def f(lst):
  lst.append(1)
  lst.append(True)
```

- Does this really make sense?
- Did we make a mistake?
- Should we have separate lists with ints and booleans?
- Should we have a data type with an int and a boolean?

Claim 3a: Static catches bugs earlier

Static typing catches tons of simple bugs as soon as you compile

- Since you know they are prevented, no need to test for them

```
; gimme all the positive numbers
(filter + '(-8 5 0 3))
```

```
//In a C++-ish language with static typing:
vector<int> v = ...
// Make a function that takes 2 ints and returns an int
function<int, int, int> add = ...
filter(add, v)  // compiler would flag as error
                // because filter must take a function
                // of one argument.
```

Claim 3b: Static catches only easy bugs

But it usually catches only the "easier" bugs, so you still have to test your functions, which should find the "easier" bugs too

Example:

```
f(int* x)
{
  cout << *x << endl;
    // (oops, forgot to check if NULL)
}
```

Claim 4a: Static typing is faster

Your code does not need to check arguments and results to make sure they're the right type.

(Faster for code execution, because the compiler does not have to insert any type checking code.)

Claim 4b: Dynamic typing is faster

Your code does not need to check arguments and results to make sure they're the right type.

(Faster for program development, because *you* don't have to insert type-checking code (i.e., listing data types of function arguments).)

Claim 5a: Code reuse easier with dynamic

By not requiring types, more code can just be reused with data of different types

- If you use cons cells for everything, libraries that work on cons cells are available
- Collections libraries are amazingly useful but often have very complicated static types (have you *seen* C++ error messages with collections [e.g., vector, map, set, etc])??

error: no match for ‘operator!’ in ‘!cache. std::map<_Key, _Tp, _Compare, _Alloc>::find [with _Key = int, _Tp = int, _Compare = std::less<int>, _Alloc = std::allocator<std::pair<const int, int> >] (((const int&)((const int*)(& n))))’

Claim 5b: Code reuse easier with static

- Modern type systems should support reasonable code reuse with features like generics and subtyping

- If you use cons cells for everything, you will confuse what represents what and get hard-to-debug errors
  - Use separate static types to keep ideas separate
  - Static types help avoid library misuse

So far

Considered 5 things you care about when writing code:

1. Convenience
2. Not preventing useful programs
3. Catching bugs asap
4. Performance
5. Code reuse

But we took the naïve view that software is developed by taking an existing spec, coding it up, testing it, and declaring victory. Reality:

- Often do a lot of prototyping *before* you have a stable spec
- Often do a lot of maintenance/evolution *after* version 1.0

Claim 6a: Dynamic better for prototyping

Early on, you don't know what cases you need in your data types and your code

- But static typing won't let you try code without having all cases; dynamic lets incomplete programs run
- So you make premature commitments to data structures
- And end up writing a lot of code to appease the type-checker that you are going to end up throwing away

Claim 6b: Static better for prototyping

What better way to document your evolving decisions on data structures and code-cases than with the type system?

Easy to put in temporary stubs as necessary, such as

```
void awesome_function(int x) {
   // do something awesome
}

void awesome_function(string s) {
   cout << "Not written yet! << endl;
}
```

Claim 7a: Dynamic better for evolution

Can change code to be more permissive without affecting old callers

- Example: Take an **`int`** or a **`string`** instead of an **`int`**

OLD

```
(define (f x)(* 2 x))
```

NEW

```
(define (f x)
  (if (number? x)
      (* 2 x)
      (string-append x x)))
```

```
f(int x) {
  return 2 * x;
}
```

```
In C++, we're saved by
  overloading.
In C, we panic.
```

Claim 7b: Static better for evolution

When we change type of data or code, the type-checker gives us a "to-do" list of everything that must change

- Avoids introducing bugs
- The more of your spec that is in your types, the more the type-checker lists what to change when your spec changes

Example: Changing the return type of a function

Counter-argument: The to-do list is mandatory, which makes evolution in pieces a pain: can't "test what I've changed so far"

"To-do list" AKA "everything breaks at once"

Coda

- Static vs. dynamic typing is too coarse a question
  - No one programming language that will be "best" for all tasks.
- There are real trade-offs here you should know
  - Allows for rational discussion informed by facts
- Ideally would have flexible languages that allow best-of-both-worlds
  - Still mostly an open and active area of research
  - Rare to find a language that lets you pick static/dynamic typing for different parts of a program.
  - Version of Racket called "Typed Racket" that lets you do this.