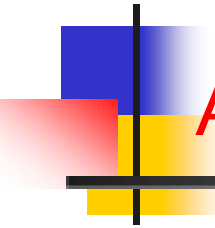
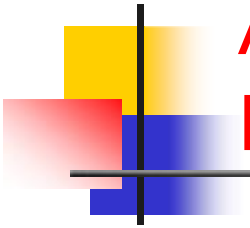


A Glimpse at Imperative Racket and Memoization

Comp 311
Rice University
Corky Cartwright



All Real "Functional" Languages Except Haskell Support Imperative Operations

Why do nearly all real "functional" languages include imperative operations?

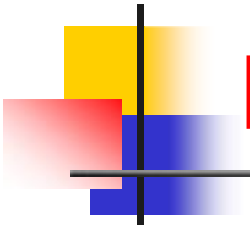
- The real world is imperative (changes state). The real world and many conceptual models evolve (change state) over time. In computations simulating these models, it is often convenient (and conceptually economical) to let *execution recapitulate evolution*. During the simulation of the model, changes in the state of the model are represented by changes in the current program state. In such contexts, the discreet use of imperativity may be logically simpler in many cases than a purely functional approach.
- Our computer hardware is imperative. At some point, even pure functional code must be executed on hardware where every computation step (execution of a machine instruction) involves mutation. To produce efficient machine code to solve a problem, we often need to describe the computation in imperative terms. Most fast algorithms perform incremental operations that are imperative.

Functional Programming Culture

Imperative computation must be clearly identified as such. When imperativity is used "internally" to improve performance, it should be encapsulated when possible behind APIs that are functional.

Examples of embedded imperativity (invisible to agents using a black box API)

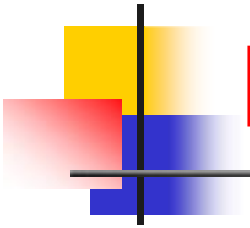
- Memoization
- Fast imperative algorithms for solving problems which may have slower functional equivalents.
- Simulation of physical systems: the change in state over short time intervals is typically small and the successor state in discrete simulation is often a simple update to the current state. In some cases, it is possible to preserve the old state and construct the new state by sharing pieces of the previous state but many data structures (like arrays) must be completely copied if the old state is to be preserved, compromising the efficiency of the update operation. Essentially all physical simulations rely on destructive updates. On the other hand, it may be convenient to build a complete new representation particularly in the context of parallelism. Parallel execution often requires copying for the sake of data locality.



FP Culture Continued

In Racket/Scheme

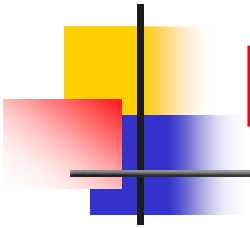
- Most mutation operations (at least those in libraries) in Racket/Scheme end with a ! (read “bang”) character. Matthias Felleisien loved to title the lecture introducing the imperative extension of Scheme as the “The Big Bang!” (He also liked the title “Church and State” where “Church” refers to Alonzo Church who invented the lambda-calculus.)
- Simulation of physical systems: the change in state over short time intervals is typically small and the successor state in discrete simulation is often a simple update to the current state. In some cases, it is possible to preserve the old state and construct the new state by sharing pieces of the previous state (which has a simple “algebraic” implementation) but many data structures (like arrays) must be completely copied if the old state is to be preserved, compromising the efficiency of the update operation. Essentially all physical simulations rely on destructive updates. On the other hand, it may be convenient to build a complete new representation particularly in the context of parallelism. Parallel execution often requires copying information across separate tasks for the sake of data locality.



Mutation Example

Naïve Fibonacci with Memoization

```
(define fib
  (local
    [(define results (make-hash))  ;; results is empty hash table
     (define (fibHelp n)
       (cond [(< n 2) 1]
             ;; if n is in the memo table, return the cached value
             [(hash-has-key? results n) (hash-ref results n)]
             [else  ;; bind sum to fib(n)
               (let [(sum (+ (fibHelp (- n 1)) (fibHelp (- n 2))))]
                 (begin
                   (hash-set! results n sum) ;; add <n,sum> to table
                   sum))]))]
    fibHelp))
```



Bottom Up Improvement

- Demo of
  - naïve fib,
  - memoized naïve fib,
  - fast fib (linear algorithm with help fun)
  - log fib
- In practice, memoization is very powerful.

Aside: Introducing local bindings

Racket/Scheme supports *three* different forms of "let" (a common name in functional languages for an expression that introduces new local bindings) that only differ on the text that is in the scope of the new bindings

- ```
  (let    [(x1 E1) ... (xn En)] E)
  ;; equivalent to ((lambda (x1 ... xn) E) E1 ... En)
  ```
- ```
  (let*   [(x1 E1) ... (xn En)] E)
  ;; equivalent to (let [(x1 E1)] ... (let [(xn En)] E) ...)
  ```
- ```
  (letrec [(x1 E1) ... (xn En)] E)
  ;; equivalent to (local [(define x1 E1) ... (define xn En)] E)
  ```

In all three constructs, the new local variables ***x1***, …, ***xn*** are "visible" in the body ***E***. In **`let`**, the new local variables are "invisible" (not in scope) in the right-hand-sides ***E1***, …, ***En*** of the new local bindings. In **`let*`**, each local variable ***xi*** is visible in *subsequent* right-hand-sides ***Ei+1***, …, ***En***. In **`letrec`**, all local variable are visible (but not *necessarily defined*) in all right-hand-sides ***E1***, …, ***En***. Forward references must be embedded inside lambda-abstractions that defer evaluation.
  ```

Observations About Various `let` forms

Ordinary **`let`** appears in most functional languages because it simply abbreviates a lambda application:

`(let [(`*`x1 E1`*`) ... (`*`xn En`*`)] `*`E`*`) ≡ ((lambda (`*`x1`* ... *`xn`*`) `*`E`*`) `*`E1`* … *`En`*`)`

The **`let*`** operation has a straightforward definition in terms of **`let`**:

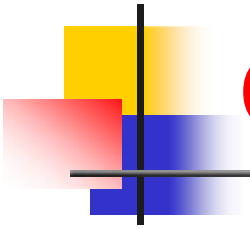
```
(let* [(x1 E1) … (xn En)] E) ≡
   (let [(x1 E1)]
       (let …
           (let [(xn En)] E) … E) … ))
```

which incidentally is how Java defines the meaning of a sequence of local bindings.

The **`letrec`** operation is alternate notation for **`local`**:

```
(letrec [(x1 E1) … (xn En)] E) ≡
   (local [(define x1 E1) … (define xn En)] E)
```

which is how Algol-like languages define the meaning of a sequence of local bindings. **`letrec`** is the most expressive of the three because it supports recursive definitions. If you use fresh names for local variables, it subsumes the others.



Core Imperative Operations

- Assignment

  (set! *v E*) rebinds variable ***v*** to the value of ***E***

- Struct field mutation (except **cons**)

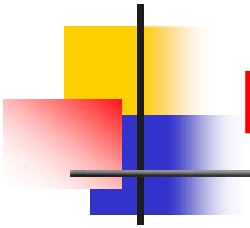
  (*<struct-name>*-*<field-name>*! *E1 E2*)

  changes the specified field in the struct identified by ***E1*** to the value of ***E2***; if the value of ***E1*** is not an instance of the specified struct, an error is thrown.

- Imperative sequencing

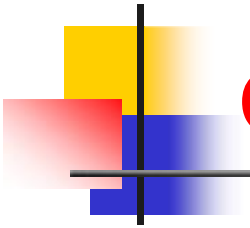
  (begin *E1 … En*)

  evaluates ***E1***, … , ***En*** and returns the value of ***En***.



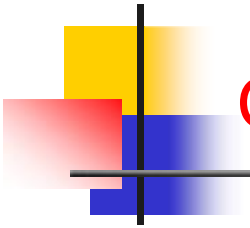
Lazy Evaluation in Racket

- The workaround: define lazy constructors as macros that expand lazy constructor applications to applications of the corresponding strict constructors composed with thunk-wrapping each argument.
- What is a macro? A rule performed by the compiler that expands a macro invocation (which typically looks just like a function application) into ordinary language code that actually implements the macro operation. The compiler completely expands all macros.
- Macros are under-utilized in modern languages because surface (concrete) program syntax is so ugly and messy to manipulate. Strings separated by varying amounts of whitespace. Ugh!
- Programs conceptually have an intelligible tree-based (abstract) syntax that programmers never see. At this level, macros are easy to express and understand. The Racket/Scheme/Lisp family of languages is ideal for macros because concrete syntax ≈ abstract syntax.



Optimizing Lazy Evaluation in Racket

- No field value (a thunk) in a lazy construction should be evaluated more than once. This optimization is a trivial form of memoization. This approach to evaluating the arguments of a construction is called "*call-by-need*". It is the preferred implementation of *call-by-name* in beta-reduction (function application) in purely functional languages as well.
- Where do we store the value? In the same cell as the pointer to the code for computing it.
- In Scheme, a **`cons`** node is mutable, so we simply use the **`first`** and **`rest`** fields in a lazy **`cons`** node to store either a pointer to a thunk or its value.
- It is easy (as we showed in last lecture) to write Scheme macros to implement optimized forms of **`lazy-cons`**, **`lazy-first`**, and **`lazy-rest`**.
- What is a macro? A rule performed by the compiler that expands a macro invocation (which typically looks just like a function application) into ordinary language code that actually implements the macro operation. The compiler completely expands all macros.
- Macros are under-utilized in modern languages because surface (concrete) program syntax is so ugly and messy to manipulate.
- Programs conceptually have an intelligible tree-based (abstract) syntax that programmers never see. At this level, macros are easy to express and understand. The Racket/Scheme/Lisp family of languages is ideal for macros because concrete syntax ≈ abstract syntax.

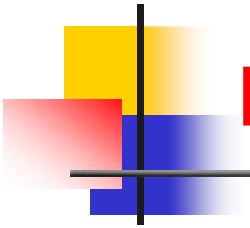


Complication in Racket: cons Is Immutable

- In Scheme, a **`cons`** struct (which is built-in to support lists) is mutable; in the official Racket languages, it is not. In official Racket, there is a mutable form of **`cons`** called **`mcons`**. The HTDP languages predate the addition **`mcons`** to Racket. In the days of DrScheme (when the first edition of HTDP was written), **`cons`** was mutable at the **Advanced Student** level and beyond (like **`R5RS, PrettyBig`**). Today, mutable **`cons`** only exists in Racket **`R5RS`** (standard Scheme) but the implementation is now done using the Racket **`mcons`** package with **`mcons`**, **`set-mcar!`**, and **`set-mcdr!`** renamed as **`cons`**, **`set-car!`**, and **`set-cdr!`** to meet the R5RS standard. Why introduce all of this complication to support an immutable definition of **`cons`**? Flexibility + Optimization! **`mcons`** is useful in some situations (like the implementation of "*call-by-need*" described below) but it is pathological from the perspective of code optimization.

Supporting Call-by-need in Racket

- Since **`cons`** is immutable, we cannot directly change the contents of the **`car`** (**`first`**) and **`cdr`** (**`rest`**) fields. We have to embed boxes (**`box`** is a unary *mutable* constructor built-in to Racket and Scheme) inside the **`cons`** struct adding a level of indirection in the machine representation. At this point, it is probably better to use **`delay`**, a built-in lazy unary constructor in Racket/Scheme (it is not part of Core Racket) which performs our optimization. Built-in primitives typically are optimized beyond the that what be achieved by equivalent source code. Since **`delay`** is lazy, it eliminates the need for thunks (it already includes similar machinery). Nevertheless, I am skeptical of this implementation of laziness because of the extra level of indirection (pointing to the delay descriptor), it will be slower that the macro I wanted to write but could not because **`cons`** is now immutable. On the other hand, there are many sound program transformations involving immutable **`cons`** that break when **`cons`** is mutable.
- I am no more optimistic about the eventual fate of Racket than I am about the fate of Scala. Both are complex platforms created primarily for use by insiders (wizards who have spent years learning, implementing, and extending them).



Practical Recommendations Regarding FP

- Only incorporate imperativity when it is semantically simpler (a judgment call) than a purely functional approach or significantly more efficient.
- If possible, encapsulate imperativity inside visible operations with purely functional contracts. Memoization is a good example of such encapsulation. Memoized naïve **`fib`** has the same contract and visible behavior (other than better performance which leaves extensional behavior unchanged) as unmemoized naïve **`fib`**.
- In Haskell, use monads that mimic explicit imperativity. I am not convinced this approach is better than tasteful, explicit imperativity but it is purely functional. When writing Haskell, do as …