CS 60

Guards and Computing Functions

September 21, 2001

Review

- A function whose code which does no work after a recursive call is said to be *tail-recursive*.
- Advantages
  - Can be more space-efficient
  - Sometimes is even faster.
- Disadvantages
  - Code is often more complex, harder to understand
  - Not all tasks benefit from tail recursion
    - e.g., `append`, `drop`, ...

Converting Numbers to Binary

- Want `toBinary(N)` to returns a list of `0`'s and `1`'s, the binary representation of the positive integer `N`.

```
toBinary(37) ==> [1, 0, 0, 1, 0, 1]
```

[since 37 = **1***2^5+**0***2^4+**0***2^3+**1***2^2+**0***2^1+**1***2^0]

```
toBinary(N) = toBinary(N, []);

toBinary(0, Acc) => Acc;
toBinary(N, Acc) => toBinary(N/2, [N%2 | Acc]);
```

Execution Sequence

```
toBinary(37, []) ==>
toBinary(18, [1]) ==>
toBinary(9, [0,1]) ==>
toBinary(4, [1,0,1]) ==>
toBinary(2, [0,1,0,1]) ==>
toBinary(1, [0,0,1,0,1]) ==>
toBinary(0, [1,0,0,1,0,1]) ==>
[1,0,0,1,0,1]
```

Conversion from Binary

- Could write **iterative** pseudo-code, then construct recursive equivalent.

```
L = …list to be converted…
Result = 0;
while (L != [ ]) {
  Result= 2*Result+first(L);
  L = rest(L);
}
… answer is in Result ...
```

```
fromBinary([],Result)=> Result;
fromBinary([F|R], Result) =>
      fromBinary(R,2*Result+F);
fromBinary(L) =
      fromBinary(L, 0);
```

```
fromBinary([1,0,0,1,0,1], 0) ==>
fromBinary([0,0,1,0,1], 1) ==>
fromBinary([0,1,0,1], 2) ==>
fromBinary([1,0,1], 4) ==>
fromBinary([0,1], 9) ==>
fromBinary([1], 18) ==>
fromBinary([], 37) ==>
37
```

Guards! Guards!

- Sometimes when choosing which case of a function to evaluate, we want more information than whether the argument matches a pattern or not.
- Rex provides two sorts of *guards*, which are extra conditions that must be satisfied before a case will be chosen.

"Normal" Guards

- Definitions of the form

  `F(`*`pattern`*`) => `*`guard`*` ? `*`expression;`*

  - This case applies only when the argument matches the given pattern *and* the guard expression evaluates to true.
- Example: Euclid's algorithm

```
gcd(0,Y) => Y;
gcd(X,Y) => X<=Y ? gcd(Y-X,X);
gcd(X,Y) => X>Y ? gcd(Y,X);
```

"Equational" "Guards"

- Definitions of the form

  `F(`*`pattern`*`) => `*`definitions`*`, `*`expression`*`;`

  Note that there's a `,` instead of a `?`
- This case applies only when the argument matches the given pattern *and* the definitions succeed
  - in this case, variables defined in the definitions can be used in computing the function's value
  - How rex defines local variables.
  - Advice: only use definitions that should *succeed*.

Example

- Raising a number to the fourth power:

```
hypercube(X) => Y=X*X, Y*Y;
```

Compare with:

```
hypercube(X) => (X*X)*(X*X);
```

Insertion Sort

```
// insertion_sort sorts its argument list
insertion_sort([]) => [];
insertion_sort([F|R]) =>
   insert(F, insertion_sort(R));

// insert(Y,Z) inserts element Y into the
// correct position in the sorted list Z.
insert(A,[]) => [A];
insert(A,[B|X]) =>
  (A<B ? [A,B|X] : [B | insert(A,X)]);
```

Selection Sort

```
// selection_sort sorts its argument list
selection_sort([]) => [];
selection_sort(L) =>
   [M | R] = min_to_front(L),
   [M | selection_sort(R)];

// min_to_front(L) moves a minimal element of
// the non-empty list L to the front (but may
// not preserve the order of the other elements)
min_to_front([A]) => [A];
min_to_front([A|L]) =>
   [B | R] = min_to_front(L),
   (A<B ? [A,B|R] : [B,A|R]);
```

Functions Returning Functions

- Consider `(X) => 5*X`
  - A perfectly valid, although anonymous, function
- Functions in rex are “first-class citizens”:
  - They can be passed as **arguments**.
  - They can be put into **lists**.
  - They can be **returned** as values.
- A function-value returning function:

  `scaleBy(Y) = (X) => Y*X;`

  - Here `scaleBy(Y)` returns a function, the function that scales its argument by `Y`.

Use of scaleBy

- Given the definition

```
scaleBy(Y) = (X) => Y*X;
```

we have

```
map(scaleBy(5), [1,2,3]) ==> [5,10,15]
scaleby(4)(7) ==> 28
```

- We could have defined

```
scale(Y, L) = map(scaleBy(Y), L);
```

`mapster`

```
mapster(F) = (L) => map(F, L);
```

- Defines a function such that, for any argument function F, returns a function that will map F over a list.

- If

```
f = mapster(even);
```

then

```
f([1,2,3,4]) ==> [0,1,0,1]
mapster(even)([1,2,3,4]) ==> [0,1,0,1]
```

Curried Functions (yum!)

- `scaleBy` and `mapster` are examples of *curried* functions
  - They take their arguments via sequential applications, each returning a function until the last argument.
- Another way to write them in rex is:

```
scaleBy(F)(X) = F*X;
mapster(F)(L) = map(F, L);
```

- Even with this notation, `scaleBy(F)` has a meaning without the `(X)`.

Anonymous returning Anonymous

- Yet another way to write `scaleBy` and `mapster` in rex is:

```
scaleBy = (F) => (X) => F*X;
mapster = (F) => (L) => map(F, L) ;
```

- The names `scaleBy` and `mapster` are incidental
  - Right-hand sides of the equations can be applied directly

Curried Functions in Math, Science, and Engineering

- In a math, science, or engineering text, you might see

  $f_k(x)$

- Typical nomenclature is that x is the “argument” and k is a “parameter”.
- **Both** k and x may be viewed as arguments to f, though typically k is “more fixed”.
- We could define this as f(k)(x) = ...
- Then we can use f(k) as f_k would be used.