Lower-Level Implementations of Higher-Level Functions

September 19, 2001

Defining Functions by Rules

• If we want to define a function taking a single list as its argument, it is sufficient to
  - Say what the function does for the empty list
  - Say what the function does for non-empty lists

• For functions taking multiple list arguments, there are even more choices:
  - Specify all four combinations of empty/non-empty
  - Cases for one argument empty/non-empty with the other argument being arbitrary.
  - Or, something else.

Example

• Define the function double_all, which multiplies every element in a list by 2:

\[
\text{double_all}([]) => []; \\
\text{double_all}([F|R]) => [F*2 | double_all(R)]
\]

• We could have used \text{map} for this function.
  - Use higher-order functions when it’s appropriate
  - Use lower-level definitions when you need to.

Review

• Last class we talked about defining functions using pattern-matching and rewrite rules:

\[
\text{equals}([], []) => 1; \\
\text{equals}([A|L],[A|M]) => \text{equals}(L,M); \\
\text{equals}(X, Y) => 0;
\]

• Today we'll look at more examples.
Tracing the Computation

double_all([1,2,3]) ==>  
[2 | double_all([2,3])] ==>  
[2 | [4 | double_all([3])]] ==>  
[2 | [4 | [6 | double_all([])]]] ==>  
[2 | [4 | [6 | []]]] ==  
[2, 4, 6]  

Using More Readable Notation

double_all([1,2,3]) ==>  
[2 | double_all([2,3])] ==>  
[2, 4 | double_all([3])] ==>  
[2, 4, 6 | double_all([])] ==>  
[2, 4, 6 | []] ==  
[2, 4, 6]  

member

- Checks whether an item is an element of a list, returning 1 if so and 0 otherwise.
  
  member(7, [2, 3, 5, 7, 11]) ==> 1

- Definition(s)?

map

- Definition for map(F, L):
reduce

• Definition for reduce(F,B,L):

append

• Definition for append?

• How much time is required to do an append?

factorial revisited

Definition using pattern-matching:

```plaintext
fact1(0) => 1;
fact1(N) => N*fact1(N-1);
```

Now consider this alternate definition:

```plaintext
fact2(N) = fact2a(N, 1);
fact2a(0,Acc) => Acc;
fact2a(N,Acc) => fact2a(N-1,N*Acc);
```

Comparison

```plaintext
fact1(3) =>
3 * fact1(2) =>
3 * (2 * fact1(1)) =>
3 * (2 * (1 * fact1(0))) =>
3 * (2 * (1 * 1)) =>
3 * (2 * 1) =>
3 * 2 =>
6.
```

```plaintext
fact2a(3,1) =>
fact2a(2,3*1) =>
fact2a(2,3) =>
fact2a(1,2*3) =>
fact2a(1,6) =>
fact2a(0,1*6) =>
6.
```
Tail Recursion

• A function whose code which does no work after a recursive call is said to be tail-recursive.
  - fact1 is not tail-recursive, but fact2a is.
  - Can be more space-efficient than non-tail-recursive code because we don't have to "stack up" work to be done later.
• Can sometimes make functions tail-recursive by adding an accumulator argument
  - But increases complexity of the definition.

reverse

• Definition using an accumulator?

reverse(L) = reverse_app(L,[]);
reverse_app([],Acc) =>
reverse_app([F|R],Acc) =>

• Which is more efficient?
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
  \[ F(pattern) \Rightarrow guard \ ? \ expression; \]
  
  Note that there's a ? but no :

- This case applies only when the argument matches the given pattern and the guard expression evaluates to true.

Example

- Euclid's algorithm:
  \[
  \begin{align*}
  \text{gcd}(0, Y) & \Rightarrow Y; \\
  \text{gcd}(X, Y) & \Rightarrow X\leq Y \ ? \ \text{gcd}(Y-X, X) ; \\
  \text{gcd}(X, Y) & \Rightarrow X>Y \ ? \ \text{gcd}(Y, X) ;
  \end{align*}
  \]

  Which of these guards is redundant?

"Equational Guards"

- Definitions of the form
  \[ F(pattern) \Rightarrow definitions, expression; \]
  
  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:

\[
\text{hypercube}(X) \Rightarrow Y = X \times X, Y \times Y;
\]

Compare with:

\[
\text{hypercube}(X) \Rightarrow (X \times X) \times (X \times X);
\]

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]) => 

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|X] : [B,A|X]);
Binary Representation

- The function toBinary(N) should return a list of 0's and 1's corresponding to the binary representation of the 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]

First Try

- How about this?

  toBinary1(0) => 0
  toBinary1(N) => [N%2 | toBinary1(N/2)]

Second Try

- How about this?

  toBinary(N) = toBinary2(N, []);
  toBinary2(0, Acc) => Acc;
  toBinary2(N, Acc) => toBinary2(N/2, [N%2 | Acc]);

Comparison

  toBinary2(37, []) =>
  toBinary2(18, [1]) =>
  toBinary2(9, [0,1]) =>
  toBinary2(4, [1,0,1]) =>
  toBinary2(2, [1,0,1,0]) =>
  toBinary2(1, [1,0,1,0,0]) =>
  toBinary2(0, [1,0,1,0,0,1]) =>
  [1,0,0,1,0,1]

  toBinary1(37) =>
  toBinary1(18) =>
  toBinary1(9) =>
  toBinary1(4) =>
  toBinary1(2) =>
  toBinary1(1) =>
  toBinary1(0) =>
  [1,0,0,1,0,1]