CSc 520

Principles of Programming Languages

12: Haskell — Function Definitions

Christian Collberg

collberg@cs.arizona.edu

Department of Computer Science

University of Arizona

Defining Functions

- When programming in a functional language we have basically two techniques to choose from when defining a new function:
  1. Recursion
  2. Composition
- Recursion is often used for basic “low-level” functions, such that might be defined in a function library.
- Composition (which we will cover later) is used to combine such basic functions into more powerful ones.
- Recursion is closely related to proof by induction.

Defining Functions...

- Here’s the ubiquitous factorial function:

```
fact ::  Int -> Int
fact n = if n == 0 then
         1
         else
         n * fact (n-1)
```

- The first part of a function definition is the type signature, which gives the domain and range of the function:

```
fact ::  Int -> Int
```

- The second part of the definition is the function declaration, the implementation of the function:

```
fact n = if n == 0 then ···
```


Defining Functions...

- The syntax of a type signature is

```
fun_name ::  argument_types
```

`fact` takes one integer input argument and returns one integer result.

- The syntax of function declarations:

```
fun_name param_names = fun_body
```

- `if` e_1 `then` e_2 `else` e_3 is a conditional expression that returns the value of e_2 if e_1 evaluates to `True`. If e_1 evaluates to `False`, then the value of e_3 is returned. Examples:

```
if False then 5 else 6       ⇒  6
if 1==2 then 5 else 6        ⇒  6
5 + if 1==1 then 3 else 2    ⇒  8
```

Defining Functions...

- `fact` is defined recursively, i.e. the function body contains an application of the function itself.
- The syntax of function application is: `fun_name arg`. This syntax is known as “juxtaposition”.
- We will discuss multi-argument functions later. For now, this is what a multi-argument function application (“call”) looks like:

```
fun_name arg_1 arg_2 ··· arg_n
```

- Function application examples:

```
fact 1        ⇒ 1
fact 5        ⇒ 120
fact (3+2)    ⇒ 120
```


Standard Recursive Functions

- Typically, a recursive function definition consists of a guard (a boolean expression), a base case (evaluated when the guard is `True`), and a general case (evaluated when the guard is False).

```
fact n =
  if n == 0 then             ⇐ guard
    1                        ⇐ base case
  else
    n * fact (n-1)           ⇐ general case
```


Simulating Recursive Functions

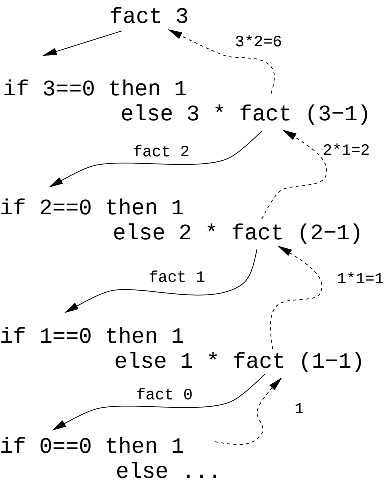
- We can visualize the evaluation of `fact 3` using a tree view, box view, or reduction view.
- The tree and box views emphasize the flow-of-control from one level of recursion to the next
- The reduction view emphasizes the substitution steps that the `hugs` interpreter goes through when evaluating a function. In our notation boxed subexpressions are substituted or evaluated in the next reduction.
- Note that the Haskell interpreter may not go through exactly the same steps as shown in our simulations. More about this later.

Tree View of `fact 3`

```
          fact 3
if 3==0 then 1
      else 3 * fact (3-1)
              fact 2
if 2==0 then 1
      else 2 * fact (2-1)
              fact 1
if 1==0 then 1
      else 1 * fact (1-1)
              fact 0
if 0==0 then 1
      else ...
```

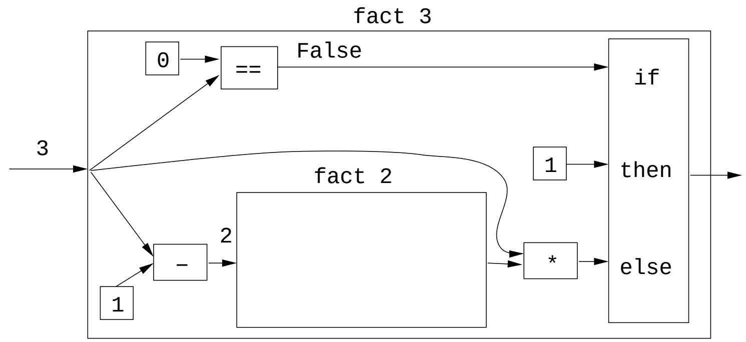
- This is a Tree View of fact 3.
- We keep going deeper into the recursion (evaluating the general case) until the guard is evaluated to `True`.

Tree View of `fact 3`

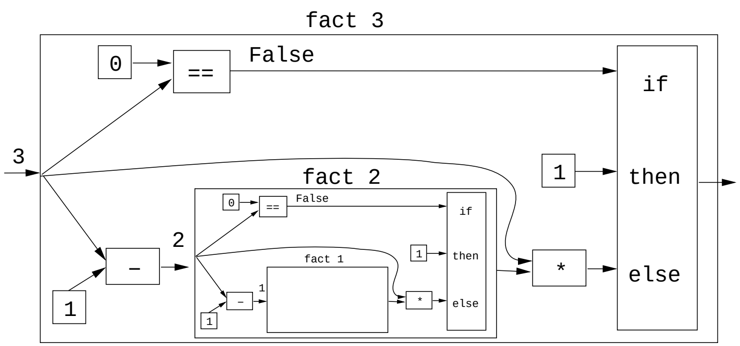


- When the guard is `True` we evaluate the base case and return back up through the layers of recursion.

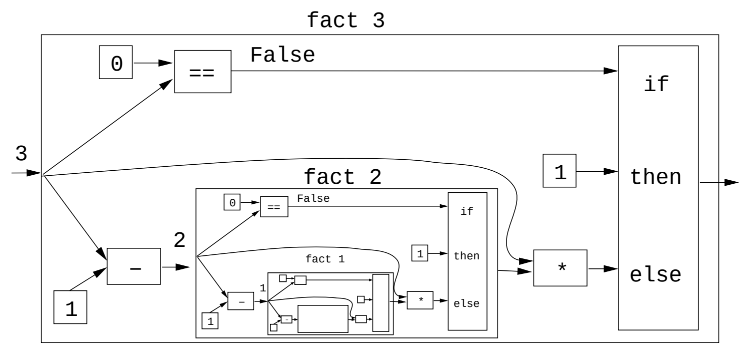
Box View of `fact 3`



Box View of `fact 3`...



Box View of `fact 3`...



Reduction View of fact 3

```
fact 3 ⇒
if 3 == 0 then 1 else 3 * fact (3-1) ⇒
if False then 1 else 3 * fact (3-1) ⇒
3 * fact (3-1) ⇒
3 * fact 2 ⇒
3 * if 2 == 0 then 1 else 2 * fact (2-1)⇒
3 * if False then 1 else 2 * fact (2-1) ⇒
3 * (2 * fact (2-1)) ⇒
3 * (2 * fact 1) ⇒
3 * (2 * if 1 == 0 then 1 else 1 * fact (1-1))
        ⇒ ···
```


Reduction View of fact 3...

```
3 * (2 * if 1 == 0 then 1 else 1 * fact (1-1)) ⇒
3 * (2 * if False then 1 else 1 * fact (1-1)) ⇒
3 * (2 * (1 * fact (1-1))) ⇒
3 * (2 * (1 * fact 0)) ⇒
3 * (2 * (1 * if 0 == 0 then 1 else 0 * fact (0-1))) ⇒
3 * (2 * (1 * if True then 1 else 0 * fact (0-1))) ⇒
3 * (2 * (1 * 1)) ⇒
3 * (2 * 1) ⇒
3 * 2 ⇒
6
```


Recursion Over Lists

- In the `fact` function the guard was `n==0`, and the recursive step was `fact(n-1)`. I.e. we subtracted 1 from `fact`'s argument to make a simpler (smaller) recursive case.
- We can do something similar to recurse over a list:
  1. The guard will often be `n==[ ]` (other tests are of course possible).
  2. To get a smaller list to recurse over, we often split the list into its head and tail, `head:tail`.
  3. The recursive function application will often be on the tail, `f tail`.

The length Function

In English:

The length of the empty list `[ ]` is zero. The length of a non-empty list S is one plus the length of the tail of S.

In Haskell:

```
len ::  [Int] -> Int
len s =  if s == [ ] then
            0
         else
            1 + len (tail s)
```

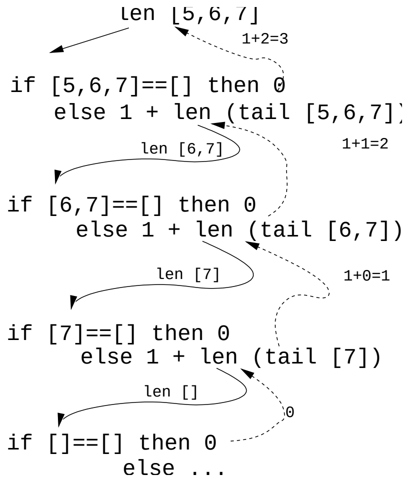
- We first check if we've reached the end of the list `s==[ ]`. Otherwise we compute the length of the tail of s, and add one to get the length of s itself.

Reduction View of `len [5,6]`

```
en s = if s == [ ] then 0 else 1 + len (tail s)

en [5,6] ⇒
  if [5,6]==[ ] then 0 else 1 + len (tail [5,6]) ⇒
  1 + len (tail [5,6]) ⇒
  1 + len [6] ⇒
  1 + (if [6]==[ ] then 0 else 1 + len (tail [6])) ⇒
  1 + (1 + len (tail [6])) ⇒
  1 + (1 + len [ ]) ⇒
  1 + (1 + (if [ ]==[ ] then 0 else 1+len (tail [ ]))) ⇒
  1 + (1 + 0)) ⇒ 1 + 1 ⇒ 2
```


Tree View of `len [5,6,7]`




```
len ::  [Int] -> Int
len s = if s==[ ] then 0
        else 1+len(tail s)
```

- Tree View of `len [5,6,7]`