15-150 Lecture 3: Recursion; Structural Induction

Lecture by Dan Licata

January 24, 2012

Today, we are going to talk about one of the most important ideas in functional programming, *structural recursion* and *structural induction*.

1 Structural Recursion

Our take on recursion will be different than what you may have seen before. The main idea is that *recursive functions come from recursive data*.

For example, we can define

> A natural number is either
>
> 0, or
>
> $1 + n$, where n is a natural number,
>
> *and that's it!*

Note that this definition is self-referential. It is what is called an *inductive definition* of the natural numbers. Using this definition, we know that 0, 1+0, 1+(1+0), ... are natural numbers. Moreover, the *and that's it!* tells us that we can compute with a natural number using what's called *structural recursion*.

To illustrate this, to a first approximation, we can represent natural numbers as integers, but this is two kinds of a lie:

- There are too many integers (negative numbers like `~3`)
- There are also too few (nothing about the above definition of natural numbers says they have fixed size).

But let's pretend and use `int` now; we'll show you how to do better later in the course, when we talk about *datatypes*.

1.1 Double

An example is the double function.

```
(* Purpose: double the number n

   Examples:
   double 0 ==> 0
```

```
   double 2 ==> 4
*)
fun double (n : int) : int =
    case n of
        0 => 0
      | _ => 2 + (double (n - 1))
```

Note that a function defined by a `fun` declaration can be used at the declared type in its own definition. E.g. `double` can be used with type `int -> int`, so the recursive call type checks.

For evaluation, we can calculate as follows:

```
    double 2
|-> case 2 of 0 => 0 | _ => 2 + (double (2 - 1))
|-> 2 + (double (2 - 1))
|-> 2 + (double 1)
|-> 2 + (case 1 of 0 => 0 | _ => 2 + (double (1 - 1)))
|-> 2 + (2 + (double (1 - 1)))
|-> 2 + (2 + (double 0))
|-> 2 + (2 + (case 0 of 0 => 0 | _ => 2 + (double (0 - 1))))
|-> 2 + (2 + 0)
|-> 2 + 2
|-> 4
```

Question: what if we forgot to subtract 1 ($n - 1$) in the argument to the recursive call? Then double calculates like this:

```
    double 2
|-> case 2 of 0 => 0 | _ => 2 + (double 2)
|-> 2 + (double 2)
==> 2 + (2 + (double 2))
==> 2 + (2 + (2 + (double 2)))
...
```

I.e. it is an infinite loop.

1.2 Case

`case` is a new operation on natural numbers. It lets you distinguish whether the number is `0` or not. On the first line, you give the result for when `n` is `0`; on the second, you give the result for when it is not—the `_` means "any other number falls into this clause."

Evaluation: It calculates as follows:

```
case 0 of 0 => <branch1> | _ => <branch2>   |->  <branch1>
case n of 0 => <branch1> | _ => <branch2>   |->  <branch2>
  if n is a non-zero numeral
```

Additionally, we always calculate the scrutinee first: e.g. in `case (1 - 1) of 0 => ... | _ => ...` you need to do the subtraction to see that it should compute to the `0` branch. Second, we don't

calculate the branches until we pick one using the above rules: the branch that is not selected never gets run.

Typing: `case e of 0 => <branch1> | _ => <branch2>` has type τ if

`e : int`

`<branch1>` : τ

`<branch2>` : τ

The two branches must have the same type. This rules out programs like
`case 0 of 0 => 1 | _ => ``hi''`.

There are two points of view on types, called *intrinsic* and *extrinsic* semantics. In intrinsic semantics, types are conceptually prior, and terms inherently inhabit particular types. From this point of view, it doesn't make sense to run an ill-typed term. This is how we will usually think of ML (cf. the picture we drew in lecture 2: valuable expressions are a subset of well-typed expressions). From this point of view, the above program doesn't contribute anything to a particular type, so it is deemed ill-typed.

An alternative is *extrinsic* semantics, where programs are conceptually prior and evaluation is defined for ill-typed terms, and types are thought of as predicates on programs: `e:T` iff `e` evaluates to a value of type `T`. Under this interpretation, `case 0 of 0 => 1 | _ => ``hi''` would have type `int`, because it would always evaluates to a numeral. However, this requires a much more complex type system where you run code during type-checking.

Questions: A student asked whether the order of the branches matters. Yes, it does, because the `_` in fact matches anything, so putting it first would catch the 0 case, too. Just follow the above order for now; we will talk about more general uses of `case` soon.

1.3 Factorial

```
(* Purpose: compute n! = n * (n-1) * (n-2) * ... * 1

   Examples:
   fact 0 ==> 1
   fact 5 ==> 120
*)

fun fact (n : int) : int =
    case n of
        0 => 1
      | _ => n * (fact (n - 1))
```

Note that it follows the same pattern as `double`: you give a case for zero, and a case for $1 + n$, in which you make a recursive call on $n - 1$.

1.4 Tests

Testing is checking that a program works by trying it on some examples. There are three ways you can write tests.

The first is just to open up the REPL and play around:

```
- fact 3;
val it = 6 : int
- fact 5;
val it = 120 : int
```

However, it's good to save your tests for later, so that when you change your code, you can run them again. This is called *regression testing*. For this class, we also want you to hand in your tests. Thus, it's good to write them in your `.sml` file.

One way to do this is to put declarations like

```
val f5 = fact 5
```

in your `.sml` file, in which case when you `use` it, you will see

```
val f5 = 120 : int
```

so you can go check the results.

Another way is to use a new feature: you can actually put a numeral (or string constant, etc.) in a `val` binding:

```
val 120 = fact 5
```

This asks SML to check that the result of `fact 5` is `120`. If it is not, then you will see

```
uncaught exception Bind [nonexhaustive binding failure]
  raised at: fact.sml:36.5-36.17
```

This means that the constant in the `val` binding did not match the actual value, and gives you the line number of the failed test.

A downside of this style is that none of the definitions in the file are loaded into the REPL until all of the tests pass. Thus, we suggest that you use the REPL directly while you are developing, and then use `val 120 = fact 5` for regression testing after the fact.

1.5 Five-Step Methodology

In writing this function, we followed an important methodology, which we will ask you to follow whenever you write a function:

1. Write the name and type signature (e.g. `fun fact (n : int) : int = ...`)
2. Write the purpose in a comment (e.g. `Purpose: compute n!`)
3. Write some examples of how the function is supposed to evaluate (e.g. `fact 5 ==> 120`)
4. Write the function body (this is the interesting part)
5. Turn your examples into tests (e.g. `val 120 = fact 5`)

1.6 Doctor

Now that I am a doctor, it's important for me to learn how to make people say "aaaaaaa". In pairs, do this:

```
(* Purpose: compute the string "aa....a" with n letters

   Examples:
   doctor 0 ==> ""
   doctor 3 ==> "aaa"
*)
fun doctor (n : int) : string =
    case n of
        0 => ""
      | _ => "a" ^ (doctor (n - 1))

(* Tests *)
val "" = doctor 0
val "aaa" = doctor 3
```

For example, `doctor 2` calculates like this:

```
    doctor 2
|-> case 2 of 0 => "" | _ => "a" ^ (doctor (2 - 1))
|-> "a" ^ (doctor (2 - 1))
|-> "a" ^ (doctor 1)
|-> "a" ^ (case 1 of 0 => "" | _ => "a" ^ (doctor (1 - 1)))
|-> "a" ^ ("a" ^ (doctor (1 - 1)))
|-> "a" ^ ("a" ^ (doctor 0))
|-> "a" ^ ("a" ^ (case 0 of 0 => "" | _ => "a" ^ (doctor (0 - 1))))
|-> "a" ^ ("a" ^ "")
|-> "a" ^ "a"
|-> "aa"
```

1.7 The Pattern of Structural Recursion

Recall the definition of a natural number:

> A natural number is either
> 0, or
> $1 + n$, where n is a natural number.
>
> *and that's it!.*

This tells us that

- The **values** of natural number type are 0, 1, 2 ...
- The **operation** on natural numbers is structural recursion.

In particular, this helps us with step 4 of the methodology (write the function body), when we are writing a function whose argument is a natural number. To define a function a function whose argument is a natural number, it suffices to (1) give a case for zero, and (2) give a case for any other number n in terms of the result on its predecessor $n - 1$.

In the abstract, this looks like:

```
fun f (n : int) : T =
   case n of
      0 => ...
    | _ => ... f (n - 1) ...
```

1.8 Just for laughs

I have this friend, who on IM, laughs a lot. If I say something even mildly funny, he'll say "hahahahahahahahahahahahahahahahaha". So I feel bad reciprocating with "haha". Thus, I wrote the following program:

```
(* Purpose: compute the string "hahaha..." with n total characters
            starting with "a" if n is odd

   Examples:
   ha 0 ==> ""
   ha 1 ==> "a"
   ha 2 ==> "ah"
   ha 3 ==> "aha"
   ha 4 ==> "haha"
   ha 6 ==> "hahaha"
*)
fun ha (n : int) : string =
    case n of
        0 => ""
      | _ => (case evenP n of
                   true => "h" ^ (ha (n - 1))
                 | false => "a" ^ (ha (n - 1)))
(* Tests *)
val "" = ha 0
val "a" = ha 1
val "ha" =  ha 2
val "aha" = ha 3
val "haha" = ha 4
val "hahaha" = ha 6
```

Here we have assumed a function `evenP : int -> bool` that tests whether an integer is even. The $n + 1$ branch of this case-analysis is more complicated than we have seen before, because it does a second case-analysis on something computed from the argument of the function—whether or not it is even.

What is a boolean?

A boolean is either	`true`, or `false`

and that's it!.

This means that the type of `bool` (booleans) is defined as follows:

- Values: `true` and `false`
- Operations: `case b of true => <branch1> | false => <branch2>`. Case is type-checked and evaluated just like for natural numbers.

In a couple of weeks, we will start using the ML `datatype` definition mechanism, which lets you define natural numbers, booleans, etc. in this style in your code, not just in words.

An important thing to realize is that *case is an expression*—there is no distinction between expressions and statements, like in C. So we can make the code more concise by pulling the recursive call out of the `case`:

```
fun ha' (n : int) : string =
    case n of
        0 => ""
      | _ => (case evenP n of
                  true => "h"
                | false => "a")
             ^ (ha' (n - 1))
```

2 Structural Induction and Equivalence

The next thing to cover is proving theorems about your code by induction. This is a really important skill: first, it lets you formally justify the correctness of your code. Second, it draws out the kind of reasoning you have to do when you're writing any code, whether you do the proofs explicitly or not.

As a running example, we'll use exponentiation:

```
(* Purpose: compute 2^n, where n is a natural number
   Examples: exp 0 ==> 1
             exp 4 ==> 16
*)

fun exp (n : int) : int =
    case n of
        0 => 1
      | n => 2 * exp (n-1)
```

How does the last case work? Well, `exp (n - 1)` computes $2^{(n-1)}$, and $2 * 2^{n-1} = 2^n$. This reasoning is exactly what it takes to prove this function correct.

2.1 Equivalence

To formally prove the correctness of `exp`, we want to state a theorem like

Theorem 1. *For all natural numbers n, `exp` $n = 2^n$.*

This theorem says that the ML program `exp` n equals the numeral 2^n. Here the n is a mathematical variable, which stands for any numeral, not an ML variable. We will allow ourselves to use math variables like n, and expressions like 2^n, in code as names for values. For example, for each numeral n, 2^n stands for the numeral $2 \times 2 \times 2 \times \ldots 2$ with n 2's, or for 1 if n is 0.

But what does `=` mean? *When are two programs equal?* This is somewhat subtle, because programs can go into an infinite loop, or raise an exception. We will define a notion of *program equivalence* that accounts for these possibilities. To make it clear that we're doing something kind of subtle, we'll use $\cong$(or `==`in ASCII) to notate equivalence.

Here's the basic idea: two programs are equivalent iff

- They both evaluate to the same value
- They both raise the same exception
- They both infinite loop

We will refine our definition of equivalence over time, but here are some rules that will let us get started:

- Equivalence is an *equivalence relation*: it is reflexive, symmetric, and transitive.

```
e == e
e1 == e2 if e2 == e1
e1 == e3 if e1 == e2 and e2 == e3
```

- Equivalence is a *congruence*: if two expressions are equal, then you can substitute one for the other inside any bigger program
- Equivalence contains evaluation: if `e |-> e'` then `e == e'`.

2.2 Example: Correctness of `exp`

Theorem 2. *For all natural numbers n, `exp` $n \cong 2^n$.*

There are two main ingredients in the proof:

- Doing calculations in ML and in math
- Appealing to the *inductive hypothesis*: when proving a theorem about natural numbers, we get to assume the theorem is true for k while proving it for $1 + k$.

We assume that `*` correctly implements $\times$, etc.

Proof. The proof is by inudction on n, using the predicate $P(x) := \texttt{exp}\, x \cong 2^x$

Case for 0**.**

To show: $\texttt{exp}\, 0 \cong 2^0$.

Proof:

	`exp` 0	
$\cong$	`case 0 of 0 => 1 | _ => 2 * exp (0 - 1)`	step
$\cong$	1	step
$\cong$	2^0	math

On the right, we write a justification for each equivalence. *step* means "these two expressions are equivalent because one steps to the other." *math* means "I'm using some basic mathematical fact" (we'll tell you what you're allowed to use). In this case the definition of 2^n says that 2^0 is 1.

Case for $1 + k$**.** Inductive hypothesis: $\texttt{exp}\, k \cong 2^k$

To show: $\texttt{exp}\, (1 + k) \cong 2^{1+k}$.

Proof:

	`exp`$(1 + k)$	
$\cong$	`case`$(1 + k)$ `of 0 => 1 | _ => 2` $*$ `exp ((`k`+1) - 1)`	step (k is a value)
$\cong$	$2 *$ `exp ((1+`k`) - 1)`	step (k is a value)
$\cong$	$2 *$ `exp` k	math
$\cong$	$2 * 2^k$	IH
$\cong$	2^{1+k}	math

In the second step, we reduce a `case` on $k+1$ to the non-zero branch—this correct since $1 + k \neq 0$, and k is a value. In the second-to-last step, we use the IH to replace `exp` k with 2^k inside the bigger expression $2 * -$, exploiting the congruence property of equivalence. In the final step, we use a mathematical fact about exponents.

□

Note that the structure of the proof mirrors the structure of the code: in the code, we have a zero case and an $n + 1$ case; same in the proof. In the code, we have a recursive call on $n - 1$; in the proof, we have an inductive hypothesis for $n - 1$.

2.3 Template for Structural Induction on Natural Numbers

Here's the format that any proof by induction on the natural numbers should have:

Induction is applicable when the statement to be proved has the form "for all natural numbers n, [some predicate] holds of n".

The proof should fill in the following skeleton:

Proof. The proof is by induction on n.

Case for 0**.**

To show: [substitute 0 into the predicate]

Proof: ...

Case for $1 + k$**.**

Inductive hypothesis: [substitute k into the predicate].

To show: [substitute $1 + k$ into the predicate].

Proof: ... Be sure to cite when you use the inductive hypothesis. ... □

Note the similarities with the template for structural recursion on the natural numbers!

The remainder of this material will be covered in lab tomorrow.

3 Additional Recursion Patterns

Sometimes, the natural way to write a function is not to recur on $n-1$. Another way to write a function on the natural numbers is to give (1) a case for 0, (2) a case for 1, and (3) a case for $n+2$ in terms of a recursive call on n. This recursion principle can be formally justified using induction, but it should be intuitively clear: every natural number is either 0, or it's 1 or it's $2+n$, where n is a natural number. We can use this to define `evenP` (note that this definition must come before `ha` in your SML file, because `ha` uses `evenP`. I flipped the order here because I wanted to talk about `ha` first.)

```
(* Purpose: determine whether the number is even

   Examples:
   evenP 0 ==> true
   evenP 3 ==> false
   evenP 12 ==> true
   evenP 27 ==> false
*)
fun evenP (n : int) : bool =
    case n of
        0 => true
      | 1 => false
      | _ => evenP (n - 2)

val true = evenP 0
val false = evenP 1
val true = evenP 12
val false = evenP 27
```

4 Pairs and Let

4.1 Pairs

The type of pairs is written $\tau_1 * \tau_2$. The values of this type are written `(e1 , e2)` where `e1` is a value of type τ_1 and `e2` is a value of type τ_2. E.g. `(3, "hi")` is a value of type `int * string`.

What operations are there on pairs? You can get the two pieces out. There are three ways to do this, using `let`, in the argument to a function, and using `case` (with one branch):

```
fun geom (p : int * int) =
    let val (x : int , y : int) = p in
        (x * y , 2 * (x + y))
    end

fun geom (x : int , y : int) = (x * y , 2 * (x + y))

fun geom (p : int * int) =
    case p of
        (x : int , y : int) => (x * y , 2 * (x + y))
```

These reduce like you might expect: when you given them an actual pair, like `(3,5)`, the components are plugged in for the variables in each position. E.g.

```
let val (x : int , y : int) = (3,5) in (x * y , 2 * (x + y)) end
|-> (3 * 5 , 2 * (3 + 5))
```

Why are there different ways of doing the same thing? They are all an instance of a general concept called pattern-matching, which we will talk about "soon".

4.2 Let

`let` is useful not just for taking apart pairs, but in general for naming intermediate results. For example:

```
fun geom (x : int , y : int) =
   let val area : int = x * y
       val perim : int =  2 * (x + y)
    in (area, perim)
    end
```

Typing is just like `val` declarations at the top-level: the expression must have the type the variable is annotated with. So is scoping: the variables are in scope in later declarations, and in the *body*, the expression between `in` and `end`. That is, `let <decls> in <expr> end` is well-typed if the `<decls>` are, and if `<expr>` is well-typed using the variables bound in the declarations.

Evaluation: To evaluate a `let`, you first evaluate the `<decls>`, substituting into subsequent declarations and `<expr>` as you go. Then, when you're done with all of the `<decls>`, you evaluate the body `<expr>`, and its value is the value of the whole `let`.

Because you substitute the result of evaluation, you can use `let` to avoid duplicating the same work twice. E.g you only add `2+3` once in the following example, even though `x` gets mentioned twice:

```
val 11 = let val x : int = 2 + 3
             val y : int = x + 1
         in x + y
         end
```