Structural Induction and Recursion

COMP 302

PART 03

Structural Induction and Recursion

Many datatypes can be defined recursively

One of the fundamental ideas of functional programming is that recursive functions can be build from recursive data

For example, we can define a Natural Number as

- 0 is a NN
- If n is a NN then 1+n is a NN
- NN is the smallest set satisfying these properties

For now we will use type int to represent these (although int is a bigger set)

Later we talk about datatypes that will allow us to represent them directly

A Function on NN

We can define a function directly tied to the definition of natural numbers:

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

Powers of 2

Here's another example (factorial is a bit tiresome)

```
fun powof2 (n : int) : int =
  case n of
     0 => 1
   | n => 2 * (powof2 (n-1))
```

Evaluation by Substitution

```
powof2 2
|-> case 2 of 0 => 0 | _ 2 + (powof2 (2-1))
|-> 2 + (powof2 (2-1))
|-> 2 + (powof2 1)
|-> 2 + (case 1 of 0 => 0 | _ 2 + (powof2 (1-1)))
|-> 2 + (2 + (powof2 (1-1)))
|-> 2 + (2 + (powof2 0))
|-> 2 + (2 + (case 0 of 0 => 0 | _ 2 + (powof2 (0-1))))
|-> 2 + (2 + 0)
|-> 2 + 2
|-> 4
```

Infinite Regression

Forgetting to decrement n

```
fun powof2 (n : int) : int =
  case n of
     0 => 1
   | n => 2 * (powof2 (n))
```

```
powof2 2
|-> case 2 of 0 => 0 | _ 2 + (powof2 2)
|-> 2 + (powof2 2)
|-> 2 + (case 1 of 0 => 0 | _ 2 + (powof2 2))
|-> 2 + (2 + (powof2 2)
|-> . . .
```

Case

Case is an operation which works on natural numbers as well as other data types

It allows us to distinguish between different alternatives

In this instance between 0 and ‘_’ which means “any other value”

This will be a very useful operation later when we discuss datatypes

Case

More generally we have

```
case e
  of 0 => <branch1>
   | _ => <branch2>
```

Type-checking

- The two branches must have the same type and that is the type of the expression

Evaluation

- The expression e is evaluated in the current environment
- The alternatives are tested in order
- The branch corresponding to the first one that matches is evaluated

Case

We will see much more general uses of Case

The if expression of SML is unnecessary

```
if e1 then e2 else e3
```

Is equivalent to

```
Case e1 of
     true => e2
   | false => e3
```

The if construct is “syntactic sugar”

Proving Equivalence

We would like to prove that powof2 (n) computes 2^n (where 2^n is shorthand for 2 x 2 x 2 …, n times)

More generally, we would like techniques that allow us to determine whether two programs are equal.

What does that mean?

Equivalence

Two programs are equivalent iff

1. They both evaluate to the same value, or
2. They both raise the same exception, or
3. They both enter an infinite loop

Properties

1. Equivalence is an equivalence relation
2. Equivalence is a congruence (one program can be substituted for another)
3. If e |-> e' then e is equivalent to e'

Proof by Induction

Theorem:

For all natural numbers `powof2 n == 2^n` (where we use == to mean “is equivalent to”)

Proof is by mathematical induction on n

Proof – Base Case

Base case:

Show that `powof2 0 == 2^0`

Proof:

```
powof2 0
== case 0 of 0 => 1 | _ => 2 * powof2 (0 - 1)
== 1
== 2^0
```

(The last step is a mathematical fact)

Proof – Inductive Step

Induction Hypothesis: `powof2 k == 2^k`

Show that `powof2 (k + 1) == 2^(k+1)`

Proof:

```
powof2 (k + 1)
== case (k + 1) of 0 => 1 | _ => 2 * powof2 ((k+1) - 1)
== 2 * powof2 ((k + 1) - 1)
== 2 * powof2 (k)
== 2*2^k      (by IH)
== 2^(k+1)    (by math)
```

Methodology

The general methodology for proof by induction on natural numbers has the following structure:

The theorem has the form “For all natural numbers n, some predicate P(n) is true”

Proof by induction on n

- Base Case: Prove P(0) directly
- Inductive Hypothesis (IH): Assume P(k) is true
- Inductive Step: Prove that P(k+1) is true making use of IH

Other Patterns

While following the inductive definition of the data type directly often works, sometimes other patterns are needed

```
(* Determine whether a number is odd *)
fun odd (n : int) : bool =
  case n of
      0 => false
    | 1 => true
    | _ => odd (n-2)
```

The proof of correctness would require a small generalization of the methodology to have two base cases

Pairs and Tuples

Functions in SML take one argument and return one result

This has the advantage that the language is very uniform and it allows us a lot of flexibility in manipulating functions

But it looks like we can write functions that take more than one argument:

```
fun max (x:real, y:real):real:
  if x < y then y else x
```

Actually max takes one argument that is an ordered pair (a 2-tuple)

There are mechanisms in SML for building compound data from simpler data

For now we will look at “pairs”

Later we will define lists and records

Pairs

Pairs are values of type `<t1>*<t2>`

The values are written (e1 , e2) where e1:<t1> and e2:<t2>

For example (“Friedman” , 3 * 7) is a value of type string * int

There are different ways to access the elements of a pair

(This generalizes to n-tuples of type (t1 * t2 * … * tn)

Accessing the components of a pair

Using “#1 e” and “#2 e” allows us to extract the components of the pair e

We can rewrite max as

```
fun max (pair : real * real) : real =
  if (#1 pair) < (#2 pair) then
    (#2 pair)
  else
    (#1 pair)
```

This is equivalent to the first definition but less readable

The first definition actually uses pattern matching (to be discussed a bit later)

Rectangle geometry

The following examples assume you are given the dimensions of a rectangle as a pair

They give three different but equivalent ways to access these dimensions to compute the area and the perimeter of the rectangle

None of them access the components of the pair directly

They use pattern matching to define the functions

Note that the result is a pair. This would be a pain to do in Java or many other imperative languages

It is generally considered bad style to access the components directly and pattern matching is almost always used to define functions with multiple arguments

Functions on pairs (with pattern matching)

Given the dimensions of a rectangle, compute the area and perimeter (as a pair)

```
fun rect (x : real, y : real): real * real = (x * y , 2 * (x + y))
----
fun rect (dim : real * real) : real * real =
  case p of
    (x : real, y : real) => (x * y , 2 * (x + y))
----
fun rect (dim : real * real) : real * real =
  let val (x : real , y : real) = p in
    (x * y , 2 * (x + y))
  end
```

Pattern Matching on Pairs

fun foo (x : t1, y : t2) : t3 = e

- This is a function declaration that takes a pair as an argument
- It extracts the components of the pair and matches them again the parameters x and y
- The types must mach

val (x : t1, y : t2) = e

- e must be a pair
- The identifiers x and y are bound to the first and second component of e

Everything that we have described easily generalizes to n-tuples

Another example

```
fun swap (p : int * real) : real * int =
  (# 2 pr, #1 pr)
----
fun swap (x : int, y :  real) : real * int =
  (y, x)
```