CSci 4223
Principles of Programming Languages
Lecture 8
Prof. Clarkson
Spring 2013

Review
- Features learned: functions, tuples, lists, let expressions, options, records, datatypes, case expressions, type synonyms, pattern matching, exceptions
- Today:
  - Tail recursion, continued
  - Type variables
  - Equality types
  - Type annotations

Recursion
Should now be comfortable with recursion:
- No harder than using a loop (whatever that is)
- For recursive datatypes, recursion is more elegant than loops:
  - When processing a tree (e.g., evaluate an arithmetic expression)
  - When appending lists
- Now:
  - How to reason about efficiency of recursion
  - The importance of tail recursion
  - Using an accumulator to achieve tail recursion
  - Functional programming idiom (no new syntax/semantics)

Call-stacks
While a program runs, there is a call stack of function calls that have started but not yet returned
- Calling a function $f$ pushes an instance of $f$ on the stack
- When a call to $f$ to finishes, it is popped from the stack

These stack-frames store information like the value of local variables and "what is left to do" in the function

Because of recursion, multiple stack-frames may be calls to the same function

Example
\[
\text{fun fact n = if n=0 then 1 else n*fact(n-1)}
\]
\[
\text{val x = fact 3}
\]
Example Revised

```ml
fun fact n = 
  let fun aux(n,acc) = 
    if n=0 
      then acc 
      else aux(n-1,acc*n) 
  in 
  aux(n,1) 
end 
val x = fact 3
```

Still recursive, more complicated, but the result of recursive calls is the result for the caller (no remaining multiplication).

The call-stacks

```
fact 3
  aux(3,1)
    aux(3,1):
      aux(3,1):
    aux(2,3):
      aux(2,3):
    aux(1,6):
      aux(1,6):
    aux(0,6):
      aux(0,6):
        Etc.
```

An optimization

It is unnecessary to keep around a stack-frame just so it can get a callee’s result and return it without any further evaluation.

ML recognizes these tail calls in the compiler and treats them differently:
- Pop the caller before the call, allowing callee to reuse the same stack space
- Implementation turns out to be as efficient as a loop
- Not just ML, any good implementation of functional language does tail-call optimization

What really happens

```
fun fact n = 
  let fun aux(n,acc) = 
    if n=0 
      then acc 
      else aux(n-1,acc*n) 
  in 
  aux(n,1) 
end 
val x = fact 3
```

```
fact 3
  aux(3,1)
    aux(3,1):
      aux(3,1):
    aux(2,3):
      aux(2,3):
    aux(1,6):
      aux(1,6):
    aux(0,6):
      aux(0,6):
```

Moral

- Tail-recursive calls can be much more efficient
  - But there’s a tradeoff: makes code a little harder to read
  - So best done when efficiency is important
- Methodology to guide transformation of recursive function:
  - Create a helper function that takes an accumulator
  - Old base case becomes initial accumulator
  - New base case becomes final accumulator

Another example

```
fun sum xs = 
  case xs of 
    [] => 0 
    | x::xs' => x + sum xs'
```

```
fun sum xs = 
  let fun aux(xs,acc) = 
    case xs of 
      [] => acc 
      | x::xs' => aux(xs',x+acc) 
  in 
  aux(xs,0) 
end
```
And another

```haskell
fun rev xs = 
case xs of 
  [] => []
  | x::xs' => (rev xs') @ [x]
```

```haskell
fun rev fun aux(xs,acc) =
  let fun aux(xs,acc) =
    | [] => acc
    | x::xs' => aux(xs',x::acc)
  in aux(xs,[]) end
```

List append can be REALLY inefficient

```haskell
fun rev xs = 
case xs of 
  [] => []
  | x::xs' => (rev xs') @ [x]
```

- For fact and sum, tail-recursion takes less space but still linear time
- But, non-tail recursive rev is quadratic because each recursive call uses append @, which must traverse the first list
  - $1+2+...+(\text{length}-1)$ is almost length*length/2
- Moral: beware @, especially within recursion
- Cons is constant-time (and fast), so the accumulator version rocks

Is tail-recursive always more efficient?

- NO.
- Sometimes recursive functions can’t be evaluated in a constant amount of space
  - E.g., functions over trees
- In these cases, the simple recursive approach is best
  - (You could get one recursive call to be a tail call, but rarely worth the complication)

Precise definition of tail call

If the result of $f \ x$ is the "immediate result" for the enclosing function body, then $f \ x$ is a tail call

Can define this notion more precisely...
- A tail call is a function call in tail position
- If an expression is not in tail position, then no subexpressions are
  - In fun $f \ p = e$, the body $e$ is in tail position
  - If if $a1$ then $e2$ else $e3$ is in tail position, then $e2$ and $e3$ are in tail position (but $a1$ is not). (Similar for case expressions)
  - If let $b1 ... bn$ in $e$ end is in tail position, then $e$ is in tail position (but no binding expressions are)
  - Function-call arguments are not in tail position
  - ...

What the difference?

Length of a list:

```haskell
fun len {xs: int list} = 
case xs of 
  [] => 0
  | _::xs' => 1 + len xs'
```

```haskell
fun len {xs: string list} = 
case xs of 
  [] => 0
  | _::xs' => 1 + len xs'
```

No algorithmic difference! Would be silly to have to write function for every kind of list type...
Type variables to the rescue

Use type variable to stand in place of an arbitrary type:

\[
\text{fun len \{xs: 'a list\} = case xs of
\begin{cases}
[\] & \Rightarrow 0 \\
_::xs' & \Rightarrow 1 + len xs'
\end{cases}
\]

- Just like we use variables to stand in place of arbitrary values
- Creates a polymorphic function ("poly" = many, "morph" = form)
- Closely related to generics in Java
- Might look like, but is rather less related to, templates in C++

Another polymorphic function

- What is type of list append?

\[
\text{fun append \{xs,ys\} = case xs of
\begin{cases}
[\] & \Rightarrow ys \\
_::xs' & \Rightarrow x :: append(xs',ys)
\end{cases}
\]

- You might expect \text{string list * string list -> string list}
- But REPL reports \text{'a list * 'a list -> 'a list}
- This is okay (such as on your homework): why?

More general

The type \text{'a list * 'a list -> 'a list}

is more general than the type \text{string list * string list -> string list}

- Polymorphic type can be "used" as any less general type, such as
  \text{int list * int list -> int list}
- But it is not more general than the type
  \text{int list * string list -> int list}
  - Must replace type variables consistently

The generalization rule for types

Easy rule you (and the type-checker) can apply without thinking:

A type \(t_1\) is more general than the type \(t_2\) if you can take \(t_1\), replace its type variables consistently, and get \(t_2\)

- Example: Replace each \('a\) with \text{int * int}
- Example: Replace each \('a\) with \text{bool} and each \('b\) with \text{bool}
- Example: Replace each \('a\) with \text{bool} and each \('b\) with \text{int}

Equivalence rules for types

- Can combine the "more general" rule with rules for equivalence
  - Use of type synonyms does not matter
  - Order of field names does not matter

Example, given \text{type foo = int * int}

the type
\[
\{\text{quux : 'b, bar : int * 'a, baz : 'b}\}
\]

is more general than
\[
\{\text{quux : string, bar : foo, baz : string}\}
\]

which is equivalent to
\[
\{\text{bar : int*int, baz : string, quux : string}\}
\]

Equality types

- You might also see type variables with a second "quote"
  - Example: \text{''a list * ''a -> bool}

- These are "equality types" that arise from using the \(=\) operator
  - The \(=\) operator works on lots of types: \text{int, string, tuples}
  - containing all equality types, ...
  - But not all types: function types, \text{real}, ...

- The rules for more general are exactly the same except you have to replace an equality-type variable with a type that can be used with \(=\)
  - A "strange" feature of ML, because \(=\) is special
Example

```haskell
(* "a * "a -> string *)
fun same_thing(x, y) =
  if x=y then "yes" else "no"

(* int -> string *)
fun is_three x =
  if x=3 then "yes" else "no"
```

(You can ignore the warning about "calling polyEqual")

We stopped writing types...

- Started off semester writing:

  ```haskell
  fun pow (x: int, y: int) =
    if y=0
    then 1
    else x * pow(x,y-1)
  ```

- But recently we’ve been leaving off argument types:

  ```haskell
  fun pow (x, y) =
    if y=0
    then 1
    else x * pow(x,y-1)
  ```

- Either way, compiler has been inferring that return type is int

How does compiler infer?

- Here’s the idea:

  ```haskell
  fun pow (x, y) =
    if y=0
    then 1
    else x * pow(x,y-1)
  ```

  - `y` must be int, since compared with `= 0`
  - `pow` must return `int`, since then branch returns 1
  - `x` must be int, since multiplied with an `int`
  - (define more carefully in later lecture)

Selectors complicate type inference

- What is the type of argument to this function?

  ```haskell
  fun sum_triple t =
    case triple of
      (x,y,z) => x+y+z
  ```

- What about this function?

  ```haskell
  fun sum_triple t =
    #1 t + #2 t + #3 t
  ```

  - “Error: unresolved flex record (need to know the names of ALL the fields in this context)”
  - Which is why we had to write

  ```haskell
  fun sum_triple (t:int*int*int) =
    #1 t + #2 t + #3 t
  ```

Pattern matching simplifies type inference

- Pattern gives enough “hints” to compiler to infer type:

  ```haskell
  fun sum_triple t =
    case triple of
      (x,y,z) => x+y+z
  ```

- That’s why we don’t need to write down types anymore
- And also why we started off writing down types

- You can add type annotations anywhere you like
  - Instead of `e` write `e:t`
  - The `":t"` is an annotation or constraint used by type inference
  - And that’s all that the “argument types” we wrote down ever were
FIRST-CLASS FUNCTIONS

Can use them anywhere we use values
- Arguments, results, parts of tuples, bound to variables, carried by
data type constructors or exceptions, ...
- First-class citizens of language, afforded all the “rights” of any other values
- Functions can take functions as arguments
- Functions can return functions as results
... so functions are higher-order
- This is not a new language feature; just a consequence of choice we made
  long ago when we said “functions are values”

Example

Can re-use n_times rather than defining many similar functions
- Computes \( f(f(...f(x))) \) where number of calls is \( n \)

```ocaml
fun n_times (f, n, x) = 
  if n=0
     then x
  else f (n_times (f,n-1,x))

fun double x = x + x
fun increment x = x + 1
val x1 = n_times (double,4,7)
val x2 = n_times (increment,4,7)
val x3 = n_times (tl,2,[4,8,12,16,20])

fun double_n_times (n, x) = n_times (double,n,x)
fun nth_tail (n, x) = n_times (tl,n,x)
```