Functional Programming

Iteration (tail-recursive functions)

Michael R. Hansen

mrh@imm.dtu.dk

Informatics and Mathematical Modelling

Technical University of Denmark

Overview

Iterative (tail-recursive) functions is a simple technique to deal with efficiency in certain situations, e.g.

- to avoid evaluations with a huge amount of pending operations
- to avoid inadequate use of @ in recursive declarations.

Iterative functions correspond to while-loops

An example: Factorial function (I)

Consider the following declaration:

```
fun fact 0 = 1
  | fact n = n * fact(n-1);
val fact = fn: int -> int
```

- What resources are needed to compute `fact`(N)?

Considerations:

- Computation time: number of individual computation steps.
- Space: the maximal memory needed during the computation to represent expressions and bindings.

An example: Factorial function (II)

Evaluation:

$$
\begin{array}{ll}
& \texttt{fact}(N) \\
\rightsquigarrow & (\texttt{n * fact(n-1)}, [\texttt{n} \mapsto N]) \\
\rightsquigarrow & N * \texttt{fact}(N-1) \\
\rightsquigarrow & N * (\texttt{n * fact(n-1)}, [\texttt{n} \mapsto N-1]) \\
\rightsquigarrow & N * ((N-1) * \texttt{fact}(N-2)) \\
\vdots & \\
\rightsquigarrow & N * ((N-1) * ((N-2) * (\cdots (4 * (3 * (2 * 1))) \cdots))) \\
\rightsquigarrow & N * ((N-1) * ((N-2) * (\cdots (4 * (3 * 2)) \cdots))) \\
\vdots & \\
\rightsquigarrow & N!
\end{array}
$$

Time and space demands: proportional to N Is this satisfactory?

Another example: Naive reversal (I)

```
fun naive_rev [] = []
  | naive_rev (x::xs) = naive_rev xs @ [x];
val naive_rev = fn : 'a list -> 'a list
```

Evaluation of naive_rev $[x_1, x_2, \ldots, x_n]$:

$$
\begin{array}{ll}
 & \texttt{naive\_rev}\,[x_1, x_2, \ldots, x_n] \\
\rightsquigarrow & \texttt{naive\_rev}\,[x_2, \ldots, x_n]@[x_1] \\
\rightsquigarrow & (\texttt{naive\_rev}\,[x_3, \ldots, x_n]@[x_2])@[x_1] \\
\vdots & \\
\rightsquigarrow & ((\cdots(([\,]@[x_n])@[x_{n-1}])@\cdots@[x_2])@[x_1])
\end{array}
$$

Space demands: proportional to n — satisfactory

Time demands: proportional to n^2 — not satisfactory

Examples: Accumulating parameters

Solution obtained by the introduction of more general functions:

$$\begin{aligned}
\texttt{itfact}(n, m) &= n! \cdot m, \text{ for } n \geq 0 \\
\texttt{itrev}([x_1, \ldots, x_n], ys) &= [x_n, \ldots, x_1] \, @ys
\end{aligned}$$

We have:

$$\begin{aligned}
n! &= \texttt{itfact}(n, 1) \\
\texttt{rev}\,[x_1, \ldots, x_n] &= \texttt{itrev}([x_1, \ldots, x_n], [\,])
\end{aligned}$$

m and ys are called *accumulating parameters*. They are used to hold the temporary result during the evaluation.

Declaration of `itfact`

```
fun itfact(0,m) = m
  | itfact(n,m) = itfact(n-1,n*m)
```

An evaluation:

$$
\begin{aligned}
& \texttt{itfact(5,1)} \\
\rightsquigarrow\ & (\texttt{itfact(n,m)}, [\texttt{n} \mapsto 5, \texttt{m} \mapsto 1]) \\
\rightsquigarrow\ & (\texttt{itfact(n-1,n*m)}, [\texttt{n} \mapsto 5, \texttt{m} \mapsto 1]) \\
\rightsquigarrow\ & \texttt{itfact(4,5)} \\
\rightsquigarrow\ & (\texttt{itfact(n,m)}, [\texttt{n} \mapsto 4, \texttt{m} \mapsto 5]) \\
\rightsquigarrow\ & (\texttt{itfact(n-1,n*m)}, [\texttt{n} \mapsto 4, \texttt{m} \mapsto 5]) \\
\rightsquigarrow\ & \texttt{itfact(3,20)} \\
\rightsquigarrow\ & \ldots \\
\rightsquigarrow\ & \texttt{itfact(0,120)} \rightsquigarrow (\texttt{m}, [\texttt{m} \mapsto 120]) \rightsquigarrow 120
\end{aligned}
$$

Space demand: constant. Time demands: proportional to n

Declaration of `itrev`

```
fun itrev([], ys)     = ys
  | itrev(x::xs, ys) = itrev(xs, x::ys)
```

An evaluation:

```
    itrev([1,2,3],[])
⇝  itrev([2,3],1::[])
⇝  itrev([2,3],[1])
⇝  itrev([3],2::[1])
⇝  itrev([3],[2,1])
⇝  itrev([],3::[2,1])
⇝  itrev([],[3,2,1])
⇝  [3,2,1]
```

Space and time demands:

proportional to n (the length of the first list)

Iterative (tail-recursive) functions (I)

The declarations of `itfact` and `itrev` are *tail-recursive functions*, where the recursive call is the *last function application* to be evaluated in the body of the declaration.

- only *one set* of bindings for argument identifiers is needed during the evaluation

Example

```
fun itfact(0,m) = m
  | itfact(n,m) =  itfact(n-1,n*m)
                   (* recursive "tail-call" *)
```

- only one set of bindings for argument identifiers is needed during the evaluation

$$
\begin{aligned}
& \texttt{itfact(5,1)} \\
\rightsquigarrow\ & (\texttt{itfact(n,m)}, [n \mapsto 5, m \mapsto 1]) \\
\rightsquigarrow\ & (\texttt{itfact(n-1,n*m)}, [n \mapsto 5, m \mapsto 1]) \\
\rightsquigarrow\ & \texttt{itfact(4,5)} \\
\rightsquigarrow\ & (\texttt{itfact(n,m)}, [n \mapsto 4, m \mapsto 5]) \\
\rightsquigarrow\ & (\texttt{itfact(n-1,n*m)}, [n \mapsto 4, m \mapsto 5]) \\
\rightsquigarrow\ & \ldots \\
\rightsquigarrow\ & \texttt{itfact(0,120)} \rightsquigarrow (\texttt{m}, [m \mapsto 120]) \rightsquigarrow 120
\end{aligned}
$$

Iterative (tail-recursive) functions (II)

Tail-recursive functions are also called *iterative functions*.

- The function $f(n, m) = (n - 1, n * m)$ is iterated during evaluations for `itfact`.
- The function $g(x :: xs, ys) = (xs, x :: ys)$ is iterated during evaluations for `itrev`.

The correspondence between tail-recursive functions and while loops are established in Chapter 18.

An example:

```
fun fact(k) = let val n = ref k
                  val m = ref 1
              in while !n <> 0
                 do  (m := !n * !m; n := !n - 1)
                 ; !m
              end;
```

Iterative functions (III)

A function $g : \tau$ `->` τ' is an *iteration of* $f : \tau$ `->` τ if it is an instance of:

```
fun g z = if p z then g(f z) else h z
```

for suitable predicate $p : \tau$ `-> bool` and function $h : \tau$ `->` τ'.

The function g is called an *iterative (or tail-recursive) function*.

Examples: `itfact` and `itrev` are easily declared in the above form

```
fun itfact(n,m) =
   if n<>0 then itfact(n-1,n*m) else m;

fun itrev(xs,ys) =
     if not (null xs)
     then itrev(tl xs, (hd xs)::ys)
     else ys;
```

Iterative functions: evaluations (I)

Consider: `fun g z = if p z then g(f z) else h z`

Evaluation of the $g\ v$:

$$
\begin{array}{ll}
 & g\ v \\
\rightsquigarrow & (\texttt{if}\ p\ z\ \texttt{then}\ g(f\ z)\ \texttt{else}\ h\ z\ ,\ [z \mapsto v]) \\
\rightsquigarrow & (g(f\ z), [z \mapsto v]) \\
\rightsquigarrow & g(f^1 v) \\
\rightsquigarrow & (\texttt{if}\ p\ z\ \texttt{then}\ g(f\ z)\ \texttt{else}\ h\ z\ ,\ [z \mapsto f^1 v]) \\
\rightsquigarrow & (g(f\ z), [z \mapsto f^1 v]) \\
\rightsquigarrow & g(f^2 v) \\
\rightsquigarrow & \ldots \\
\rightsquigarrow & (\texttt{if}\ p\ z\ \texttt{then}\ g(f\ z)\ \texttt{else}\ h\ z\ ,\ [z \mapsto f^n v]) \\
\rightsquigarrow & (h\ z, [z \mapsto f^n v]) \qquad \text{suppose } p(f^n v) \rightsquigarrow \text{false} \\
\rightsquigarrow & h(f^n v)
\end{array}
$$

Iterative functions: evaluations (II)

Observe two desirable properties:

- there are n recursive calls of g, and
- at most *one binding* for the argument pattern z is 'active' at any stage in the evaluation.

Example: Fibonacci numbers (I)

A declaration based directly on the mathematical definition:

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib(n-1) + fib(n-2);
val fib = fn : int -> int
```

is highly inefficient. For example:

```
    fib 4
⇝  fib 3 + fib 2
⇝  (fib 2 + fib 1) + fib 2
⇝  ((fib 1 + fib 0) + fib 1) + fib 2
⇝  ··· ⇝ 2 + (fib 1 + fib 0)
⇝  ···
```

Ex: `fib 44` requires around 10^9 evaluations of base cases.

Example: Fibonacci numbers (II)

An iterative solution gives high efficiency:

```
fun itfib(n,a,b) = if n <> 0 then itfib(n-1,a+b,a)
                                  else a;
```

The expression $\texttt{itfib}(n, 0, 1)$ evaluates to F_n, for any $n \geq 0$:

- Case $n = 0$: $\texttt{itfib}(0, 0, 1) \rightsquigarrow 0 \; (= F_0)$
- Case $n > 0$:

$$
\begin{array}{ll}
 & \texttt{itfib}(n, 0, 1) \\
\rightsquigarrow & \texttt{itfib}(n-1,\ 1,\ 0) = \texttt{itfib}(n-1,\ F_1,\ F_0) \\
\rightsquigarrow & \texttt{itfib}(n-2,\ F_1 + F_0,\ F_1) \\
\rightsquigarrow & \texttt{itfib}(n-2,\ F_2,\ F_1) \\
\vdots & \\
\rightsquigarrow & \texttt{itfib}(0, F_n, F_{n-1}) \\
\rightsquigarrow & F_n
\end{array}
$$

Recommendations

Have iterative functions in mind when dealing with efficiency, e.g.

- to avoid evaluations with a huge amount of pending operations
- to avoid inadequate use of `@` in recursive declarations.