S4.2: Expression Evaluation: Improving Performance

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2018

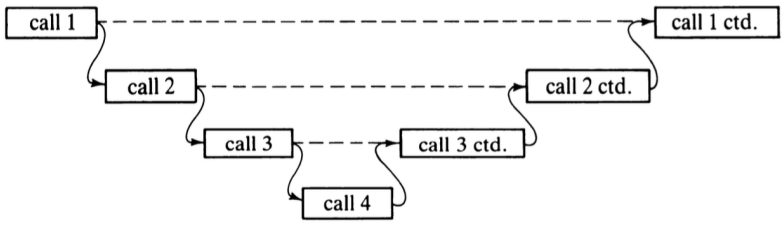
Improving Performance of Expression Evaluation

- We'll look at a few examples of functions whose performance can be improved.
- These, along with notes about improving their performance, are to be found in the files `tail.ml`, `continuation.ml`, and `tail_recursive_tree_functions.ml`.
  These are in the `SamplePrograms/Sec01` and `SamplePrograms/Sec10` directories in the public class repository.
- Some of this material comes from section 9.4 of Chris Reade's book "Elements of Functional Programming".
  This is in the `Resources` directory of the public class repository.

Tail recursion

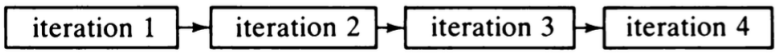
- The functions `rev_a` and `sum_a` are both using "tail recursion."
- We call them "tail recursive."
- All recursive calls are in "tail position."
  This position is outermost in the function body or outermost in conditionals if the conditional is outermost.
- See Figure 9.1 in Reade's book, and in the following slides.
- Compilers can convert tail recursive functions into loops, thus avoiding
  - the space overhead of storing local values for each function instantiation,
  - the time overhead of saving this context and restoring it on returns from called functions.

Stacking up recursion

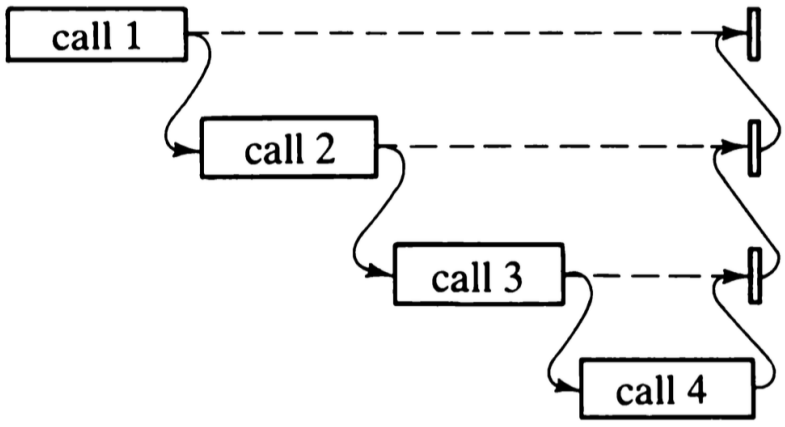


See examples at the beginning of `tail.ml`

iteration



tail recursion



Tail recursion can be compiled into iteration.

Recursion versus iteration

- Iteration, that is looping, is linear in nature
- The expressive power of recursion is that it does not need to be linear.
  Recursion over trees, for example. It is not always clear what a linear traversal over that tree would be.
- Our aim here, is to see how to linearize certain computations.

Accumulating parameters

- One technique we can use is to rewrite functions to use an "accumulating parameter".
- See discussion of `sum` in `tail.ml`.

Folds

Recall:

```
let rec fold_left (f: 'b -> 'a -> 'b) (acc: 'b)
                  (lst: 'a list) : 'b =
  match lst with
  | [] -> acc
  | x::xs -> fold_left f (f acc x) xs

let rec fold_right (f: 'a -> 'b -> 'b) (lst: 'a list)
                   (base: 'b) : 'b =
  match lst with
  | [] -> base
  | x::xs -> f x (fold_right f xs base)
```

Which one is tail recursive?

fold left

- Note how fold-left already captures this notion.
- It is tail recursive and does this work for us.
- fold-left: tail recursive and fast
- fold-right: a more general pattern, replace constructors (cons `::` and nil `[ ]`) with functions or values.

"tree" shaped computations

- The notion of continuations can also lead to tail-recursive functions, especially for computations that naively have a tree-shaped structure.
- See the examples in `continuations.ml`:
  - flattening trees
  - factorial
  - map
  - fold-right
  - ...
- Some additional examples in `tail_recursive_tree_functions.ml`.