Tail Recursion,
do I have it? how to get it?

Paul Bone

What to expect

What to expect from this presentation.

- Some definitions.
- Do I have tail recursion?
- How do I get it?

We'll mostly be discussing strict languages even though some of the syntax is Haskell-ish, the semantics are usually not.

We'll also assume that compilers targeting C, Java, etc, cannot rely on the underlying language to provide LCO/TCO. Although we assume that compilers targeting ILs such as LLVM, JVM, BEAM etc and also assembly, can provide LCO.

Self recursion

```
map f [] = []
map f x:xs = (f x):(map f xs)
```

`map` is recursive, because it calls it**self**.

Mutual recursion

```
odd 0 = False
odd n = even (n - 1)

even 0 = True
even n = odd (n - 1)
```

`odd` can call `even` and `even` can call `odd`, the recursion is **mutual**.

Each group of such functions is called a *strongly connected component (SCC)*. Technically even a self-recursive function like `map`, or a non-recursive function is a SCC, the SCC just contains a single item.

Tail position

Tail position is the last thing a function does before it exits. In this syntax it is the **outermost** expression on the RHS of `=`.

```
f1 = tail_position 1 2

f2 = 1 + (not_tail_position 2)
```

In `f2` it's the addition that's in **tail position**.

Tail recursion

Tail recursion is when a recursive call (self or mutual) is in tail position.

In other words, the last thing that the function does is to call itself. `foldl` is **tail-recursive**.

```
foldl _ [] acc = acc
foldl f x:xs acc = foldl f xs (f acc x)
```

This call is tail recursive. Or sometimes more specifically: self tail recursive or mutually tail recursive.

Tail call optimisation (TCO)

Tail call optimisation (TCO) (also called last call optimisation) ensures that functions use a fixed amount of stack space regardless of how much they need to recurse (how many items there are in `foldl`'s list).

Each call can re-use the stack frame of the current call.

Different languages/compilers can (and sometimes promise) to optimize:

- Only self-recursive tail calls (eg: with a loop).
- Sibling calls.
- Mutually recursive tail calls (eg: with a trampoline, inlining etc).
- Any call in tail position (eg: with a jump).

Sibling call

Like a mutual call, except with some extra constraints. Different C compilers require different constraints, depending on what they're willing to optimise. These are usually:

- Caller and callee calling conventions match.
- Return types match.
- Parameter lists either
  - match completely or
  - the callee's list matches the initial part of the caller's.

And of course, the call must be in tail position for the optimisation to work.

Do I have it?

Audience participation!

Quiz question 1

Is this tail recursive?

```
foldl _ [] acc = acc
foldl f x:xs acc = foldl f xs (f acc x)
```

Audience participation! Call out **Yes!** now. Of course, this was the practice question, we saw it in the above examples.

Quiz question 1a

Is this tail recursive?

```
foldl _ [] acc = acc
foldl f x:xs acc = foldl f xs (f acc x)
```

Sometimes the answer is **no**. Depending on the language/compiler, debugging and profiling builds can interfere with TCO.

Okay, that was an unfair trick, but I want to introduce it now so I can refer to it later.

Quiz question 2

How about this?

```
map f [] = []
map f x:xs = (f x):(map f xs)
```

Quiz question 2

How about this?

```
map f [] = []
map f x:xs = (f x):(map f xs)
```

No, this is not tail recursive. `map` is **not in tail position**, `:` is in **in tail position**.

Quiz question 3

But what if the language is lazy?

```
map f [] = []
map f x:xs = (f x):(map f xs)
```

Quiz question 3

But what if the language is lazy?

```
map f [] = []
map f x:xs = (f x):(map f xs)
```

Yes, When execution enters either branch, it was because the *thunk* was forced to *WHNF*: The **cons cell** will be constructed containing two thunks, one for the head, and one for the tail. The tail, containing the recursive call, will not be evaluated at this time and `map` will return the cons cell containing the thunk which contains the **unevaluated recursive call**.

However, if `+` was the symbol in tail position, rather than `:` then this **would not** be tail recursive.

Laziness creates the somewhat analogous problem of space leaks.

Quiz question 4

Okay, so it's not a functional language, but how about the equivalent Prolog code?

```
map(_, [], []).
map(P, [X | Xs], [Y | Ys]) :-
    P(X, Y),
    map(P, Xs, Ys).
```

Prolog also makes heavy use of recursion, so tail recursion is important.

Quiz question 4

```
map(_, [], []).
map(P, [X | Xs], [Y | Ys]) :-
    P(X, Y),
    map(P, Xs, Ys).
```

Yes.

- The arguments are unified with their parameters. In particular the "output" argument is unified with the cons cell, `[Y | Ys]`, whose head and tail are *free*.
- `P(X, Y)` is called, giving a value to `Y` which implicitly fills in the aliased head of the cons cell.
- `map(P, Xs, Ys)` is **tail-called** since it is the last *conjunct* in this *clause*. It fills in the value for `Ys` which implicitly fills in the tail of the cons cell.

Quiz question 5

Mercury

```
:- pred map(pred(A, B), list(A), list(B)).
:- mode map(pred(in, out) is det, in, out) is det.

map(_, [], []).
map(P, [X | Xs], [Y | Ys]) :-
    P(X, Y),
    map(P, Xs, Ys).
```

Maybe this is getting unfair..

Quiz question 5

Mercury

```
:- pred map(pred(A, B), list(A), list(B)).
:- mode map(pred(in, out) is det, in, out) is det.

map(_, [], []).
map(P, [X | Xs], [Y | Ys]) :-
    P(X, Y),
    map(P, Xs, Ys).
```

No. Unlike Prolog, Mercury does not support logic variables (aliasing). The construction of the **cons cell** must occur after the **recursive call**, and therefore that call cannot be a tail call.

Quiz question 6

Last one

```
:- pred foo(..., ab,  ab).
:- mode foo(..., out, out) is det.

foo(..., a, b).
foo(..., A, B) :-
    ...,
    foo(..., B, A).
```

There's no construction this time, maybe this is tail recursive?

Quiz question 6

Last question, this time there are two outputs, something you can't do in most functional languages.

```
:- pred foo(..., ab,  ab).
:- mode foo(..., out, out) is det.

foo(..., a, b).
foo(..., A, B) :-
    ...,
    foo(..., B, A).
```

No sorry, Mercury must swap the output parameters after the recursive call, so it is not in tail position.

However in Prolog this is **Yes**. The variables are passed into the recursive call by reference, they're swapped before the call.

How do I get it?

Now is a good itme for pizza!

Maybe you don't need it?

Have you considered?

- Just add more memory! (allocate more stack memory)
- Or use a segmented stack

This may be much easier than modifying your code! But the code will still be inefficient in both time and space.

It also simply shifts the bound of the amount of data you can handle, you may also crash **harder** when the system runs out of RAM and begins to *thrash*.

Accumulator introduction

```
map f []     = []
map f x:xs = (f x) : (map f xs)
```

becomes

```
map f xs = map' f xs []
```

```
map' _ []     acc = reverse acc
map' f x:xs acc = map' xs (f x):acc
```

The **accumulator** now stores the information that was previously on the stack, this trades stack memory for heap memory.

Either the developer or the compiler can perform this transformation.

Reduce the required stack depth

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc0 xs =
    let r = foldl' 500 f acc0 xs in
        case r of
            Complete acc        -> acc
            Incomplete list acc -> foldl f acc list

data Result input acc = Complete acc
                      | Incomplete [input] acc

foldl' :: Int -> (b -> a -> b) -> b -> [a] -> Result a b
foldl' 0 f acc  xs         = Incomplete xs acc
foldl' n _ acc  []         = Complete acc
foldl' n f acc0 (x0:xs0) = let acc = (f acc0 x0) in
                           foldl' (n-1) f acc xs0
```

Reduce the required stack depth

This technique is useful when your functional language **doesn't** support tail recursion, eg: debugging and profiling builds. It works for code that would be tail recursive if the language supported it.

It uses two loops, when the inner loop has exceeded its depth limit it returns, freeing the stack it consumed.

The limit is usually tuned manually and therefore the whole transformation is usually done manually.

Last call modulo constroctor (LCMC)

```
map _ [] = []
map f x:xs = {
      let cons = (f x):_
      map' f xs address_of(cons, field 1)
      return cons
   }
```

```
map' f [] result_ptr = {
      *result_ptr := []
   }
map' f x:xs result_ptr = {
      let cons = (f x):_
      *result_ptr := cons
      map' f xs address_of(cons, field 1)
   }
```

Last call modulo constroctor (LCMC)

Based on the intuition from the lazy functional and Prolog examples, we can optimize tail-calls in strict functional languages by moving the construction before the recursive call.

This optimisation can usually only be done by the compiler.

Loops and state machines

Often it's very easy to transform a recursive call into a while loop.

```
foldl(f, list, acc) {
  while (true) {
    switch (list) {
      case []:
        return acc;
      case x:xs:
        acc1 = f(y, acc);
        // Replace the input variables and loop.
        acc = acc1;
        list = xs;
    }
  }
}
```

Loops and state machines

A state machine can be used for mutually-recursive loops.

```
foo(foo_arg1, foo_arg2) {
  foo:
    if (...) {
      return w;
    } else {
      bar arg = x;
      goto bar;
    }
  bar:
    ... code for bar ...;
    foo_arg1 = y;
    foo_arg2 = z;
    goto foo;
  }
}
```

Trampoline

This works even when the compiler cannot see the whole call graph, for example there are module boundaries or higher order calls.

```
typedef void* Func (args);

void driver (Func* entry)
{
  struct arg_struct args;

  Func* fp = entry;
  while (fp != NULL) {
    fp = (Func*) (*fp)(&args);
  }
}
```

This has a lot in common with continuation passing style.

Thank you

Mercury

http://mercurylang.org

Plasma

http://plasmalang.org

Paul Bone

http://paul.bone.id.au

These slides are typeset with Prince

http://princexml.com