Tail Modulo Async/Await

Vivien Gachet Gabriel Radanne Ludovic Henrio

The Functional Programmer's Best Friend

```
1 let rec fact n =
2   if n = 0 then
3     1
4   else
5     n * fact (n-1)
```

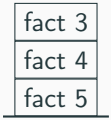
fact 5

The Functional Programmer's Best Friend

```
1 let rec fact n =
2   if n = 0 then
3     1
4   else
5     n * fact (n-1)
```

The Functional Programmer's Best Friend

```
1 let rec fact n =
2   if n = 0 then
3     1
4   else
5     n * fact (n-1)
```

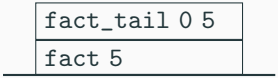


The Functional Programmer's Best Friend

```
1 let rec fact n =
2   if n = 0 then
3     1
4   else
5     n * fact (n-1)
```

Tail calls – An alternative calling convention

```
1 let rec fact_tail acc n =
2   if n = 0 then
3     acc
4   else
5     fact (n*acc) (n-1)
6
7 let fact = fact_tail 1
```



$\Rightarrow$ For calls in tail position, we can use *a different calling convention*

Tail recursion – A compiler transformation

```
1  let fact n0 =
2    let n = ref n0 in
3    let acc = ref 1 in
4    while true do
5      if n = 0 then
6        break
7      else
8        acc := !n*!acc
9    done;
10   !acc
11
```

fact 5

⇒ If all recursive calls are tail, we can de-recursify !

Asynchronous code

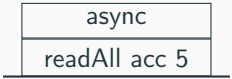
```
1 val readAll : int -> string Future.t
2 let async readAll n =
3   if n = 0 then
4     ""
5   else
6     let s = await(readFile "foo%i.txt" n) in
7     s ^ await(readAll (n-1))
```

Asynchronous code – Tail recursive???

```
1 let async readAll' acc n =
2   if n = 0 then
3     acc
4   else
5     let s = await(readFile "foo%i.txt" n) in
6     await(readAll' (s^acc) (n-1))
7
8 let readAll n = readAll' "" n
```

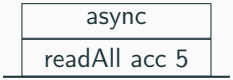
Asynchronous code – Tail recursive???

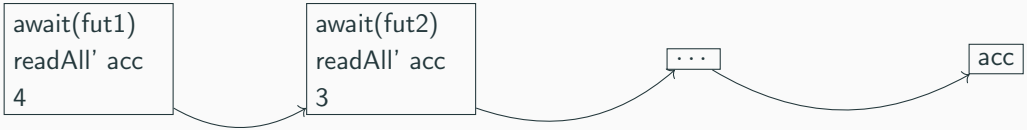
```
1 let async readAll' acc n =
2   if n = 0 then
3     acc
4   else
5     let s = await(readFile "foo%i.txt" n) in
6     await(readAll' (s^acc) (n-1))
7
8 let readAll n = readAll' "" n
```



Asynchronous code – Tail recursive???

```
1 let async readAll' acc n =
2   if n = 0 then
3     acc
4   else
5     let s = await(readFile "foo%i.txt" n) in
6     await(readAll' (s^acc) (n-1))
7
8 let readAll n = readAll' "" n
```





Contribution: Tail modulo Async/Await

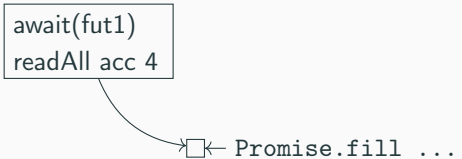
Our contribution:

- The notion of "tail asynchronous calls"
- A code transformation to "destiny passing style"
- An extension to Tail modulo Cons
- Full semantics
- WIP proof
- Partial implementation

Tail modulo async/await

First Key Idea: Use Promises!

Promise = Future + Write-once pointer



Destiny Passing style

```
1 let async readAll' acc n =
2   if n = 0 then
3     acc
4   else
5     let s = await(readFile "foo%i.txt" n) in
6     await(readAll' (s^acc) (n-1))
7
8 let readAll n = readAll' "" n
```

Calls in tail position can be rewritten to use the *destiny*, i.e. the eventual position of the computation[1]

[1] see "Forward to a Promising Future" by K Fernandez-Reyes, et al

Destiny Passing style

```
1  let readAll' d acc n =
2    if n = 0 then
3      ... acc ...
4    else
5      let s = await(readFile "foo%i.txt" n) in
6      readAll' d (s^acc) (n-1)
7
8  let readAll n =
9    let d = ... in
10   readAll' d "" n
```

Calls in tail position can be rewritten to use the *destiny*, i.e. the eventual position of the computation[1]

[1] see "Forward to a Promising Future" by K Fernandez-Reyes, et al

Destiny Passing style

```
1  let readAll' d acc n : unit =
2    if n = 0 then
3      Promise.fill d acc
4    else
5      let s = await(readFile "foo%i.txt" n) in
6      readAll' d (s^acc) (n-1)
7
8  let readAll n =
9    let d = ... in
10   readAll' d "" n
```

Calls in tail position can be rewritten to use the *destiny*, i.e. the eventual position of the computation[1]

[1] see "Forward to a Promising Future" by K Fernandez-Reyes, et al

Destiny Passing style

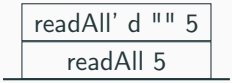
```
1  let readAll' d acc n : unit =
2    if n = 0 then
3      Promise.fill d acc
4    else
5      let s = await(readFile "foo%i.txt" n) in
6      readAll' d (s^acc) (n-1)
7
8  let readAll n =
9    let fut, d = Promise.create () in
10   readAll' d "" n;
11   fut
```

Calls in tail position can be rewritten to use the *destiny*, i.e. the eventual position of the computation[1]

[1] see "Forward to a Promising Future" by K Fernandez-Reyes, et al

Destiny Passing style

```
1  let readAll' d acc n : unit =
2    if n = 0 then
3      Promise.fill d acc
4    else
5      let s = await(readFile "foo%i.txt" n) in
6      readAll' d (s^acc) (n-1)
7
8  let readAll n =
9    let fut, d = Promise.create () in
10   readAll' d "" n;
11   fut
```

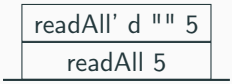


Calls in tail position can be rewritten to use the *destiny*, i.e. the eventual position of the computation[1]

[1] see "Forward to a Promising Future" by K Fernandez-Reyes, et al

Destiny Passing style

```
1  let readAll' d acc n : unit =
2    if n = 0 then
3      Promise.fill d acc
4    else
5      let s = await(readFile "foo%i.txt" n) in
6      readAll' d (s^acc) (n-1)
7
8  let readAll n =
9    let fut, d = Promise.create () in
10   readAll' d "" n;
11   fut
```



A look at Constructors

Tail recursion modulo cons

We can also get tail recursion modulo Cons (Frédéric Bour, et al)!

```
1 let rec map f = function
2   | [] -> []
3   | x :: xs ->
4     let y = f x in
5     y :: map f xs
```

Tail recursion modulo cons

We can also get tail recursion modulo Cons (Frédéric Bour, et al)!

```
1 let rec map f = function
2   | [] -> []
3   | x :: xs ->
4     let y = f x in
5     y :: map f xs
```

```
1 let rec map_dps dst i f = function
2   | [] ->
3     dst.i <- []
4   | x::xs ->
5     let y = f x in
6     let dst' = y :: Hole in
7     dst.i <- dst';
8     map_dps dst' 1 f xs
```

Key idea: Use *Destination* Passing style

What about functions on trees ?

```
1 type tree =
2   | Leaf
3   | Node of int * tree * tree
4
5 let rec map f = function
6   | Leaf -> Leaf
7   | Node(v, tl, tr) ->
8     let tl' = map f tl in
9     Node(f v, tl', map f tr)
```

What about functions on trees ?

```
1 type tree =
2   | Leaf
3   | Node of int * tree * tree
4
5 let rec map f = function
6   | Leaf -> Leaf
7   | Node(v, tl, tr) ->
8     let tl' = map f tl in
9     Node(f v, tl', map f tr)
```

Only one of the call is “tail modulo cons”

An Async trick

```
let rec async map f = function
  | Leaf -> Leaf
  | Node(v, tl, tr) ->
    Node(f v,
         await (map f tl),
         await (map f tr))
```

What if we:

- Allocate the cons first (In the style of Tail mod Cons)
- Launch all recursive calls concurrently

Values with holes

Key idea: Destiny = Promise ∩ Destination

- Destiny are holes in the to-be-computed value
- We can keep multiple write-once pointers to the holes
- Values is ready when there are no more holes

Tail modulo Cons+Async/Await

```
let rec async map f =
    function
  | Leaf -> Leaf
  | Node(v, tl, tr) ->
    Node(f v,
         await (map f tl),
         await (map f tr))
```

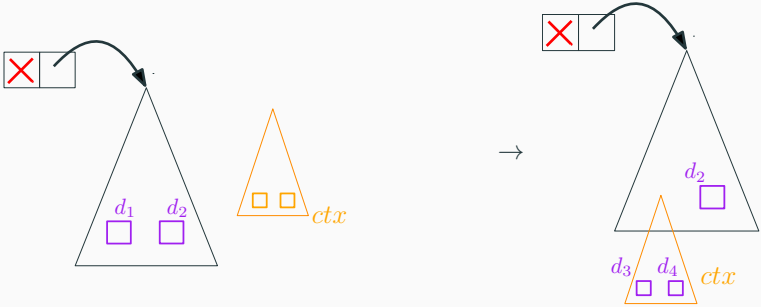
```
let rec map_dps d f = function
  | Leaf -> Destiny.refine d Leaf
  | Node(v, tl, tr) ->
    let ctx = Node(f v, Hole, Hole) in
    let dl, dr = Destiny.refine d ctx in
    fork_call map_dps dl f tl;
    fork_call map_dps dr f tr;
    ()

let map f l =
  let d, fut = Destiny.fresh () in
  map_dps d f l;
  fut
```

Destiny = Promise ∩ Destination

- `type 'a Destiny.t`
  A destiny whose hole will be filled by a value of type `'a`.
  Must be used linearly.
- `Destiny.fresh : unit -> 'a Destiny.t * 'a Future.t`
  Fresh destiny, and its future
- `Destiny.refine : 'a Destiny.t -> 'a -> 'a1 Destiny.t * ...`
  Fill a hole with a value (which might have several holes).

Destiny.refine



Refine an existing hole, and return all destinies

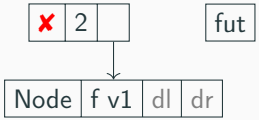
Destiny fulfillment

Detailled look at the code

```
1  let rec map_dps d f = function
2    | Leaf -> Destiny.refine d Leaf
3    | Node(v, tl, tr) ->
4      let ctx = Node(f v, Hole, Hole) in
5      let dl, dr = Destiny.refine d ctx in
6      fork_call map_dps dl f tl;
7      fork_call map_dps dr f tr;
8      ()
9
10 let map f l =
11   let d, fut = Destiny.fresh () in
12   map_dps d f l;
13   fut
```

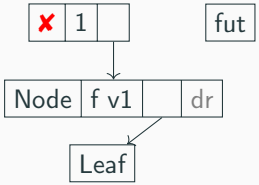
Detailled look at the code

```
1  let rec map_dps d f = function
2    | Leaf -> Destiny.refine d Leaf
3    | Node(v, tl, tr) ->
4      let ctx = Node(f v, Hole, Hole) in
5      let dl, dr = Destiny.refine d ctx in
6      fork_call map_dps dl f tl;
7      fork_call map_dps dr f tr;
8      ()
9
10 let map f l =
11   let d, fut = Destiny.fresh () in
12   map_dps d f l;
13   fut
```



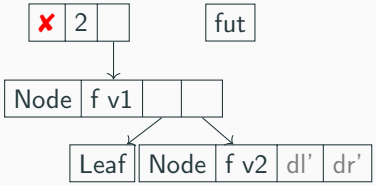
Detailled look at the code

```
1  let rec map_dps d f = function
2    | Leaf -> Destiny.refine d Leaf
3    | Node(v, tl, tr) ->
4      let ctx = Node(f v, Hole, Hole) in
5      let dl, dr = Destiny.refine d ctx in
6      fork_call map_dps dl f tl;
7      fork_call map_dps dr f tr;
8      ()
9
10 let map f l =
11   let d, fut = Destiny.fresh () in
12   map_dps d f l;
13   fut
```



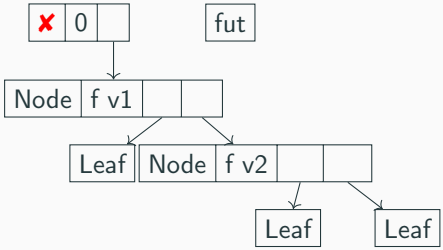
Detailled look at the code

```
1  let rec map_dps d f = function
2    | Leaf -> Destiny.refine d Leaf
3    | Node(v, tl, tr) ->
4      let ctx = Node(f v, Hole, Hole) in
5      let dl, dr = Destiny.refine d ctx in
6      fork_call map_dps dl f tl;
7      fork_call map_dps dr f tr;
8      ()
9
10 let map f l =
11   let d, fut = Destiny.fresh () in
12   map_dps d f l;
13   fut
```



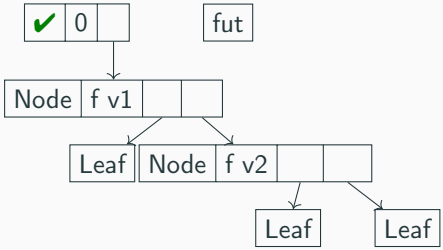
Detailled look at the code

```
1  let rec map_dps d f = function
2    | Leaf -> Destiny.refine d Leaf
3    | Node(v, tl, tr) ->
4      let ctx = Node(f v, Hole, Hole) in
5      let dl, dr = Destiny.refine d ctx in
6      fork_call map_dps dl f tl;
7      fork_call map_dps dr f tr;
8      ()
9
10 let map f l =
11   let d, fut = Destiny.fresh () in
12   map_dps d f l;
13   fut
```



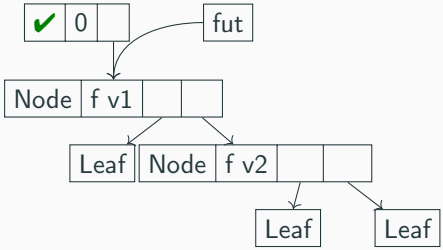
Detailled look at the code

```
1  let rec map_dps d f = function
2    | Leaf -> Destiny.refine d Leaf
3    | Node(v, tl, tr) ->
4      let ctx = Node(f v, Hole, Hole) in
5      let dl, dr = Destiny.refine d ctx in
6      fork_call map_dps dl f tl;
7      fork_call map_dps dr f tr;
8      ()
9
10 let map f l =
11   let d, fut = Destiny.fresh () in
12   map_dps d f l;
13   fut
```



Detailled look at the code

```
1  let rec map_dps d f = function
2    | Leaf -> Destiny.refine d Leaf
3    | Node(v, tl, tr) ->
4      let ctx = Node(f v, Hole, Hole) in
5      let dl, dr = Destiny.refine d ctx in
6      fork_call map_dps dl f tl;
7      fork_call map_dps dr f tr;
8      ()
9
10 let map f l =
11   let d, fut = Destiny.fresh () in
12   map_dps d f l;
13   fut
```



Current Status

- ✔ Define a semantics with destinies and forking calls
- ∼ WIP simulation proof of the translation
- ✔ Proof-of-concept implementation of destinies in (unsafe) OCaml
- ✘ Forking-call are not available in source OCaml
- ✘ Async is not available in OCaml IR

⇒ How do we actually put that into OCaml ?