# Tail Modulo Async/Await - Extended Abstract

Vivien Gachet, Ludovic Henrio, Gabriel Radanne

# Tail Modulo Async/Await - Extended Abstract

Vivien Gachet
EnsL, Inria, UCBL, CNRS, LIP
Lyon, France

Ludovic Henrio
CNRS, EnsL, Inria, UCBL, LIP
Lyon, France
ludovic.henrio@ens-lyon.fr

Gabriel Radanne
Inria, EnsL, UCBL, CNRS, LIP
Lyon, France
gabriel.radanne@inria.fr

## 1 Introduction

Tail-calls [9] are an essential feature of functional programming languages: they allow to write iterations in a declarative fashion, using simple recursive functions, unbothered by implementation details such as stack space. They work by giving special treatment to function calls in "terminal position", i .e. the last thing a function would do. Indeed, such calls never need to be returned from, meaning that adding an entry in the stack is not necessary. Tail-recursion exploits this characteristic further by transforming appropriate recursive functions into for-loops, ensuring they only use $O(1)$ stack space. Tail-call and tail-recursion, since their introduction in the 70s, have been mainstay in many programming languages and are available in virtually all modern production compilers (C, Rust, Scheme, Haskell, OCaml, ...).

Tail-modulo-cons is a classic extension, originally described in Prolog, which aims to allow function calls "under a constructor". For instance, let us consider the code in Figure 1. The last call on Line 9 is not normally in "tail-position", as a *Node* still have to be allocated. Tail-modulo-cons is a code transformation that first allocates the *Node*, with an appropriately placed hole, then makes a tail-call to the function. Tail-modulo-cons was recently added to OCaml [1] and expanded to richer contexts [4].

Unfortunately, such a transformation doesn't handle multiple recursive calls, such as the ones present in the map function on trees, as shown in Figure 1. Indeed, what would even be the semantics in a sequential context? Which calls to *map* should run first?

This problem takes on a different meaning in a concurrent context! Consider a concurrent map on binary trees implemented in Figure 2 using the async/await paradigm. The whole function is marked as `async`, meaning that it returns a promise [5] representing the computation in progress. `await` waits for a promise to be completed and returns its value. Each recursive call, on Line 5 and 6, thus run concurrently. Squinting a little bit, we can observe that both recursive calls are, in spirit, tail-calls (modulo cons): the only thing that remains to do after them is to allocate the `Node`. Unfortunately, we have *two* recursive calls, both hidden under an `await`, which is out of scope of existing transformations.

Our contribution is a code transformation, dubbed "Tail modulo Async/Await" and inspired by Bour et al. [1], which precisely transform such functions to run in constant stack space. We now give some ideas how this transformation proceeds on examples.

*Conference'17, July 2017, Washington, DC, USA*
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
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)
```

**Figure 1.** A map on binary trees. The right recursive call to map will be terminal with Tail-Modulo-Cons [1], but the whole function is not.

```
1 let rec async map f = function
2   | Leaf -> Leaf
3   | Node(v, tl, tr) ->
4     Node(f v,
5          await (map f tl),
6          await (map f tr))
```

**Figure 2.** An *asynchronous* concurrent map on binary trees. With our transformation, this function runs in $O(1)$ stack.

## 2 Tail-Modulo-Async/Await in Action

We now showcase our code transformation on the `map` example from Figure 2. The transformed code is shown in source syntax in Figure 3. It has been modified in several meaningful ways which we detail in the rest of this section. First, *map* now relies on `map_dps` in *Destiny passing style*, using an output-argument `d`. Then, `map_dps` itself uses a new operator, `refine`, to build the result in-place. Finally, recursive calls to `map` have been replaced by tail-calls under `fork`.

### 2.1 Destiny Passing Style

The core of a tail-modulo [4] transformation is moving the necessary context from the function body to its parameter. An example of this is the continuation-passing-style where the whole function continuation is passed as an argument. Another case is destination-passing-style, used by Bour et al. [1], that exposes the memory location to-be-filled during

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

**Figure 3.** Source version of the transformed code of the concurrent *map* in Destiny Passing style.

computation. For Tail Modulo Async/Await, the context of the computation is the *destiny* [3], i.e. where does the end-result of the asynchronous computation should go. In practice, we rely on value with multiple holes, in the style of Minamide [6].

With that in mind, we can take another look at `map_dps` and explain the transformation. First, the top-level *map* function generates a destiny `d` and its associated promise with `Destiny.fresh`. The destiny `d` points to a hole, and the promise will contain the value when `d` is fully filled. `map` then calls `map_dps` with the destiny `d`. `map_dps` never returns any value, but will fill the holes in `d`. This allow us to transform all its recursive calls to tail-calls under fork, on Line 11 and 12. Looking back at the original code in Figure 2, all we now need to deal with are the two constructors `Leaf` and `Node`.

### 2.2 Constructors with `refine`

To implement constructors with mutliholes, and more generally "values with holes", we rely on `Destiny.refine`, whose behavior is shown graphically on Figures 4 and 5.

In the simplest case, `Destiny.refine` takes as argument a destiny `d`, a value `v` and fill `d` with `v`. This is illustrated in the base case of `map_dps`, on Line 7, where `d` is filled (in place) with `Leaf`. Naturally, `v` might itself contains holes! This is the case in the `Node` case, as shown on Line 9-10. In this case, `Destiny.refine` returns a set of destinies pointing to all the new holes in the structure. For instance, Figure 4 illustrate the situation where `let d3, d4 = Destiny.refine d1 ctx`.

Finally, at some point in the execution, the complete structure will become hole-free, as illustrated in Figure 5. At this point, `refine` raises a flag to resolve the promise introduced by `Destiny.fresh`.

Crucially, destinies are *write-only*, and only used *linearly*. Indeed, since all calls are in tail position, the value inside a

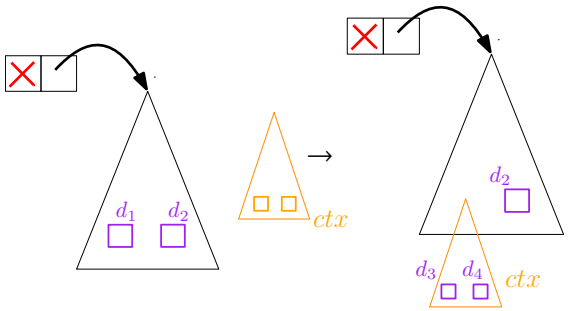


**Figure 4.** *refine* filling a destiny with a structure that has holes. It allocates destinies to the holes of the structure, and returns them.

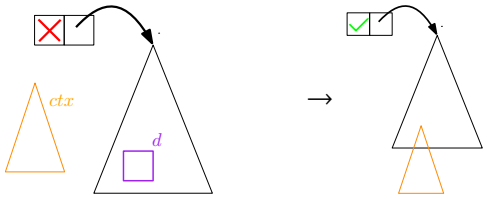


**Figure 5.** `refine` tracks internally the number of holes. When it reaches zero, it sets a flag to True, and the promise resolver can be called

destiny will not only be read once all computation is done, through the promise. With that in mind, we can look at Figure 4 and decompose what `refine` does here : it allocates locations for each hole in the new context ($d_3$ and $d_4$), it replaces what $d_1$ points to with $ctx$, and updates the number of holes inside the overall structure. If that count is 0, it sets a flag to True, which means the promise can be resolved, as seen in Figure 5

### 2.3 Chaining

Our DPS transformation turns asynchronous tail-modulo-cons functions into asynchronous tail-recursive functions. We can in fact handle more general cases, such as any expression `await e` in tail position. For instance, let's go back to Figure 2, but replacing `f v` with `await (f v)`. `f` might have a DPS version, which we could readily use. Otherwise, we can still avoid having to await on `f v` by using `chain`.

Chains, introduced by Fernandez-Reyes et al. [2], allow to delegate some asynchronous computation to another function. In our context, a chain is equivalent to adding a callback indicating that, when `f v` is done, it should fill a given destiny `d`. A naive implementation might cause long chains of indirections. Following Fernandez-Reyes et al. [2], we implement an optimized version adapted to our setup.

## 3 Content of the Talk

In the talk, we will present the formalization of the operations and transformation, along with a partial proof of bisimulation. We will showcase transformations on more complex examples, and a work-in-progress implementation in OCaml which leverages OCaml 5's new concurrency capabilities [7, 8].

## References


[1] Frédéric Bour, Basile Clément, and Gabriel Scherer. 2021. Tail Modulo Cons. *CoRR* abs/2102.09823 (2021). arXiv:2102.09823 https://arxiv.org/abs/2102.09823

[2] Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, and Huu-Phuc Vo. 2018. Forward to a Promising Future. In *Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018. Proceedings (Lecture Notes in Computer Science, Vol. 10852)*, Giovanna Di Marzo Serugendo and Michele Loreti (Eds.). Springer, 162–180. https://doi.org/10.1007/978-3-319-92408-3_7

[3] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2010. ABS: A Core Language for Abstract Behavioral Specification. In *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers (Lecture Notes in Computer Science, Vol. 6957)*, Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue (Eds.). Springer, 142–164. https://doi.org/10.1007/978-3-642-25271-6_8

[4] Daan Leijen and Anton Lorenzen. 2023. Tail Recursion Modulo Context: An Equational Approach. *Proc. ACM Program. Lang.* 7, POPL (2023), 1152–1181. https://doi.org/10.1145/3571233

[5] Barbara Liskov and Liuba Shrira. 1988. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, Richard L. Wexelblat (Ed.). ACM, 260–267. https://doi.org/10.1145/53990.54016

[6] Yasuhiko Minamide. 1998. A Functional Representation of Data Structures with a Hole. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, David B. MacQueen and Luca Cardelli (Eds.). ACM, 75–84. https://doi.org/10.1145/268946.268953

[7] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 113:1–113:30. https://doi.org/10.1145/3408995

[8] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 206–221. https://doi.org/10.1145/3453483.3454039

[9] Wikipedia contributors. 2024. Tail call — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Tail_call&oldid=1221167354. [Online; accessed 28-May-2024].