

# Tail Modulo Async-Await

EMMA NARDINO, ENS Lyon, Univ Lyon, UCBL, CNRS, Inria, LIP, France

LUDOVIC HENRIO, CNRS, Univ Lyon, ENS Lyon, UCBL, Inria, LIP, France

GABRIEL RADANNE, Inria, Univ Lyon, ENS Lyon, UCBL, CNRS, LIP, France

YANNICK ZAKOWSKI, Inria, Univ Lyon, ENS Lyon, UCBL, CNRS, LIP, France

This article extends tail-call optimisation by applying it to asynchronous calls. We first introduce *Tail-Modulo-Await*, a novel code transformation for asynchronous tail recursive functions that prevents the creation of unnecessary tasks. We then show how to combine Tail-Modulo-Await with the existing Tail-Modulo-Cons optimisation; we obtain an optimisation able to turn a recursive function with multiple tail calls under constructors into a parallel version of the function, also optimised in space.

We formalise both optimisations over representative calculi, and prove them correct through backward simulations. Finally, we provide a proof-of-concept implementation as an OCaml syntax extension and evaluate it experimentally, showing our approach optimises both memory and execution time.

## ACM Reference Format:

Emma Nardino, Ludovic Henrio, Gabriel Radanne, and Yannick Zakowski. 2025. Tail Modulo Async-Await. 1, 1 (July 2025), 40 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 Introduction

The Tail Call Optimisation has been a staple of functional programming languages since its introduction in 1977 in Steele Jr. [23]’s seminal paper “LAMBDA: The Ultimate GOTO”. This optimisation hinges on the observation that, when a function call is in *tail position*, i.e., a return position in the program, it can be compiled as a simple JUMP instruction, thus dramatically improving the efficiency of procedure calls. This transformation, which also saves space in the function stack, made its way in numerous languages and compilers, whether enabled by default in languages such as OCaml, Haskell, or Scala; or as an on-demand optimisation in compilers such as LLVM and GCC.

The notion of tail position was initially quite restrictive: only calls in exact return position were considered. Tail-Call *modulo Constructor* [13, 21] extends the class of accepted program with single tail-positions under a data constructor. Thanks to this notion, most functions over lists are now considered tail-recursive, and thus liberated from stack constraints.

However, while lists remain a data structure of choice for functional programmers, shouldn’t trees deserve the same care? More broadly, can we hope for a similar optimisation in presence of *multiple* calls in tail position? Surely, this makes little sense in a sequential context: how could we launch several tail-calls without any synchronisation? However, this makes perfect sense in an *asynchronous* context, and more specifically in the presence of futures [7],<sup>1</sup> which are entities representing the result of an ongoing computation. In fact, a notion similar to asynchronous tail calls already exists for Objective Oriented method calls in asynchronous languages, dubbed *forward* [10].

In this article, we further extend the notion of tail-position to be “modulo Await”. Similarly to how tail-calls in sequential contexts are optimised to use constant stack space, we show how to optimise tail-calls in asynchronous context in order to use no spurious scheduler space. Furthermore, when

<sup>1</sup>Coincidentally, both works were published in 1977, and both originated from the LISP community!

---

Authors’ Contact Information: Emma Nardino, [emma.nardino@ens-lyon.fr](mailto:emma.nardino@ens-lyon.fr), ENS Lyon, Univ Lyon, UCBL, CNRS, Inria, LIP, Lyon, France; Ludovic Henrio, [ludovic.henrio@cnrs.fr](mailto:ludovic.henrio@cnrs.fr), CNRS, Univ Lyon, ENS Lyon, UCBL, Inria, LIP, Lyon, France; Gabriel Radanne, [gabriel.radanne@inria.fr](mailto:gabriel.radanne@inria.fr), Inria, Univ Lyon, ENS Lyon, UCBL, CNRS, LIP, Lyon, France; Yannick Zakowski, [yannick.zakowski@inria.fr](mailto:yannick.zakowski@inria.fr), Inria, Univ Lyon, ENS Lyon, UCBL, CNRS, LIP, Lyon, France.

```

1 type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree
2
3 val map : ('a -> 'b) -> 'a tree -> 'b tree future
4 let%async rec map f t = match t with
5 | Leaf -> Leaf
6 | Node (x , tl, tr) -> Node (f x, await (map f tl), await (map f tr))

```

Fig. 1. An asynchronous map on trees.

combined with Tail-Modulo-Cons (tmc), we obtain a Tail-Modulo-Cons-Await optimisation (tmca) able to optimise functions with multiple recursive calls under a constructor. We demonstrate this feature over trees.

### A tail-recursive map on trees

Fig. 1 showcases an implementation of an asynchronous `map` function on trees in an OCaml dialect with asynchronous functions.<sup>2</sup> Before going over details, let us state our selling point: our code transformation parallelizes such functions (calls to the mapped function are executed concurrently), uses a constant memory space per thread, and doesn't use any superfluous scheduler space.

We provide the signature for `map` on Line 3: it takes a function `f`, a tree `t`, and returns the tree where `f` has been applied to every element of `t`. In our code, asynchronous-related operations are indicated in blue and code internal to our optimisation in peach. Concretely, we consider a parametric tree type, defined on Line 1, where each `Node` is binary and contains a value. As is common for asynchronous functions, `map` returns more precisely a *future* for this tree. By tagging the function as `async` on Line 4, calls to this function create a future and launch the associated computation in a parallel sub-task. The result can then be retrieved using the `await` primitive: this is done for both recursive calls on Line 6. Finally, we assemble the value into a `Node` and return it, implicitly in a future. In the absence of any optimisation, the behaviour of this function would be, on each `Node` constructor, to launch the task for one branch, wait for it to finish, launch the task for the other branch, wait for it to finish, then allocate the new constructor and return it. The stack would grow linearly in the height of the tree; furthermore, the execution would be overall sequential, while paying additional cost for synchronisation.

The seasoned asynchronous programmer would however favour a more parallel version and parallelize the treatment of the left and right branches: both sub-tasks are launched by the calls first, and run simultaneously before being both `awaited` under the `Node`. The main drawback is that this code consumes a lot of scheduler space because all the finished tasks which have at least one unfinished task below it in the tree structure are still blocked in an `await` statement.

We introduce a new keyword `asyncpar` to tag functions to be optimized, i.e., replacing Line 4 with:

```
let%asyncpar rec map f t = match t with
```

With this tag, our approach parallelizes the subtasks, and automatically terminates all tasks that are only awaiting on others. To achieve this, we translate the function to *Destiny Passing Style*, an asynchronous interpretation of Destination Passing Style [22]. An idealized version of the transformed function `map_dps` is shown on Fig. 2a; its execution is illustrated on Fig. 2b.

At the top level, a single future `fut` is tied to the filling of a memory cell, a destiny `d` (Line 11). From there, a helper function `map_dps` is called. The function `map_dps` takes as an additional argument a *destiny* `d` (Line 1) which represents the memory location where the result of the asynchronous computation should be written to. When returning a value, such as `Leaf` on Line 2, we set the

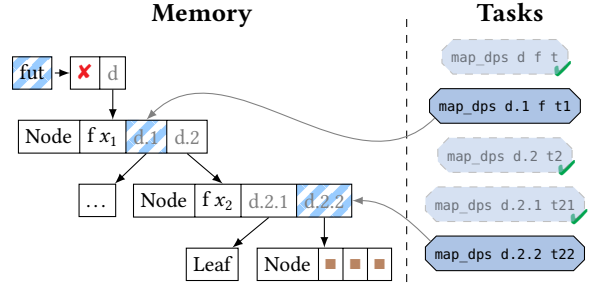
<sup>2</sup>Our dialect is implemented with an OCaml syntax extension and available at <https://zenodo.org/records/15757165>.

```

1 let rec map_dps d f t = match t with
2 | Leaf -> d ◀ Leaf
3 | Node (x, tl, tr) ->
4   let v = Node (■, ■, ■) in
5   d ◀ v;
6   fork_dps map_dps v.1 f tl;
7   fork_dps map_dps v.2 f tr;
8   v.0 ◀ f x
9
10 let map f l =
11 let d, fut = create_future () in
12 fork_dps map_dps d f l;
13 fut

```

(a) Code after tmca



(b) Snapshot of the execution of map f t after tmca

Fig. 2. Translation of the map on trees from Fig. 1 using Tail-Modulo-Cons-Await (if annotated `asynpcpar`)

destiny  $d$  with  $d \triangleleft \text{Leaf}$ . In the recursive case, we perform the allocation ahead of time—similarly to what happens in `tmc`—but put empty holes in lieu of its arguments. The current destiny can therefore be immediately set to  $v$ , the freshly allocated node. Both recursive calls are then forked in their own thread, one for each child: we must provide them with their respective destiny, i.e., the corresponding hole in  $v$  they are responsible for filling—we address fields of constructors by position. The calls to `fork_dps` handling recursive calls each fork a new task running the function `map_dps` with the arguments passed as parameters; they create no stack for executing `map_dps`—the only stack needed is for  $f$ —and discard the value returned by the function so that the call uses a minimal space. Finally, we compute the value  $f x$  and set it in  $v$ . When all parallel tasks have finished, there are no more holes in the structure, the future is marked as *resolved*, and the value can be retrieved. Fig. 2b shows an intermediate state of the execution: the status in memory of the tree being computed on the left, and the tasks involved in the computation on the right. Scheduler space is represented in blue. Futures and destinies both play a synchronisation role and serve as pointers to actual data, and are thus striped. The original future `fut` is yet unresolved (X). Tasks `(map_dps d f t)`, `(map_dps d.2 f t2)`, and `(map_dps d.2.1 f t21)` are finished, the corresponding destinies have been filled and the tasks have been garbage-collected. Task `(map_dps d.2.2 f t22)` has just performed the allocation of a `Node` (with holes), and filled its destination. Task `(map_dps d.1 f t1)` is still ongoing. This illustrates that only the tasks actually computing are still active on the scheduler side, contrarily to the versions that do not use `tmca`.

This code transformation automatically reveals parallelism present in the source program. But as such, it doesn't fully respect the sequential semantics. Indeed, the transformed code can *interleave* subtasks computing `map f tl` and `map f tr`, while the sequential semantics would run without interleaving. Thus, this automatic parallelisation comes at the risk of introducing undesired behaviors.

Thankfully, the programmer can easily prevent such an aggressive parallelisation by using the `async` annotation, as was done in Fig. 1. In such case, our approach preserves sequentiality while still optimising the code by reducing its memory footprint: it terminates all tasks which are only awaiting for others. Finally, it is worth noting that we only modify code *at the level of a constructor in tail position*: the programmer can therefore fine tune which computations should be preserved sequential by `awaiting` tasks outside the constructor inside an `asynpcpar` function.

## Contribution and Plan

In this paper, we introduce the following contributions.

- In [Section 2](#), we introduce a novel code transformation for asynchronous tail-recursive function dubbed Tail-Modulo-Await. This code transformation generalises and automates the treatment of `forward` present in the literature on asynchronous computing.
- In [Section 3](#), we combine Tail-Modulo-Await and Tail-Modulo-Cons to derive a transformation allowing for having multiple tail calls launched concurrently. This code transformation can, depending on the user’s choice, maximise parallelism, or strictly preserve the sequential semantics.
- We formalise both on minimal calculi and prove them sound via backward simulations.
- We provide a proof-of-concept implementation as an OCaml syntax extension and evaluate it experimentally in [Section 4](#).

## 2 Tail-Modulo-Await

In this section, we set aside constructors and focus on an optimisation dubbed Tail-Modulo-Await (`tma`)—we shall extend this optimisation to handle our introductory example later, in [Section 3](#). After illustrating informally the optimisation on an example, we introduce two core calculi, a source and a target, over which we formalise the transformation.

### 2.1 Motivating example

[Fig. 3](#) presents a toy program illustrating `tma`. The `check_urls` function iterates recursively over a list of urls in order to check whether they actually point to some resource. Such checks require to communicate over the network, which takes time: to avoid halting the whole system, we hence perform them via a call to an asynchronous operation `check_url: url -> bool future`. This operation, provided by our favourite Web Client, is wrapped in an `await` on [Line 6](#). But a similar issue arises on [Line 7](#) in the recursive call! The `check_urls` function itself must therefore be declared asynchronous, and recursive calls must be wrapped in an `await`.

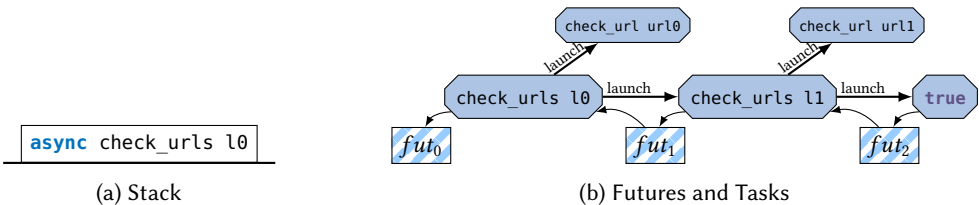
```

1 val check_urls : url list -> bool future
2 let%async rec check_urls files =
3   match files with
4   | [] -> true
5   | url :: rest ->
6     if await (check_url url) then
7       await (check_urls rest)
8     else false

```

[Fig. 3](#). Checking all URLs, with `async/await`

Recursion and `async/await` seem to play well together: we have elegantly turned our recursive function into a non-blocking one. But one may wonder what the space behaviour of such a function is? Indeed, the synchronous equivalent (where `async/awaits` are removed) is clearly tail-recursive, and hence its recursive calls can operate in constant stack. Hopefully, the same is true of this asynchronous variant. But in this case, `check_urls`’s memory behaviour deserves closer inspection, as illustrated on [Fig. 4](#). Strictly speaking, the function does not consume stack-space either: it delegates to `async`, where each call to `await` is in tail position. However, as illustrated in [Section 1](#), in an asynchronous context we must worry about *scheduler-space* as well! Each recursive call creates a future, which needs to be awaited for, forming a chain the size of the list `l0` taken as argument. As the chain grows, the memory gets cluttered with un-active futures and tasks.



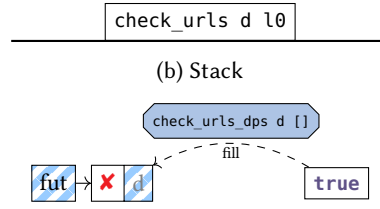
[Fig. 4](#). Memory behaviour of `check_urls`

```

1 let rec check_urls_dps d files =
2   match files with
3   | [] -> d ◀ true
4   | url::rest ->
5     if await (check_url url) then
6       check_urls_dps d rest
7     else d ◀ false
8
9 let check_urls l =
10  let d, fut = create_future () in
11  fork_dps check_urls_dps d l;
12  fut

```

(a) Code after tma



(c) Futures, Destinies and Tasks.

Here, the task reaching the end of the list is filling the destiny d. Only one task for `check_urls_dps` is alive at any point. There is only one call in the stack for `check_urls`.

Fig. 5. Checking all URLs with the Tail-Modulo-Await transformation

As per its name, Tail-Modulo-Await optimises such functions that are almost tail recursive, but whose tail calls happen under `await`. Consider the effect of the optimisation in Fig. 5: the stack remains of constant size, only one recursive task is ever alive, and the chain of futures is reduced to length one. To achieve this, the recursive computation `check_urls_dps` is crucially not asynchronous any more. Furthermore, it takes a destiny `d` as argument, a *write-once* memory cell in which we shall set the boolean resulting from its computation. The function `check_urls` is then defined as a top level wrapper that creates a pair of a destiny to be passed to the helper, and of a future that will get resolved by this destiny: such a pair is referred to as a *promise* [20] in the literature. To ensure we are non-blocking, the helper is forked in a separate thread, before returning the future. Naturally, calls to `await` which are not in tail position remain, such as the one to `check_url` on Line 5.

One might wonder what happens if a terminal position contains a call to a function which is not in Destiny-Passing-Style? For instance, if we were to unroll the `check_urls` function once, we would obtain the following extra-clause in our pattern matching:

```
| [url] -> await (check_url url)
```

The `check_url` function, which is provided by an external library, doesn't necessarily have a DPS version. Nevertheless, we would like to avoid creating an intermediary task whose sole purpose is to wait before filling the destiny. This exactly corresponds to `forward` [10] from the literature. Specifically, `forward (d, fut)` registers that when future `fut` is resolved, the result should be used to set destiny `d`. This can be implemented efficiently in the scheduler to avoid an extra task. Furthermore, it works for any future, not just asynchronous function applications. This efficiently bridges the gap between the DPS world and the rest of the asynchronous world.

## 2.2 Calculus

We now present our calculus, a first order imperative and asynchronous  $\lambda$ -calculus which formalises this first transformation, Tail-Modulo-Await, without any constructor. Fig. 6 shows at once the syntax of both the source and the target language, using the following colour code: *Light blue* elements are source-specific, *peach* elements are target-specific, and Grey elements are runtime syntax.

*Shared core.* Both languages include standard control flow constructs as well as support for references, with operations for creation (`newref(e)`), dereferencing (`!e`), and update (`e := e`). As motivated earlier, this first calculus contains no data type, static values are reduced to basic types. Its functional side is first order: we assume all functions are  $\lambda$ -lifted in a table of function definitions *Fun*. In both languages, functions are tagged with their calling behaviour when declared: sequential by default, or with an asynchronous behaviour. When a task *t* is created, a future *fut* is associated

(expressions) $Expr \ni e ::= v \mid x \in Var \mid \mathbf{let} \ x = e \ \mathbf{in} \ e' \mid (f \ e) \mid \mathbf{if} \ e \ \mathbf{then} \ e_t \ \mathbf{else} \ e_f \mid \mathbf{newref}(e) \mid !e \mid e := e \mid \mathbf{await}(e) \mid (f \#_t(e', e)) \mid \mathbf{forward}(e, e') \mid \mathbf{refine}(e, e')$	(functions) $fd ::= f \mapsto \lambda x. e \mid f \mapsto \lambda_{\text{async}} x. e \mid f \mapsto \lambda_{\text{dps}} \delta, x. e$ $Fun ::= \left[ \overline{fd} \right]$
(values) $Val \ni v ::= b \in \mathbb{B} \mid c \in Const \mid \ell \in Loc \mid fut \in \mathbb{F} \mid d \in Dest$	

Fig. 6. Syntax – **Light blue** elements only exist in the source language and **peach** elements only in the target language. Grey elements denote runtime syntax.

to it, and any task can retrieve the result of the computation by performing  $\mathbf{await}(fut)$ . If  $t$  is not finished yet, we say that  $fut$  is *unresolved*, in which case  $\mathbf{await}(fut)$  is blocking; otherwise, it returns the computed value. Futures are runtime values.

*Source language.* The only syntax for function calls is  $(f \ e)$ . The behaviour of the call directly depends on the tag associated to  $f$  in its declaration: synchronous by default, unless tagged by  $\lambda_{\text{async}}$ . In the latter case, the behaviour is asynchronous: each call spawns both a task and a fresh future, whose resolution is tied to the task.

*Target language.* Here, there are no such coupling between a future and a task responsible for fulfilling it. Instead, each future is mapped to exactly one *destiny*: a value acting as a write-once memory location which can be used to fill the future. A pair of a read-only future and a write-once reference is often referred to as a *promise* in asynchronous programming languages. These destinies are not bound to a particular task, and can be passed by argument to subsequent calls to asynchronous functions. To this end, asynchronous functions are declared in destiny-passing style (dps), i.e., of the shape  $(\lambda_{\text{dps}} \delta, x. e)$ , where  $\delta$  is the variable used to pass the destiny.

There are two distinct ways to call a dps function. The first one, as  $(f \ e)$ , is similar to the source language: it creates a task, a promise (read a pair  $fut, d$ , where  $d$  is the destiny resolving future  $fut$ ) and maps the destiny to  $d$ . This corresponds to automatically introducing the wrapper function in Fig. 5. The second one is specific to tail calls to asynchronous functions: we explicitly represent such calls in the syntax as  $(f \#_t(d, e))$ . Instead of creating a fresh promise, such tail calls *delegate* the resolution of the promise to the newly created task, by passing its destiny  $d$  as the new task’s destiny. This behaviour corresponds to  $\mathbf{fork\_dps}$  calls in Fig. 5.

Unlike futures, the resolution of destinies is not associated to the end of a task but must be performed explicitly, via the statement  $\mathbf{refine}(d, e)$ . In the absence of constructor, this corresponds to  $d \leftarrow e$  in our informal syntax.<sup>3</sup> Finally,  $\mathbf{forward}(fut, d)$  binds the resolution of a promise to the resolution of another one: the destiny  $d$  is automatically filled when the promise  $fut$  is resolved.

### 2.3 Semantics

We now equip both languages with an operational, small-step, semantics. We assume a table of function definitions  $Fun$  implicitly parametrising the semantics.

*Shared core.* At runtime, values are extended with locations and futures (Fig. 6). Both languages share a thread-local core, whose reduction is defined over *local* configurations  $(\sigma, e)$  carrying a store and an expression. The local reduction  $\cdot \rightarrow \cdot$  is provided on Fig. 8. This reduction relation is completely standard: evaluation contexts are defined on Fig. 7, and the four other rules specify the operations over the store, and the beta reduction for sequential function calls.

<sup>3</sup>As the name hints,  $\mathbf{refine}$  takes on more responsibilities when constructors are introduced, in the next section

(store)  $\sigma ::= [\overline{\ell \mapsto v}]$   
 (evaluation contexts)  $Ectx \ni C ::= \square \mid \mathbf{let} \ x = C \ \mathbf{in} \ e \mid \mathbf{if} \ C \ \mathbf{then} \ e_t \ \mathbf{else} \ e_f \mid (f \ C)$   
 $\mid \mathbf{newref}(C) \mid C := e \mid \ell := C \mid \mathbf{await}(C)$   
 $\mid (f \#_t C) \mid \mathbf{forward}(C, e') \mid \mathbf{refine}(e, C)$

Fig. 7. Runtime syntax for Stores  $\sigma$  and Contexts  $C$ 

$$\frac{\sigma, e \rightarrow \sigma', e'}{\sigma, C[e] \rightarrow \sigma', C[e']} \quad \frac{\ell \notin \text{dom}(\sigma)}{\sigma, \mathbf{newref}(v) \rightarrow \sigma[\ell \mapsto v], \ell} \quad \frac{(\ell \mapsto v) \in \sigma}{\sigma, !\ell \rightarrow \sigma, v}$$

$$\frac{}{\sigma, (\ell := v) \rightarrow \sigma[\ell \mapsto v], ()} \quad \frac{(f \mapsto \lambda x. e) \in \text{Fun}}{\sigma, (f \ v) \rightarrow \sigma, e[x \mapsto v]}$$
Fig. 8. Sequential semantics (rules for **if** and **let** are omitted) –  $\sigma, e \rightarrow \sigma', e'$ 

(unordered list of tasks)  $\text{Task} ::= (\text{main} \mapsto e) \sqcup (\text{fut} \mapsto e)$   
 (source configurations)  $c ::= \sigma, \text{Task}$

(a) Runtime configurations

$\sigma_s = \emptyset$   
 $\text{Task}_s = (\text{main} \mapsto e)$

(b) Initial configurations

Fig. 9. Source Configuration

STEP

$$\frac{\sigma, e \rightarrow \sigma', e'}{\sigma, \text{Task} \sqcup (\text{fut} \mapsto e) \rightarrow \sigma', \text{Task} \sqcup (\text{fut} \mapsto e')}$$

AWAIT-RESOLVED

$$\frac{(\text{fut}' \mapsto v) \in \text{Task}}{\sigma, \text{Task} \sqcup (\text{fut} \mapsto C[\mathbf{await}(\text{fut}')]) \rightarrow \sigma, \text{Task} \sqcup (\text{fut} \mapsto C[v])}$$

ASYNC-CALL

$$\frac{(f \mapsto \lambda_{\text{async}} x. e) \in \text{Fun} \quad \text{fut}' \notin \text{dom}(\text{Task}) \quad \text{fut}' \neq \text{fut}}{\sigma, \text{Task} \sqcup (\text{fut} \mapsto C[(f \ v)]) \rightarrow \sigma, \text{Task} \sqcup (\text{fut}' \mapsto e[x \mapsto v]) \sqcup (\text{fut} \mapsto C[\text{fut}'])}$$
Fig. 10. Global source reduction –  $\sigma, \text{Task} \rightarrow \sigma', \text{Task}'$ 

*Source language.* In the source, task creation and synchronisation is entirely managed through futures: runtime configurations (Fig. 9) pair a store with a set of *tasks*, each bound to a different *future* (with the exception of the main thread). Initial configurations have an empty store and a single *main* task, while final configurations have all tasks evaluated down to a value.

The concurrent reductions are given on Fig. 10. The **STEP** rule allows any task to progress according to the local semantics. Rule **AWAIT-RESOLVED** evaluates an **await**(*fut'*) statement: if the computation of *fut'* is reduced down to a value, this value is substituted. Finally, the **ASYNC-CALL** rule handles task creation by creating a fresh future and a new entry in the set of tasks.

*Target language.* Runtime configurations for the target language are described on Fig. 11. Contrary to the source, *any* thread can fulfil a future: *Fut* hence keeps track of their status. We write  $[ \text{fut} \mapsto \mathcal{U} d ]$  when *fut* is yet unresolved, and tied to the destiny *d*; and  $[ \text{fut} \mapsto \mathfrak{R} v ]$  when *fut* is resolved with value *v*. The target tasks are not indexed by futures, but rather by anonymous identifiers, written *tid*, that cannot be awaited. Initial configurations are similar to the source language, with an empty

(maps of futures)	$Fut ::= \mathbb{F} \xrightarrow{\text{fin}} Dest + Val$	$\sigma_t = \emptyset$
(Tasks)	$Task ::= (main \mapsto e) \sqcup (tid \mapsto e)$	$Task_t = (main \mapsto e)$
(target configurations)	$c ::= \sigma, Task, Fut$	$Fut_t = \emptyset$

(a) Runtime configuration

(b) Initial configuration

Fig. 11. Target Configuration

STEP	$\frac{\sigma, e \rightarrow \sigma', e'}{\sigma, Task \sqcup (tid \mapsto e), Fut \rightarrow \sigma', Task \sqcup (tid \mapsto e'), Fut}$
AWAIT-TARGET	$\frac{Fut [ fut ] = \mathfrak{R} v}{\sigma, Task \sqcup (tid \mapsto C[\mathbf{await}(fut)]), Fut \rightarrow \sigma, Task \sqcup (tid \mapsto C[v]), Fut}$
CHAIN	$\frac{Fut [ fut ] = \mathfrak{U} d \quad Fut [ fut' ] = \mathfrak{R} v}{\sigma, Task \sqcup (tid \mapsto \mathbf{forward}(fut', d)), Fut \rightarrow \sigma, Task, Fut [ fut \mapsto \mathfrak{R} v ]}$
DPS-CALL	$\frac{(f \mapsto \lambda_{\text{dps}} \delta, x. e) \in Fun \quad fut', d, tid' \text{ fresh}}{\sigma, Task \sqcup (tid \mapsto C[(f v)]), Fut \rightarrow \sigma, Task \sqcup (tid \mapsto C[ fut' ] ) \sqcup (tid' \mapsto e[\delta \mapsto d][x \mapsto v]), Fut [ fut' \mapsto \mathfrak{U} d ]}$
TAIL-CALL	$\frac{(f \mapsto \lambda_{\text{dps}} \delta, x. e) \in Fun \quad tid' \notin \text{dom}(Task)}{\sigma, Task \sqcup (tid \mapsto (f \#_t (d, v))), Fut \rightarrow \sigma, Task \sqcup (tid' \mapsto e[x \mapsto v][\delta \mapsto d]), Fut}$
FUTURE-FILL	$\frac{Fut [ fut ] = \mathfrak{U} d}{\sigma, Task \sqcup (tid \mapsto \mathbf{refine}(d, v)), Fut \xrightarrow{\varepsilon} \sigma, Task, Fut [ fut \mapsto \mathfrak{R} v ]}$

Fig. 12. Global target reduction –  $\sigma, Task, Fut \rightarrow \sigma', Task', Fut'$ 

map of futures. A configuration is *final* if its main thread is reduced to a value, it is the only thread left, and every future in  $Fut$  is resolved.

Reduction is defined on Fig. 12. Local reductions can still be lifted to any task (**STEP**), and rule **AWAIT-TARGET** handles futures synchronisation:  $\mathbf{await}(fut)$  can only be evaluated if  $fut$  is resolved, i.e., if  $Fut [ fut ] = \mathfrak{R} v$  for some value  $v$ .

**CHAIN** evaluates a  $\mathbf{forward}(fut', d)$  statement when  $fut'$  is resolved, and in turn resolves the future linked to destiny  $d$ . Its behaviour is seemingly similar to the one of  $\mathbf{await}$ : it blocks until  $fut'$  is resolved. However  $\mathbf{forward}$  can only occur in tail position, as captured by the absence of an evaluation context in **CHAIN**. Thus tasks of the form  $(tid \mapsto \mathbf{forward}(fut, d))$  are kept asleep in the configuration until  $fut$  is resolved, but do not block pending computations. An optimised implementation could store such **forwarded** destinies directly on the future  $fut$  itself.

**DPS-CALL** and **TAIL-CALL** handle asynchronous function calls, as described previously. **DPS-CALL** creates a fresh promise (thus a new mapping  $[ fut \mapsto \mathfrak{U} d ]$  in  $Fut$ ), and a new task resolving said



source. These steps exactly correspond to **FUTURE-FILL**, whose transitions are identified by an “ $\varepsilon$ ”. We hence need to ensure that a transformed program cannot take infinitely many  $\varepsilon$  steps in a row.

*Definition 2.1 (backward simulation).* We say that  $\succsim$  is a *backward simulation* if for any  $c_s, c_t, c'_t$  such that  $c_s \succsim c_t$ , we have:

- if  $c_t \xrightarrow{\varepsilon} c'_t$ , then  $c_s \succsim c'_t$
- if  $c_t \rightarrow c'_t$ , there exists  $c'_s$  such that  $c_s \rightarrow^+ c'_s$ , and  $c'_s \succsim c'_t$

Which allows us to state the correctness theorem for **tma**

**THEOREM 2.2.** *There exists a relation  $\succsim$  such that:*

- (1)  $\succsim$  is a backward simulation
- (2) forall  $c_s$  initial source configuration,  $c_s \succsim \mathcal{D}[[c_s]]$
- (3) if  $c_s \succsim c_t$ , and  $c_t$  is a final configuration, then there is a final source configuration  $c'_s$  such that  $c_s \rightarrow^* c'_s$ , and  $c'_s$  and  $c_t$  have the same final main value.

In the remaining of this section, we define the relation  $\succsim$  and sketch the simulation proof.

**2.5.1 Definition of  $\succsim$ .** We define  $\succsim$  on **Fig. 14c** in three successive layers. First at the level of expressions, the relation  $\sim_d$  is parametrised by a destiny  $d$ . It relates a source expression  $e$  to its syntactic transformation, in which the free variable  $\delta$  is substituted by  $d$  (**EXPR-TRANSFO**). This alignment can however be broken: the target may reduce to expressions like **refine**( $d$ , **await**( $e$ )), with synchronous function calls in tail-positions (**EXPR-REFINE**).

At the level of tasks,  $\simeq$  (**Fig. 14b**) relates sets of threads which collectively capture the same computation. The relation is of the form  $T_s \simeq T_t, [fut_0 \mapsto \mathfrak{R}v|\mathfrak{U}d]$  where  $T_{s,t}$  denote sets of threads computing the same future. On the source side, a task computing an asynchronous function and its subsequent recursive calls is related on the target side to the corresponding pair of thread and future/destiny mapping. One of the main purpose of **tma** is to prevent chains of hanging tasks all awaiting the following, i.e., groups of tasks with the shape  $(fut_0 \mapsto \mathbf{await}(fut_1)) \sqcup \dots \sqcup (fut_{n-1} \mapsto \mathbf{await}(fut_n))$ . We hence introduce the notation  $\mathbf{Aw}[fut_0..fut_n]$  for such chains, and capture this intuition in **TASK-COMP**. We must furthermore keep track of resolved futures on either sides (**TASK-VAL**), and bookkeep intermediary states featuring a resolved future yet to be garbage collected (**TASK-AWAIT-VAL**).

Finally, **Fig. 14c** composes task-level relations into a simulation relation between configurations. The stores  $(\sigma_{s,t})$  and the main thread  $(\mathbf{main} \mapsto e)$  are in exact agreement. The other threads on each side can be partitioned into  $n$  pools  $(T_{s,t}^{(i)})$ , one for each future in the target, such that they are pairwise related by  $\simeq$ . Intuitively, a new pool is created when an asynchronous function is called from a non-tail position (by rule **ASYNC-CALL/DPS-CALL**), and are never removed from the configuration. Finally, we maintain two invariants: (6) that no two pools can use the same destiny name, and (7) that the futures resulting from tail-calls in a pool are *internal*, i.e., they cannot appear in the store or any other thread.

**2.5.2 Proof sketch.** The proof of backward simulation relies on the observation that (1) a step in a thread belonging to a pool leaves all other pools unchanged, and (2) it is simulated in the source by steps taken in the corresponding pool. We then distinguish two broad cases: a step in **main** must be exactly mirrored, while a step in pool  $T_t^{(i)}$  must be matched in  $T_s^{(i)}$  and restore (5) for index  $i$ . We refer the reader to **Appendix C** for the full case analysis, and detail here two informative cases.

*Stepping in main.* Since the target **main** is syntactically identical to the source one, the only steps it can take are local **STEPS**, **DPS-CALL**, and **AWAIT-TARGET**.

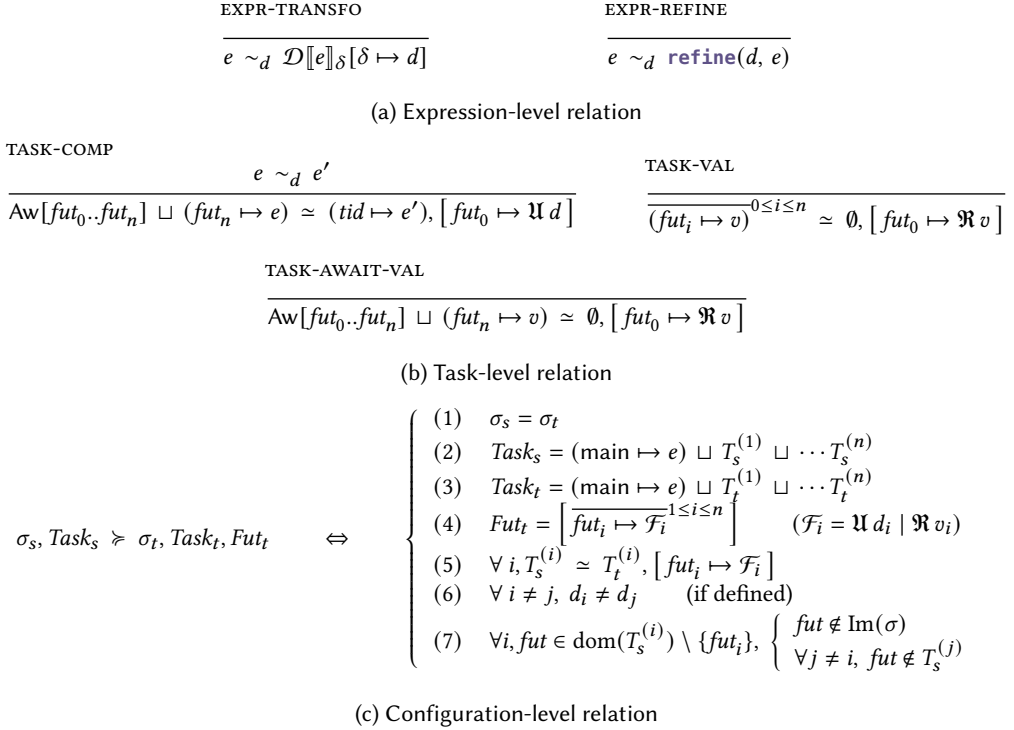


Fig. 14. Simulation relation

To illustrate thread creation, consider a **DPS-CALL** step: it is matched by an **ASYNC-CALL** in the source, using the same future name. The stores are omitted from the configurations.

$$\begin{array}{ccc}
 Task_s \sqcup (\text{main} \mapsto C[(f v)]) & \succcurlyeq & Task_t \sqcup (\text{main} \mapsto C[(f v)], Fut_t \\
 \downarrow \text{ASYNC-CALL} & & \downarrow \text{DPS-CALL} \\
 \downarrow (f \mapsto \lambda_{\text{async}} x. e) \in Fun_s & & \downarrow (f \mapsto \lambda_{\text{dps}} \delta, x. \mathcal{D}[[e]]_\delta) \in Fun_t \\
 Task_s \sqcup (\text{main} \mapsto C[fut']) & \succcurlyeq & Task_t \sqcup (\text{main} \mapsto C[fut']) \\
 \sqcup (fut' \mapsto e[x \mapsto v]) & \succcurlyeq & \sqcup (tid \mapsto \mathcal{D}[[e]]_\delta[\delta \mapsto d][x \mapsto v]), \\
 & & Fut_t \sqcup [fut' \mapsto \mathcal{U} d]
 \end{array}$$

The main threads (highlighted in **red**) are still identical, and the newly created tasks (in **blue**) verify clause (5) by **TASK-COMP**.  $d$  and  $fut'$  are fresh so (6) and (7) are satisfied.

*Stepping in a parallel thread.* Given a target step in pool  $T_t^{(i)}$ , clause (5) constrains the shape of the corresponding pool  $T_s^{(i)}$ . Considering a **TAIL-CALL** step as illustration:

$$\begin{array}{ccc}
 Task_s \sqcup \text{Aw}[fut_0..fut_n] & \succcurlyeq & Task_t \sqcup (tid \mapsto (f \#_t (d, v))), \\
 \sqcup (fut_n \mapsto \text{await}(f v)) & & Fut_t \sqcup [fut_0 \mapsto \mathcal{U} d] \\
 \downarrow \text{ASYNC-CALL} & & \downarrow \text{TAIL-CALL} \\
 Task_s \sqcup \text{Aw}[fut_0..fut_{n+1}] & \succcurlyeq & Task_t \sqcup (tid' \mapsto \mathcal{D}[[e]]_\delta[\delta \mapsto d][x \mapsto v]), \\
 \sqcup (fut_{n+1} \mapsto e[x \mapsto v]) & \succcurlyeq & Fut_t \sqcup [fut_0 \mapsto \mathcal{U} d]
 \end{array}$$

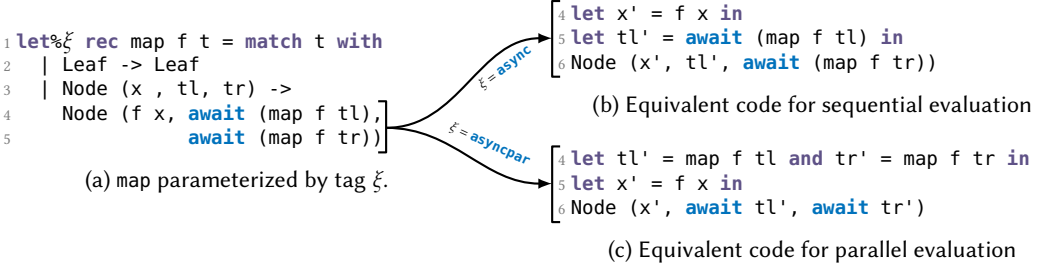


Fig. 15. A parametric map on trees

The task-level invariant  $\simeq$  ensures via **TASK-COMP** that we can take a matching **ASYNC-CALL** in the source. Since  $fut_{n+1}$  is fresh, (7) is preserved.

*Initial and final states.* Initial configurations are trivially related. If  $c_s \succ c_t$ , with  $c_t$  a final target configuration, then the main thread is reduced to a value  $v$ , and the same is true in  $c_s$  by clauses (2) and (3). Furthermore, every future is resolved, so each  $T_s^{(i)}$  is either of the form  $(\overline{fut_i} \mapsto v_i)$  by **TASK-VAL**, and thus satisfy finality; or  $\text{Aw}[fut_0..fut_n] \sqcup (fut_n \mapsto v)$ , in which  $n$  **AWAIT-RESOLVED** steps can be taken to reach a final configuration.

### 3 Tail-Modulo-Cons-Await

Before adding constructors to the calculus, let us revisit some subtle semantic considerations, using the example of a tail-recursive map on trees introduced in [Section 1](#).

As discussed in the introduction, we consider two possible semantics for asynchronous functions in the presence of **async/await**. If the function is tagged as **async** the arguments of the constructors placed in tail position are evaluated in sequence, while if the function is tagged **asyncpar** then parallelism is maximised. These behaviours are illustrated on [Fig. 15](#), which features the **map** on trees from [Fig. 1](#) with two possible function tags. We express the semantic meaning of these tags by means of a translation of lines 4 and 5 of **map** into two programs whose order of evaluation is made explicit through **let** bindings. Beware, this is *not* the result of the Tail-Modulo-Cons-Await transformation, which would also pre-allocate the constructor and avoid intermediate task creations, as showcased in [Section 1](#).

For the **async** tag, the *sequential* semantics showcased in [Fig. 15b](#) prevents any interleaving, and fixes an order of execution (here, left-to-right). In this case, only one **await** statement can be considered in tail position, the rest of the body must be awaited on to launch the final call. This semantics is similar to the Tail-Modulo-Cons transformation formalized by Allain et al. [4].

For the **asyncpar** tag, evaluation can maximise parallelism, as showcased in [Fig. 15c](#). The semantics is equivalent to first launching both sub-tasks (here,  $\text{map } f \text{ tl}$  and  $\text{map } f \text{ tr}$ ), and then the sequential code ( $f \ x$ ), hence removing any synchronisation between the tasks.

In both cases, **tmca** pre-allocates the **Node** constructor and constructs the structure in memory piece-by-piece, as shown in [Section 1](#). But crucially in the case of the parallel semantics, the intermediary structure during execution might have several holes. The top-level future is resolved when there are no more holes in the structure.

In the remainder of this section, we extend the calculus introduced in [Section 2](#) in order to formalise the Tail-Modulo-Cons-Await code transformation, and prove its correctness. It would not be convenient to tag functions in our calculus. Instead we tag constructors as **seq** or **||** with an equivalent semantics: a function tagged **async** instead has all its constructors tagged **seq** in the calculus, while a function tagged **asyncpar** has all its constructors tagged **||**. Tagging constructors

(expressions) $Expr \ni e ::= \dots$ $\quad   \{ t, e, e \}   [ t, e, e ]$ $\quad   e.(e)   e.(e) \leftarrow e$ $\quad   \blacksquare$ $\quad   \text{refine}(e, e)$	(values) $Val \ni v ::= \dots$ <span style="border: 1px solid grey; padding: 2px;"><math>d, \ell \in Loc</math></span> $\quad   t \in Tag   i \in Index$ $\quad   \diagup$ (tags) $Tag \ni t ::= t_{seq}   t_{  }$ (offsets) $Index \ni i ::= \{0, 1, 2\}$
--	--

Fig. 16. Syntax extension for constructors — Extends Fig. 6 — Light blue elements only exist in the source language and peach elements only in the target language. Grey elements denote runtime syntax.

(store) $\sigma ::= \dots$ (eval contexts) $Ectx \ni C ::= \dots$ <span style="border: 1px solid grey; padding: 2px;"><math>[ t, \blacksquare, C ]</math></span> $\quad   [ t, C, e ]   [ t, v, C ]   e.(C)   C.(v)$ $\quad   e.(e) \leftarrow C   e.(C) \leftarrow v   C.(v) \leftarrow v$ $\quad   \text{refine}(e, C)$
---

Fig. 17. Runtime syntax for constructors — Extends Fig. 7

$$\text{sta}(C_n^{tl}[\text{await}(e_1) | \dots | \text{await}(e_n)]) ::= C_n^{tl}[e_1 | \dots | e_n]$$

Fig. 18. Utility function to strip awaits in tail positions —  $\text{sta}(e)$

theoretically allows for more flexibility than tagging functions. But parallelisation only preserves the semantics if it is safe to have several executions of the function running concurrently. It thus makes more sense to ensure this safety property at the function level, and we consider tagging functions or constructors as equivalent in practice.

### 3.1 Calculus

To model data-constructors, we add binary constructors to the language as tagged mutable memory blocks  $\{ t, e_1, e_2 \}$ , with notations and semantics inspired by Allain et al.'s formalization of Tail-Modulo-Cons [4], but with the additional *sequential* ( $t_{seq}$ ) and *parallel* ( $t_{||}$ ) tags discussed above. As in [4], we have a separate, runtime-only, block construction  $[ t, e_1, e_2 ]$ , in which evaluation order is always left-to-right, and which performs allocation when all fields have finished evaluating.

Memory locations are still runtime-only, and there is no general pointer arithmetic. Once allocated, constructor fields are accessible using  $\ell.(i)$ , where  $i$  is an offset in  $\{0, 1, 2\}$ . Destinies are now actual memory locations, in the store.

The purpose of the refine instruction is extended: refine( $d, v$ ) takes a value  $v$  which may have some *holes*—syntactic markers (written  $\blacksquare$ ) denoting positions in a value that we want to compute later—, plugs said value at location  $d$ , and returns the locations of said *holes* in  $v$ . The positions marked are initialized with value  $\diagup$  (read *Undef*). For simplicity, we keep arguments of refine as *shallow* constructors (no sub-constructors) with at most two arguments. We extend slightly this construct for our implementation in Section 4.1.

### 3.2 Semantics

Both the source and target runtime configurations are the same as for the Tail-Modulo-Await transformation. The same is true for initial and final configurations.

*Shared core.* We add support for constructors with contexts in Fig. 17, utility functions in Fig. 18 and reduction rules in Fig. 19.

$\frac{\text{CONSTR-READ} \quad \sigma[\ell + i] = v}{\sigma, \ell.(i) \rightarrow \sigma, v}$	$\frac{\text{CONSTR-WRITE} \quad \ell + i \in \text{dom}(\sigma)}{\sigma, \ell.(i) \leftarrow v \rightarrow \sigma[\ell + i \mapsto v], ()}$	$\frac{\text{CONSTR-ALLOC} \quad \forall i \in \text{Index}, \ell + i \notin \text{dom}(\sigma)}{\sigma, [t, v_1, v_2] \rightarrow \sigma[\ell \mapsto t, v_1, v_2], \ell}$
$\frac{\text{CONSTR-FULL-PAR} \quad \text{sta}(e_1) = e'_1 \quad \text{sta}(e_2) = e'_2}{\sigma, \{t_{  }, e_1, e_2\} \rightarrow \sigma, \text{let } x_1 = e'_1 \text{ in } \text{let } x_2 = e'_2 \text{ in } [t_{  }, \text{await}(x_1), \text{await}(x_2)]}$	$\frac{\text{CONSTR-PAR-LEFT} \quad \text{sta}(e_1) = e'_1 \quad \text{sta}(e_2) \text{ undefined}}{\sigma, \{t_{  }, e_1, e_2\} \rightarrow \sigma, \text{let } x_1 = e'_1 \text{ in } \text{let } x_2 = e_2 \text{ in } [t_{  }, \text{await}(x_1), x_2]}$	
$\frac{\text{CONSTR-PAR-RIGHT} \quad \text{sta}(e_2) = e'_2 \quad \text{sta}(e_1) \text{ undefined}}{\sigma, \{t_{  }, e_1, e_2\} \rightarrow \sigma, \text{let } x_2 = e'_2 \text{ in } \text{let } x_1 = e_1 \text{ in } [t_{  }, x_1, \text{await}(x_2)]}$	$\frac{\text{CONSTR-SEQ-LEFT} \quad \text{sta}(e_1) = e'_1 \quad \text{sta}(e_2) \text{ undefined}}{\sigma, \{t_{\text{seq}}, e_1, e_2\} \rightarrow \sigma, \text{let } x_2 = e_2 \text{ in } \text{let } x_1 = e'_1 \text{ in } [t_{\text{seq}}, \text{await}(x_1), x_2]}$	
$\frac{\text{CONSTR-BASE} \quad \text{other cases}}{\sigma, \{t, e_1, e_2\} \rightarrow \sigma, \text{let } x_1 = e_1 \text{ in } \text{let } x_2 = e_2 \text{ in } [t, x_1, x_2]}$		

Fig. 19. Sequential semantics for constructors —  $\sigma, e \rightarrow \sigma', e'$  — Extends Fig. 8

To specify their reduction, we need to capture expressions of shape  $C_n^{tl}[\text{await}(e_1) \mid \dots \mid \text{await}(e_n)]$  where  $C_n^{tl}$  is a tail context (see Fig. 13a). In such expressions, all execution path lead to an **await**. Remarkably, such tail contexts “commute” with **awaits**, which is crucial to formalize which positions in a constructor are indeed task creations. To capture this, we define a partial function over expressions: **strip-tail-awaits** (shortened to **sta**), defined in Fig. 18, which removes all tail awaits. Intuitively, this removal separates synchronisation from computation: consider  $e' = \text{sta}(e) = C_n^{tl}[e_1 \mid \dots \mid e_n]$ , then  $e$  is semantically equivalent to first binding *fut* to  $e'$ , launching a parallel task, and then evaluating **await**(*fut*). Conversely, if  $\text{sta}(e)$  is undefined, expression  $e$  must be treated sequentially.

We can define reduction for constructors on Fig. 19. Stores are unchanged and evaluation contexts are easily extended in Fig. 17. Read (**CONSTR-READ**) and write (**CONSTR-WRITE**) accesses to constructors are defined by memory accesses. The final constructor allocation (**CONSTR-ALLOC**) is immediate by store access. Recall that  $\{t, e_1, e_2\}$  denotes syntactic constructors, which can have sub-expressions, while  $[t, x_1, x_2]$  is runtime syntax for a constructor allocation and can not have sub-expressions. The main challenge is in defining the evaluation of sub-expressions in a constructor depending on its tag. While the rules are numerous, they simply enumerate all possible node tags and positions for holes (one or two holes, left or right). We use let-bindings to represent evaluation order: for *parallel* constructors  $t_{||}$ , we first evaluate the “parallel” fields (i.e., such that **sta** is defined), then the sequential fields (similarly to **tmc**), and then perform the **await** for the parallel fields just before allocation; conversely for *sequential* constructors  $t_{\text{seq}}$ : non-parallel fields are computed first, thus preserving evaluation order.

*Source language.* The concurrent reduction of the source semantics is unchanged.

*Target language.* Our target semantics crucially relies on the fact that we can *resolve* a future only if it has no holes. To model this, we assume a magic function  $\text{holes} : \ell \rightarrow \mathbb{N}$  (see Section 4 for an efficient implementation) which returns the number of locations set to  $\swarrow$  in the tree rooted at  $\ell$ .

*Sequential Semantics.* We add rules for **refine** in the sequential semantics. In a transformed program, the second argument of a **refine** can be one of four cases, each described by one of the rules in Fig. 20: either a block with a parallel tag and two holes (**REFINE-PAR**), a block with a sequential

$$\begin{array}{c}
\text{REFINE-PAR} \\
\frac{\sigma [d] = / \quad \ell, \ell + 1, \ell + 2 \notin \text{dom}(\sigma)}{\sigma' = \sigma [d \mapsto \ell, \ell \mapsto t_{||}, \ell + 1, \ell + 2 \mapsto /]} \\
\sigma, \text{refine}(d, [t_{||}, \blacksquare, \blacksquare]) \rightarrow \sigma', (\ell + 1, \ell + 2)
\end{array}
\qquad
\begin{array}{c}
\text{REFINE-SEQ-RIGHT} \\
\frac{\sigma [d] = / \quad \ell, \ell + 1, \ell + 2 \notin \text{dom}(\sigma)}{\sigma' = \sigma [d \mapsto \ell, \ell \mapsto t_{\text{seq}}, \ell + 1 \mapsto v_1, \ell + 2 \mapsto /]} \\
\sigma, \text{refine}(d, [t_{\text{seq}}, v_1, \blacksquare]) \rightarrow \sigma', (\ell + 2)
\end{array}$$

$$\begin{array}{c}
\text{REFINE-SEQ-LEFT} \\
\frac{\sigma [d] = / \quad \ell, \ell + 1, \ell + 2 \notin \text{dom}(\sigma)}{\sigma' = \sigma [d \mapsto \ell, \ell \mapsto t_{\text{seq}}, \ell + 1 \mapsto / , \ell + 2 \mapsto v_2]} \\
\sigma, \text{refine}(d, [t_{\text{seq}}, \blacksquare, v_2]) \rightarrow \sigma', (\ell + 1)
\end{array}
\qquad
\begin{array}{c}
\text{FILL} \\
\frac{\sigma [d] = /}{\sigma, \text{refine}(d, v) \xrightarrow{\varepsilon} \sigma [d \mapsto v], ()}
\end{array}$$

Fig. 20. Sequential semantics for **refine**

$$\begin{array}{c}
\text{CHAIN-UNRESOLVED} \\
\frac{\text{Fut}[fut] = \mathfrak{U} d' \quad C \neq \square \quad tid' \text{ fresh}}{\sigma, \text{Task} \sqcup (tid \mapsto C[\text{forward}(fut, d)]), \text{Fut} \xrightarrow{\varepsilon} \sigma, \text{Task} \sqcup (tid \mapsto C[()]) \sqcup (tid' \mapsto \text{forward}(fut, d)), \text{Fut}}
\end{array}$$

$$\begin{array}{c}
\text{CHAIN-RESOLVED} \\
\frac{\text{Fut}[fut] = \mathfrak{R} v}{\sigma, \text{Task} \sqcup (tid \mapsto \mathbf{C}[\text{forward}(fut, d)]), \text{Fut} \rightarrow \sigma[d \mapsto v], \text{Task} \sqcup (tid \mapsto C[()]), \text{Fut}}
\end{array}$$

$$\begin{array}{c}
\text{DPS-CALL} \\
\frac{(f \mapsto \lambda_{\text{dps}} \delta, x. e) \in \text{Fun} \quad fut', d, tid' \text{ fresh}}{\sigma, \text{Task} \sqcup (tid \mapsto C[(f v)]), \text{Fut} \rightarrow \sigma[d \mapsto /], \text{Task} \sqcup (tid \mapsto C[fut']) \sqcup (tid' \mapsto e[\delta \mapsto d][x \mapsto v]), \text{Fut}[fut' \mapsto \mathfrak{U} d]}
\end{array}$$

$$\begin{array}{c}
\text{TAIL-CALL} \\
\frac{(f \mapsto \lambda_{\text{dps}} \delta, x. e) \in \text{Fun} \quad tid' \text{ fresh}}{\sigma, \text{Task} \sqcup (tid \mapsto \mathbf{C}[(f \#_t (d, v))]), \text{Fut} \rightarrow \sigma, \text{Task} \sqcup (tid \mapsto C[()]) \sqcup (tid' \mapsto e[x \mapsto v][\delta \mapsto d]), \text{Fut}}
\end{array}$$

$$\begin{array}{c}
\text{FUTURE-RESOLVE} \\
\frac{\text{Fut}[fut] = \mathfrak{U} d \quad \text{holes}(d) = 0 \quad \sigma [d] = v}{\sigma, \text{Task}, \text{Fut} \xrightarrow{\varepsilon} \sigma \setminus d, \text{Task}, \text{Fut}[fut \mapsto \mathfrak{R} v]}
\end{array}
\qquad
\begin{array}{c}
\text{PRUNE-TASKS} \\
\frac{}{\sigma, \text{Task} \sqcup (tid \mapsto ()), \text{Fut} \rightarrow \sigma, \text{Task}, \text{Fut}}
\end{array}$$

Fig. 21. Global Target reduction –  $\sigma, \text{Task}, \text{Fut} \rightarrow \sigma', \text{Task}', \text{Fut}'$  – Extends Fig. 12. Greyed out rules are similar to `tma`, with the modified parts highlighted in **viola**.

tag and one hole (**REFINE-SEQ-LEFT** or **REFINE-SEQ-RIGHT**), or a value (**FILL**). Except for that last case, **refine** performs the allocation when every field of the block is either a value or a hole—setting the location to `/` in that case—, sets the destiny argument to the corresponding value, and returns all locations initialized at `/`.

*Concurrent Semantics.* To define the target concurrent semantics with constructors, we extend the one defined for Tail-Modulo-Await, but with additional contexts. The language constructs that previously appeared exclusively in tail-position (and therefore could only be reduced under empty evaluation context), can now appear under context: before a sequence, when evaluating asynchronous fields of transformed constructor allocations. Therefore, we add contexts in the reduction rules for these constructs (namely **forward** and  $(\cdot \#_t \cdot)$ )—for clarity, we write the rule in grey and highlight the new additions in **viola**. Furthermore, these rules cannot kill the current

task any more, and must instead return (). Instead, the new rule **PRUNE-TASKS** kills the threads which “have nothing left to do”. Similarly, since **forward** can now appear under non-empty context, it shouldn’t block its thread evaluation if the future to forward is still unresolved. Rule **CHAIN-UNRESOLVED** delegates the forward to a dedicated task, until it can be evaluated once the future is resolved.

Filling a destiny does not necessarily resolve a future immediately. The rule **FUTURE-RESOLVE** sets a future to  $\mathfrak{R}$  in *Fut* when it has no holes left; it also removes its associated destiny from the store. This last removal is not necessary, but simplifies proofs by keeping stores consistent between source and target configurations (these locations do not exist in the source).

### 3.3 Transformation

Finally, Fig. 22 describes the Tail-Modulo-Cons-Await transformation, extending Tail-Modulo-Await with constructors. It uses the *sta* function (see Fig. 18) to determine which positions end in **awaits** or not, and re-order the expressions accordingly. The bulk of the constructor is always replaced by an allocation and a **refine** to obtain the appropriate destinies. Note that, in the  $t_{\text{seq}}$  case, the transformation is not recursively called on sub-expressions as there is only one tail-position.

$$\begin{aligned}
 \mathcal{D}[[e]]_{\delta} &::= \dots \\
 \mathcal{D}[[\{ t_{||}, e_1, e_2 \}]]_{\delta} &::= \text{let } \delta_1, \delta_2 = \text{refine}(\delta, [ t_{||}, \blacksquare, \blacksquare ]) \text{ in } && \text{(when } \text{sta}(e_2) \text{ defined,} \\
 & \mathcal{D}[[e_2]]_{\delta_2}; \mathcal{D}[[e_1]]_{\delta_1} && \text{and } \text{sta}(e_1) \text{ undefined)} \\
 \mathcal{D}[[\{ t_{||}, e_1, e_2 \}]]_{\delta} &::= \text{let } \delta_1, \delta_2 = \text{refine}(\delta, [ t_{||}, \blacksquare, \blacksquare ]) \text{ in } && \text{(otherwise)} \\
 & \mathcal{D}[[e_1]]_{\delta_1}; \mathcal{D}[[e_2]]_{\delta_2} \\
 \mathcal{D}[[\{ t_{\text{seq}}, e_1, e_2 \}]]_{\delta} &::= \text{let } \delta_1 = \text{refine}(\delta, [ t_{\text{seq}}, \blacksquare, e_2 ]) \text{ in } && \text{(when } \text{sta}(e_1) \text{ defined)} \\
 & \mathcal{D}[[e_1]]_{\delta_1} \\
 \mathcal{D}[[\{ t_{\text{seq}}, e_1, e_2 \}]]_{\delta} &::= \text{let } \delta_2 = \text{refine}(\delta, [ t_{\text{seq}}, e_1, \blacksquare ]) \text{ in } && \text{(when } \text{sta}(e_1) \text{ undefined)} \\
 & \mathcal{D}[[e_2]]_{\delta_2}
 \end{aligned}$$

Fig. 22. The Tail-Modulo-Cons-Await transformation –  $\mathcal{D}[[\cdot]]$  – Extends Fig. 13

### 3.4 Correctness of *tmca*

We extend the proof of correctness introduced in Section 2.5 for *tma*. The simulation relation essentially extends the previous one. However, the amount of administrative details we need to track of gets hairy: we therefore refer the interested reader to the Appendix C for details, and favour here a graphical representation to convey the intuition of the proof.

The management of the stuttering aspect of the backward simulation is slightly more involved than in the *tma* case: the **FILL** reduction step can be stuttering in some contexts, but not all. The identification of  $\varepsilon$  steps is therefore now contextual, rather than purely hardcoded in the semantics.

**3.4.1 Definition of  $\succ$ .** Figs. 23 and 24 introduce graphical representations for respectively source and target configurations in relation: we take this informal graphical depiction as definition for  $\succ$ . As for *tma*, the *main* thread is identical. However, because we allocate constructors in the store at different points in execution, and also because destinies are now actual memory locations, the stores cannot be synchronised: nonetheless, the store for a source configuration is always included in the target store ( $\sigma$ ), which we single out on the left of the representation.

The remaining threads of the configurations are still divided in matching pairs of pools, but with additional structure, represented as boxes. Each box is associated to exactly one future. When, in



Fig. 23. Source configurations

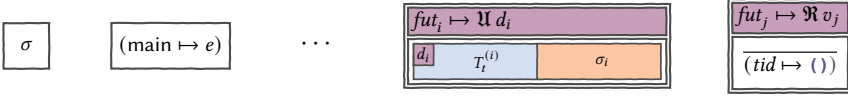
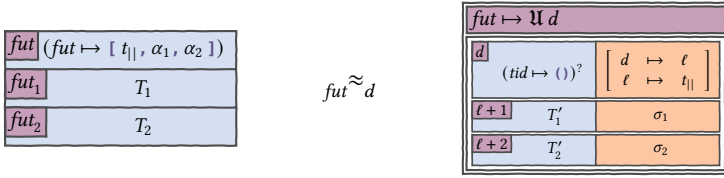


Fig. 24. Target configurations



$$T_i \text{ fut}_i \approx_{l+i} \sigma_i, T'_i, i \in \{1, 2\}$$

Fig. 25. The last case of the relation for parallel constructors

the target, this future  $fut_i$  maps to an unresolved destiny  $d_i$ , it owns the share of store  $\sigma_i$ , which holds the tree of constructors “in construction” on which the resolution of  $d_i$  depends. As in `tma`, the source and target threads pools are related via a relation  $T_s^{(i)}$   $fut_i \approx d_i \sigma_i, T_t^{(i)}$ , annotated with the future and destiny being resolved by the box. Otherwise, if the future is resolved (case  $fut_j$ ), the box carries the remaining set of lingering anonymous tasks on the target side, and resolved futures on the source side.

Recall the tree-like memory layout at runtime presented in Fig. 2b. This structure, internal to a “box”, therefore needs to be captured in  $\approx$ . It therefore reflects two aspects of the current state of the structure in memory. First, at each node, it recursively ensures that the subtrees are related. Furthermore, it must keep track of the status of the allocation of a given node as one of three successive states: whether none, one, or two subtasks have been launched yet.<sup>4</sup> By identifying this last case, we can identify the operation in the target filling the final hole, and against which the source program can perform the corresponding allocations, resynchronising both configurations.

Fig. 25 illustrates the case of a *parallel* constructor with both fields asynchronous, in the situation where both subtasks have already been launched. On the source side, we have  $\alpha_i = \text{await}(fut_i)$  if  $fut_i$  is not resolved yet, and  $\alpha_i = v_i$  where  $\sigma_i(l+i) = v_i$  otherwise.

**3.4.2 Proof sketch.** First, observe that similarly to the `tma` case, the  $\approx$  relation allows us to frame out part of the configurations when proving the backward simulation. We can therefore strengthen the statement of simulation we establish, specifying the shape of the configurations we reach.

Consider  $c_s = \begin{matrix} \sigma_s, \\ Task_s \sqcup T_s \end{matrix} \succ \begin{matrix} \sigma_t \sqcup s_t, \\ Task_t \sqcup T_t, \\ Fut_t \end{matrix} = c_t$ , s.t.  $T_s \text{ fut} \approx_d s_t, T_t$ , then:

<sup>4</sup>In the case of sequential constructors, at most one subtask is launched.

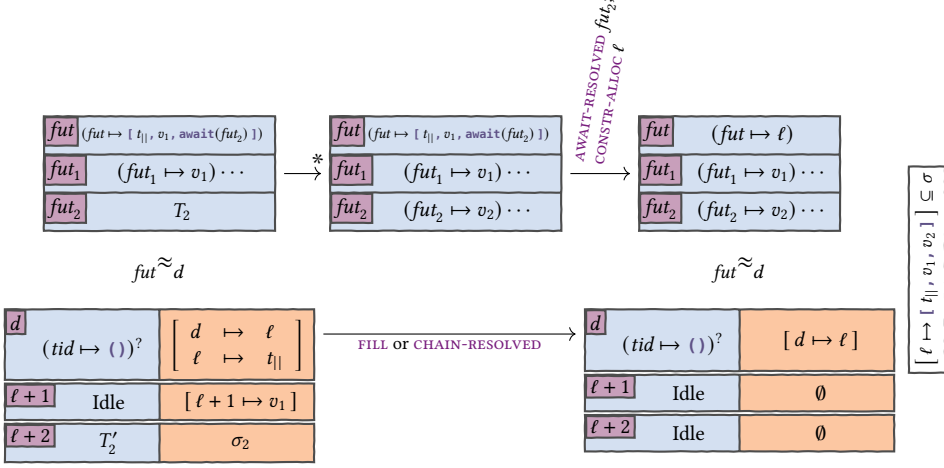


Fig. 26. Sub-configurations before a synchronisation step. Idle denotes a possibly empty list of threads all reduced to ()

LEMMA 3.1. *If we have*

$$c_t \rightarrow \begin{array}{l} \sigma'_t \sqcup s'_t, \\ Task_t \sqcup T'_t (\sqcup T_t^{\text{new}})^?, \\ Fut_t (\sqcup F_t^{\text{new}})^? \end{array}$$

*then there exist  $T'_s, \sigma'_s$  (and maybe  $T_s^{\text{new}}$ ) such that*

$$c_s \rightarrow^* \begin{array}{l} \sigma'_s, \\ Task_s \sqcup T'_s (\sqcup T_s^{\text{new}})^? \end{array} \quad \text{and } T'_s \text{ fut} \approx_d s'_t, T'_t$$

Moreover, we can characterise more finely  $T'_s$  in the case where we finish computing the tasks associated with  $d$ . Indeed, if in  $c_t$  we have  $\text{holes}(d) = 1$ , and if the step under consideration in the target fills this last hole in the subtree of  $d$ , then  $T'_s = (fut \mapsto v) \sqcup (\overline{fut_i \mapsto v_i})$ , with  $v = s'_t(d)$ . This situation arises each time a destiny is introduced, but most importantly at the end of the evaluation of constructors. In the target, as soon as all values are filled, we are done with evaluating the constructor, and its future can be resolved and awaited. Therefore, the last step filling a destiny needs to be matched, in the source, by all allocations and administrative steps required to get to a configuration where the block can be allocated, and finally performing said allocation.

This lemma relies in particular on the fact that for related configuration, for any destiny  $d$  such that  $\text{holes}(d) = 0$  associated to related pools  $T_s \text{ fut} \approx_d \sigma_t T_t$ , we have that  $fut$  is resolved and  $T_s \supseteq (fut \mapsto v)$  where  $v = \sigma_t(d)$ .

*Backward simulation: illustrative case.* Finally, we illustrate the case from Fig. 25 in the proof of lemma 3.1 conducted by induction on the hypothesis  $T_s \text{ fut} \approx_d s_t, T_t$ .

Fig. 26 depicts the situation, where the target is at the bottom. The step it takes fills the last location under destiny  $d$ . We know that  $\text{holes}(d) = 1$ , with said hole necessarily in either the left or right sub-tree. Let's assume it is in the right sub-tree, rooted at  $\ell + 2$ , with the other one completely resolved. By invariant, we have  $\alpha_1 = v_1$ , where  $v_1 = \sigma_t[\ell + 1]$  and  $T_2 \text{ fut}_2 \approx_{\ell+2} \sigma_2, T'_2$ , with  $\text{holes}(\ell + 2) = 1$ . By induction hypothesis, the step can be matched in the source, until  $fut_2$  is resolved, to value  $v_2$  with  $v_2$  the value at  $\ell + 2$  in the target configuration, represented on Fig. 26 by the first sequence of transition at the top of the figure.

At this point, the `await(fut2)` remaining in the constructor can be resolved, and the block allocated. As this block is now allocated in the store on the source side, it is already accounted for in the “synchronised” logical view of the target store, and the corresponding mappings are removed from the  $\sigma_i$  sub-parts.

We finally reach a “value” state of the  $\approx$  relation: all source threads are resolved futures, all target thread idle (read reduced to `()` or already pruned), with a single mapping from the current destiny to its value in the sub-part of the store; which satisfies the lemma, and the invariant.

## 4 Practical considerations

In this section, we showcase an implementation and evaluation of our code transformation. We first describe a refined version of Tail-Modulo-Cons-Await which avoids creating unnecessary destinies. We then describe our prototype OCaml implementation, and evaluate it in practice.

### 4.1 Optimized Code Transformation

The `tmca` code transformation proposed in Section 3.3 is quite inefficient in the presence of nested constructors. This inefficiency, already noted by Allain et al. [4], is best demonstrated on an example. Let us consider the source code below on the left, which builds a value using constructors  $A_{||}$  and  $B_{||}$  (both in a parallel fashion). It will result in the target code on the right.

$$\{ A_{||}, \{ B_{||}, \text{await}(e_1), e_2 \}, \text{await}(e_3) \} \Rightarrow \begin{array}{l} \text{let } \delta_l, \delta_r = \text{refine}(\delta, [ A_{||}, \blacksquare, \blacksquare ]) \text{ in} \\ \text{forward}(e_3, \delta_r); \\ \text{let } \delta_{ll}, \delta_{lr} = \text{refine}(\delta_l, [ B_{||}, \blacksquare, \blacksquare ]) \text{ in} \\ \text{forward}(e_1, \delta_{lr}); \text{refine}(\delta_{ll}, e_2) \end{array}$$

While semantically fine, the destiny  $\delta_l$  is unnecessary: it is immediately filled by the allocated block  $[ B_{||}, \blacksquare, \blacksquare ]$ . As we show in next section, creating a destiny in practice is a fairly costly operation: it requires cooperation from the scheduler, and, in a language like OCaml, induces a write-barrier when filled. It would be better to set the value directly. As such, and following a similar idea from Allain et al. [4], we emit the following code:

$$\begin{array}{l} \text{let } \delta_{ll}, \delta_{lr}, \delta_r = \text{refine}(\delta, [ A_{||}, [ B_{||}, \blacksquare, \blacksquare ], \blacksquare ]) \text{ in} \\ \text{forward}(e_3, \delta_r); \text{forward}(e_1, \delta_{lr}); \text{refine}(\delta_{ll}, e_2) \end{array}$$

This new code builds a single more complex “value with holes”, and has only one call to `refine`. It nonetheless preserves the execution order of sub-expressions  $e_1$ ,  $e_2$  and  $e_3$  induced by the naive transformation.

The detailed transformation is shown in Fig. 27. Only the base case of the  $\mathcal{D}[\cdot]_\delta$  changes, by calling a collecting function,  $\mathcal{D}_{\text{cons}}[\cdot]$ , which accumulates two things: a list of expressions to be evaluated (either synchronously or asynchronously), and a “value with holes”, made of constructors and constants. This transformation ensures that only one call to `refine` is done per tail-position. Except for this optimisation, our prototype implementation closely follows the formalisation we presented in the previous sections.

### 4.2 Proof-of-concept Implementation

We provide a proof-of-concept implementation available at <https://zenodo.org/records/15757165> (and will be available as free software online for publication). The syntactic transformation is implemented as a PPX syntax extension on OCaml’s syntax. This is not a battle-ready implementation, which should be implemented directly inside a compiler, but is sufficient to test our transformation on real-world examples. The runtime is implemented as a thin layer atop an existing

OCaml concurrency library.<sup>5</sup> An overview of the runtime API is provided on Figs. 28 and 30, and the actual translation of `map` on trees on Fig. 29.

<sup>5</sup>Most existing library would satisfy our needs; we use Picos (<https://github.com/ocaml-multicore/picos>) for its clear semantics and Moonpool (<https://github.com/c-cube/moonpool/>) for its efficient scheduler.

$$\begin{aligned}
\mathcal{D}[[C_n^{tl}[e_1 \mid \dots \mid e_n]]]_\delta &::= C_n^{tl}[\mathcal{D}[[e_1]]_\delta \mid \dots \mid \mathcal{D}[[e_n]]_\delta] \\
\mathcal{D}[[\text{await}(e)]]_\delta &::= \mathcal{D}_a[[e]]_\delta \\
\mathcal{D}[[e]]_\delta &::= \frac{\text{let } \overline{\delta}_k = \text{refine}(\delta, c) \text{ in}}{\mathcal{D}[[e_k]]_{\delta_k}; \quad \forall k. \text{sta}(e_k) \text{ defined} \quad \text{where } \overline{e}_k, c = \mathcal{D}_{\text{cons}}[[e]]} \\
&\quad \mathcal{D}[[e_k]]_{\delta_k}; \quad \forall k. \text{sta}(e_k) \text{ not defined} \\
\frac{\overline{e}_i^i, c = \mathcal{D}_{\text{cons}}[[e]} \quad \overline{e}_j^j, c' = \mathcal{D}_{\text{cons}}[[e]]}{\mathcal{D}_{\text{cons}}[[\{t_{||}, e, e'\}]] &::= \overline{e}_i^i; \overline{e}_j^j, \{t_{||}, c, c'\}} \\
\mathcal{D}_{\text{cons}}[[c]] &::= \emptyset, c \quad \mathcal{D}_{\text{cons}}[[x]] &::= \emptyset, x \quad \mathcal{D}_{\text{cons}}[[e]] &::= e, \blacksquare
\end{aligned}$$

Fig. 27. Optimized Tail-Modulo-Cons-Await transformation for nested constructors –  $\mathcal{D}[[\cdot]]$  – Extends Fig. 22

```

1 module Loc : sig
2 (** Location inside an OCaml memory block,
3   composed of a value and an offset. *)
4 type 'h t = { pointer : Obj.t ; offset : int }
5
6 (** An heterogeneous list of locations. *)
7 type 'a list =
8   | [] : unit list
9   | (::) : 'h t * 'h2 list -> ('h * 'h2) list
10
11 (** A dummy value to insert into blocks. *)
12 val dummy : 'a
13
14 (** [Loc.mk o i] is the location in value [o]
15   at offset [i]. [o] must be a pointer to a
16   block. *)
17 val mk : 'a -> int -> 'b t
18 end

```

Fig. 28. Runtime library for locations

```

1 let rec map$dps d4 f t = match t with
2 | Leaf -> Destiny.refine d4 Leaf []
3 | Node (v, tl, tr) ->
4   let v5 = Node(Loc.dummy, Loc.dummy, Loc.dummy) in
5   let [d6; d7; d8] =
6     Destiny.refine d4 v5
7     [Loc.mk v5 0; Loc.mk v5 1; Loc.mk v5 2]
8   in
9   fork (fun () -> map$dps d8 f tr);
10  fork (fun () -> map$dps d7 f tl);
11  Destiny.refine d6 (f v) []
12
13 let map f l =
14   let d, c = Destiny.mk () in
15   fork (fun () -> map$dps d f l);
16   c

```

Fig. 29. Real translation of parallel map from Fig. 1 with an `asynpar` annotation

```

1 (** Launch a new task. *)
2 val fork : (unit -> unit) -> unit
3
4 module Destiny : sig
5 (** A Destiny with a hole of type ['a]. *)
6 type 'a t = {
7   counter : int Atomic.t; (* Number of holes *)
8   resolve : unit -> unit; (* Top-level resolve *)
9   set : 'a -> unit; (* Set the current destiny *)
10 }
11
12 (** An heterogeneous list of destinies. *)
13 type 'a list =
14   | [] : unit list
15   | (::) : 'h t * 'h2 list -> ('h * 'h2) list
16
17 (** [mk ()] returns a synchronised destiny and
18   future. The future will be fulfilled only
19   when all holes in the destiny are filled. *)
20 val mk : unit -> 'a t * 'a Future.t
21
22 (** [refine d v hs] fills the destiny [d] with a
23   value [v] with holes [hs]. It returns a list
24   of destinies corresponding to each hole.
25   [d] should not be used again. *)
26 val refine : 'b t -> 'b -> 'a Loc.list -> 'a list
27
28 (** [forward c d] will fill the destiny [d] with
29   the value of the asynchronous computation [c]
30   when it is finished. *)
31 val forward : 'a Future.t -> 'a t -> unit
32 end

```

Fig. 30. Runtime library for destinies

The main difference between our implementation and the formalism presented so far is that hole creation, memory locations and destinies were previously fused into the near-magical `refine` operation. These operations are now split to allow finer control. To pre-allocate a constructor, we first create a value from a constructor, with holes implemented by a dummy value `Loc.dummy`. We then take locations for each of its holes, using the `Loc.mk` function, and gather them in an heterogeneous `Loc.list`. We then use `Destiny.refine` with the explicit list of locations to obtain the list of destinies (as a `Destiny.list`). This is demonstrated in Line 4 to 7 in Fig. 29.

The implementation of `Loc` relies on details of the OCaml memory model and some unsafe internal API (`Obj`). `Destiny`, in turn, only uses regular OCaml with concurrency. Destinies are a triplet composed of a set to a location, a reference to a shared atomic counter representing the current number of holes in the full structure, and a `resolve` function which marks the top-level future as resolved. The functions `forward` and `refine` decrement and increment the counter when holes are added or removed from the structure. The key trick here is that, as soon as the atomic counter hits zero, there are no more holes in the structure and the counter can never rise up again. We can then resolve the top-level future immediately.

To implement DPS calls (Line 9, 10 and 15 in Fig. 29), we use a regular `fork` function which takes a callback containing the actual call. The call is syntactically a tail-call, ensuring that the OCaml's compiler uses the desired call convention. Finally, for the purpose of this implementation we consider all constructors to be parallel: indeed, as described in Section 3, sequential constructors can be emulated by let-bindings. The rest of the implementation follows our formalism closely.

### 4.3 Experimental Evaluation

We now evaluate our full code transformation experimentally. Our evaluation setup is an Intel Xeon E5-2630 equipped with 12 cores and 32GiB of memory. All benchmarks are run until obtaining a 99% confidence interval. We focus on evaluation using synthetic benchmarks on trees with results ranging over Figs. 32 to 35.

*Example codes.* We consider 4 main variants of a `map` over binary trees, mapping a function `f`:

- (1) synchronous with `tmc` enabled (`map_tmc`, in black);
- (2) asynchronous that sequentializes the recursive calls (`map_async`, in red);
- (3) running in parallel both `f` and the two recursive calls (`map_asyncpar_full`, in green);
- (4) running `f` first, and then the recursive calls in parallel (`map_asyncpar_width`,<sup>6</sup> in blue).

```

1 let%async rec map_asyncpar_width_tmca f t =
2   match t with
3   | L -> L
4   | N (v , tl , tr ) ->
5     let y = f v in
6     N (y,
7       await (map_asyncpar_width_tmca f tl),
8       await (map_asyncpar_width_tmca f tr))

```

Fig. 31. `map_asyncpar_width`

Furthermore, for variants (2)-(4), we consider two versions in each case, hence adding up to 7 variants in our benchmarks. We consider as baseline an implementation hand-written by a programmer familiar with asynchronous programming. We compare them against the versions generated by our optimisation from a simple code, suffixing their respective names by `_tmca`. The latter are written in the style of Fig. 15a, using parameter `async` for (2), and `asyncpar` for (3) and (4). The `map_async` and `map_asyncpar_full` implementations correspond to the semantics respectively presented in Fig. 15b and Fig. 15c; the third implementation, for `map_asyncpar_width_tmca`, is presented in Fig. 31.

<sup>6</sup>The suffix indicates that in this case, we limit the parallelism in order to lower the contention on the scheduler.

		tmc	Peak Mem. (MB)	Allocated Mem. (MB)	Peak Stack (MB)	Peak Total (MB)
asynpar	width	manual	1.2	124.6	1.95	3.15
		tmca	1.0	107.3	1.33	2.33
	full	manual	1.5	24.4	1.63	3.13
		tmca	0.6	20	$5.31e^{-4}$	0.6
	full	manual	1.5	25	1.63	3.7
		tmca	0.6	21.9	$1.48e^{-3}$	0.6

Fig. 32. Memory usage on a tree of  $10^5$  elements (i.e., a tree occupying 0.6 MB in memory)

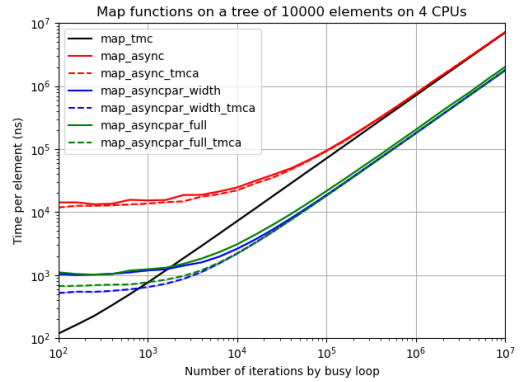


Fig. 33. Impact of the computation per element

For each of these variants, we map a function consisting of a busy loop with a variable number of iterations. Our benchmarks are run on a randomly generated tree with a width bounded by 40. Bounding the width of the tree allows us to explore both parallelisation—that grows with the width of the tree—and stack management—that grows with long branches in the tree.

Manually written codes are plotted in contiguous lines, while tmca optimised ones are dashed.

*Memory consumption.* Fig. 32 shows the memory footprint measured when mapping over a tree of  $10^5$  elements using a “random” scheduler running on 4 cores. We display the maximum live memory reported by the GC (*Peak Mem.*), the total amount allocated by the GC during execution (*Allocated Mem.*), and the maximum sum of the sizes of the stacks in all the currently live fibers reached (*Peak Stack*). The final column shows the sum of the first and third columns (*Peak Total*). Stack measurements are obtained by instrumenting a scheduler to collect data on all fibers that are currently either suspended or in execution.

First, we stress that *the objective is reached*: both tmca with asynpar variants eliminate the stack consumption observed in their manual variants—it actually does so even more than tmc. Indeed, tmca can treat both recursive calls as tail recursive. Naturally, by creating threads comes a memory cost. However, most of this memory is short lived, so that the peak of the parallel versions is very close to the one of tmc.<sup>7</sup> Comparing map\_asynpar\_width\_tmca with map\_asynpar\_full\_tmca, the fully parallel version uses slightly more stack and more memory, which is consistent with the increased number of concurrent tasks. The asynchronous case suffers from the worst of both worlds: it has a significant stack size, as many task stay alive for a while before they can be continued, even in the tmca case. We believe a less naive scheduling policy could improve stack usage, notably for map\_async\_tmca.

Overall, both map\_asynpar\_\*\_tmca variants are remarkably lean, with a memory consumption barely above the tmc version, while opening the way to parallel execution, as we will observe now.

*Time measurements.* We now present the performances of the different implementations in Figs. 33 and 35. In particular, we consider how performance scales with three metrics: the amount of work done per element, the size of the tree, and the parallelism factor. The first graph varies the quantity of work per element for a randomly generated tree of 10 000 elements, while the

<sup>7</sup>The GC reports for tmc a peak lower than the size of the tree itself, which is likely an artifact of a measuring error: 0.6MB is the theoretical minimum here.

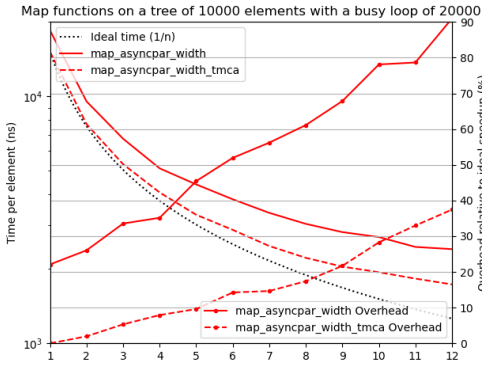


Fig. 34. Parallel scaling

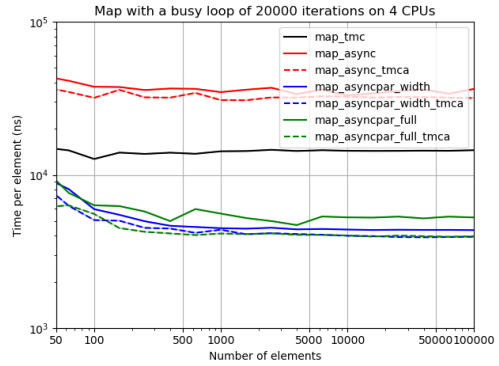


Fig. 35. Impact of the number of elements

second graph varies the number of elements, both ran with four CPUs. All graphs show the time per element in nanosecond (lower is better), and use log/log scales. All experiments are run with a reasonably optimized work-stealing scheduler.<sup>8</sup>

The first key observation for all graphs is that *dotted lines are below solid lines*, meaning that the optimisations brought by our transformation leads to better performances than manually written code. This reflects that using DPS allows us to reduce the number of synchronisations and optimises memory accesses. As shown in Fig. 33, parallelisation (green and blue lines) induces a speed up factor of four (the number of CPUs) when tasks are big enough. Asymptotically we also notice that `map_async` and `tmc` become similar: the performance gain brought by `tmc` and the additional synchronisation cost from `map_async` both get negligible compared to the computation time. On the contrary, when the work load is small, synchronisation between asynchronous processes becomes noticeable. This is why the `tmc` version is significantly faster when no computation is performed—notice here the cutoff around tasks of roughly 1000ns, which is quite small. With a busy loop that has between  $10^3$  and  $10^4$  iterations, synchronisation and computation have similar cost and parallel benchmarks get faster than the sequential ones.

Thus, we have considered a loop size of 20 000 iterations and studied the impact of the size of the tree in Fig. 35. This graph illustrates the fact that asymptotically the cost of synchronisation in the sequentialized asynchronous cases is still significant: for such tasks, the `map_async` versions are twice slower than the `tmc` one. Indeed, we perform here a light computation per element, balancing out the computation time and the synchronisation time. On parallel versions, we again observe the impact of parallelisation: our maps run 4 times faster than the `tmc` version. Indeed the parallel `tmca` versions do not incur any synchronisation cost. Only the sequential nature of `map_async` imposes a synchronisation cost. On small and average sizes, measures are a bit less stable but we observe that the `tmca` optimisation improves performances (dotted lines are significantly below solid ones).

Finally, Fig. 34 shows the parallel scaling of `map_asyncpar_full` with and without `tmca` optimisation. The left axis shows the time per element while the right axis gives the relative cost compared to an ideal speedup computed by dividing the time necessary with 1 CPU (on `map_asyncpar_full_tmca`) by the number of CPUs. Again we notice that `tmca` is always faster than manually written parallel code and that we stay relatively close to the ideal speedup. With 12 CPUs, `map_asyncpar_full_tmca` is less than 40% slower than the ideal speedup, while the non `tmca` version goes up to 90% slower.

<sup>8</sup>More precisely: [https://c-cube.github.io/moonpool/moonpool/Moonpool/Ws\\_pool/index.html](https://c-cube.github.io/moonpool/moonpool/Moonpool/Ws_pool/index.html)

To conclude, our experimental evaluation shows that `tmca` definitely achieves the promised gain for `asynctpar` functions: they consume no stack, barely any more memory than sequential versions, provide automatic parallelism close to the ideal speedup and improve performances over the manually written versions. For non parallel versions, `tmca` does improve performance, but provides less memory gains. We believe this can be improved by fine-tuning the scheduler, but consider this out of scope of this paper. Naturally, our transformation is only really useful when the task per element is non-trivial compared to the synchronisation cost. The cutoff point highly depends on the scheduler, but is already promisingly small, around 1000ns/task, with the off-the-shelf scheduler we have considered—which is efficient, but not highly tuned for maximum performance.

## 5 Related Work and Perspective

The use of continuation for implementing or optimising structured concurrent programming is well-studied, in contexts such as OpenMP or Habanero-Java [3, 5, 16, 17] notably. Our implementation relies on the parallelism based on effect handlers introduced in OCaml 5, but our approach can be adapted to other paradigms for parallel programming. Furthermore, our work focuses on the efficient implementation of asynchronous tasks, and in particular on extending the notion of futures and promises. We hence focus our related work on contributions that target tail-recursion and asynchronous programming paradigms.

### 5.1 Tail recursion

The Tail-Modulo-Cons transformation is as old as functional programming, seeing birth in the LISP community in the early 70th. During this decade, two academic publications describe the transformation, first Risch for the REMREC system [21], then Friedman and Wise [13] over a pure LISP with cons compiled down to a machine language.

After having remained folklore for half a century, the transformation has seen revived interest in the community through essentially two independent pieces of work. Leijen and Lorenzen [19] have implemented `tmc` for Koka, a strongly typed language with effect types and handlers. In doing so, they have rephrased the transformation in an equational style, generalised it to work with an abstract notion of *contexts*, and adapted it to the linear typing discipline followed by Koka. Independently, Allain et al. [4] have also implemented `tmc`, but in the OCaml compiler. In doing so, they have formalised the optimisation over a core calculus from which we took inspiration for designing our own, and have furthermore mechanised its proof of soundness using a variant of Simuliris [14], a simulation technique based on separation logic.

### 5.2 Asynchronous Programming

*Future and Promises.* Futures are limited in the sense that they must be fulfilled by the task that was associated with their creation. This limitation guarantees that futures are eventually fulfilled. On the contrary, promises [20] tie their fulfilment to a handler given to the programmer. Hence, they do not suffer from this limitation, but it cannot be enforced that a promise is resolved exactly once [1]. Observing that promises allow for optimising the scheduler space compared to futures was already the key idea behind the `forward` construct [10]. In this work, `forward` promises are only used internally for optimisation purposes, much as we do in the present work. Indeed, the forward optimisation is only valid when applied to an asynchronous call in tail position. The present article can therefore be seen as an extension of [10] with more automation, added parallelism, and a broader application setting.

Dataflow synchronisation on futures [11] is a paradigm that makes it impossible to observe chains of futures: chains are automatically traversed upon each synchronisation (synchronisation

is insensible to successive asynchronous tail calls). Chappe et al. [8] showed that with dataflow futures, it is safe to optimise tail-calls with promises, and that `return` has the same semantics as `forward` in this setting. This is another setting that makes it possible to optimise scheduler space for futures (through promises as well) and in which the `tmaa` approach should be applicable to broaden the setting. Unfortunately dataflow synchronisation on futures implies a type system where 'a future unifies with ('a future) future. Such typing rules, where the monadic join is an axiom, are particularly difficult to integrate in ML-like languages such as OCaml.

*Other Asynchronous Paradigms.* Futures are massively used in actor and active object languages [2, 9]: this family of languages thus offers a privileged application domain for all optimisations of asynchronous calls, such as `tmca`. Indeed, every call to an actor creates a future, which is often done in tail-position, making future optimisation crucial in these languages. Notably, `forward` was created in an active object language, and the present work would benefit languages with prominent Actors libraries [2, 15, 18], including the recently designed OCaml active object library [6].

Our parallel-tagged constructors could also be compared to the `parT` construct of the Encore language [12], or the `par` construct of Haskell [24, 25]. Both combinators generate parallelism and allow the coordination of parallel tasks. Rather, our vision here is to create parallelism automatically when using data constructors. We believe the integration of data production and parallelism fit well with the notion of futures and promises, but approaches based on parallel combinators feature richer synchronisation patterns compared to our work. Investigating the integration of richer coordination patterns in our context, which could for example enable some form of pipelining when performing two asynchronous maps in a row (even over complex data structures), is left to further work.

### 5.3 Application to Other Programming Languages

In Section 4, we have presented an implementation in OCaml of `tmca`, the optimization we introduced in Section 3. However, `tmca` is not specifically tied to OCaml in any way: the transformation aims at being applicable to most languages where tail recursive functions can be the target of optimisations. In this section, we review more specifically the requirements `tmca` relies on.

First, `tma` and `tmca` depend internally on the presence of promises. Externally, the transformations are designed to apply to asynchronous function calls based on futures. As futures can be implemented using promises, any language that provides promises, or at least a mean to implement them, should therefore qualify. Second, the correctness of `tmca` relies on assumptions about constructor values: the allocated locations for blocks must be inaccessible while partially initialized. Even in a sequential context, this assumption is already required by `tmc` to ensure memory safety.

Overall, our approach is hence theoretically applicable to any language with memory safety and support for parallel computations. However, it should be noted that implementing efficiently promises and data structures with holes, if they are not available, relies on tricky modifications of the execution environment. Thankfully, numerous mainstream languages already provide all the required facilities: notably, languages such as Scala, Haskell, Rust (although tail-call-optimisation is not guaranteed by the compiler so far), or C# do qualify.

### Data-Availability Statement

The full proof-of-concept implementation is available as an artifact on Zenodo (url not provided for anonymisation, an anonymous archive is provided to the reviewers instead) along with the experimental data and instructions for how to reproduce it. This artifact has been submitted for Artifact Evaluation.

## References

- [1] Erika Ábrahám, Immo Grabe, Andreas Grüner, and Martin Steffen. 2009. Behavioral interface description of an object-oriented language with futures and promises. *The Journal of Logic and Algebraic Programming* 78, 7 (2009), 491 – 518. doi:10.1016/j.jlap.2009.01.001 The 19th Nordic Workshop on Programming Theory (NWPT 2007).
- [2] Gul Agha. 1986. *Actors: a model of concurrent computation in distributed systems*. MIT Press.
- [3] Aditya Agrawal and V. Krishna Nandivada. 2023. UWompro: UWomp++ with Point-to-Point Synchronization, Reduction and Schedules. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 27–38. doi:10.1109/PACT58117.2023.00011
- [4] Clément Allain, Frédéric Bour, Basile Clément, François Pottier, and Gabriel Scherer. 2025. Tail Modulo Cons, OCaml, and Relational Separation Logic. *Proceedings of the ACM on Programming Languages* 9, POPL (Jan. 2025), 2337–2363. doi:10.1145/3704915
- [5] Raghesh Aloor and V. Krishna Nandivada. 2019. Efficiency and expressiveness in UW-OpenMP. In *Proceedings of the 28th International Conference on Compiler Construction (Washington, DC, USA) (CC 2019)*. Association for Computing Machinery, New York, NY, USA, 182–192. doi:10.1145/3302516.3307360
- [6] Martin Andrieux, Ludovic Henrio, and Gabriel Radanne. 2024. Active Objects based on Algebraic Effects. In *Active Object Languages: Current Research Trends*. Lecture Notes in Computer Science, Vol. 14360. 3–36. doi:10.1007/978-3-031-51060-1\_1
- [7] Henry. G. Baker Jr. and Carl Hewitt. 1977. The Incremental Garbage Collection of Processes. In *Proc. Symp. on Artificial Intelligence and Programming Languages*. New York, NY, USA, 55–59.
- [8] Nicolas Chappe, Ludovic Henrio, Amaury Maillé, Matthieu Moy, and Hadrien Renaud. 2021. An Optimised Flow for Futures: From Theory to Practice. *CoRR* abs/2107.07298 (2021). arXiv:2107.07298 <https://arxiv.org/abs/2107.07298>
- [9] Frank de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. 2017. A Survey of Active Object Languages. *ACM Comput. Surv.* 50, 5, Article 76 (Oct. 2017), 39 pages. doi:10.1145/3122848
- [10] Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, and Huu-Phuc Vo. 2018. Forward to a Promising Future. In *Conference proceedings COORDINATION 2018*.
- [11] Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen, and Tobias Wrigstad. 2019. Godot: All the Benefits of Implicit and Explicit Futures. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:28. doi:10.4230/LIPIcs.ECOOP.2019.2 Distinguished artefact.
- [12] Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain. 2016. ParT: An Asynchronous Parallel Abstraction for Speculative Pipeline Computations. In *Proc. 18th IFIP WG 6.1 Intl. Conf. on Coordination Models and Languages (COORDINATION 2016)*, Alberto Lluch-Lafuente and José Prouença (Eds.). Vol. 9686. 101–120.
- [13] Daniel P Friedman and David S Wise. 1975. Unwinding stylized recursions into iterations. *Comput. Sci. Dep., Indiana University, Bloomington, IN, Tech. Rep* 19 (1975).
- [14] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.* 6, POPL, Article 28 (Jan. 2022), 31 pages. doi:10.1145/3498689
- [15] Philipp Haller and Martin Odersky. 2009. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science* 410, 2 (2009), 202 – 220. doi:10.1016/j.tcs.2008.09.019 Distributed Computing Techniques.
- [16] Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 84)*, Dale Miller (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 18:1–18:19. doi:10.4230/LIPIcs.FSCD.2017.18
- [17] Shams Imam and Vivek Sarkar. 2014. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 618–643.
- [18] Einar Broch Johnsen and Olaf Owe. 2007. An Asynchronous Communication Model for Distributed Concurrent Objects. *Softw. Syst. Model.* 6, 1 (2007), 39–58. doi:10.1007/s10270-006-0011-2
- [19] Daan Leijen and Anton Lorenzen. 2023. Tail Recursion Modulo Context: An Equational Approach. *Proc. ACM Program. Lang.* 7, POPL, Article 40 (Jan. 2023), 30 pages. doi:10.1145/3571233
- [20] B. Liskov and L. Shrira. 1988. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (Atlanta, Georgia, USA) (PLDI '88)*. Association for Computing Machinery, New York, NY, USA, 260–267. doi:10.1145/53990.54016

- [21] Tore Risch. 1973. REMREC - A program for Automatic Recursion Removal. <https://api.semanticscholar.org/CorpusID:60315470>
- [22] Amir Shaikhha, Andrew W. Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing, FHPC@ICFP 2017, Oxford, UK, September 7, 2017*, Phil Trinder and Cosmin E. Oancea (Eds.). ACM, 12–23. doi:10.1145/3122948.3122949
- [23] Guy L. Steele Jr. 1977. Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, LAMBDA: The Ultimate GOTO. In *Proceedings of the 1977 annual conference, ACM '77, Seattle, Washington, USA, October 16-19, 1977*, James S. Ketchel, Harvey Z. Kriloff, H. Blair Burner, Patricia E. Crockett, Robert G. Herriot, George B. Houston, and Cathy S. Kitto (Eds.). ACM, 153–162. doi:10.1145/800179.810196
- [24] Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. 1996. GUM: A Portable Parallel Implementation of Haskell. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadelphia, Pennsylvania, USA, May 21-24, 1996*, Charles N. Fischer (Ed.). ACM, 79–88. doi:10.1145/231379.231392
- [25] Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. 1998. Algorithms + Strategy = Parallelism. *J. Funct. Program.* 8, 1 (1998), 23–60. doi:10.1017/S0956796897002967

## A Yet Another example: a medley of map

We consider several asynchronous versions of the function `map_async` over lists, mapping a synchronous or asynchronous function over a list, returning an asynchronous computation itself. We first consider in Fig. 36 a version where only the recursive call to `map_async` is asynchronous, with the mapped function synchronous (`map_async_sync : ('a -> 'b) -> 'a list -> 'b list future`). Using Tail-Modulo-Cons-Await, we can transform it into the code in Fig. 36b.

```
1 let%async rec map_async_sync f l =
2   match l with
3   | Nil -> Nil
4   | Cons (x, xs) ->
5     Cons (f x, await (map_async_sync f xs))
```

(a) Source implementation

```
1 let rec map_async_sync_dps d f l =
2   match l with
3   | Nil -> d ◀ Nil
4   | Cons (x, xs) ->
5     let v = Cons (f x, ■) in
6     d ◀ v;
7     fork_dps map_async_sync_dps v.1 f xs
```

(b) Translation using `tmca`

The wrapping code for the initial destiny is identical to Fig. 5

Fig. 36. An asynchronous, sequential, map on lists with `async/await`, mapping a sequential function

When both calls are asynchronous (`map_async : ('a -> 'b future) -> 'a list -> 'b list future`), the function admits two semantics: *sequential* where an application of  $f$  must terminate before the next, or *parallel* which launches all tasks at once. These two semantics are differentiated in our formalisation by the annotation over constructor tags. Let us first consider the sequential version in Fig. 37. The Tail-Modulo-Cons-Await transformation affects this version in the same way as this version in Fig. 36.

```
1 let%async rec map_async_seq f l =
2   match l with
3   | Nil -> Nil
4   | Cons (x, xs) ->
5     let y = await (f x) in
6     Cons (y, await (map_async_seq f xs))
```

(a) Source implementation

```
1 let rec map_async_seq_dps d f l =
2   match l with
3   | Nil -> d ◀ Nil
4   | Cons (x, xs) ->
5     let y = await (f x) in
6     let v = Cons (y, ■) in
7     d ◀ v;
8     fork_dps map_async_seq_dps v.1 f xs
```

(b) Translation using `tmca`

The wrapping code for the initial destiny is identical to Fig. 5

Fig. 37. An asynchronous, sequential, map on lists with `async/await`

Finally, the parallel version is illustrated in Fig. 38. The transformed code in Fig. 38b uses `forward` for the call to  $f$ , rather than `fork_dps`, as  $f$  might be an external asynchronous function without a `dps` version.

```

1 let%async rec map_async_par f l =
2   match l with
3   | Nil -> Nil
4   | Cons (x, xs) ->
5     Cons (await (f x),
6           await (map_async_par f xs))

```

(a) Source implementation

```

1 let rec map_async_par_dps d f l =
2   match l with
3   | Nil -> d ◀ Nil
4   | Cons (x, xs) ->
5     let v = Cons (■, ■) in
6     d ◀ v;
7     let fut = f x in
8     forward v.0 fut;
9     fork_dps map_async_par_dps v.1 f xs

```

(b) Translation using tmca

The wrapping code for the initial destiny is identical to Fig. 5

Fig. 38. An asynchronous, parallel, map on lists with async/await

## B Supporting code for the Experimental Evaluation

```

1 type 'a tree = N of 'a * 'a tree * 'a tree | L
2
3 let[@tail_mod_cons] rec map_tmca f t =
4   match t with
5   | L -> L
6   | N (v, tl, tr) ->
7     let y = f v in
8     let tl' = map_tmca f tl in
9     N (y, tl', map_tmca f tr)
10
11 let rec map_async f t =
12   async @@ fun () -> match t with
13   | L -> L
14   | N (v, tl, tr) ->
15     let y = f v in
16     let tl' = await (map_async f tl) in
17     let tr' = await (map_async f tr) in
18     N (y, tl', tr')
19
20 let rec map_asyncpar_width f t =
21   async @@ fun () -> match t with
22   | L -> L
23   | N (v, tl, tr) ->
24     let y = f v in
25     let task_tl = map_asyncpar_width f tl in
26     let task_tr = map_asyncpar_width f tr in
27     N (y, await task_tl, await task_tr)
28
29 let rec map_asyncpar_full f t =
30   async @@ fun () -> match t with
31   | L -> L
32   | N (v, tl, tr) ->
33     let task_tl = map_asyncpar_full f tl in
34     let task_tr = map_asyncpar_full f tr in
35     N (f v, await task_tl, await task_tr)

```

```

1 let%async rec map_async_tmca f t =
2   match t with
3   | L -> L
4   | N (v, tl, tr) ->
5     let y = f v in
6     let tl' = await (map_async_tmca f tl) in
7     N (y, tl', await (map_async_tmca f tr))
8
9 let%asyncpar rec map_asyncpar_width_tmca f t =
10  match t with
11  | L -> L
12  | N (v, tl, tr) ->
13    let y = f v in
14    N (y,
15      await (map_asyncpar_width_tmca f tl),
16      await (map_asyncpar_width_tmca f tr))
17
18 let%asyncpar rec map_asyncpar_full_tmca f t =
19  match t with
20  | L -> L
21  | N (v, tl, tr) ->
22    N (f v,
23      await (map_asyncpar_full_tmca f tl),
24      await (map_asyncpar_full_tmca f tr))

```

Fig. 39. Implementation of map on trees

For the corresponding formal versions, we first need some syntactic sugar (inspired by the formalisation of `tmc[4]`):

$$\begin{aligned}
\text{match } e \text{ with} & & \text{let } m = e \text{ in} \\
| () \rightarrow e_1 & := & \text{if } m = () \text{ then } e_1 \\
| t \ x \ y \rightarrow e_2 & & \text{else let } x, y = m.(1), m.(2) \text{ in } e_2 \\
\\
\{ t, e_1, e_2, e_3 \} & := & \{ t^\uparrow, e_1, \{ t^\downarrow, e_2, e_3 \} \} \\
\\
\text{NIL, LEAF} & := & ()
\end{aligned}$$

and assume the following constructor tags:  $\text{CONS}, \text{NODE}^\uparrow, \text{NODE}^\downarrow$ .

We can then write the roughly corresponding formal versions of the previous functions:

$$\begin{aligned}
\text{mapNaive} \mapsto & \lambda f \ l. \ \text{match } l \ \text{with} \\
& | \text{NIL} \rightarrow \text{NIL} \\
& | \text{CONS } x \ t \rightarrow \\
& \quad \{ \text{CONS}, f \ x, \text{mapNaive } f \ t \} \\
\\
\text{mapAsyncSync} \mapsto & \lambda_{\text{async}} f \ l. \ \text{match } l \ \text{with} \\
& | \text{NIL} \rightarrow \text{NIL} \\
& | \text{CONS } x \ t \rightarrow \\
& \quad \{ \text{CONS}, f \ x, \text{await}(\text{mapAsyncSync } f \ t) \} \\
\\
\text{mapAsyncSeq} \mapsto & \lambda_{\text{async}} f \ l. \ \text{match } l \ \text{with} \\
& | \text{NIL} \rightarrow \text{NIL} \\
& | \text{CONS}_{\text{seq}} \ x \ t \rightarrow \\
& \quad \{ \text{CONS}_{\text{seq}}, \text{await}(f \ x), \text{await}(\text{mapAsyncSeq } f \ t) \} \\
\\
\text{mapAsyncPar} \mapsto & \lambda_{\text{async}} f \ l. \ \text{match } l \ \text{with} \\
& | \text{NIL} \rightarrow \text{NIL} \\
& | \text{CONS}_{\text{par}} \ x \ t \rightarrow \\
& \quad \{ \text{CONS}_{\text{par}}, \text{await}(f \ x), \text{await}(\text{mapAsyncPar } f \ t) \}
\end{aligned}$$

Fig. 40. Formal definitions, map on lists

$$\begin{aligned}
\text{mapTreeSeq} \mapsto & \lambda f t. \text{ match } t \text{ with} \\
& | \text{LEAF} \rightarrow \text{LEAF} \\
& | \text{NODE } l x r \rightarrow \\
& \quad \{ \text{NODE}, \text{mapTreeSeq } f l, f x, \text{mapTreeSeq } f r \} \\
\text{mapTreeParWidth} \mapsto & \lambda_{\text{async}} f t. \text{ match } t \text{ with} \\
& | \text{LEAF} \rightarrow \text{LEAF} \\
& | \text{NODE}_{\text{par}} l x r \rightarrow \text{let } v = f x \text{ in} \\
& \quad \{ \text{NODE}_{\text{par}}, \text{await}(\text{mapTreeParWidth } f l), v, \\
& \quad \text{await}(\text{mapTreeParWidth } f r) \} \\
\text{mapTreeParFull} \mapsto & \lambda_{\text{async}} f t. \text{ match } t \text{ with} \\
& | \text{LEAF} \rightarrow \text{LEAF} \\
& | \text{NODE}_{\text{par}} l x r \rightarrow \\
& \quad \{ \text{NODE}_{\text{par}}, \text{await}(\text{mapTreeParFull } f l), f x, \\
& \quad \text{await}(\text{mapTreeParFull } f r) \}
\end{aligned}$$

Fig. 41. formal definitions, map on trees

## C Backward-simulation proofs

### C.1 Tail-Modulo-Await backward simulation proof

Cases for the main thread. :

Since the main thread is unaffected by the transformation, its term is only comprised of source syntax. Thus, the only steps that can be taken are local steps, **DPS-CALL** and **AWAIT-TARGET**

**STEP** any local step is immediately closed if they don't affect the store. If they affect the store, then they affect it in the same way in source and target, and since stores are equal, we close immediately.

**DPS-CALL:**

$$\begin{array}{ccc}
\text{Task}_s \sqcup (\text{main} \mapsto C[(f v)]) & \cong_D & \text{Task}_t \sqcup (\text{main} \mapsto C[(f v)]), \text{Fut}_t \\
\downarrow \text{ASYNC-CALL} & & \downarrow \text{DPS-CALL} \\
(f \mapsto \lambda_{\text{async}} x. e) \in \text{Fun} & & (f \mapsto \lambda_{\text{dps}} \delta. x. \mathcal{D}[e]_{\delta}) \in \text{Fun}' \\
\text{Task}_s \sqcup (\text{main} \mapsto C[\text{fut}']) & \cong_{D \sqcup \{d\}} & \text{Task}_t \sqcup (\text{main} \mapsto C[\text{fut}']) \\
\sqcup (\text{fut}' \mapsto e[x \mapsto v]) & & \sqcup (\text{tid} \mapsto \mathcal{D}[e]_{\delta}[\delta \mapsto d][x \mapsto v]), \\
& & \text{Fut}_t \sqcup [\text{fut}' \mapsto \mathcal{U} d]
\end{array}$$

Main threads in red still equal, blue closed by **TASK-COMP**

**AWAIT-TARGET:** The challenge is:

$$\begin{array}{ccc}
\text{Task}_s \sqcup (\text{main} \mapsto C[\text{await}(\text{fut})]) & \cong_D & \text{Task}_t \sqcup (\text{main} \mapsto C[\text{await}(\text{fut})]), \text{Fut}_t \\
& & \downarrow \text{AWAIT-TARGET} \\
& & \text{Fut}_t [ \text{fut} ] = \mathfrak{R} v \\
& & \text{Task}_t \sqcup (\text{main} \mapsto C[v]), \text{Fut}_t
\end{array}$$

We need to have, on the LHS,  $(fut \mapsto v)$ , which we will get by exploiting  $Fut_t [ fut ] = \mathfrak{R} v$  in the RHS. There are two cases:

**TASK-VAL:**

$$\begin{array}{ccc} Task_s \sqcup (\text{main} \mapsto C[\mathbf{await}(fut)]) & \cong_D & Task_t \sqcup (\text{main} \mapsto C[\mathbf{await}(fut)]), Fut_t \\ \downarrow \text{AWAIT-RESOLVED} & & \downarrow \text{AWAIT-TARGET} \\ Task_s \sqcup (\text{main} \mapsto C[v]) & \cong_D & Task_t \sqcup (\text{main} \mapsto C[v]), Fut_t \end{array}$$

$\downarrow_{Fut_t [ fut ] = \mathfrak{R} v}$

**TASK-AWAIT-VAL:** We have some  $T_s^{(j)} = \text{Aw}[fut, fut_1..fut_n] \sqcup (fut_n \mapsto v) \subseteq Task_s$ . We close by  $n + 1$  applications of **AWAIT-RESOLVED**.

$$\begin{array}{ccc} Task_s \sqcup (\text{main} \mapsto C[\mathbf{await}(fut)]) & \cong_D & Task_t \sqcup (\text{main} \mapsto C[\mathbf{await}(fut)]), Fut_t \\ \downarrow \text{AWAIT-RESOLVED} & & \downarrow \text{AWAIT-TARGET} \\ Task_s \sqcup (\text{main} \mapsto C[v]) & \cong_D & Task_t \sqcup (\text{main} \mapsto C[v]), Fut_t \end{array}$$

$\downarrow_{i=n..0}$        $\downarrow_{Fut_t [ fut ] = \mathfrak{R} v}$

We are left with  $T_s^{(j)'} = (fut \mapsto v) \sqcup \overline{(fut_1 \mapsto v)}^{1 \leq i \leq n}$ , which is closed using **TASK-VAL**. Which concludes the cases for the main thread.

*other groupings.* In the following, we thus assume that we are in one of the  $T_t^{(i)}$ . We omit writing the await chains when they are not affected.

Note that we have the following property: For any  $C_t, e$  such that  $e_t = C_t[e]$ , there exists a context  $C_s$  such that  $e_s = C[e]$ .

Moreover, for any  $e', C_s[e'] \sim_d C_t[e']$

**Case STEP.** All sub-cases follow directly from the properties presented above, and the arguments mentioned for the main thread.

**Case AWAIT-RESOLVED.** For the reasons explained above:

$$\begin{array}{ccc} Task_s \sqcup (fut \mapsto C_s[\mathbf{await}(fut_r)]) & \cong_{D \sqcup \{d\}} & Task_t \sqcup (tid \mapsto C_t[\mathbf{await}(fut_r)]), Fut_t \\ \downarrow \text{AWAIT-RESOLVED} & & \downarrow \text{AWAIT-RESOLVED} \\ Task_s \sqcup (fut_0 \mapsto C_s[v]) & \cong_{D \sqcup \{d\}} & Task_t \sqcup (tid \mapsto C_t[v]), Fut_t \end{array}$$

$\downarrow_{(fut_r \mapsto v) \in Task_s (*)}$        $\downarrow_{Fut_t [ fut_r ] = \mathfrak{R} v}$

(\*) by exploiting  $Fut_t [ fut_r ] = \mathfrak{R} v$ , in the case **TASK-VAL**. See the proof in the case of the main thread for the case of **TASK-AWAIT-VAL**

**Case CHAIN.**

$$\begin{array}{ccc} Task_s \sqcup \text{Aw}[fut_0..fut_n] \sqcup (fut_n \mapsto \mathbf{await}(fut_r)) & \cong_{D \sqcup \{d\}} & Task_t \sqcup (tid \mapsto \mathbf{forward}(fut_r, d)), \\ & & Fut_t \sqcup [ fut_0 \mapsto \mathfrak{U} d ] \\ \downarrow \text{AWAIT-RESOLVED} & & \downarrow \text{CHAIN} \\ Task_s \sqcup \text{Aw}[fut_0..fut_n] \sqcup (fut_n \mapsto v) & \cong_D & Task_t, Fut_t \sqcup [ fut_0 \mapsto \mathfrak{R} v ] \end{array}$$

$\downarrow_{(fut_r \mapsto v) \in Task_s (*)}$        $\downarrow_{Fut_t [ fut_r ] = \mathfrak{R} v}$

Using the same arguments as above to obtain (\*).

Closed with **TASK-VAL** or **TASK-AWAIT-VAL** depending on the number of awaits in LHS (i.e. depending on  $n$ )

Case **DPS-CALL**.

$$\begin{array}{ccc}
 \text{Task}_s \sqcup (fut \mapsto C_s[(f v)]) & \cong_{D \sqcup \{d\}} & \text{Task}_t \sqcup (tid \mapsto C_t[(f v)], Fut_t \sqcup [fut \mapsto \mathcal{U} d]) \\
 \downarrow \text{ASYNC-CALL} & & \downarrow \text{DPS-CALL} \\
 (f \mapsto \lambda_{\text{async}} x. e) \in \text{Fun} & & (f \mapsto \lambda_{\text{dps}} \delta, x. \mathcal{D}[[e]]_\delta) \in \text{Fun}' \\
 \text{Task}_s \sqcup (fut \mapsto C_s[fut']) & \cong_{D \sqcup \{d, d'\}} & \text{Task}_t \sqcup (tid \mapsto C_t[fut']) \\
 \sqcup (fut' \mapsto e[x \mapsto v]) & & \sqcup (tid' \mapsto \mathcal{D}[[e]]_\delta[\delta \mapsto d'][x \mapsto v]), \\
 & & Fut_t \sqcup [fut \mapsto \mathcal{U} d] \sqcup [fut' \mapsto \mathcal{U} d']
 \end{array}$$

Chain of awaits omitted. Both painted subsets closed with **TASK-COMP**.

Case **TAIL-CALL**.

$$\begin{array}{ccc}
 \text{Task}_s \sqcup \text{Aw}[fut_0..fut_n] & \cong_{D \sqcup \{d\}} & \text{Task}_t \sqcup (tid \mapsto (f \#_t(d, v))), \\
 \sqcup (fut_n \mapsto \text{await}((f v))) & & Fut_t \sqcup [fut_0 \mapsto \mathcal{U} d] \\
 \downarrow \text{ASYNC-CALL} & & \downarrow \text{TAIL-CALL} \\
 \text{Task}_s \sqcup \text{Aw}[fut_0..fut_{n+1}] & \cong_{D \sqcup \{d\}} & \text{Task}_t \sqcup (tid' \mapsto \mathcal{D}[[e]]_\delta[\delta \mapsto d][x \mapsto v]), \\
 \sqcup (fut_{n+1} \mapsto e[x \mapsto v]) & & Fut_t \sqcup [fut_0 \mapsto \mathcal{U} d]
 \end{array}$$

With  $fut_{n+1}$  fresh and awaited nowhere else. Closed with **TASK-COMP**.

Case **FUTURE-FILL** ( $\varepsilon$ ).

$$\begin{array}{ccc}
 \text{Task}_s \sqcup \text{Aw}[fut_0..fut_n] & \cong_{D \sqcup \{d\}} & \text{Task}_t \sqcup (tid \mapsto \text{refine}(d, v)), Fut_t \sqcup [fut_0 \mapsto \mathcal{U} d] \\
 \sqcup (fut_n \mapsto v) & \cong_D & \downarrow \text{FUTURE-FILL} \\
 & & \varepsilon \\
 & & \text{Task}_t, Fut_t \sqcup [fut_0 \mapsto \mathfrak{R} v]
 \end{array}$$

Closed using **TASK-AWAIT-VAL**.

There can be at most as many **FUTURE-FILL** steps as there are unresolved futures in  $Fut_t$ , of which there are finitely many at any given point; which ensures that we cannot stutter indefinitely.

Which concludes the proof.

## C.2 Tail-Modulo-Cons-Await backward simulation proof

**C.2.1 Simulation relation.** We first define the full backward simulation relation, and all necessary sub-relations.

$$\begin{array}{ccc}
 \frac{\text{EXPR-TRANSFO}}{e \sim_d \mathcal{D}[[e]]_\delta[\delta \mapsto d]} & \frac{\text{EXPR-FILL-AWAIT}}{e \sim_d \text{refine}(d, e)} & \frac{\text{EXPR-AWAIT}}{e \stackrel{\text{await}}{\sim}_d \mathcal{D}_a[[e]]_\delta[\delta \mapsto d]}
 \end{array}$$

Fig. 42. Expr-level relations

$$\frac{\text{THREAD-DEST} \quad e \sim_d e'}{(fut \mapsto e) \quad fut \approx_d [d \mapsto /], (tid \mapsto e')}$$

Fig. 43. Thread-level relation

$$\frac{\text{THREAD-AWAIT} \quad T_s \quad fut_n \approx_d \sigma_t, T_t}{\text{Aw}[fut_0..fut_n] \sqcup T_s \quad fut_0 \approx_d^{\{fut_1..n\}} \sigma_t, T_t} \quad \frac{\text{THREAD-VAL}}{(fut \mapsto v) \sqcup (\overline{fut_i \mapsto v}) \quad fut \approx_d^{\overline{fut_i}} [d \mapsto v], (\overline{tid \mapsto ()})^+}$$

Fig. 44. Pool relation. The right index annotates the set of “internal” futures

$$\frac{\text{FIELD-THREAD-AWAIT} \quad \alpha = \mathbf{await}(fut) \quad T_s \quad fut \approx_d^J \sigma_t, T_t \quad d \in \text{dom}(\sigma_t)}{\{\alpha\} : T_s \mathcal{R}_d^J \sigma_t, T_t} \quad \frac{\text{FIELD-RESOLVED} \quad T_s = (\overline{fut_i \mapsto v_i}) \quad \exists i, v_i = v}{\{v\} : T_s \mathcal{R}_d^{\overline{fut_i}} [d \mapsto v], \emptyset}$$

$$\frac{\text{FIELD-CHAIN} \quad \alpha = \mathbf{await}(fut) \quad T_t = \mathbf{forward}(d, fut)}{\{\alpha\} : \emptyset \quad \text{fwd} \mathcal{R}_d^{\emptyset} [d \mapsto /], T}$$

Fig. 45. For keeping sub-tasks for constructors together

$$\begin{array}{c}
\text{PAR-BOTH-0} \\
\frac{e_1 \overset{\text{await}}{\sim}_{d_1} e'_1 \quad e_2 \overset{\text{await}}{\sim}_{d_2} e'_2}{\text{let } x_1 = e_1 \text{ in } \quad [d \mapsto \ell, \ell \mapsto t_{||}, /, /],} \\
(fut \mapsto \text{let } x_2 = e_2 \text{ in } \quad ) \overset{fut \simeq_d}{\sim} (tid \mapsto \text{let } (d_1, d_2) = (\ell + 1, \ell + 2) \text{ in } \quad ) \\
[ t_{||}, \text{await}(x_1), \text{await}(x_2) ] \quad e'_1; e'_2 \\
\\
\text{PAR-BOTH-1} \\
\frac{e_1 \overset{\text{await}}{\sim}_{\ell+1} e'_1 \quad e_2 \overset{\text{await}}{\sim}_{\ell+2} e'_2}{\text{let } x_1 = e_1 \text{ in } \quad [d \mapsto \ell, \ell \mapsto t_{||}, /, /], (tid \mapsto e'_1; e'_2)} \\
(fut \mapsto \text{let } x_2 = e_2 \text{ in } \quad ) \overset{fut \simeq_d}{\sim} \\
[ t_{||}, \text{await}(x_1), \text{await}(x_2) ] \\
\\
\text{PAR-BOTH-2} \\
\frac{\vee \left\{ \begin{array}{l} \{\alpha_1\} : T_1 \overset{\text{fwd}}{\mathcal{R}}_{\ell+1} \sigma_1, T'_1 \\ \{\alpha_1\} : T_1 \mathcal{R}_{\ell+1}^I \sigma_1, T'_1 \end{array} \right. \quad e_2 \overset{\text{await}}{\sim}_{\ell+2} e'_2}{(fut \mapsto \text{let } x_2 = e_2 \text{ in } \quad ) \sqcup T_1 \overset{fut \simeq_d^I}{\sim} [d \mapsto \ell, \ell \mapsto t_{||}, \ell + 2 \mapsto /] \sqcup \sigma_1, (tid \mapsto (():)^? e'_2) \sqcup T'_1} \\
[ t_{||}, \alpha_1, \text{await}(x_2) ] \\
\\
\text{PAR-BOTH-3} \\
\frac{\vee \left\{ \begin{array}{l} \{\alpha_1\} : T_1 \overset{\text{fwd}}{\mathcal{R}}_{\ell+1} \sigma_1, T'_1 \\ \{\alpha_1\} : T_1 \mathcal{R}_{\ell+1}^{I_1} \sigma_1, T'_1 \end{array} \right. \quad \{\alpha_2\} : T_2 \mathcal{R}_{\ell+2}^{I_2} \sigma_2, T'_2 \quad T = (tid \mapsto (():)^?) \quad \text{holes}(d) \geq 1}{(fut \mapsto [ t_{||}, \alpha_1, \alpha_2 ]) \sqcup T_1 \sqcup T_2 \overset{I_1 \sqcup I_2}{\sim}_d [d \mapsto \ell, \ell \mapsto t_{||}] \sqcup \sigma_1 \sqcup \sigma_2, T \sqcup T'_1 \sqcup T'_2}
\end{array}$$

Fig. 46. paratag, both awaits

$$\begin{array}{c}
\text{PAR-LEFT-0} \\
\frac{e_1 \stackrel{\text{await}}{\sim}_{d_1} e'_1 \quad e_2 \sim_{d_2} e'_2}{(fut \mapsto \text{let } x_1 = e_1 \text{ in } [t_{||}, \text{await}(x_1), x_2]) \text{ fut} \simeq_d (tid \mapsto \text{let } (d_1, d_2) = (\ell + 1, \ell + 2) \text{ in } e'_1; e'_2) [d \mapsto \ell, \ell \mapsto t_{||}, /, /]}, \\
\text{PAR-LEFT-1} \\
\frac{e_1 \stackrel{\text{await}}{\sim}_{\ell+1} e'_1 \quad e_2 \sim_{\ell+2} e'_2}{(fut \mapsto \text{let } x_1 = e_1 \text{ in } [t_{||}, \text{await}(x_1), x_2]) \text{ fut} \simeq_d [d \mapsto \ell, \ell \mapsto t_{||}, /, /], (tid \mapsto e'_1; e'_2)} \\
\text{PAR-LEFT-2} \\
\frac{\bigvee \left\{ \begin{array}{l} \{\alpha_1\} : T_1 \text{ fwd } \mathcal{R}_{\ell+1} \sigma_1, T'_1 \\ \{\alpha_1\} : T_1 \mathcal{R}_{\ell+1}^I \sigma_1, T'_1 \end{array} \right. \quad e_2 \sim_{\ell+2} e'_2}{(fut \mapsto \text{let } x_2 = e_2 \text{ in } [t_{||}, \alpha_1, x_2]) \sqcup T_1 \text{ fut} \simeq_d^I [d \mapsto \ell, \ell \mapsto t_{||}, \ell_2 \mapsto /] \sqcup \sigma_1, (tid \mapsto (();)^? e'_2) \sqcup T'_1} \\
\text{PAR-LEFT-3} \\
\frac{\bigvee \left\{ \begin{array}{l} \{\alpha_1\} : T_1 \text{ fwd } \mathcal{R}_{\ell+1} \sigma_1, T'_1 \\ \{\alpha_1\} : T_1 \mathcal{R}_{\ell+1}^I \sigma_1, T'_1 \end{array} \right. \quad T = (tid \mapsto (();)^?)^? \quad \text{holes}(d) \geq 1}{(fut \mapsto [t_{||}, \alpha_1, v_2]) \sqcup T_1 \sqcup T_2 \text{ fut} \simeq_d [d \mapsto \ell, \ell \mapsto t_{||}, \ell + 2 \mapsto v_2] \sqcup \sigma_1, T \sqcup T'_1}
\end{array}$$

Fig. 47. paratag, one await on the left (symmetrically for right)

$$\begin{array}{c}
\text{PAR-NONE-0} \\
\frac{e_1 \sim_{d_1} e'_1 \quad e_2 \sim_{d_2} e'_2}{(fut \mapsto \text{let } x_1 = e_1 \text{ in } [t_{||}, x_1, x_2]) \text{ fut} \simeq_d (tid \mapsto \text{let } (d_1, d_2) = (\ell + 1, \ell + 2) \text{ in } e'_1; e'_2) [d \mapsto \ell, \ell \mapsto t_{||}, /, /]}, \\
\text{PAR-NONE-1} \\
\frac{e_1 \sim_{\ell+1} e'_1 \quad e_2 \sim_{\ell+2} e'_2}{(fut \mapsto \text{let } x_1 = e_1 \text{ in } [t_{||}, x_1, x_2]) \text{ fut} \simeq_d [d \mapsto \ell, \ell \mapsto t_{||}, /, /], (tid \mapsto e'_1; e'_2)} \\
\text{PAR-NONE-2} \\
\frac{e_2 \sim_{\ell+2} e'_2}{(fut \mapsto \text{let } x_2 = e_2 \text{ in } [t_{||}, v_1, x_2]) \text{ fut} \simeq_d [d \mapsto \ell, \ell \mapsto t_{||}, v_1, /], (tid \mapsto (();)^? e'_2)}
\end{array}$$

Fig. 48. partag, no awaits

$$\begin{array}{c}
\text{SEQ-NONE-0} \\
\frac{e_2 \sim_{d_2} e'_2}{(fut \mapsto \text{let } x_2 = e_2 \text{ in } [t_{\text{seq}}, v_1, x_2]) \text{ fut} \approx_d (tid \mapsto \text{let } d_2 = \ell + 2 \text{ in } e'_2) \quad [d \mapsto \ell, \ell \mapsto t_{\text{seq}}, v_1, /]}, \\
\text{SEQ-NONE-1} \\
\frac{e_2 \sim_{\ell+2} e'_2}{(fut \mapsto \text{let } x_2 = e_2 \text{ in } [t_{\text{seq}}, v_1, x_2]) \text{ fut} \approx_d [d \mapsto \ell, \ell \mapsto t_{\text{seq}}, v_1, /], (tid \mapsto e'_2)}
\end{array}$$

Fig. 49. seqtag, no awaits

$$\begin{array}{c}
\text{SEQ-LEFT-0} \\
\frac{e_1 \overset{\text{await}}{\sim}_{d_1} e'_1}{(fut \mapsto \text{let } x_1 = e_1 \text{ in } [t_{\text{seq}}, \text{await}(x_1), v_2]) \text{ fut} \approx_d (tid \mapsto \text{let } d_1 = \ell + 1 \text{ in } e'_1) \quad [d \mapsto \ell, \ell \mapsto t_{\text{seq}}, /, v_2]}, \\
\text{SEQ-LEFT-1} \\
\frac{e_1 \overset{\text{await}}{\sim}_{\ell+1} e'_1}{(fut \mapsto \text{let } x_1 = e_1 \text{ in } [t_{\text{seq}}, \text{await}(x_1), v_2]) \text{ fut} \approx_d (tid \mapsto e'_1) \quad [d \mapsto \ell, \ell \mapsto t_{\text{seq}}, /, v_2]}, \\
\text{SEQ-LEFT-2} \\
\frac{\bigvee \left\{ \begin{array}{l} \{\alpha_1\} : T_1 \text{ fwd } \mathcal{R}_{\ell+1} \sigma_1, T'_1 \\ \{\alpha_1\} : T_1 \mathcal{R}_{\ell+1}^I \sigma_1, T'_1 \end{array} \right. \quad T = (tid \mapsto ())^? \quad \text{holes}(d) \geq 1}{(fut \mapsto [t_{\text{seq}}, \alpha_1, v_2]) \sqcup T_1 \text{ fut} \approx_d \frac{[d \mapsto \ell, \ell \mapsto t_{\text{seq}}, \ell + 2 \mapsto v_2] \sqcup \sigma_1}{T \sqcup T'_1}}
\end{array}$$

Fig. 50. seqtag, await on the left (sym for right)

$$\begin{array}{c}
\text{BOX-THREAD} \\
\frac{T_s \text{ fut} \approx_d^I s_t, T_t}{T_s \cong^I s_t, T_t, [fut \mapsto \mathfrak{U} d]} \\
\text{BOX-VAL} \\
\frac{}{(fut \mapsto v) \sqcup \overline{(fut_i \mapsto v)}^i \cong \overline{fut_i}^i [d \mapsto v], \emptyset, [fut \mapsto \mathfrak{U} d]} \\
\text{BOX-RESOLVED} \\
\frac{T = \overline{(tid \mapsto ())}}{(fut \mapsto v) \sqcup \overline{(fut_i \mapsto v)}^i \cong \overline{fut_i}^i \emptyset, T, [fut \mapsto \mathfrak{R} v]}
\end{array}$$

Fig. 51. Top-level-future-level rel (=boxes)

*Config-level.* The simulation relation between source and target configurations:

$$\sigma_s, Task_s \succcurlyeq \sigma_t, Task_t, Fut_t \Leftrightarrow \begin{cases} (1) & Task_s = (\text{main} \mapsto e) \sqcup T_s^{(1)} \sqcup \dots \sqcup T_s^{(n)} \\ (2) & Task_t = (\text{main} \mapsto e) \sqcup T_t^{(1)} \sqcup \dots \sqcup T_t^{(n)} \\ (3) & \sigma_t = \sigma_s \sqcup \sigma_1 \sqcup \dots \sqcup \sigma_n \\ (4) & Fut_t = \left[ \overline{fut_i \mapsto \mathcal{F}_i}^{1 \leq i \leq n} \right] \quad (\mathcal{F}_i = \mathcal{U} d_i \mid \mathcal{R} v_i) \\ (5) & \forall i, T_s^{(i)} \cong^{I_i} \sigma_i, T_t^{(i)}, [fut_i \mapsto \mathcal{F}_i] \end{cases}$$

with each  $I_i$  internal sets of futures in the sense given in Section 2.5.

Like in tma, steps are “local” to one thread pool.

*C.2.2 Proof sketch.* We follow the same sketch as in tma:

- (1) isolate in which set of threads the step operates: either the main, or one of the parallel pools
- (2) case analysis on  $\cong$ , then on the step.
- (3) in the parallel pools, in case **BOX-THREAD**, use lemma 3.1

*Steps in the main thread.* The same arguments as for tma are still valid here, with one caveat: the stores are not fully synchronised. However, only the synchronised parts are accessible by the main thread.

*Steps in parallel thread pools.* We are thus in a  $T_t^{(i)}$ . We write the configs as

$$\sigma_s, Task_s \sqcup T_s^{(i)} \succcurlyeq \begin{array}{l} \sigma_s \sqcup \sigma_{\text{others}} \sqcup \sigma_i, \\ Task_t \sqcup T_t^{(i)}, \\ Fut_t \sqcup F_i \end{array}$$

where  $T_i \cong^{I_i} \sigma_i, T_i, F_i$

We also have the same property as for single-hole: if  $e_s \sim_d e_t$ , then for any  $C_t, e$  such that  $e_t = C_t[e]$ , there exists a context  $C_s$  such that  $e_s = C_s[e]$ .

Moreover, for any  $e', C_s[e'] \sim_d C_t[e']$

**BOX-VAL.** Only step that can be taken is **FUTURE-RESOLVE** ( $\epsilon$ ). We close immediately with **BOX-RESOLVED**. Cannot have infinitely many **FUTURE-RESOLVE** steps in a row: each consumes an unresolved future, which there are finitely many at each step.

**BOX-RESOLVED.** Only possible step is **PRUNE-TASKS** ( $\epsilon$ ) (only if  $T \neq \emptyset$ ), and we close immediately.

**BOX-THREAD.** We give here most elements of the proof of lemma 3.1, from which the proof for this case follows immediately

**THREAD-DEST.** This case handles almost every computation, except constructors being built

**local steps** All local steps except the ones for **refine** are treated the same as for the single-hole optimisation, using the contexts property, and the same arguments as in the main thread. For the store argument, only refine can read or write to unsynchronised parts of the store.

**refine** ( $\neq$  **FILL**) A refine step necessarily happens in a context of the form **let**  $\delta^+ = \text{refine}(d, \square)$  **in**  $e$ . There are several cases depending on the shape of the second argument of **refine** (sequential or parallel tag, which and how many fields are in the domain of  $\text{sta}$ ). They are all matched with one of the **CONSTR-PAR/SEQ** rules, and we enter one of the chains described earlier, in a state **PAR/SEQ-<\_>-0**.

**FILL** We close with **THREAD-VAL** after  $n$  applications of **AWAIT-RESOLVED**.

$$\begin{array}{ccc}
\text{Aw}[fut_0..fut_n] \sqcup (fut_n \mapsto v) & fut_0 \approx_d & [d \mapsto /], (tid \mapsto \mathbf{refine}(d, v)) \\
\downarrow \text{AWAIT-RESOLVED} & & \downarrow \text{FILL} \\
\downarrow_{i=n-1..0} & & \\
(fut_0 \mapsto v) \sqcup \overline{(fut_i \mapsto v)}^{i=1..n} & fut_0 \approx_d & [d \mapsto v], (tid \mapsto ())
\end{array}$$

Even though we might answer the challenge with 0 steps on the source side, we cannot take infinitely many **FILL** steps in a row: each **FILL** step consumes a binding  $[d \mapsto /]$  from the store. The number of these bindings in the store is thus a strictly decreasing measure for possibly- $\varepsilon$  steps. It is moreover always finite, thus we cannot stutter indefinitely. This also verifies the second part of the lemma.

**CHAIN-RESOLVED** We have  $e \sim_d e' = C[\mathbf{forward}(fut', d)]$ . The only possible case for  $C$  is  $\square$ . Thus, with  $Fut_t[fut'] = \mathfrak{R} v$ , we get by **BOX-RESOLVED** that  $(fut' \mapsto v) \in Task_s$

$$\begin{array}{ccc}
\text{Aw}[fut_0..fut_n] \sqcup (fut_n \mapsto \mathbf{await}(fut')) & fut_0 \approx_d & [d \mapsto /], (tid \mapsto \mathbf{forward}(fut', d)) \\
\downarrow \text{AWAIT-RESOLVED} & & \downarrow \text{CHAIN-RESOLVED} \\
\downarrow_{i=n..0} & & \\
(fut_0 \mapsto v) \sqcup \overline{(fut_i \mapsto v)}^{i=1..n} & fut_0 \approx_d & [d \mapsto v], (tid \mapsto ())
\end{array}$$

Closing again with **THREAD-VAL**, same as fill, with possibly  $n + 1$  awaits.

This also verifies the second part of the lemma.

**DPS-CALL** Very similar to the case for the single-hole optimisation. Here, we create a new “box”:

$$\begin{array}{ccc}
(fut \mapsto C[(f v)]) & fut \approx_d & \sigma_t, \\
& & (tid \mapsto C[(f v)]), \\
\downarrow \text{ASYNC-CALL} & & \downarrow \text{DPS-CALL} \\
\downarrow_{(f \mapsto \lambda_{\text{async}} x. e) \in \text{Fun}} & & \downarrow_{(f \mapsto \lambda_{\text{dps}} \delta, x. \mathcal{D}[e]_{\delta}) \in \text{Fun}'} \\
(fut \mapsto C[fut']) & fut \approx_d & \sigma_t, \\
& & (tid \mapsto C[fut']) \\
\sqcup & & \sqcup \\
(fut' \mapsto e[x \mapsto v]) & \cong & [d' \mapsto /], \\
& & (tid' \mapsto \mathcal{D}[e]_{\delta}[\delta \mapsto d'] [x \mapsto v]) \\
& & [fut' \mapsto \mathfrak{U} d']
\end{array}$$

Red part is the same, blue by **BOX-THREAD**, **THREAD-DEST**

**TAIL-CALL** Exactly the same as for the single-hole optimisation.

**CHAIN-UNRESOLVED**, **FUTURE-RESOLVE**, **PRUNE-TASKS** None of these rules applicable in this case.

Continuing the cases of  $\approx$ :

**THREAD-VAL**. Two rule can be applied: either **PRUNE-TASKS**, or **FUTURE-RESOLVE**.

**PRUNE-TASKS** stutters and we either stay in the same state, or if it was the last task to prune, go to **BOX-VAL**.

**FUTURE-RESOLVE** matched with  $n$  awaits, then closed with **BOX-RESOLVED**

*The sequence PAR-BOTH-X.* We give details here for one of the chains handling constructors: the case of parallel-tagged constructors, with both fields “asynchronous” (read: in the domain of `sta`). The arguments for the other sequences are the same.

**PAR-BOTH-0** Only case is reducing the `let`, stuttering, going to **PAR-BOTH-1**

**PAR-BOTH-1** Any step under context in  $e'_1$  will be matched in  $e_1$ , stay in same state. Last steps can be either **TAIL-CALL** or one of the chain ones, and need to be matched with the corresponding source step, then the substitution ( $e_1$  is now just a future), and we go to **PAR-BOTH-2**

**PAR-BOTH-2** Any step will be either in the thread *tid*, or in  $T'_1$ . Steps in  $T'_1$  keep us in the same state, we go to **PAR-BOTH-3** only with the last step in  $e'_2$ .

**in *tid*** First step will be getting rid of the  $((;))^\dagger$ , which is epsilon. Then, any subsequent steps in  $e'_2$  will be matched in  $e_2$  in the same way as  $e'_1$  in the previous case. The last step will either create a  $T'_2$ , or directly resolve to a value, in the same way.

**in  $T'_1$**  if its under **FIELD-CHAIN**, only possible step is **CHAIN-RESOLVED**, ok. Otherwise, induction hypothesis.

After the last step (which by ind ends with  $(fut_1 \mapsto v_1)$  on the LHS), we must resolve the `await`( $fut_1$ ), and we close this part with **FIELD-RESOLVED**.

**PAR-BOTH-3** Same as the previous one, but now steps might also be in  $T'_2$ . They are however for the most part treated in the same way.

The last step in this state is already detailed in [Section 3.4](#)