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 (April 2025), 36 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. [17]'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* [3] 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 [5],[1] which are entities representing the result of an ongoing computation. In fact, a notion similar to asynchronous tail calls already exists for OO method calls in asynchronous languages, dubbed `forward` [8].

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

[1] Coincidentally, both works were published in 1977, and both originated from the LISP community!

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

2025. ACM XXXX-XXXX/2025/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
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. A parallel and 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.[2] Before going over details, let us state our selling point: using our code transformation, this function is fully parallel (all calls to `f` 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 asynchronous programmer would however not write the code above and would favour a more parallel version by replacing Line 6 with:

```
let tl' = map f tl in let tr' = map f tr in Node (f x, await tl', await tr')
```

This version parallelizes the treatment of the left and right branches (this introduces task interleaving as discussed below). Its main drawback is that it can have as many "active" tasks as there are nodes in the tree; indeed all the finished tasks which have at least one unfinished task below it in the tree structure are still blocked in an `await` statement.

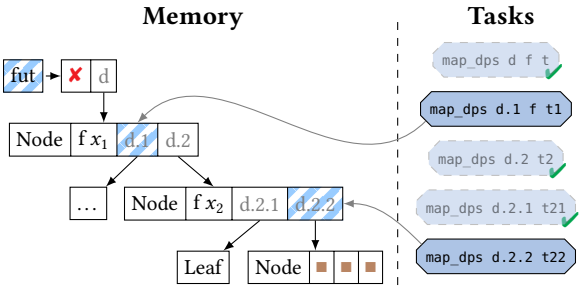
Our approach optimises the two preceding programs by both parallelizing subtasks, and terminating all tasks that are only awaiting others. To achieve this, we translate the function to *Destiny Passing Style*, an asynchronous interpretation of Destination Passing Style [16]. An idealized version of the 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 destiny `d` with `d ◂ Leaf`. In the recursive case, we perform the allocation ahead of time—similarly to what

[2]Our dialect is implemented with an OCaml syntax extension.

```
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 with `tmca`



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

Fig. 2. Translation of `map` on trees from Fig. 1 using Tail-Modulo-Cons-Await

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 pass them 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` 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 left side shows the status in memory of the tree being computed, the right side shows the tasks involved in the computation. Scheduler space is represented in blue. Futures and destiny both play a synchronisation role and serve as pointers to actual data, and are thus striped. The original future `fut` is yet unresolved (✘). 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). 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. In fact, it doesn't fully respect the sequential semantics. Indeed on Line 6 in the code above, the transformed code can *interleave* subtasks in `map f tl` and `map f tr`, while the sequential semantics would enforce an execution order, without interleaving. Thus, this automatic parallelisation comes at the risk of introducing undesired behaviors. We mitigate this risk in two ways.

First, we only modify code *at the level of a constructor in tail position*. This means the programmer can easily enforce sequentiality via simple let-bindings. For instance, we could replace Line 6 by:

```
let x' = f x in Node (x', await (map f tl), await (map f tr))
```

In this case, it would wait for `f` to return before proceeding to the two subtrees in parallel (thus bounding the parallelism by the width of the tree). This aligns with good practice in languages like OCaml, where users who care about order of operation should state it explicitly via let-bindings.[3]

Second, we introduce two variants for each constructor: *parallel* constructors which behave as above, and *sequential* ones which don't introduce parallelism and only allow one tail position. With these variants, we show that our transformation preserves the semantics.

[3]Naturally, this might come with some cost in scheduler- or stack- space, since the only way to optimise these aspects is through parallelism and thus interleaving of tasks.

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 automatises 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 users' choices, maximise parallelism, or strictly preserve 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 calculus, a source and a target, over which we formalise the transformation.

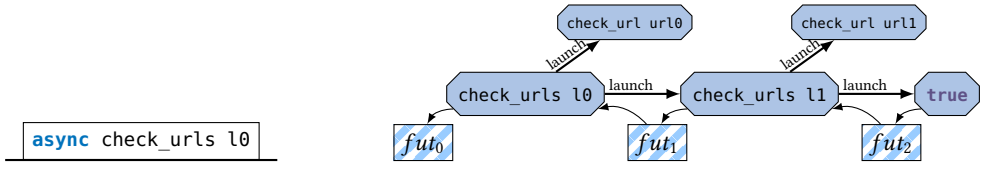
2.1 Motivating example

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`/`await`s 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

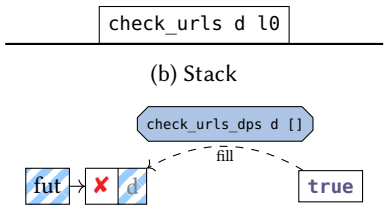


(a) Stack (b) Futures and Tasks

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`



(b) Stack

(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 call in the stack for `check_urls`.

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

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.

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* 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`** [8] from the literature. **`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 application. 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 λ-calculus which formalises this first transformation, Tail-Modulo-Await, without any constructor. Fig. 6 shows at once the syntax of both the source language and the target one, using the following code colour: *Light blue* elements are source-specific, *peach* elements are target-specific, and Grey elements are runtime syntax. dynamic semantics of the language.

Shared core. Both languages includes standard control flow and support for references, with operations for creation (**`newref`**(e)), reads ($!\, e$), and updates ($e := e$). 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 λ-lifted in a table of function definitions *Fun*. In both languages,

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

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.

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 to it, and any task can get the result of the computation by performing `await`(*fut*). If t is not finished yet, we say *fut* is *unresolved*, in which case `await`(*fut*) is blocking; otherwise, it retrieves 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 λ_{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 $(\lambda_{\mathsf{dps}} \delta,\, x.\, e)$, where δ 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 `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 `refine`(d, e). In the absence of constructor, this corresponds to `d ◂ e` in our informal syntax.[4] Finally, `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 (σ, e) carrying a store and an expression. The local reduction $\cdot \longrightarrow \cdot$ is provided on Fig. 8. This reduction relation is

[4] As the name hints, `refine` takes on more responsibilities when constructors are introduced, in the next section

$$
\begin{array}{lllll}
\text{(store)} & & \sigma & ::= & \left[\, \overline{\ell \mapsto v} \,\right] \\
\text{(evaluation contexts)} & \mathit{Ectx} \ni C & & ::= & \Box \mid \texttt{let}\, x = C\, \texttt{in}\, e \mid \texttt{if}\, C\, \texttt{then}\, e_t\, \texttt{else}\, e_f \mid (f\ C) \\
& & & & \mid \texttt{newref}(C) \mid C := e \mid \ell := C \mid \texttt{await}(C) \\
& & & & \mid (f \#_t\, C) \mid \texttt{forward}(C,\, e') \mid \texttt{refine}(e,\, C)
\end{array}
$$

Fig. 7. Runtime syntax for Stores σ and Contexts C

$$
\frac{\sigma, e \;\to\; \sigma', e'}{\sigma, C[e] \;\to\; \sigma', C[e']}
\qquad
\frac{\ell \notin \mathrm{dom}(\sigma)}{\sigma, \texttt{newref}(v) \;\to\; \sigma\,[\,\ell \mapsto v\,],\ell}
\qquad
\frac{(\ell \mapsto v) \in \sigma}{\sigma, !\,\ell \;\to\; \sigma, v}
$$

$$
\frac{}{\sigma, (\ell := v) \;\to\; \sigma\,[\,\ell \mapsto v\,], ()}
\qquad
\frac{(f \mapsto \lambda x.\, e) \in \mathit{Fun}}{\sigma, (f\ v) \;\to\; \sigma, e[x \mapsto v]}
$$

Fig. 8. Sequential semantics (rules for `if` and `let` are omitted) — $\sigma, e \;\to\; \sigma', e'$

$$
\begin{array}{lllll}
\text{(unordered list of tasks)} & \mathit{Task} & ::= & (\mathsf{main} \mapsto e) \sqcup \overline{(\mathit{fut} \mapsto e)} \\
\text{(source configurations)} & c & ::= & \sigma, \mathit{Task}
\end{array}
$$

(a) Runtime configurations

$$
\begin{aligned}
\sigma_s &= \emptyset \\
\mathit{Task}_s &= (\mathsf{main} \mapsto e)
\end{aligned}
$$

(b) Initial configurations

Fig. 9. Source Configuration

$$
\textsc{step}\quad \frac{\sigma, e \;\to\; \sigma', e'}{\sigma, \mathit{Task} \;\sqcup\; (\mathit{fut} \mapsto e) \;\to\; \sigma', \mathit{Task} \sqcup (\mathit{fut} \mapsto e')}
$$

$$
\textsc{await-resolved}\quad \frac{(\mathit{fut}' \mapsto v) \in \mathit{Task}}{\sigma, \mathit{Task} \sqcup (\mathit{fut} \mapsto C[\texttt{await}(\mathit{fut}')]) \;\to\; \sigma, \mathit{Task} \sqcup (\mathit{fut} \mapsto C[v])}
$$

$$
\textsc{async-call}\quad \frac{(f \mapsto \lambda_{\mathsf{async}} x.\, e) \in \mathit{Fun} \qquad \mathit{fut}' \notin \mathrm{dom}(\mathit{Task}) \qquad \mathit{fut}' \neq \mathit{fut}}{\sigma, \mathit{Task} \sqcup (\mathit{fut} \mapsto C[(f\ v)]) \;\to\; \sigma, \mathit{Task} \sqcup (\mathit{fut}' \mapsto e[x \mapsto v]) \sqcup (\mathit{fut} \mapsto C[\mathit{fut}'])}
$$

Fig. 10. Global source reduction — $\sigma, \mathit{Task} \;\to\; \sigma', \mathit{Task}'$

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.

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 $[\, \mathit{fut} \mapsto \mathfrak{U}\ d\,]$

$$
\begin{array}{lrcl}
\text{(maps of futures)} & \mathit{Fut} & ::= & \mathbb{F} \xrightarrow{\text{fin}} \mathit{Dest} + \mathit{Val} \\
\text{(Tasks)} & \mathit{Task} & ::= & (\mathsf{main} \mapsto e) \sqcup \overline{(\mathit{tid} \mapsto e)} \\
\text{(target configurations)} & c & ::= & \sigma, \mathit{Task}, \mathit{Fut}
\end{array}
$$

(a) Runtime configuration

$$
\begin{aligned}
\sigma_t &= \emptyset \\
\mathit{Task}_t &= (\mathsf{main} \mapsto e) \\
\mathit{Fut}_t &= \emptyset
\end{aligned}
$$

(b) Initial configuration

Fig. 11. Target Configuration

STEP

$$\frac{\sigma, e \;\longrightarrow\; \sigma', e'}{\sigma, \mathit{Task} \;\sqcup\; (\mathit{tid} \mapsto e), \mathit{Fut} \;\longrightarrow\; \sigma', \mathit{Task} \sqcup (\mathit{tid} \mapsto e'), \mathit{Fut}}$$

AWAIT-TARGET

$$\frac{\mathit{Fut}\,[\,\mathit{fut}\,] = \mathfrak{R}\, v}{\sigma, \mathit{Task} \sqcup (\mathit{tid} \mapsto C[\mathtt{await}(\mathit{fut})]), \mathit{Fut} \;\longrightarrow\; \sigma, \mathit{Task} \sqcup (\mathit{tid} \mapsto C[v]), \mathit{Fut}}$$

CHAIN

$$\frac{\mathit{Fut}\,[\,\mathit{fut}\,] = \mathfrak{U}\, d \qquad \mathit{Fut}\,[\,\mathit{fut}'\,] = \mathfrak{R}\, v}{\sigma, \mathit{Task} \sqcup (\mathit{tid} \mapsto \mathtt{forward}(\mathit{fut}', d)), \mathit{Fut} \;\longrightarrow\; \sigma, \mathit{Task}, \mathit{Fut}\,[\,\mathit{fut} \mapsto \mathfrak{R}\, v\,]}$$

DPS-CALL

$$\frac{(f \mapsto \lambda_{\mathsf{dps}}\delta,\, x.\, e) \in \mathit{Fun} \qquad \mathit{fut}', d, \mathit{tid}' \text{ fresh}}{\begin{array}{l}\sigma, \mathit{Task} \sqcup (\mathit{tid} \mapsto C[(f\ v)]), \mathit{Fut} \;\longrightarrow \\ \sigma, \mathit{Task} \sqcup (\mathit{tid} \mapsto C[\mathit{fut}']) \sqcup (\mathit{tid}' \mapsto e[\delta \mapsto d][x \mapsto v]), \mathit{Fut}\,[\,\mathit{fut}' \mapsto \mathfrak{U}\, d\,]\end{array}}$$

TAIL-CALL

$$\frac{(f \mapsto \lambda_{\mathsf{dps}}\delta,\, x.\, e) \in \mathit{Fun} \qquad \mathit{tid}' \notin \mathrm{dom}(\mathit{Task})}{\sigma, \mathit{Task} \sqcup (\mathit{tid} \mapsto (f \#_t (d, v))), \mathit{Fut} \;\longrightarrow\; \sigma, \mathit{Task} \sqcup (\mathit{tid}' \mapsto e[x \mapsto v][\delta \mapsto d]), \mathit{Fut}}$$

FUTURE-FILL

$$\frac{\mathit{Fut}\,[\,\mathit{fut}\,] = \mathfrak{U}\, d}{\sigma, \mathit{Task} \sqcup (\mathit{tid} \mapsto \mathtt{refine}(d, v)), \mathit{Fut} \xrightarrow{\varepsilon} \sigma, \mathit{Task}, \mathit{Fut}\,[\,\mathit{fut} \mapsto \mathfrak{R}\, v\,]}$$

Fig. 12. Global target reduction — $\sigma, \mathit{Task}, \mathit{Fut} \;\longrightarrow\; \sigma', \mathit{Task}', \mathit{Fut}'$

when *fut* is yet unresolved, and tied to the destiny d; and $[\,\mathit{fut} \mapsto \mathfrak{R}\, v\,]$ 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 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 12. Local reductions can still be lifted to any task (STEP). AWAIT-TARGET handles futures synchronisation: `await`(*fut*) can only be evaluated if *fut* is resolved, i.e., if $\mathit{Fut}\,[\,\mathit{fut}\,] = \mathfrak{R}\, v$ for some value v.

CHAIN evaluates a `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 `await`: it blocks until fut' is resolved. However `forward` can only occur in tail position, as captured by the absence of an evaluation context in CHAIN. Thus tasks of the form ($\mathit{tid} \mapsto$ `forward`(*fut*, d)) are kept asleep

$$\text{(tail contexts)}\quad Tctx_n \ni C^{tl}_n ::= \square : Tctx_1 \mid \texttt{let}\, x = e\, \texttt{in}\, C^{tl}_n : Tctx_n \mid \texttt{if}\, e\, \texttt{then}\, C^{tl}_n\, \texttt{else}\, {C^{tl}}_m : Tctx_{n+m}$$

$$\square_1[e] := e$$

$$(\texttt{let}\, x = e\, \texttt{in}\, C^{tl}_n)[e_1 \mid \cdots \mid e_n] := \texttt{let}\, x = e\, \texttt{in}\, (C^{tl}_n[e_1 \mid \cdots \mid e_n])$$

$$(\texttt{if}\, e\, \texttt{then}\, C^{tl}_n\, \texttt{else}\, C^{tl}_m)[e_1 \mid \cdots \mid e_{n+m}] := \texttt{if}\, e\, \texttt{then}\, (C^{tl}_n[e_1 \mid \cdots \mid e_n])\, \texttt{else}\, (C^{tl}_m[e_{n+1} \mid \cdots \mid e_{n+m}])$$

(a) Tail contexts: definition and filling operation

$$Fun_s \Rightarrow Fun_t$$

$$f \mapsto \lambda x.\, e \rightsquigarrow f \mapsto \lambda x.\, e$$

$$f \mapsto \lambda_{\text{async}} x.\, e \rightsquigarrow f \mapsto \lambda_{\text{dps}} \delta,\, x.\, \mathcal{D}[\![e]\!]_\delta$$

(b) Transformation over function definitions

$$\mathcal{D}[\![\emptyset, T_{\text{main}}]\!] = \emptyset, T_{\text{main}}, \emptyset$$

(c) Transformation over initial configurations

$$\mathcal{D}[\![C^{tl}_n[e_1 \mid \cdots \mid e_n]]\!]_\delta ::= C^{tl}_n[\mathcal{D}[\![e_1]\!]_\delta \mid \cdots \mid \mathcal{D}[\![e_n]\!]_\delta]$$

$$\mathcal{D}[\![\texttt{await}(e)]\!]_\delta ::= \mathcal{D}_{\text{a}}[\![e]\!]_\delta$$

$$\mathcal{D}[\![e]\!]_\delta ::= \texttt{refine}(\delta,\, e) \quad \text{(otherwise)}$$

$$\mathcal{D}_{\text{a}}[\![C^{tl}_n[e_1 \mid \cdots \mid e_n]]\!]_\delta ::= C^{tl}_n[\mathcal{D}_{\text{a}}[\![e_1]\!]_\delta \mid \cdots \mid \mathcal{D}_{\text{a}}[\![e_n]\!]_\delta]$$

$$\mathcal{D}_{\text{a}}[\![(f\ e)]\!]_\delta ::= (f \#_t\ (\delta, e)) \quad \text{if } (f \mapsto \lambda_{\text{async}} x.\, e') \in Fun$$

$$\mathcal{D}_{\text{a}}[\![e]\!]_\delta ::= \texttt{forward}(e,\, \delta) \quad \text{(otherwise)}$$

(d) Transformation over expressions

Fig. 13. The Tail-Modulo-Await transformation — $\mathcal{D}[\![\cdot]\!]$

in the configuration until *fut* is resolved, but do not block pending computations. An optimised implementation could store such **forward**ed 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 promise by executing the called function. TAIL-CALL handles specifically calls in tail position: it passes the current destiny to the called function and hosts it in a new task, but furthermore kills the current task as we know there is nothing left to computer afterwards. Finally, FUTURE-FILL handles future resolution, killing the current task as well, as **refine** statements only exist in tail position.

2.4 The Tail-Modulo-Await Transformation

We can finally define the Tail-Modulo-Await transformation in Fig. 13. It is applied to all asynchronous functions $\lambda_{\text{async}} \cdot .$ in *Fun* (Fig. 13b). The rest of the program, and in particular the main thread, is not affected (Fig. 13c).

Its core is a recursive function over expressions $\mathcal{D}[\![\cdot]\!]_\delta$ where δ is a variable name bound to a destiny (Fig. 13d). It relies on the notion of *tail-context* (Fig. 13a) which captures in an expression the set of positions at which a call would be tail.

The function $\mathcal{D}[\![\cdot]\!]$ itself proceeds by pattern-matching, recursing through the program towards tail positions, leaving the rest unperturbed. If it reaches an **await**(e), it switches mode, now recursing e via $\mathcal{D}_{\text{a}}[\![\cdot]\!]$, until it reaches either an asynchronous function call to transform it into a tail-call, or any other expression to transform it into a **forward**. Finally, for other expressions e, $\mathcal{D}[\![\cdot]\!]$ emits **refine**$(\delta,\, e)$ to fill the current destiny.

2.5 Correctness of Tail-Modulo-Await

We prove that the `tma` transformation does not introduce new behaviors: if a source program p_s is transformed into a target program p_t, then any execution of p_t can be matched by an execution of p_s, leading to *similar* final states, and identical stores (i.e., p_t is a *behavioural refinement* of p_s). As is standard, we prove it by exhibiting a *backward simulation*. The simulation admits *stuttering* steps, capturing "administrative" steps occurring in the target language but matched by no step in the source. These steps exactly correspond to FUTURE-FILL, whose transitions are identified by an "ε". We hence need to ensure that a transformed program cannot take infinitely many ε steps in a row.

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

- if $c_t \xrightarrow{\varepsilon} c_t'$, then $c_s \succcurlyeq c_t'$
- if $c_t \longrightarrow c_t'$,, there exists c_s' such that $c_s \longrightarrow^+ c_s'$, and $c_s' \succcurlyeq c_t'$

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

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

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

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

2.5.1 Definition of $\succcurlyeq$. We define $\succcurlyeq$ 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 δ is substituted by d (EXPR-TRANSFO). This alignment can however be broken: the target may reduce to expressions like $\texttt{refine}(d, \texttt{await}(e))$, 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 $T_s \simeq T_t, \left[fut_0 \mapsto \mathfrak{R}\, v | \mathfrak{U}\, d \right]$ 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 $(fut_0 \mapsto \texttt{await}(fut_1)) \sqcup \cdots (fut_{n-1} \mapsto \texttt{await}(fut_n))$. We hence introduce the notation $\mathrm{Aw}[fut_0..fut_n]$ 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 ($\mathsf{main} \mapsto e$) 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

$$\textsc{expr-transfo}\ \frac{}{e \sim_d \mathcal{D}[\![e]\!]_\delta[\delta \mapsto d]} \qquad \textsc{expr-refine}\ \frac{}{e \sim_d \texttt{refine}(d, e)}$$

(a) Expression-level relation

$$\textsc{task-comp}\ \frac{e \sim_d e'}{\mathsf{Aw}[fut_0..fut_n] \sqcup (fut_n \mapsto e) \simeq (tid \mapsto e'), \left[\, fut_0 \mapsto \mathfrak{U}\, d \,\right]} \qquad \textsc{task-val}\ \frac{}{\overline{(fut_i \mapsto v)}^{0 \le i \le n} \simeq \emptyset, \left[\, fut_0 \mapsto \mathfrak{R}\, v \,\right]}$$

$$\textsc{task-await-val}\ \frac{}{\mathsf{Aw}[fut_0..fut_n] \sqcup (fut_n \mapsto v) \simeq \emptyset, \left[\, fut_0 \mapsto \mathfrak{R}\, v \,\right]}$$

(b) Task-level relation

$$\sigma_s, Task_s \succcurlyeq \sigma_t, Task_t, Fut_t \quad \Leftrightarrow \quad \begin{cases} (1) & \sigma_s = \sigma_t \\ (2) & Task_s = (\mathsf{main} \mapsto e) \sqcup T_s^{(1)} \sqcup \cdots T_s^{(n)} \\ (3) & Task_t = (\mathsf{main} \mapsto e) \sqcup T_t^{(1)} \sqcup \cdots T_t^{(n)} \\ (4) & Fut_t = \left[\, \overline{fut_i \mapsto \mathcal{F}_i}^{1 \le i \le n} \,\right] \qquad (\mathcal{F}_i = \mathfrak{U}\, d_i \mid \mathfrak{R}\, v_i) \\ (5) & \forall\, i, T_s^{(i)} \simeq T_t^{(i)}, \left[\, fut_i \mapsto \mathcal{F}_i \,\right] \\ (6) & \forall\, i \neq j,\ d_i \neq d_j \qquad \text{(if defined)} \\ (7) & \forall i, fut \in \mathrm{dom}(T_s^{(i)}) \setminus \{fut_i\}, \begin{cases} fut \notin \mathrm{Im}(\sigma) \\ \forall j \neq i,\ fut \notin T_s^{(j)} \end{cases} \end{cases}$$

(c) Configuration-level relation

Fig. 14. Simulation relation

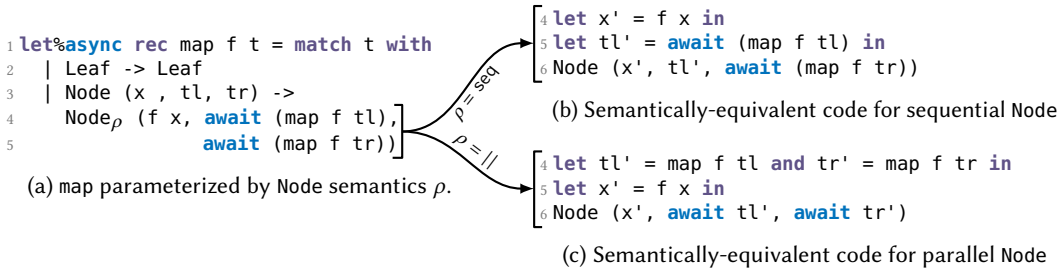
steps taken in the corresponding pool. We then distinguish two broad cases: a step in `main` must be exactly mirrored, while a step in pool $T_t^{(i)}$ must be matched in $T_s^{(i)}$ and restore (5) for index i. We refer the reader to Appendix C for the full case analysis, and detail here two informative 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.

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.

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

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.



(a) `map` parameterized by `Node` semantics ρ.

(b) Semantically-equivalent code for sequential `Node`

(c) Semantically-equivalent code for parallel `Node`

Fig. 15. A parametric `map` on trees

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

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

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 $c_s \succcurlyeq c_t$, 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 $\mathsf{Aw}[fut_0..fut_n] \sqcup (fut_n \mapsto v)$, 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, we consider two possible semantics for constructors in the presence of **`async`**/**`await`**. Constructors tagged as sequential ensure their arguments are evaluated in sequence, while those tagged *parallel* maximise parallelism. These different behaviours are illustrated on Fig. 15, which features the `map` on trees from Fig. 1, but where the final `Node` constructor is parametrised by a tag ρ. We capture the meaning of this tag by translating `map` into two programs semantically equivalent to the original one, depending on the value of ρ, but whose order of evaluation is made explicit through let bindings. Beware, this is *not* the result Tail-Modulo-Cons-Await transformation, which aims to pre-allocate the constructor and avoid intermediate task creations, as showcased in Section 1.

For ρ = seq, 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. [3].

For ρ = ||, evaluation can maximise parallelism, as showcased in Fig. 15c. The semantics is equivalent to first launching both sub-tasks (here, `map f tl` and `map f tr`), and then the sequential code (`f x`), hence removing any synchronisation between the tasks.

$$
\begin{array}{llll}
\text{(expressions)} & \mathit{Expr} \ni e & ::= & \ldots \\
& & & \mid \texttt{\{}\, t, e, e \,\texttt{\}} \mid \texttt{[}\, t, e, e \,\texttt{]} \\
& & & \mid e\texttt{.(}e\texttt{)} \mid e\texttt{.(}e\texttt{)} \leftarrow e \\
& & & \mid \blacksquare \\
& & & \mid \texttt{refine}(e, e)
\end{array}
\qquad
\begin{array}{llll}
\text{(values)} & \mathit{Val} \ni v & ::= & \ldots \mid d, \ell \in \mathit{Loc} \\
& & & \mid t \in \mathit{Tag} \mid i \in \mathit{Index} \\
& & & \mid \diagup \\
\text{(tags)} & \mathit{Tag} \ni t & ::= & t_{\text{seq}} \mid t_{||} \\
\text{(offsets)} & \mathit{Index} \ni i & ::= & \{0, 1, 2\}
\end{array}
$$

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.

$$
\begin{array}{llll}
\text{(store)} & \sigma & ::= & \ldots \\
\text{(eval contexts)} & \mathit{Ectx} \ni C & ::= & \cdots \mid \texttt{[}\, t, C, e \,\texttt{]} \mid \texttt{[}\, t, v, C \,\texttt{]} \mid \texttt{[}\, t, \blacksquare, C \,\texttt{]} \mid e\texttt{.(}C\texttt{)} \mid C\texttt{.(}v\texttt{)} \\
& & & \mid e\texttt{.(}e\texttt{)} \leftarrow C \mid e\texttt{.(}C\texttt{)} \leftarrow v \mid C\texttt{.(}v\texttt{)} \leftarrow v \\
& & & \mid \texttt{refine}(e, C)
\end{array}
$$

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

$$\texttt{sta}(C_n^{tl}[\texttt{await}(e_1) \mid \cdots \mid \texttt{await}(e_n)]) ::= C_n^{tl}[e_1 \mid \cdots \mid e_n]$$

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

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.

3.1 Calculus

To model data-constructors, we add binary constructors to the language as tagged mutable memory blocks $\texttt{\{}\, t, e_1, e_2 \,\texttt{\}}$, with notations and semantics inspired by Allain et al.'s formalization of Tail-Modulo-Cons [3], but with the additional *sequential* (t_{seq}) and *parallel* ($t_{||}$) discussed above. As in [3], we have a separate, runtime-only block construction $\texttt{[}\, t, e_1, e_2 \,\texttt{]}$, 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 $\ell\texttt{.(}i\texttt{)}$, 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: $\texttt{refine}(d, v)$ takes a value v which may have some *holes*—syntactic markers (written $\blacksquare$) to denote 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.

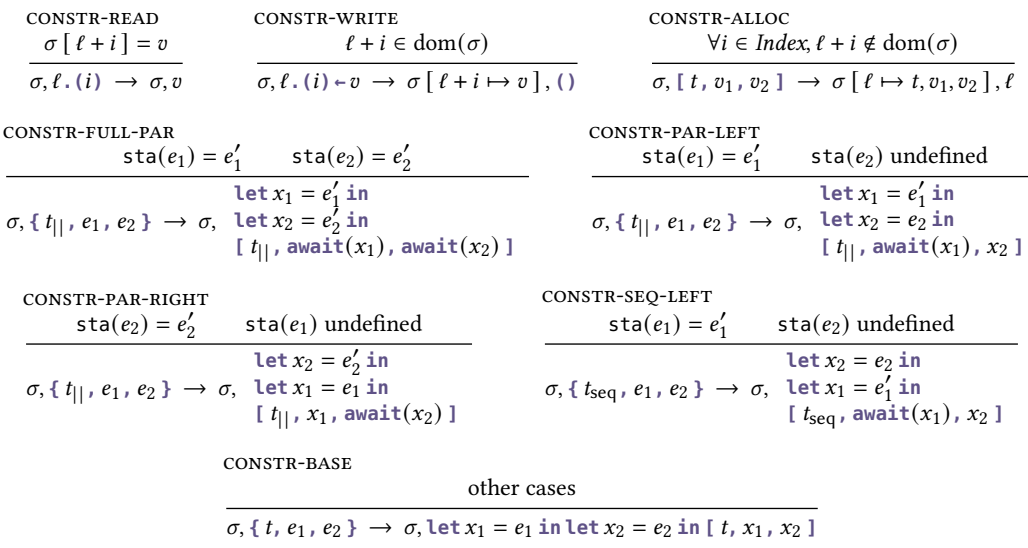


Fig. 19. Sequential semantics for constructors — $\sigma, e \;\to\; \sigma', e'$ — Extends Fig. 8

To specify their reduction, we need to capture expressions of shape $C_n^{tl}[\mathtt{await}(e_1) \mid \cdots \mid \mathtt{await}(e_n)]$ 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 `await`s, 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 $e' = \mathtt{sta}(e) = C_n^{tl}[e_1 \mid \cdots \mid e_n]$, then e is semantically equivalent to first binding *fut* to e', launching a parallel task, and then evaluating $\mathtt{await}(\mathit{fut})$. Conversely, if $\mathtt{sta}(e)$ 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_{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 $\mathtt{holes} : \ell \to \mathbb{N}$ (see Section 4 for an efficient implementation) which returns the number of locations set to $\nearrow$ in the tree rooted at ℓ.

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

REFINE-PAR

$$\frac{\sigma\,[\,d\,] = \diagup \qquad \ell, \ell+1, \ell+2 \notin \mathrm{dom}(\sigma) \qquad \sigma' = \sigma\left[\, d \mapsto \ell, \ell \mapsto t_{||}, \ell+1, \ell+2 \mapsto \diagup \,\right]}{\sigma, \texttt{refine}(d, \texttt{[}\, t_{||}\texttt{,}\ \blacksquare\texttt{,}\ \blacksquare\ \texttt{]}) \ \to\ \sigma', (\ell+1, \ell+2)}$$

REFINE-SEQ-RIGHT

$$\frac{\sigma\,[\,d\,] = \diagup \qquad \ell, \ell+1, \ell+2 \notin \mathrm{dom}(\sigma) \qquad \sigma' = \sigma\left[\, d \mapsto \ell, \ell \mapsto t_{\mathrm{seq}}, \ell+1 \mapsto v_1, \ell+2 \mapsto \diagup \,\right]}{\sigma, \texttt{refine}(d, \texttt{[}\, t_{\mathrm{seq}}\texttt{,}\ v_1\texttt{,}\ \blacksquare\ \texttt{]}) \ \to\ \sigma', (\ell+2)}$$

REFINE-SEQ-LEFT

$$\frac{\sigma\,[\,d\,] = \diagup \qquad \ell, \ell+1, \ell+2 \notin \mathrm{dom}(\sigma) \qquad \sigma' = \sigma\left[\, d \mapsto \ell, \ell \mapsto t_{\mathrm{seq}}, \ell+1 \mapsto \diagup, \ell+2 \mapsto v_2 \,\right]}{\sigma, \texttt{refine}(d, \texttt{[}\, t_{\mathrm{seq}}\texttt{,}\ \blacksquare\texttt{,}\ v_2\ \texttt{]}) \ \to\ \sigma', (\ell+1)}$$

FILL

$$\frac{\sigma\,[\,d\,] = \diagup}{\sigma, \texttt{refine}(d, v) \xrightarrow{\varepsilon} \sigma\,[\,d \mapsto v\,], \texttt{()}}$$

Fig. 20. Sequential semantics for `refine`

CHAIN-UNRESOLVED

$$\frac{Fut\,[\,fut\,] = \mathfrak{U}\,d' \qquad C \neq \square \qquad tid' \text{ fresh}}{\sigma, Task \sqcup (tid \mapsto C[\texttt{forward}(fut, d)]), Fut \xrightarrow{\varepsilon} \sigma, Task \sqcup (tid \mapsto C[\texttt{()}]) \sqcup (tid' \mapsto \texttt{forward}(fut, d)), Fut}$$

CHAIN-RESOLVED

$$\frac{Fut\,[\,fut\,] = \mathfrak{R}\,v}{\sigma, Task \sqcup (tid \mapsto C[\texttt{forward}(fut, d)]), Fut \ \to\ \sigma[\,d \mapsto v\,], Task \sqcup (tid \mapsto C[\texttt{()}]), Fut}$$

DPS-CALL

$$\frac{(f \mapsto \lambda_{\mathrm{dps}}\delta, x.\, e) \in Fun \qquad fut', d, tid' \text{ fresh}}{\begin{array}{c}\sigma, Task \sqcup (tid \mapsto C[(f\ v)]), Fut \ \to \\ \sigma[\,d \mapsto \diagup\,], Task \sqcup (tid \mapsto C[fut']) \sqcup (tid' \mapsto e[\delta \mapsto d][x \mapsto v]), Fut\left[\, fut' \mapsto \mathfrak{U}\,d\,\right]\end{array}}$$

TAIL-CALL

$$\frac{(f \mapsto \lambda_{\mathrm{dps}}\delta, x.\, e) \in Fun \qquad tid' \text{ fresh}}{\begin{array}{lcl}\sigma, & & \sigma, \\ Task \sqcup (tid \mapsto C[(f\ \#_t\ (d, v))]), & \to & Task \sqcup (tid \mapsto C[\texttt{()}]) \sqcup (tid' \mapsto e[x \mapsto v][\delta \mapsto d]), \\ Fut & & Fut\end{array}}$$

FUTURE-RESOLVE

$$\frac{Fut\,[\,fut\,] = \mathfrak{U}\,d \qquad \mathtt{holes}(d) = 0 \qquad \sigma\,[\,d\,] = v}{\sigma, Task, Fut \xrightarrow{\varepsilon} \sigma \setminus d, Task, Fut\,[\,fut \mapsto \mathfrak{R}\,v\,]}$$

PRUNE-TASKS

$$\frac{}{\sigma, Task \sqcup (tid \mapsto \texttt{()}), Fut \ \to\ \sigma, Task, Fut}$$

Fig. 21. Global Target reduction — $\sigma, Task, Fut \to \sigma', Task', Fut'$ — Extends Fig. 12.
Greyed out rules are similar to `tma`, with the modified parts highlighted in *violet*.

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 $\diagup$ in that case—, sets the destiny argument to the corresponding value, and returns all locations initialized at $\diagup$.

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 *violet*. 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 $\Re$ 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 `await`s 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_{seq} case, the transformation is not recursively called on sub-expressions as there is only one tail-position.

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

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 ε steps is therefore now contextual, rather than purely hardcoded in the semantics.

3.4.1 Definition of $\succcurlyeq$. Figs. 23 and 24 introduce graphical representations for respectively source and target configurations in relation: we take this informal graphical depiction as definition for $\succcurlyeq$. 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 (σ), 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

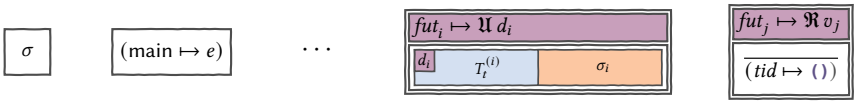


Fig. 24. Target configurations

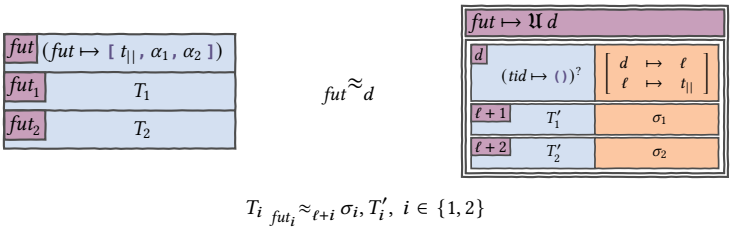


Fig. 25. The last case of the relation for parallel constructors

the target, this future fut_i maps to an unresolved destiny d_i, it owns the share of store σ_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 $T_s^{(i)} \; \underset{fut_i}{\approx}_{d_i} \; \sigma_i, T_t^{(i)}$, 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.[5] 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 $\alpha_i = \texttt{await}(fut_i)$ 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 $c_s = \begin{matrix}\sigma_s,\\ Task_s \sqcup T_s\end{matrix} \succcurlyeq \begin{matrix}\sigma_t \sqcup s_t,\\ Task_t \sqcup T_t,\\ Fut_t\end{matrix} = c_t$, s.t. $T_s \; \underset{fut}{\approx}_d \; s_t, T_t$, then:

[5] In the case of sequential constructors, at most one subtask is launched.

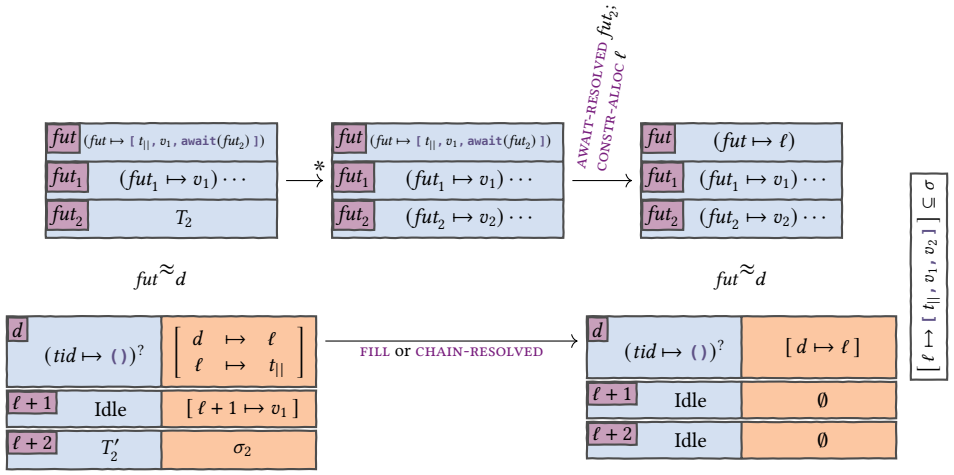


Fig. 26. Sub-configurations before a synchronisation step. Idle denotes a possibly empty list of threads all reduced to ()

Lemma 3.1. *If we have*

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

then there exist T'_s, σ'_s *(and maybe* T_s^{new}*) such that*

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

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 $\texttt{holes}(d) = 1$, 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 $\texttt{holes}(d) = 0$ associated to related pools $T_s \ {}_{fut}{\approx}_d \ \sigma_t T_t$, 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 $T_s \ {}_{fut}{\approx}_d \ s_t, T_t$.

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 $\texttt{holes}(d) = 1$, 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 $T_2 \ {}_{fut_2}{\approx}_{\ell+2} \ \sigma_2, T'_2.$, with $\texttt{holes}(\ell + 2) = 1$. 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 $\texttt{await}(fut_2)$ 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 σ_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. [3], 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.

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

While semantically fine, the destiny δ_l is unnecessary: it is immediately filled by the allocated block $\texttt{[}\, B_{||}\texttt{,}\ \blacksquare\texttt{,}\ \blacksquare\ \texttt{]}$. 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. [3], we emit the following code:

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

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 in OCaml. The implementation is available in the joint anonymous material (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.[6] An overview of the runtime API is provided on Figs. 28 and 30, and the actual translation of `map` on trees on Fig. 29.

[6]Most existing library would satisfy our needs, we use Picos (https://github.com/ocaml-multicore/picos) for its clear semantics and to allow testing with different schedulers

$$\mathcal{D}[\![C_n^{tl}[e_1 \mid \cdots \mid e_n]]\!]_\delta ::= C_n^{tl}[\mathcal{D}[\![e_1]\!]_\delta \mid \cdots \mid \mathcal{D}[\![e_n]\!]_\delta]$$

$$\mathcal{D}[\![\texttt{await}(e)]\!]_\delta ::= \mathcal{D}_{\text{a}}[\![e]\!]_\delta$$

$$\mathcal{D}[\![e]\!]_\delta ::= \begin{array}{ll} \texttt{let}\ \overline{\delta_k} = \texttt{refine}(\delta,\ c)\ \texttt{in} & \\ \overline{\mathcal{D}[\![e_k]\!]_{\delta_k}}; & \forall k.\texttt{sta}(e_k)\ \text{defined} \\ \overline{\mathcal{D}[\![e_k]\!]_{\delta_k}}; & \forall k.\texttt{sta}(e_k)\ \text{not defined} \end{array} \qquad \text{where } \overline{e_k}, c = \mathcal{D}_{\text{cons}}[\![e]\!]$$

$$\frac{\overline{e_i}^i, c = \mathcal{D}_{\text{cons}}[\![e]\!] \qquad \overline{e'_j}^j, c' = \mathcal{D}_{\text{cons}}[\![e]\!]}{\mathcal{D}_{\text{cons}}[\![\{\ t_{||}\ ,\ e,\ e'\ \}]\!] ::= \overline{e_i}^i; \overline{e'_j}^j,\ \{\ t_{||}\ ,\ c,\ c'\ \}}$$

$$\mathcal{D}_{\text{cons}}[\![c]\!] ::= \emptyset, c \qquad \mathcal{D}_{\text{cons}}[\![x]\!] ::= \emptyset, x \qquad \mathcal{D}_{\text{cons}}[\![e]\!] ::= e, \blacksquare$$

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

```
module Loc : sig
(** Location inside an OCaml memory block,
    composed of a value and an offset. *)
type 'h t = { pointer : Obj.t ; offset : int }

(** An heterogeneous list of locations *)
type 'a list =
  | [] : unit list
  | (::) : 'h t * 'h2 list -> ('h * 'h2) list

(** A dummy value to insert into blocks. *)
val dummy : 'a

(** [Loc.mk o i] is the location in value [o]
    at offset [i]. [o] must b a pointer to a
    block. *)
val mk : 'a -> int -> 'b t
end
```

Fig. 28. Runtime library for locations

```
let rec map$dps d4 f t = match t with
  | Leaf -> Destiny.refine d4 Leaf []
  | Node (v, tl, tr) ->
    let v5 = Node(Loc.dummy,Loc.dummy,Loc.dummy) in
    let [d6; d7; d8] =
      Destiny.refine d4 v5
        [Loc.mk v5 0; Loc.mk v5 1; Loc.mk v5 2]
    in
    fork (fun () -> map$dps d8 f tr);
    fork (fun () -> map$dps d7 f tl);
    Destiny.refine d6 (f v) []

let map f l =
  let d, c = Destiny.mk () in
  fork (fun () -> map$dps d f l);
  c
```

Fig. 29. Real translation of parallel `map` from Fig. 1

```
(** Launch a new task. *)
val fork : (unit -> unit) -> unit

module Destiny : sig
(** A Destiny with a hole of type ['a] *)
type 'a t = {
  counter : int Atomic.t; (* Number of holes *)
  resolve : unit -> unit; (* Top-level resolve *)
  set : 'a -> unit; (* Set the current destiny *)
}

(** An heterogeneous list of destinies *)
type 'a list =
  | [] : unit list
  | (::) : 'h t * 'h2 list -> ('h * 'h2) list

(** [mk ()] returns a synchronised destiny and
    future. The future will be fulfilled only
    when all holes in the destiny are filled. *)
val mk : unit -> 'a t * 'a Future.t

(** [refine d v hs] fills the destiny [d] with a
    value [v] with holes [hs]. It returns a list
    of destinies corresponding to each holes.
    [d] should not be used again. *)
val refine : 'b t -> 'b -> 'a Loc.list -> 'a list

(** [forward c d] will fill the destiny [d] with
    the value of the asynchronous computation [c]
    when it is finished. *)
val forward : 'a Future.t -> 'a t -> unit
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.

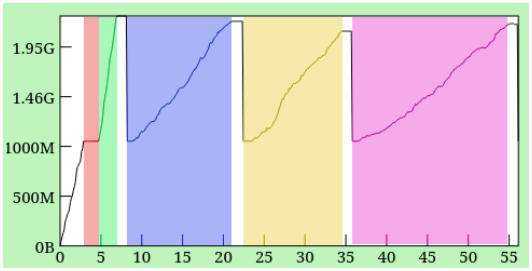
To implement DPS calls (Line 9, 10 and 15 in Fig. 29), we use a regular `fork` function which takes a callback containing the actual call. The call is syntactically a tail-call, ensuring that OCaml's compiler uses the desired call convention. Finally, for the purpose of this implementation we consider 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 a Thinkpad T470 equipped with i5-7200U with 4 cores and 11GiB of memory. All the implementations evaluated are provided in Appendix B. We explore the following scientific questions:

What is the effect of Tail-Modulo-Cons-Await on memory consumption? Evaluating stack and scheduler space precisely is quite difficult. We settle for a lesser goal: to evaluate the total live memory during the execution of a program. More specifically, we consider the execution of several variants of `List.map` on a list of 50000000 elements. The implementations (see Fig. 36 in Appendix B) are: a naive non-tail-rec `List.map`, a version using Tail-Modulo-Cons, an asynchronous version with a sequential function parameter, an asynchronous version with a sequential constructor, and an asynchronous version with a parallel constructor. For more details, see Appendix A. The overall memory footprint of the overall trace is shown in Fig. 31. There is only one run, since memory footprint is deterministic. The `List.map` runs out of stack before allocating anything, and crashes. All the other functions complete their execution while consuming the exact same amount of memory, which corresponds exactly to an allocation of the output list. Neither stack nor scheduler space is used in this execution. The synchronous version using Tail-Modulo-Cons runs faster than the asynchronous version, which is expected since it doesn't involve a scheduler. This confirms that Tail-Modulo-Cons-Await behaves as advertised and avoids superfluous stack and scheduler space.

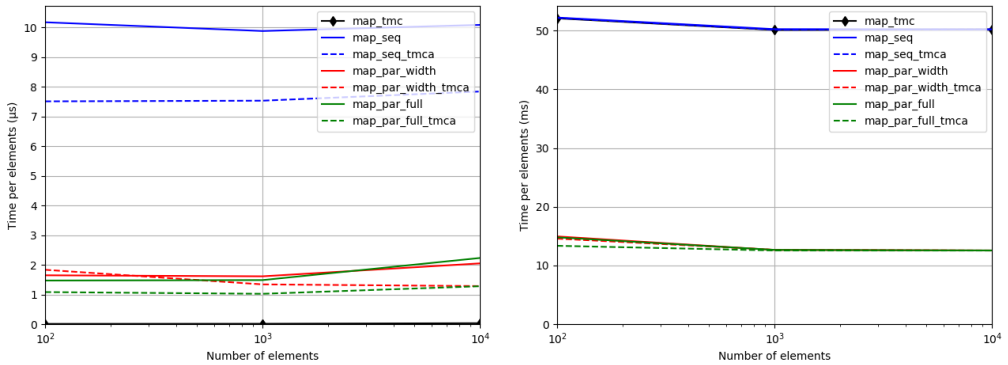
What is the effect of Tail-Modulo-Cons-Await on performances and parallelism? To evaluate the effect of Tail-Modulo-Cons-Await on time, we run several variants of `map` on randomly generated trees of growing size. Figure 32 shows the *time per elements*, i.e., the total time divided by the number of elements, for `Tree.map f`. We consider two distinct `f` functions which are applied at each `Node`: a very fast function on the left (multiply by 2) and a slow function (sleep a random amount between 0 and 50ms). The evaluated `map` implementations (see Fig. 37 in Appendix B) are as follows: `map_tmc` is a synchronous sequential version which uses Tail-Modulo-Cons (with only one recursive tail call); all the others are asynchronous, with and without Tail-Modulo-Cons-Await: `map_seq` uses a



Order of execution:

- Creation of the list of 50000000 elements
- `map_naive`: Not tail-recursive
- `map_tmc`: With Tail-Modulo-Cons
- `map_async_sync`: Synchronous function f
- `map_async_seq`: Sequential constructor `::`
- `map_async_par`: Parallel constructor `::`

Fig. 31. Memory measurement of several version of `List.map`. Global Memory footprint vs. Time elapsed since start of execution in seconds.



(a) With `f = fun x -> x*.2.`. Time in μs.

(b) With `f = Unix.sleepf`. Time in ms.

Fig. 32. Time measurement for several versions of `Tree.map`. Lower is better.

sequential semantics `Node` (as in Fig. 15b); `map_par_full` uses a full *parallel* semantics (as in Fig. 15c); `map_par_width` also uses a *parallel* semantics, but applies `f` before any recursive calls, thus bounding the parallelism by *width* (see Section 1). The function compiled with Tail-Modulo-Cons-Await are marked with `_tmca` and are represented using dashed lines. The scheduler is allowed to spawn tasks on all four available processors. All time measurements are obtained with repeated runs until a correlation of $r^2 > 0.8$ is obtained.

First, we can observe that the parallel semantics is, indeed, parallel! Regardless of the function `f` used, the parallel versions are exactly 4 times faster than the sequential versions.

Second, let us look at raw performance. With the fast `f` function, we can observe that Tail-Modulo-Cons-Await consistently improves performances, especially on large trees, although never to the point of equating the performance of the synchronous version. The cost of the slow function completely dominates any of the costs introduced by the implementation of `map`, and the cohort are separated into two groups: the sequential and the parallel versions.

5 Related Work

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 [15], then Friedman and Wise [11] 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 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. [3] 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 [12], 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 gives guarantees on the fact that futures are eventually fulfilled. On the contrary, promises associate their fulfilment with a handler given to the programmer. Hence, they do not suffer from this limitation, but it cannot be ensured that a promise is resolved exactly once [1]. The fact that promises allow for the optimisation of scheduler space compared to futures was already the key idea of the `forward` construct [8]. In this work, `forward` promises are only used internally for optimisation purposes, as we do in the work presented in this paper. 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 [8] with more automation, added parallelism, and more general application setting.

Dataflow synchronisation on futures [9] 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. [6] 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 (also by replacing futures by promises) and in which the `tmaa` approach should be applicable to make the approach more general. Unfortunately dataflow synchronisation on futures implies a type system where `'a future` unifies with `('a future) future`. Such typing rules, where monadic flatten is an axiom, are particularly difficult to integrate in ML-like languages such as OCaml.

Other Asynchronous Paradigms. Futures are massively used in actors and active object languages [2, 7]: this family of languages thus offers a privileged application domain for all optimisations of asynchronous calls, notably the one we propose. The fact that, by nature, all calls to an actor create a future multiplies the number of futures, often with tail-calls, and hence makes future optimisation crucial in these languages. Indeed `forward` was created in an active object language, and the present work would clearly benefit languages with prominent Actors libraries [2, 13, 14], including the recently designed OCaml active object library [4].

Our parallel constructors could also be compared to the `parT` construct of the Encore language [10] or the `par` construct of Haskell [18, 19]. 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 fits well with the notion of futures and promises, but approaches based on parallel combinator features richer synchronisation patterns compared to our work. We might study if it would be possible to integrate a few 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).

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] 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

[4] 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

[5] 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.

[6] 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

[7] 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

[8] Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, and Huu-Phuc Vo. 2018. Forward to a Promising Future. In *Conference proceedings COORDINATION 2018.*

[9] 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.

[10] 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 Proença (Eds.). Vol. 9686. 101–120.

[11] Daniel P Friedman and David S Wise. 1975. Unwinding stylized recursions into iterations. *Comput. Sci. Dep., Indiana University, Bloomington, IN, Tech. Rep* 19 (1975).

[12] 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

[13] 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.

[14] 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

[15] Tore Risch. 1973. REMREC - A program for Automatic Recursion Removal. https://api.semanticscholar.org/CorpusID:60315470

[16] 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

[17] 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

[18] 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), Philadephia, Pennsylvania, USA, May 21-24, 1996*, Charles N. Fischer (Ed.). ACM, 79–88. doi:10.1145/231379.231392

[19] 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 asynhcronous function over a list, returning an asynchronous computation itself. We first consider in Fig. 33 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. 33b.

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

(a) Source implementation

```
let rec map_async_sync_dps d f l =
  match l with
  | Nil -> d ◀ Nil
  | Cons (x, xs) ->
    let v = Cons (f x, ■) in
    d ◀ v;
    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. 33. 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. 34. The Tail-Modulo-Cons-Awaittransformation affects this version in the same way as ths version in Fig. 33.

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

(a) Source implementation

```
let rec map_async_seq_dps d f l =
  match l with
  | Nil -> d ◀ Nil
  | Cons (x, xs) ->
    let y = await (f x) in
    let v = Cons (y, ■) in
    d ◀ v;
    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. 34. An asynchronous, sequential, `map` on lists with async/await

Finally, the parallel version is illustrated in Fig. 35. The transformed code in Fig. 35b uses `forward` for the call to `f`, rather than `fork_dps`, as `f` might be an external asynchronous function without a dps version.

```
let%async rec map_async_par f l =
  match l with
  | Nil -> Nil
  | Cons (x, xs) ->
    Cons (await (f x),
          await (map_async_par f xs))
```

(a) Source implementation

```
let rec map_async_par_dps d f l =
  match l with
  | Nil -> d ◂ Nil
  | Cons (x, xs) ->
    let v = Cons (■, ■) in
    d ◂ v;
    let fut = f x in
    forward v.0 fut;
    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. 35. An asynchronous, parallel, `map` on lists with async/await

B Supporting code for the Experimental Evaluation

```
let rec map_naive f = function
  | [] -> []
  | h :: t -> f h :: map_naive f t

let[@tail_mod_cons] rec map_tmc f l =
  match l with
  | [] -> []
  | x :: xs ->
    let y = f x in
    y :: map_tmc f xs

let%async rec map_async_sync f l =
  match l with
  | [] -> []
  | x :: xs ->
    f x :: await (map_async_sync f xs)

let%async rec map_async_seq f l =
  match l with
  | [] -> []
  | x :: xs ->
    let y = await (f x) in
    y :: await (map_async_seq f xs)

let%async rec map_async_par f l =
  match l with
  | [] -> []
  | x :: xs ->
    await (f x) :: await (map_async_par f xs)
```

Fig. 36. Implementation of map on lists

```
type 'a tree = N of 'a * 'a tree * 'a tree | L

let[@tail_mod_cons] rec map_tmc f t = match t with
  | L -> L
  | N (v , tl , tr ) ->
    let y = f v in
    let tl' = map_tmc f tl in
    N (y, tl', map_tmc f tr)

let rec map_seq f t =
  async @@ fun () -> match t with
  | L -> L
  | N (v , tl , tr ) ->
    let y = f v in
    let tl' = await (map_seq f tl) in
    let tr' = await (map_seq f tr) in
    N (y, tl', tr')

let rec map_par_width f t =
  async @@ fun () -> match t with
  | L -> L
  | N (v , tl , tr ) ->
    let y = f v in
    let task_tl = map_par_width f tl in
    let task_tr = map_par_width f tr in
    N (y, await task_tl, await task_tr)

let rec map_par_full f t =
  async @@ fun () -> match t with
  | L -> L
  | N (v , tl , tr ) ->
    let task_tl = map_par_full f tl in
    let task_tr = map_par_full f tr in
    N (f v, await task_tl, await task_tr)

let%async rec map_seq_tmca f t = match t with
  | L -> L
  | N (v , tl , tr ) ->
    let y = f v in
    let tl' = await (map_seq_tmca f tl) in
    N (y, tl', await (map_seq_tmca f tr))

let%async rec map_par_width_tmca f t = match t with
  | L -> L
  | N (v , tl , tr ) ->
    let y = f v in
    N (y,
       await (map_par_width_tmca f tl),
       await (map_par_width_tmca f tr))

let%async rec map_par_full_tmca f t = match t with
  | L -> L
  | N (v , tl , tr ) ->
    N (f v,
       await (map_par_full_tmca f tl),
       await (map_par_full_tmca f tr))
```

Fig. 37. Implementation of 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:

$$\begin{array}{lcl} Task_s \sqcup (\mathsf{main} \mapsto C[(f\ v)]) & \cong_D & Task_t \sqcup (\mathsf{main} \mapsto C[(f\ v)]), Fut_t \\ \downarrow_{\text{ASYNC-CALL}}^{(f \mapsto \lambda_{\text{async}} x.\, e) \in Fun} & & \downarrow_{\text{DPS-CALL}}^{(f \mapsto \lambda_{\text{dps}} \delta, x.\, \mathcal{D}[\![e]\!]_\delta) \in Fun'} \\ Task_s \sqcup (\mathsf{main} \mapsto C[fut']) \sqcup (fut' \mapsto e[x \mapsto v]) & \cong_{D \sqcup \{d\}} & Task_t \sqcup (\mathsf{main} \mapsto C[fut']) \sqcup (tid \mapsto \mathcal{D}[\![e]\!]_\delta[\delta \mapsto d][x \mapsto v]), \\ & & Fut_t \sqcup [\, fut' \mapsto \mathfrak{U}\, d\,] \end{array}$$

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

AWAIT-TARGET: The challenge is:

$$\begin{array}{lcl} Task_s \sqcup (\mathsf{main} \mapsto C[\mathtt{await}(fut)]) & \cong_D & Task_t \sqcup (\mathsf{main} \mapsto C[\mathtt{await}(fut)]), Fut_t \\ & & \downarrow_{Fut_t\,[\,fut\,] = \mathfrak{R}\, v}^{\text{AWAIT-TARGET}} \\ & & Task_t \sqcup (\mathsf{main} \mapsto C[v]), Fut_t \end{array}$$

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:

$$\begin{array}{lcl} Task_s \sqcup (\mathsf{main} \mapsto C[\mathtt{await}(fut)]) & \cong_D & Task_t \sqcup (\mathsf{main} \mapsto C[\mathtt{await}(fut)]), Fut_t \\ \downarrow_{\text{AWAIT-RESOLVED}} & & \downarrow_{Fut_t\,[\,fut\,] = \mathfrak{R}\, v}^{\text{AWAIT-TARGET}} \\ Task_s \sqcup (\mathsf{main} \mapsto C[v]) & \cong_D & Task_t \sqcup (\mathsf{main} \mapsto C[v]), Fut_t \end{array}$$

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

$$\begin{array}{lcl} Task_s \sqcup (\mathsf{main} \mapsto C[\mathtt{await}(fut)]) & \cong_D & Task_t \sqcup (\mathsf{main} \mapsto C[\mathtt{await}(fut)]), Fut_t \\ \downarrow_{\text{AWAIT-RESOLVED}}^{i=n..0} & & \downarrow_{Fut_t\,[\,fut\,] = \mathfrak{R}\, v}^{\text{AWAIT-TARGET}} \\ Task_s \sqcup (\mathsf{main} \mapsto C[v]) & \cong_D & Task_t \sqcup (\mathsf{main} \mapsto C[v]), Fut_t \end{array}$$

We are left with $T_s^{(j)\prime} = (fut \mapsto v) \sqcup \overline{(fut_1 \mapsto v)}^{1 \le i \le n}$, 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:

$$\begin{array}{lcl} Task_s \sqcup (fut \mapsto C_s[\texttt{await}(fut_r)]) & \cong_{D \sqcup \{d\}} & Task_t \sqcup (tid \mapsto C_t[\texttt{await}(fut_r)]), Fut_t \\ \Big\downarrow {\scriptstyle \text{AWAIT-RESOLVED} \atop (fut_r \mapsto v) \in Task_s\,(*)} & & \Big\downarrow {\scriptstyle \text{AWAIT-RESOLVED} \atop Fut_t\,[\,fut_r\,] = \mathfrak{R}\,v} \\ Task_s \sqcup (fut_0 \mapsto C_s[v]) & \cong_{D \sqcup \{d\}} & Task_t \sqcup (tid \mapsto C_t[v]), Fut_t \end{array}$$

$(*)$ 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.

$$\begin{array}{lcl} Task_s \sqcup \mathsf{Aw}[fut_0..fut_n] \sqcup (fut_n \mapsto \texttt{await}(fut_r)) & \cong_{D \sqcup \{d\}} & Task_t \sqcup (tid \mapsto \texttt{forward}(fut_r,\, d)), \\ & & Fut_t \sqcup [\,fut_0 \mapsto \mathfrak{U}\,d\,] \\ & & \Big\downarrow {\scriptstyle \text{CHAIN} \atop Fut_t\,[\,fut_r\,] = \mathfrak{R}\,v} \\ Task_s \sqcup \mathsf{Aw}[fut_0..fut_n] \sqcup (fut_n \mapsto v) & \cong_D & Task_t, Fut_t \sqcup [\,fut_0 \mapsto \mathfrak{R}\,v\,] \end{array}$$

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.

$$\begin{array}{lcl} Task_s \sqcup (fut \mapsto C_s[(f\ v)]) & \cong_{D \sqcup \{d\}} & Task_t \sqcup (tid \mapsto C_t[(f\ v)]), Fut_t \sqcup [\,fut \mapsto \mathfrak{U}\,d\,] \\ \Big\downarrow {\scriptstyle \text{ASYNC-CALL} \atop (f \mapsto \lambda_{\text{async}} x.\, e) \in Fun} & & \Big\downarrow {\scriptstyle \text{DPS-CALL} \atop (f \mapsto \lambda_{\text{dps}} \delta, x.\, \mathcal{D}[\![e]\!]_\delta) \in Fun'} \\ Task_s \sqcup (fut \mapsto C_s[fut']) & & Task_t \sqcup (tid \mapsto C_t[fut']) \\ \quad \sqcup (fut' \mapsto e[x \mapsto v]) & \cong_{D \sqcup \{d, d'\}} & \quad \sqcup (tid' \mapsto \mathcal{D}[\![e]\!]_\delta[\delta \mapsto d'][x \mapsto v]), \\ & & Fut_t \sqcup [\,fut \mapsto \mathfrak{U}\,d\,] \sqcup [\,fut' \mapsto \mathfrak{U}\,d'\,] \end{array}$$

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

Case TAIL-CALL.

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

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

Case FUTURE-FILL *(ε).*

$$\begin{array}{lcl} Task_s \sqcup \mathsf{Aw}[fut_0..fut_n] \\ \quad \sqcup (fut_n \mapsto v) & \cong_{D \sqcup \{d\}} & Task_t \sqcup (tid \mapsto \texttt{refine}(d, v)), Fut_t \sqcup [\, fut_0 \mapsto \mathfrak{U}\, d \,] \\ & \cong_D & \downarrow \text{FUTURE-FILL}\ \varepsilon \\ & & Task_t, Fut_t \sqcup [\, fut_0 \mapsto \mathfrak{R}\, v \,] \end{array}$$

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.

$$\text{EXPR-TRANSFO}\ \frac{}{e \sim_d \mathcal{D}[\![e]\!]_\delta[\delta \mapsto d]} \qquad \text{EXPR-FILL-AWAIT}\ \frac{}{e \sim_d \texttt{refine}(d, e)} \qquad \text{EXPR-AWAIT}\ \frac{}{e \overset{\text{await}}{\sim}_d \mathcal{D}_{\mathsf{a}}[\![e]\!]_\delta[\delta \mapsto d]}$$

Fig. 38. Expr-level relations

$$\text{THREAD-DEST}\ \frac{e \sim_d e'}{(fut \mapsto e)\ {}_{fut}\!\simeq_d\ [\, d \mapsto \nearrow \,], (tid \mapsto e')}$$

Fig. 39. Thread-level relation

$$\text{THREAD-AWAIT}\ \frac{T_s\ {}_{fut_n}\!\simeq_d \sigma_t, T_t}{\mathsf{Aw}[fut_0..fut_n] \sqcup T_s\ {}_{fut_0}\!\approx_d^{\{fut_{1..n}\}} \sigma_t, T_t} \qquad \text{THREAD-VAL}\ \frac{}{(fut \mapsto v) \sqcup \overline{(fut_i \mapsto v)}\ {}_{fut}\!\approx_d^{\overline{fut_i}}\ [\, d \mapsto v \,], \overline{(tid \mapsto ())}^{+}}$$

Fig. 40. Pool relation. The right index annotates the set of "internal" futures

$$\textsc{field-thread-await}\quad \frac{\alpha = \texttt{await}(fut) \qquad T_s \;{}_{fut}{\approx}^{I}_{d}\; \sigma_t, T_t \qquad d \in \mathrm{dom}(\sigma_t)}{\{\alpha\} : T_s \,\mathcal{R}^{I}_{d}\, \sigma_t, T_t}$$

$$\textsc{field-resolved}\quad \frac{T_s = \overline{(fut_i \mapsto v_i)} \qquad \exists i,\ v_i = v}{\{v\} : T_s \,\mathcal{R}^{\overline{fut_i}}_{d}\, [\, d \mapsto v\,], \emptyset}$$

$$\textsc{field-chain}\quad \frac{\alpha = \texttt{await}(fut) \qquad T_t = \texttt{forward}(d,\ fut)}{\{\alpha\} : \emptyset \;{}^{\mathsf{fwd}}\mathcal{R}^{\emptyset}_{d}\, [\, d \mapsto \diagup\,], T}$$

Fig. 41. For keeping sub-tasks for constructors together

$$\textsc{par-both-0}\quad \frac{e_1 \overset{\mathsf{await}}{\sim}_{d_1} e_1' \qquad e_2 \overset{\mathsf{await}}{\sim}_{d_2} e_2'}{\left(fut \mapsto \begin{array}{l}\texttt{let}\, x_1 = e_1\, \texttt{in}\\ \texttt{let}\, x_2 = e_2\, \texttt{in}\\ \texttt{[}\, t_{||}\texttt{,}\ \texttt{await}(x_1)\texttt{,}\ \texttt{await}(x_2)\,\texttt{]}\end{array}\right) \;{}_{fut}{\simeq}_{d}\; \left(\begin{array}{l}[\, d \mapsto \ell, \ell \mapsto t_{||}, \diagup, \diagup\,],\\ (tid \mapsto \begin{array}{l}\texttt{let}\,(d_1, d_2) = (\ell+1, \ell+2)\,\texttt{in}\\ e_1'; e_2'\end{array})\end{array}\right)}$$

$$\textsc{par-both-1}\quad \frac{e_1 \overset{\mathsf{await}}{\sim}_{\ell+1} e_1' \qquad e_2 \overset{\mathsf{await}}{\sim}_{\ell+2} e_2'}{\left(fut \mapsto \begin{array}{l}\texttt{let}\, x_1 = e_1\, \texttt{in}\\ \texttt{let}\, x_2 = e_2\, \texttt{in}\\ \texttt{[}\, t_{||}\texttt{,}\ \texttt{await}(x_1)\texttt{,}\ \texttt{await}(x_2)\,\texttt{]}\end{array}\right) \;{}_{fut}{\simeq}_{d}\; [\, d \mapsto \ell, \ell \mapsto t_{||}, \diagup, \diagup\,], (tid \mapsto e_1'; e_2')}$$

$$\textsc{par-both-2}\quad \frac{\vee \begin{cases} \{\alpha_1\} : T_1 \;{}^{\mathsf{fwd}}\mathcal{R}_{\ell+1}\, \sigma_1, T_1' \\ \{\alpha_1\} : T_1 \,\mathcal{R}^{I}_{\ell+1}\, \sigma_1, T_1' \end{cases} \qquad e_2 \overset{\mathsf{await}}{\sim}_{\ell+2} e_2'}{\left(fut \mapsto \begin{array}{l}\texttt{let}\, x_2 = e_2\, \texttt{in}\\ \texttt{[}\, t_{||}\texttt{,}\ \alpha_1\texttt{,}\ \texttt{await}(x_2)\,\texttt{]}\end{array}\right) \sqcup T_1 \;{}_{fut}{\simeq}^{I}_{d}\; [\, d \mapsto \ell, \ell \mapsto t_{||}, \ell+2 \mapsto \diagup\,] \sqcup \sigma_1, (tid \mapsto ((\,);)^{?} e_2') \sqcup T_1'}$$

$$\textsc{par-both-3}\quad \frac{\vee \begin{cases} \{\alpha_1\} : T_1 \;{}^{\mathsf{fwd}}\mathcal{R}_{\ell+1}\, \sigma_1, T_1' \\ \{\alpha_1\} : T_1 \,\mathcal{R}^{I_1}_{\ell+1}\, \sigma_1, T_1' \end{cases} \qquad \{\alpha_2\} : T_2 \,\mathcal{R}^{I_2}_{\ell+2}\, \sigma_2, T_2' \qquad T = (tid \mapsto \texttt{()})^{?} \qquad \texttt{holes}(d) \geq 1}{(fut \mapsto \texttt{[}\, t_{||}\texttt{,}\ \alpha_1\texttt{,}\ \alpha_2\,\texttt{]}) \sqcup T_1 \sqcup T_2 \;{}_{fut}{\simeq}^{I_1 \sqcup I_2}_{d}\; [\, d \mapsto \ell, \ell \mapsto t_{||}\,] \sqcup \sigma_1 \sqcup \sigma_2, T \sqcup T_1' \sqcup T_2'}$$

Fig. 42. paratag, both awaits

PAR-LEFT-0

$$\frac{e_1 \overset{\text{await}}{\sim}_{d_1} e_1' \qquad e_2 \sim_{d_2} e_2'}{\left(fut \mapsto \begin{array}{l}\texttt{let}\, x_1 = e_1 \,\texttt{in}\\ \texttt{let}\, x_2 = e_2 \,\texttt{in}\\ [\, t_{||}, \texttt{await}(x_1), x_2\,]\end{array}\right) \;{}_{fut}\!\simeq_d \left(\begin{array}{l}[\, d \mapsto \ell, \ell \mapsto t_{||}, /, /\,],\\ tid \mapsto \begin{array}{l}\texttt{let}\,(d_1, d_2) = (\ell+1, \ell+2)\,\texttt{in}\\ e_1'; e_2'\end{array}\end{array}\right)}$$

PAR-LEFT-1

$$\frac{e_1 \overset{\text{await}}{\sim}_{\ell+1} e_1' \qquad e_2 \sim_{\ell+2} e_2'}{\left(fut \mapsto \begin{array}{l}\texttt{let}\, x_1 = e_1 \,\texttt{in}\\ \texttt{let}\, x_2 = e_2 \,\texttt{in}\\ [\, t_{||}, \texttt{await}(x_1), x_2\,]\end{array}\right) \;{}_{fut}\!\simeq_d [\, d \mapsto \ell, \ell \mapsto t_{||}, /, /\,], (tid \mapsto e_1'; e_2')}$$

PAR-LEFT-2

$$\frac{\vee \begin{cases} \{\alpha_1\} : T_1 \;{}^{\text{fwd}}\mathcal{R}_{\ell+1}\, \sigma_1, T_1' \\ \{\alpha_1\} : T_1 \;\mathcal{R}^I_{\ell+1}\, \sigma_1, T_1' \end{cases} \qquad e_2 \sim_{\ell+2} e_2'}{\left(fut \mapsto \begin{array}{l}\texttt{let}\, x_2 = e_2 \,\texttt{in}\\ [\, t_{||}, \alpha_1, x_2\,]\end{array}\right) \sqcup T_1 \;{}_{fut}\!\simeq^I_d [\, d \mapsto \ell, \ell \mapsto t_{||}, \ell_2 \mapsto \diagup\,] \sqcup \sigma_1, (tid \mapsto ((\,);)^? e_2') \sqcup T_1'}$$

PAR-LEFT-3

$$\frac{\vee \begin{cases} \{\alpha_1\} : T_1 \;{}^{\text{fwd}}\mathcal{R}_{\ell+1}\, \sigma_1, T_1' \\ \{\alpha_1\} : T_1 \;\mathcal{R}^I_{\ell+1}\, \sigma_1, T_1' \end{cases} \qquad T = (tid \mapsto ())^? \qquad \texttt{holes}(d) \geq 1}{(fut \mapsto [\, t_{||}, \alpha_1, v_2\,]) \sqcup T_1 \sqcup T_2 \;{}_{fut}\!\simeq_d [\, d \mapsto \ell, \ell \mapsto t_{||}, \ell+2 \mapsto v_2\,] \sqcup \sigma_1, T \sqcup T_1'}$$

Fig. 43. paratag, one await on the left (symmetrically for right)

PAR-NONE-0

$$\frac{e_1 \sim_{d_1} e_1' \qquad e_2 \sim_{d_2} e_2'}{\left(fut \mapsto \begin{array}{l}\texttt{let}\, x_1 = e_1 \,\texttt{in}\\ \texttt{let}\, x_2 = e_2 \,\texttt{in}\\ [\, t_{||}, x_1, x_2\,]\end{array}\right) \;{}_{fut}\!\simeq_d \left(\begin{array}{l}[\, d \mapsto \ell, \ell \mapsto t_{||}, /, /\,],\\ tid \mapsto \begin{array}{l}\texttt{let}\,(d_1, d_2) = (\ell+1, \ell+2)\,\texttt{in}\\ e_1'; e_2'\end{array}\end{array}\right)}$$

PAR-NONE-1

$$\frac{e_1 \sim_{\ell+1} e_1' \qquad e_2 \sim_{\ell+2} e_2'}{\left(fut \mapsto \begin{array}{l}\texttt{let}\, x_1 = e_1 \,\texttt{in}\\ \texttt{let}\, x_2 = e_2 \,\texttt{in}\\ [\, t_{||}, x_1, x_2\,]\end{array}\right) \;{}_{fut}\!\simeq_d [\, d \mapsto \ell, \ell \mapsto t_{||}, /, /\,], (tid \mapsto e_1'; e_2')}$$

PAR-NONE-2

$$\frac{e_2 \sim_{\ell+2} e_2'}{\left(fut \mapsto \begin{array}{l}\texttt{let}\, x_2 = e_2 \,\texttt{in}\\ [\, t_{||}, v_1, x_2\,]\end{array}\right) \;{}_{fut}\!\simeq_d [\, d \mapsto \ell, \ell \mapsto t_{||}, v_1, /\,], (tid \mapsto ((\,);)^? e_2')}$$

Fig. 44. partag, no awaits

SEQ-NONE-0

$$\frac{e_2 \sim_{d_2} e_2'}{(fut \mapsto \begin{array}{l}\texttt{let}\, x_2 = e_2\, \texttt{in} \\ [\, t_{\text{seq}}, v_1, x_2\,]\end{array}) \; {}_{fut}{\simeq}_{d} \; (tid \mapsto \begin{array}{l}[\, d \mapsto \ell, \ell \mapsto t_{\text{seq}}, v_1, \diagup\,], \\ \texttt{let}\, d_2 = \ell + 2\, \texttt{in} \\ e_2'\end{array})}$$

SEQ-NONE-1

$$\frac{e_2 \sim_{\ell+2} e_2'}{(fut \mapsto \begin{array}{l}\texttt{let}\, x_2 = e_2\, \texttt{in} \\ [\, t_{\text{seq}}, v_1, x_2\,]\end{array}) \; {}_{fut}{\simeq}_{d} \; [\, d \mapsto \ell, \ell \mapsto t_{\text{seq}}, v_1, \diagup\,], (tid \mapsto e_2')}$$

Fig. 45. seqtag, no awaits

SEQ-LEFT-0

$$\frac{e_1 \overset{\text{await}}{\sim}_{d_1} e_1'}{(fut \mapsto \begin{array}{l}\texttt{let}\, x_1 = e_1\, \texttt{in} \\ [\, t_{\text{seq}}, \texttt{await}(x_1), v_2\,]\end{array}) \; {}_{fut}{\simeq}_{d} \; (tid \mapsto \begin{array}{l}[\, d \mapsto \ell, \ell \mapsto t_{\text{seq}}, \diagup, v_2\,], \\ \texttt{let}\, d_1 = \ell + 1\, \texttt{in} \\ e_1'\end{array})}$$

SEQ-LEFT-1

$$\frac{e_1 \overset{\text{await}}{\sim}_{\ell+1} e_1'}{(fut \mapsto \begin{array}{l}\texttt{let}\, x_1 = e_1\, \texttt{in} \\ [\, t_{\text{seq}}, \texttt{await}(x_1), v_2\,]\end{array}) \; {}_{fut}{\simeq}_{d} \; \begin{array}{l}[\, d \mapsto \ell, \ell \mapsto t_{\text{seq}}, \diagup, v_2\,], \\ (tid \mapsto e_1')\end{array}}$$

SEQ-LEFT-2

$$\frac{\vee \begin{cases} \{\alpha_1\} : T_1 \; {}^{\text{fwd}}\mathcal{R}_{\ell+1}\, \sigma_1, T_1' \\ \{\alpha_1\} : T_1 \; \mathcal{R}^{I}_{\ell+1}\, \sigma_1, T_1' \end{cases} \qquad T = (tid \mapsto \texttt{()})^{?} \qquad \texttt{holes}(d) \geq 1}{(fut \mapsto [\, t_{\text{seq}}, \alpha_1, v_2\,]) \sqcup T_1 \; {}_{fut}{\simeq}_{d} \; \begin{array}{l}[\, d \mapsto \ell, \ell \mapsto t_{\text{seq}}, \ell + 2 \mapsto v_2\,] \sqcup \sigma_1, \\ T \sqcup T_1'\end{array}}$$

Fig. 46. seqtag, await on the left (sym for right)

BOX-THREAD

$$\frac{T_s \; {}_{fut}{\approx}^{I}_{d} \; s_t, T_t}{T_s \cong^{I} s_t, T_t, [\, fut \mapsto \mathfrak{U}\, d\,]}$$

BOX-VAL

$$\frac{}{(fut \mapsto v) \sqcup \overline{(fut_i \mapsto v)}^{\,i} \cong^{\overline{fut_i}^{\,i}} [\, d \mapsto v\,], \emptyset, [\, fut \mapsto \mathfrak{U}\, d\,]}$$

BOX-RESOLVED

$$\frac{T = \overline{(tid \mapsto \texttt{()})}}{(fut \mapsto v) \sqcup \overline{(fut_i \mapsto v)}^{\,i} \cong^{\overline{fut_i}^{\,i}} \emptyset, T, [\, fut \mapsto \mathfrak{R}\, v\,]}$$

Fig. 47. Top-level-future-level rel (=boxes)

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

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

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 $T_i \cong^I \sigma_i, T_i, F_i$
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 $e_s = C[e]$.
Moreover, for any e', $C_s[e'] \sim_d C_t[e']$

BOX-VAL. Only step that can be taken is FUTURE-RESOLVE (ε). 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 (ε) (only if $T \neq \emptyset$), and we close immediately.

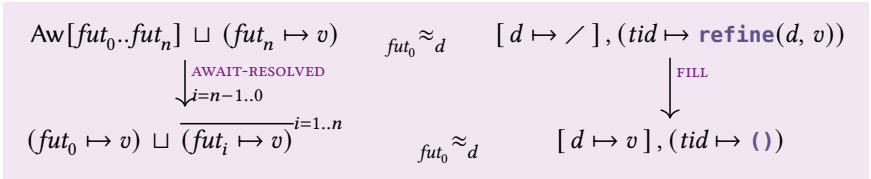
BOX-THREAD. We give here most elements of the proof of lemma 3.1, from which the proof for this case follows immediately

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.

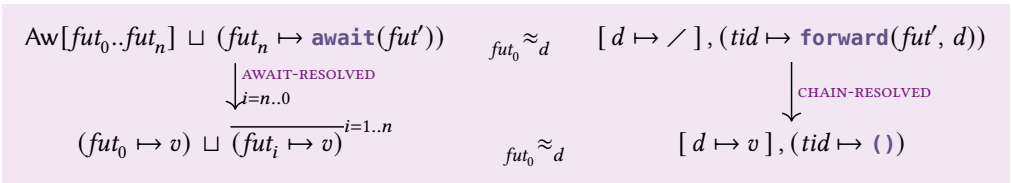
refine ($\neq$ FILL) A refine step necessarily happens in a context of the form **`let`** $\delta^+ =$ **`refine`**$(d, \square)$ **`in`** e. There are several cases depending on the shape of the second argument of **`refine`** (sequential or parallel tag, which and how many fields are in the domain of `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.



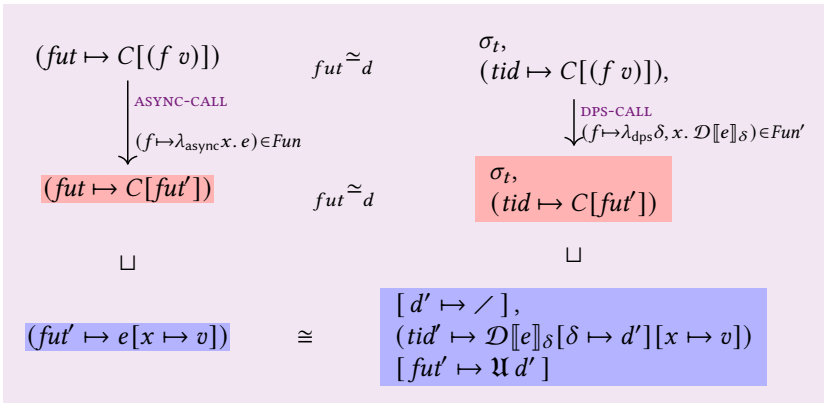
Even though we might answer the challenge with 0 steps on the source side, we cannot take infinitely many FILL steps in a row: each FILL step consumes a binding $[\, d \mapsto \nearrow \,]$ from the store. The number of these bindings in the store is thus a strictly decreasing measure for possibly-ε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

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



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":



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 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 fro 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 $(\texttt{()};)^?$, 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