

Tail Modulo Cons, OCAML, and Relational Separation Logic

ANONYMOUS AUTHOR(S)

Common functional languages incentivize tail-recursive functions, as opposed to general recursive functions that consume stack space and may not scale to large inputs. This distinction occasionally requires writing functions in a tail-recursive style that may be more complex and slower than the natural, non-tail-recursive definition.

This work describes our implementation of the *tail modulo constructor* (TMC) transformation in the OCAML compiler, an optimization that provides stack-efficiency for a larger class of functions — tail-recursive *modulo constructors* — which includes in particular the natural definition of `List.map` and many similar recursive data-constructing functions.

We prove the correctness of this program transformation in a simplified setting — a small untyped calculus — that captures the salient aspects of the OCAML implementation. Our proof is mechanized in the COQ proof assistant, using the Iris base logic. An independent contribution of our work is an extension of the SIMULIRIS approach to define simulation relations that support different calling conventions. To our knowledge, this is the first use of SIMULIRIS to prove the correctness of a compiler transformation.

1 INTRODUCTION

1.1 Prologue

“OCAML”, we teach our students, “is a functional programming language. We can write the beautiful function `List.map` as follows:”

```
let rec map f = function
| [] → []
| x :: xs → f x :: map f xs
```

“Well, actually”, we continue, “OCAML is an effectful language, so we need to be careful about the evaluation order. We want `map` to process elements from the beginning to the end of the input list, and the evaluation order of `f x :: map f xs` is unspecified. So we write:”

```
let rec map f = function
| [] → []
| x :: xs →
  let y = f x in
  y :: map f xs
```

“Well, actually, this version fails with a `Stack_overflow` exception on large input lists. If you want your `map` to behave correctly on all inputs, you should write a *tail-recursive* version. For this you can use the accumulator-passing style:”

```
let map f li =
  let rec map_ acc = function
  | [] → List.rev acc
  | x :: xs → map_ (f x :: acc) xs
  in map_ [] f li
```

“Well, actually, this version works fine on large lists, but it is less efficient than the original version. One approach is to start with a non-tail-recursive version, and switch to a tail-recursive version for large inputs; even there you can use some manual unrolling to reduce the overhead of the accumulator. For example, the nice [Containers](#) library does it as follows:”.

```

50 let tail_map f l =
51   (* Unwind the list of tuples, reconstructing the full list front-to-back.
52    @param tail_acc a suffix of the final list; we append tuples' content
53    at the front of it *)
54   let rec rebuild tail_acc = function
55     | [] → tail_acc
56     | (y0, y1, y2, y3, y4, y5, y6, y7, y8) :: bs →
57       rebuild (y0 :: y1 :: y2 :: y3 :: y4 :: y5 :: y6 :: y7 :: y8 :: tail_acc) bs
58   in
59   (* Create a compressed reverse-list representation using tuples
60    @param tuple_acc a reverse list of chunks mapped with [f] *)
61   let rec dive tuple_acc = function
62     | x0 :: x1 :: x2 :: x3 :: x4 :: x5 :: x6 :: x7 :: x8 :: xs →
63       let y0 = f x0 in let y1 = f x1 in let y2 = f x2 in
64       let y3 = f x3 in let y4 = f x4 in let y5 = f x5 in
65       let y6 = f x6 in let y7 = f x7 in let y8 = f x8 in
66       dive ((y0, y1, y2, y3, y4, y5, y6, y7, y8) :: tuple_acc) xs
67     | xs →
68       (* Reverse direction, finishing off with a direct map *)
69       let tail = List.map f xs in
70       rebuild tail tuple_acc
71   in
72   dive [] l

```

```

let direct_depth_default_ = 1000

let map f l =
  let rec direct f i l = match l with
  | [] → []
  | [x] → [f x]
  | [x1;x2] → let y1 = f x1 in [y1; f x2]
  | [x1;x2;x3] →
    let y1 = f x1 in let y2 = f x2 in [y1; y2; f x3]
  | _ when i=0 → tail_map f l
  | x1::x2::x3::x4::l' →
    let y1 = f x1 in
    let y2 = f x2 in
    let y3 = f x3 in
    let y4 = f x4 in
    y1 :: y2 :: y3 :: y4 :: direct f (i-1) l'
  in
  direct f direct_depth_default_ l

```

At this point, unfortunately, some students leave the class and never come back.

We propose a new feature for the OCAML compiler, an explicit, opt-in “Tail Modulo Cons” transformation, to retain our students. After the first version (or maybe, if we are teaching an advanced class, after the second version), we could show them the following version:

```

71 let[@tail_mod_cons] rec map f = function
72 | [] → []
73 | x :: xs → f x :: map f xs

```

This version is as fast as the simple implementation, tail-recursive, and easy to write.

The catch, of course, is to teach when this `[@tail_mod_cons]` annotation can be used. Maybe we would not show it at all, and pretend that the direct map version with `let y` is fine. This would be a much smaller lie than it currently is, a `[@tail_mod_cons]`-sized lie.

Finally, experts must be very happy. They know about all these versions, but they do not have to write them by hand anymore. Have a program perform (some of) the program transformations that they are currently doing manually.

1.2 TMC transformation example

A function call is in *tail position* within a function definition if the definition has “nothing to do” after evaluating the function call – the result of the call is the result of the whole function at this point of the program. (A precise definition will be given in Section 3.2.) A function is *tail recursive* if all its recursive calls are tail calls.

In the definition of `map`, the recursive call is not in tail position: after computing the result of `map f xs` we still have to compute the final list cell, `y :: ■`. We say that a call is *tail modulo cons* when the work remaining is formed of data *constructors* only, such as `(::)` here.

```

92 let[@tail_mod_cons] rec map f = function
93 | [] → []
94 | x :: xs →
95   let y = f x in
96   y :: map f xs

```

Other datatype constructors may also be used; the following example is also tail-recursive *modulo cons*:

```
let[@tail_mod_cons] rec tree_of_list = function
| [] → Empty
| x :: xs → Node(Empty, x, tree_of_list xs)
```

The TMC transformation returns an equivalent function in *destination-passing style* where the calls in *tail modulo cons* position have been turned into *tail* calls. In particular, for `map` it gives a tail-recursive function, which runs in constant stack space; many other list functions also become tail-recursive. This works for other datatypes as well, such as binary trees, but in this case some recursive calls may remain non-tail-recursive.

The transformed code of `map` can be described as follows:

```
let rec map f = function
| [] → []
| x::xs →
  let y = f x in
  let dst = y :: ■ in
  map_dps dst 1 f xs;
  dst

and map_dps dst i f = function
| [] →
  dst.i ← []
| x::xs →
  let y = f x in
  let dst' = y :: ■ in
  dst.i ← dst';
  map_dps dst' 1 f xs
```

The transformed code has two variants of the `map` function. The `map_dps` variant is in *destination-passing style*, it expects additional parameters that specify a memory location, a *destination*, and will write its result to this *destination* instead of returning it. It is tail-recursive, and it performs a single traversal of the list. The `map` variant provides the same interface as the non-transformed function, we say that it is in *direct style*. It is not tail-recursive, but it does not call itself recursively, it calls the tail-recursive `map_dps` on non-empty lists.

The key idea of the transformation is that the expression `y :: map f xs`, which contained a non-tail-recursive call, is transformed into first the computation of a *partial* list cell, written `y :: ■`, followed by a call to `map_dps` that is asked to write its result in the position of the `■`. The recursive call thus happens after the cell creation (instead of before), in tail-recursive position in the `map_dps` variant. In the direct variant, the value of the destination `dst` has to be returned after the call.

The transformed code is pseudo-OCAML, it is not a valid OCAML program: we use a magical `■` constant, and our notation `dst.i ← ...` to update constructor parameters in-place is also invalid in source programs. The transformation is implemented on a lower-level, untyped intermediate representation of the OCAML compiler (Lambda), where those operations do exist. The OCAML type system is not expressive enough to type-check the transformed program: the list cell is only partially-initialized at first, each partial cell is mutated exactly once, and in the end the whole result is returned as an *immutable* list. Some type system are expressive enough to represent this transformed code, notably Mezzo [Balabonski, Pottier and Protzenko 2016], based on a permission system inspired by *separation logic* [O'Hearn 2019], or the linear types used in Minamide [1998].

TMC has been implemented in Lisp [Friedman and Wise 1975; Risch 1973], Prolog and (simultaneously to our work) Koka [Leijen and Lorenzen 2023]. A simpler variant of TMC, where constructors are replaced by associative operations and transformed to *accumulator-passing style*, is in gcc and clang.

The first main contribution of our work is an implementation of TMC in the OCAML compiler as an on-demand program transformation. We describe the non-trivial design choices in terms

of user interface, and evaluate performance through microbenchmarks. Various functions in the standard library and third-party codebases have been rewritten to use it, to become tail-recursive, gain in performance, or (when an efficient but complex tail-recursive version was used) simplify considerably the implementation.

The second main contribution of this work is a mechanized proof of correctness for the core of this transformation on a small untyped calculus. For any input source program, we establish a termination-preserving *behavioral refinement* between the source program and the corresponding transformed program: any behavior of the transformed program, be it converging, diverging or stuck, is a behavior of the source program.

Our proof technique is to define a relational program logic for our small untyped calculus, to show the correctness of the TMC transformation using this program logic, and to get the behavioral refinement by proving adequacy of our program logic. We build on top of SIMULIRIS [Gäher, Sammler, Spies, Jung, Dang, Krebbers, Kang and Dreyer 2022], a framework for simulations in separation logic over the Iris base logic. The use of separation logic nicely captures certain aspects of the proof argument, in particular the fact that the destination-passing-style function uniquely owns the destination location it receives.

To our knowledge, previous works on SIMULIRIS have verified *examples* of interesting optimizations and program transformations, by formally proving relations between pairs of concrete programs. Our work may be the first proof of correctness of a *program transformation* (as a function or relation) using a simulation-based approach, establishing correctness for all input programs. (Earlier Iris work proves program transformations using logical relations, see for example Tassarotti, Jung and Harper [2017].)

At this level of generality, we found that the SIMULIRIS simulation is not expressive enough to reason about transformations that introduce new function calling conventions. We generalize the SIMULIRIS handling of function calls by parameterizing the simulation relation over an *abstract protocol*, inspired by de Vilhena and Pottier [2021]. This sub-contribution of our work is independent from the TMC transformation and our small calculus, and we tried to express it in general terms, beyond the specific needs of TMC. In particular, we believe that Iris-based relational separation logics could be a powerful yet pleasant proof technique for compiler verification.

The core of the soundness proof is the specification of the two variants of each TMC-transformed function — direct style and destination-passing style. It concisely conveys the essence of destination-passing style: computing the same thing and writing it to an owned destination. For instance, to give a taste of the formalism, the specification of the variants of `map` looks as follows:

$$\begin{aligned} & \{\text{True}\} \text{@map } (@f, v_s) \succeq \text{@map } (@f, v_t) \{\approx\} \\ & \{(\ell + i) \mapsto \blacksquare\} \text{@map } (@f, v_s) \succeq \text{@map_dps } ((\ell, i), @f, v_t) \{v'_s, (). \exists v'_t. (\ell + i) \mapsto v'_t * v'_s \approx v'_t\} \end{aligned}$$

To sum up, our main contributions are:

- (1) an implementation of the TMC transformation in the OCAML compiler, with a discussion of the user interface, a performance evaluation, and a survey of its early usage;
- (2) a mechanized proof of soundness for an idealized TMC transformation on a small calculus, using a relational separation program logic;
- (3) a generalization of the SIMULIRIS handling of function calls with abstract *protocols* to reason about different calling conventions.

197	Index	$\ni i$	$::= 0 \mid 1 \mid 2$		
198	B	$\ni b$	$::= \text{true} \mid \text{false}$	Def	$\ni d ::= \text{rec } x = e$
199	Tag	$\ni t$		Prog	$\ni p ::= \mathbb{F} \xrightarrow{\text{fin}} \text{Def}$
200	L	$\ni \ell$		State	$\ni \sigma ::= \mathbb{L} \xrightarrow{\text{fin}} \text{Val}$
201	F	$\ni f$		Config	$\ni \rho ::= \text{Expr} \times \text{State}$
202	X	$\ni x, y$			
203	Val	$\ni v, w$	$::= () \mid i \mid t \mid b \mid \ell \mid @f$		
204	Expr $\ni e$	$::= v$		Ectx $\ni E$	$::= \square$
205		$ x \mid \text{let } x = e_1 \text{ in } e_2 \mid e_1 e_2$			$ \text{let } x = E \text{ in } e_2 \mid e_1 E \mid E v_2$
206		$ e_1 = e_2$			$ e_1 = E \mid E = v_2$
207		$ \text{if } e_0 \text{ then } e_1 \text{ else } e_2$			$ \text{if } E \text{ then } e_1 \text{ else } e_2$
208		$ \{ t, e_1, e_2 \} \mid [t, e_1, e_2]$			$ e_1.(E) \mid E.(v_2)$
209		$ e_1.(e_2) \mid e_1.(e_2) \leftarrow e_3$			$ e_1.(e_2) \leftarrow K \mid e_1.(E) \leftarrow v_3 \mid E.(v_2) \leftarrow v_3$

Fig. 1. DATA LANG syntax

213	$e_1 ; e_2$	$::= \text{let } x = e_1 \text{ in } e_2$		
214	(e_1, e_2)	$::= \{ \text{PAIR}, e_1, e_2 \}$	$\text{rec } (x_1, x_2) = e$	$::= \text{rec } y = \text{let } (x_1, x_2) = y \text{ in } e$
215	$\text{let } (x_1, x_2) = e_1 \text{ in } e_2$	$::= \text{let } y = e_1 \text{ in}$	$[]$	$::= ()$
216		$\text{let } x_1 = y.(1) \text{ in}$	$e_1 :: e_2$	$::= \{ \text{CONS}, e_1, e_2 \}$
217		$\text{let } x_2 = y.(2) \text{ in}$		
218		e_2		
219	$\text{match } e_0 \text{ with } [] \rightarrow e_1 \mid x :: xs \rightarrow e_2$	$::= \text{let } y = e_0 \text{ in}$		
220		$\text{if } y = []$		
221		$\text{then } e_1$		
222		$\text{else let } (x, xs) = y \text{ in } e_2$		
223	\blacksquare	$::= ()$		

Fig. 2. DATA LANG syntactic sugar

Remark. A preliminary, work-in-progress version of this work was presented in a previous publication at a national conference.¹ We explain our choice to submit a full paper to an international conference in [Appendix A](#).

2 TMC ON AN IDEALIZED LANGUAGE

In this section, we formalize the “tail modulo cons” (TMC) transformation in an idealized language, DATA LANG, that is expressive enough to account for the main aspects of TMC but does not support all features of OCAML.

This serves as a precise presentation of the transformation, and it is those definitions for which we formally prove correctness. We intentionally kept this presentation very close to our Coq development, which can be referred to for full details.

2.1 Language

The syntax of DATA LANG is given in [Figure 1](#) and its semantics in [Figure 3](#). We also introduce syntactic sugar in [Figure 2](#), including some for lists. Going back to our motivating example, we can define the `map` function on lists as in [Figure 4](#).

¹Citation removed for anonymity.

$$\begin{array}{c}
\boxed{- \xrightarrow[\text{head}]{p} - : \text{Config} \rightarrow \text{Config} \rightarrow \text{Prop}} \qquad \boxed{- \xrightarrow{p} - : \text{Config} \rightarrow \text{Config} \rightarrow \text{Prop}} \\
\\
\text{STEPLET} \qquad \frac{(\text{let } x = v \text{ in } e, \sigma) \xrightarrow[\text{head}]{p} (e[x \setminus v], \sigma)}{\text{STEPLET}} \qquad \frac{\text{STEPCALL} \quad p[f] = (\text{rec } x = e)}{(\text{@f } v, \sigma) \xrightarrow[\text{head}]{p} (e[x \setminus v], \sigma)} \\
\\
\text{STEPBLOCK1} \quad (\{t, e_1, e_2\}, \sigma) \xrightarrow[\text{head}]{p} \left(\begin{array}{l} \text{let } x_1 = e_1 \text{ in} \\ \text{let } x_2 = e_2 \text{ in} \\ [t, x_1, x_2] \end{array} , \sigma \right) \qquad \text{STEPBLOCK2} \quad (\{t, e_1, e_2\}, \sigma) \xrightarrow[\text{head}]{p} \left(\begin{array}{l} \text{let } x_2 = e_2 \text{ in} \\ \text{let } x_1 = e_1 \text{ in} \\ [t, x_1, x_2] \end{array} , \sigma \right) \\
\\
\text{STEPBLOCKDET} \quad \frac{\forall i \in \text{Index}, \ell + i \notin \text{dom}(\sigma)}{([t, v_1, v_2], \sigma) \xrightarrow[\text{head}]{p} (\ell, \sigma[\ell \mapsto t, v_1, v_2])} \qquad \text{STEPLoad} \quad \frac{\sigma[\ell] = v}{(\ell.(i), \sigma) \xrightarrow[\text{head}]{p} (v, \sigma)} \\
\\
\text{STEPSTORE} \quad \frac{\ell + i \in \text{dom}(\sigma)}{(\ell.(i) \leftarrow v, \sigma) \xrightarrow[\text{head}]{p} ((), \sigma[\ell + i \mapsto v])} \qquad \text{STEPCTX} \quad \frac{(e, \sigma) \xrightarrow[\text{head}]{p} (e', \sigma')}{(E[e], \sigma) \xrightarrow{p} (E[e'], \sigma')}
\end{array}$$

Fig. 3. DATA LANG semantics (excerpt)

```

map := rec (fn, xs) =
  match xs with
  | [] →
    []
  | x :: xs →
    let y = fn x in
    y :: @map (fn, xs)

```

Fig. 4. Natural implementation of map in DATA LANG

DATA LANG is an untyped sequential calculus with mutable state. A DATA LANG program p is a finite mapping from function names $f \in \mathbb{F}$ to definitions d , which are themselves (mutually-)recursive functions whose body is written $\text{rec } x = e$. Functions have a single parameter for simplicity, with pairs used to pass several values.

DATA LANG has booleans $b \in \{\text{true}, \text{false}\}$, an if-then-else construct, and a runtime equality test between (untyped) values $e_1 = e_2$.

DATA LANG does not feature general lambda expressions – its programs correspond to closure-converted or lambda-lifted source programs. Functions names f can be turned into values written @f , to be used directly in function calls or as parameters to higher-order functions.

To express constructors, DATA LANG features mutable memory blocks with an abstract *tag* ($t \in \text{Tag}$), and two *fields* which are arbitrary values (e_1, e_2). One can allocate a block with $\{t, e_1, e_2\}$, access its fields with $e_1.(e_2)$ and modify them with $e_1.(e_2) \leftarrow e_3$. Allocation returns a location $\ell \in \mathbb{L}$, which may not appear in source programs.

$$p_s \rightsquigarrow p_t := \exists \xi. \bigwedge \left[\begin{array}{l} \text{dom}(\xi) \subseteq \text{dom}(p_s) \\ \text{dom}(p_t) = \text{dom}(p_s) \cup \text{codom}(\xi) \\ \forall f, d_s. p_s[f] = d_s \implies \exists d_t. p_t[f] = d_t \wedge d_s \xrightarrow[\text{dir}]{\xi} d_t \\ \forall f, f_{dps}, d_s. p_s[f] = d_s \wedge \xi[f] = f_{dps} \implies \exists d_t. p_t[f_{dps}] = d_t \wedge d_s \xrightarrow[\text{dps}]{\xi} d_t \end{array} \right]$$

Fig. 5. TMC transformation

To avoid over-simplifying, the evaluation order of subexpressions e_1 and e_2 in $\{t, e_1, e_2\}$ is unspecified as in OCAML. To model this in the semantics, we introduce a separate, deterministic block construction $[t, e_1, e_2]$ construction which cannot appear in source programs. A block expression $\{t, e_1, e_2\}$ first nondeterministically reduces to either `let $x_1 = e_1$ in let $x_2 = e_2$ in $[t, x_1, x_2]$` through **STEPBLOCK1** or `let $x_2 = e_2$ in let $x_1 = e_1$ in $[t, x_1, x_2]$` through **STEPBLOCK2**. $[t, v_1, v_2]$ performs the allocation through **STEPBLOCKDET**.

The values in **DATALANG** are functions $@f$, locations ℓ of allocated blocks, booleans b , tags t (taken in an arbitrary, denumerable set), the unit value $()$, and indices $i \in \{0, 1, 2\}$ inside blocks. (Our transformation never mutates tags in position 0, nor do OCAML programs, but for example **MEZZO** supports it.)

On top of these basic language features, **Figure 2** introduces syntactic sugar for pairs (e_1, e_2) as blocks with a specific tag **PAIR**, for decomposing blocks in **let**-bindings (`let $(x, y) = e_1$ in e_2`) and in arguments of toplevel functions ($f \mapsto \text{rec } (x, y) = e$), and for (untyped) lists by defining the empty list `[]` as $()$, and for (mutable) cons-cells as blocks with a specific tag **CONS**. Pattern-matching on lists can be expressed by comparing the list with `[]`, using our block-deconstructing **let** (which ignores the tag) to deconstruct cons-cells.

As a side note, we use named expression variables here but the Coq mechanization actually adopts de Bruijn syntax better suited to define transformations involving binders. More precisely, our formalisation relies on the **AUTOSUBST** library [Schäfer, Tebbi and Smolka 2015]. Our definitions respect α -equivalence on term variables x, y : we implicitly assume any term variable in bound position to be chosen distinct from all other variables in context. Function names f are not α -renamed, as the transformation relates names in the source and target of the transformation.

2.2 Transformation

We now define the TMC transformation as a relation $p_s \rightsquigarrow p_t$ between programs and their transformation. The relation is total, in the sense that any **DATALANG** program p_s can be related to at least one transformed program p_t , but it not deterministic, for each input program it captures a (finite) set of admissible transformations, which we all prove valid. This non-determinism captures several choices that have to be done by the user through a user interface to control the transformation, or by the compiler implementation, influencing performance and evaluation order of the result. In this section, we do not describe how these choices are resolved – there is a large design space. We present the choices we made for OCAML compiler in **Section 3**.

As formalized in **Figure 5**, transforming a **DATALANG** program p consists in:

1. *Choosing a subset of toplevel functions to be TMC-transformed.* In OCAML, the programmer has to annotate these. For each such function f , we also require a fresh function name $\xi[f]$ (that is not defined in p_s) that will be defined to the *destination-passing style* (DPS) version of f in the transformed program p_t .

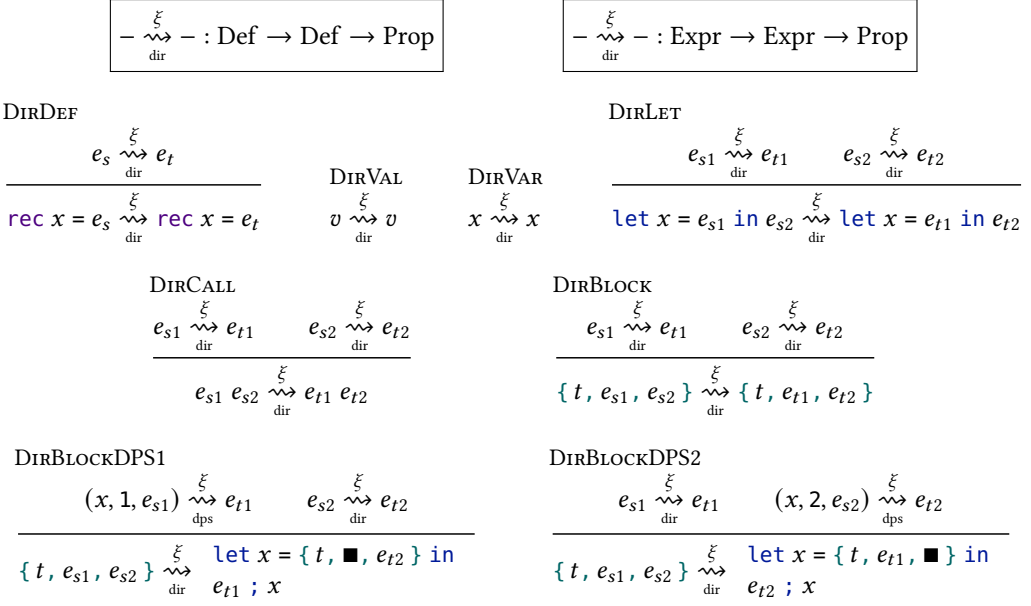


Fig. 6. Direct TMC transformation (omitting congruence rules similar to **DIRCALL**)

Formally, the subset is determined by the domain of the renaming function ξ , which is passed as a parameter to the auxiliary transformations we describe next.

2. For each function f defined in p , computing its direct transform. We introduce in Figure 6 the relations $d_s \xrightarrow[\text{dir}]{\xi} d_t$ for definitions and $e_s \xrightarrow[\text{dir}]{\xi} e_t$ for expressions.

$d_s \xrightarrow[\text{dir}]{\xi} d_t$ expresses that: 1) d_t has the same calling convention as d_s . 2) The body of d_t is the direct transform of the body of d_s .

$e_s \xrightarrow[\text{dir}]{\xi} e_t$ expresses that e_t is the direct transform of e_s . Intuitively, this means e_t computes the same thing as e_s .

The direct-style transform corresponds to the case where we do not have a block available to serve as a destination: this version is used in an arbitrary calling context, not necessarily under a constructor. Most rules are straightforward congruences – we recursively transform subexpressions and preserve the term constructor. We omit the rules for loads $e_1 . (e_2)$, stores $e_1 . (e_2) \leftarrow e_3$, and the deterministic blocks $[t, e_1, e_2]$ which are such simple congruences, just like calls $e_1 \ e_2$.

The key cases are for a block construct $\{t, e_1, e_2\}$. We can use this block as a destination, and switch to the destination-passing-style calling convention – these rules are the only places in the direct-style transformation where destination-passing-style is introduced. **DIRBLOCK** is a simple congruence rule that keeps both arguments in direct style. The rules (**DIRBLOCKDPS1**, **DIRBLOCKDPS2**) allow to choose a block argument to be evaluated in destination-passing style. (It is also possible to transform both arguments in DPS style, and we include extra rules for this in our formalization.)

In details, the terms produced by these rules proceed as follows: 1) They partially initialize a new memory block, with a hole for one of their arguments. 2) They evaluate the DPS transformation of

$$\begin{array}{c}
\boxed{- \xrightarrow[\text{dps}]{\xi} - : \text{Def} \rightarrow \text{Def} \rightarrow \text{Prop}} \quad \boxed{- \xrightarrow[\text{dps}]{\xi} - : \text{Expr} \times \text{Expr} \times \text{Expr} \rightarrow \text{Expr} \rightarrow \text{Prop}} \\
\\
\text{DPSDEF} \\
\frac{(x_{dst}, x_{idx}, e_s) \xrightarrow[\text{dps}]{\xi} e_t}{\text{rec } x = e_s \xrightarrow[\text{dps}]{\xi} \text{rec } ((x_{dst}, x_{idx}), x) = e_t} \\
\\
\text{DPSLET} \quad \text{DPSIF} \\
\frac{e_{s1} \xrightarrow[\text{dir}]{\xi} e_{t1} \quad (e_{dst}, e_{idx}, e_{s2}) \xrightarrow[\text{dps}]{\xi} e_{t2}}{\left(e_{dst}, e_{idx}, \text{let } x = e_{s1} \text{ in } e_{s2} \right) \xrightarrow[\text{dps}]{\xi} \text{let } x = e_{t1} \text{ in } e_{t2}} \quad \frac{e_{s0} \xrightarrow[\text{dir}]{\xi} e_{t0} \quad (e_{dst}, e_{idx}, e_{s1}) \xrightarrow[\text{dps}]{\xi} e_{s2} \quad (e_{dst}, e_{idx}, e_{s2}) \xrightarrow[\text{dps}]{\xi} e_{t2}}{\left(e_{dst}, e_{idx}, \text{if } e_{s0} \text{ then } e_{s1} \text{ else } e_{s2} \right) \xrightarrow[\text{dps}]{\xi} \text{if } e_{t0} \text{ then } e_{t1} \text{ else } e_{t2}} \\
\\
\text{DPSBLOCK1} \quad \text{DPSBLOCK2} \\
\frac{(x, 1, e_{s1}) \xrightarrow[\text{dps}]{\xi} e_{t1} \quad e_{s2} \xrightarrow[\text{dir}]{\xi} e_{t2}}{\left(e_{dst}, e_{idx}, \{ t, e_{s1}, e_{s2} \} \right) \xrightarrow[\text{dps}]{\xi} \text{let } x = \{ t, \blacksquare, e_{t2} \} \text{ in } e_{dst} \cdot (e_{idx}) \leftarrow x ; e_{t1}} \quad \frac{e_{s1} \xrightarrow[\text{dir}]{\xi} e_{t1} \quad (x, 2, e_{s2}) \xrightarrow[\text{dps}]{\xi} e_{t2}}{\left(e_{dst}, e_{idx}, \{ t, e_{s1}, e_{s2} \} \right) \xrightarrow[\text{dps}]{\xi} \text{let } x = \{ t, e_{t1}, \blacksquare \} \text{ in } e_{dst} \cdot (e_{idx}) \leftarrow x ; e_{t2}} \\
\\
\text{DPSCALL} \quad \text{DPSBASE} \\
\frac{f \in \text{dom}(\xi) \quad e_s \xrightarrow[\text{dir}]{\xi} e_t}{(e_{dst}, e_{idx}, @f e_s) \xrightarrow[\text{dps}]{\xi} @ \xi[f] ((e_{dst}, e_{idx}), e_t)} \quad \frac{e_s \xrightarrow[\text{dir}]{\xi} e_t}{(e_{dst}, e_{idx}, e_s) \xrightarrow[\text{dps}]{\xi} e_{dst} \cdot (i) \leftarrow e_t}
\end{array}$$

Fig. 7. Destination-passing style TMC transformation (in full)

the corresponding argument. passing the uninitialized field as destination. 3) They return the now fully initialized block.

An implementation would typically determine which subexpression would benefit from destination-passing style, that is, contains function calls $@f e$ in tail position (relatively to the subexpression) that have a destination-passing variant $\xi[f]$.

3. For each TMC-transformed function f , choosing a destination-passing-style transform. We introduce in Figure 7 the relations $d_s \xrightarrow[\text{dps}]{\xi} d_t$ for definitions and $(e_{dst}, e_{idx}, e_s) \xrightarrow[\text{dps}]{\xi} e_t$ for expressions.

$d_s \xrightarrow[\text{dps}]{\xi} d_t$ expresses that: 1) The function defined in d_t has an additional parameter representing the destination where it must write its result; it is a pair of the location of a memory block x_{dst} along with the index x_{idx} of a particular field in this block. 2) The body of d_t is a DPS transform of the body of d_s under the given destination.

$(e_{dst}, e_{idx}, e_s) \xrightarrow[\text{dps}]{\xi} e_t$ expresses that e_t is a DPS transform of e_s under destination (e_{dst}, e_{idx}) . Intuitively, this means e_t computes the same thing as e_s but writes it into the destination instead of returning it. We will formalize this intuition in Section 4.

Note that the rule **DPSDEF**, which relates the two judgments, uses the expression-level relation $(\overset{\xi}{\rightsquigarrow}_{\text{dps}})$ with a term variable x_{idx} to represent the index, not just a constant index 1 or 2 as in block rules. These are the only two sort of expressions used to represent offsets in the transformation.

In the direct-style relation, congruence rules apply the same direct-style transformation to all subexpressions. The congruence-like rule of the DPS relation, for example **DPSLET** and **DPSIF**, are different. They apply the DPS transformation to sub-expressions which are in tail position relative to the expression, and the direct-style transformation to all other subexpressions. (The if-then-else construct can have two different subexpressions in tail position, because at most one of them is evaluated at runtime.)

The rules **DPSBLOCK1** and **DPSBLOCK2** correspond to the rules **DIRBLOCKDPS1** and **DIRBLOCKDPS2** in the direct-style transformation, but the transformed code is different. Consider the translation of $(e_{\text{dst}}, e_{\text{idx}}, \{t, e_{s1}, e_{s2}\})$ into `let $x = \{t, e_{t1}, \blacksquare\}$ in $e_{\text{dst}}.(e_{\text{idx}}) \leftarrow x$; e_{t2}` by **DIRBLOCKDPS2**. First we create a new destination x , with a hole in second position. Then, instead of computing the corresponding subterm, we write this new destination x into the *current* destination $(e_{\text{dst}}, e_{\text{idx}})$. Finally we evaluate e_{t2} , which is the DPS transform of the subexpression e_{s2} , with the destination $(x, 2)$. Notice that e_{t2} is in tail position relative to the transformed expression, while it was not in tail position (but under a constructor) in the source expression. This is the key step of the TMC transformation. Both rules **DIRBLOCKDPS2** puts the second subterm in tail position, and **DIRBLOCKDPS1** puts the first subterm in tail position. In the case of lists as in our running example `y :: map f xs`, we obviously want to put the second subterm in tail position, and to use **DIRBLOCKDPS2**. But consider a map function on binary trees `Node(map f left, map f right)`, the implementation must choose one subterm to put in tail position and another to keep in non-tail position.

The rule **DPSCALL** applies only to calls `@f es` to a known function f , on the condition that a DPS variant has been generated for f : $f \in \text{dom}(\xi)$. In this case, the function call can be compiled to a call to the DPS variant $\xi[f]$, transferring the responsibility to write to the destination to the callee. This is the case where the DPS transform is beneficial, as this transformation may turn a non-tail-call into a tail-call – when it occurs under a block, in a subterm that was moved to tail position. This rule is selected for `map` in our `y :: map f xs` example.

Finally, there is a catch-all rule **DPSBASE** that applies in any case, in particular whenever none of the other rules can be selected. This case trivially realizes the DPS calling convention by evaluating the subterm to a result and writing this result in the desired destination. This is what happens in the base case of `map_dps`, where the empty list `[]` is transformed into `dst.(idx) $\leftarrow []$` .

2.3 Realizing the relation as a function

Our Coq formalization includes a function that takes an input program and outputs a related program, following the one-pass implementation approach that we introduced in the OCAML compiler (Section 3.5).

3 OCAML IMPLEMENTATION

For reasons of space, we moved some of the content in this section to [Appendix B](#):

- [Appendix B.1](#) discusses alternative language implementation techniques that do not require TMC.
- [Appendix B.2](#) provides a summary of the history of our implementation (started in 2015, restarted in 2020, merged in 2021).
- [Appendix B.3](#) exhibits an applicative functor structure that makes our implementation nicely compact and readable.

- [Appendix B.4](#) surveys the adoption of the TMC transformation in the standard library, and in third-party OCAML codebases, that happened since the feature was released in 2022.

3.1 Examples

Many functions that consume and produce lists are tail-recursive-modulo-cons, in the sense that all they have a TMC decomposition where all recursive calls are in TMC position. Notable functions include `map`, as already discussed, but also for example:

```
let[@tail_mod_cons] rec filter p =      let[@tail_mod_cons] rec merge cmp l1 l2 =
  function                               match l1, l2 with
  | [] → []                             | [], l | l, [] → l
  | x :: xs →                             | h1 :: t1, h2 :: t2 →
    if p x                               if cmp h1 h2 <= 0
    then x :: filter p xs                 then h1 :: merge cmp t1 l2
    else filter p xs                     else h2 :: merge cmp l1 t2
```

TMC is not useful only for lists or other “linear” data types, with at most one recursive occurrence of the datatype in each constructor.

A non-example. Consider a `map` function on binary trees:

```
let[@tail_mod_cons] rec map f = function
| Leaf v → Leaf (f v)
| Node(t1, t2) → Node(map f t1, (map[@tailcall]) f t2)
```

In this function, there are two recursive calls, but only one of them can be optimized; we used the `[@tailcall]` attribute to direct our implementation to optimize the call to the right child, as we will discuss later. This is a *bad* example of TMC usage in most cases, given that

- If the tree is arbitrary, there is no reason that it would be right-leaning rather than left-leaning. Making only the right-child calls tail-calls does not protect us from stack overflows.
- If the tree is known to be balanced, then in practice the depth is probably very small in both directions, so the TMC transformation is not necessary to have a well-behaved function.

Interesting non-linear examples. There are interesting examples of TMC-transformation on functions operating on tree-like data structures, when there are natural assumptions about which child is likely to contain a deep subtree. The OCAML compiler itself contains a number of them; consider for example the following function from the `Cmm` module, one of its lower-level program representations:

```
let[@tail_mod_cons] rec map_tail f = function
| Clet(id, exp, body) →
  Clet(id, exp, map_tail f body)
| Cifthenelse(cond, ifso, ifnot) →
  Cifthenelse(cond, map_tail f ifso, (map_tail[@tailcall]) f ifnot)
| Csequence(e1, e2) →
  Csequence(e1, map_tail f e2)
| Cswitch(e, tbl, el) →
  Cswitch(e, tbl, Array.map (map_tail f) el)
[...]
```

```

TailCtxFrame    $\ni T$  ::= let x = e in  $\square$  | if e then  $\square$  else  $\square$ 
ConsCtxFrame    $\ni K$  ::= { t, e,  $\square$  } | { t,  $\square$ , e }
TMCFrame        $\ni U$  ::= T | K
TailCtx         $\ni T^*$  ::=  $\square$  | T |  $T^*[T^*]$ 
TMCCTX          $\ni U^*$  ::=  $\square$  | U |  $U^*[U^*]$ 

```

Fig. 8. DATA_{LANG} contexts for optimizable calls

This function is traversing the “tail” context of an arbitrary program term – a meta-example! The `Cifthenelse` node acts as our binary-node constructor, we do not know which side is likely to be larger, so TMC is not so interesting. The recursive calls for `Cswitch` are not in TMC position. But on the other hand the `Clet`, `Csequence` cases are very beneficial to have in TMC: while they have several recursive subtrees, they are in practice only deeply nested in the direction that is turned into a tailcall by the transformation. The OCAML compiler does sometimes encounter machine-generated programs with a unusually long sequence of either constructions, and the TMC transformation may very well avoid a stack overflow in this case.

Another example would be [#9636](#), a patch to the OCAML compiler proposed in June 2020 by Mark Shinwell, to get a partially-tail-recursive implementation of the “Common Subexpression Elimination” (CSE) pass through a manual contiation-passing-style transform. Xavier Leroy remarked that the existing implementation in fact fits the TMC fragment. Not all recursive calls become tail-calls (this would require a more powerful transformation or a longer, less readable patch), but the behavior of TMC on the unchanged code matches the tail-call-ness proposed in the human-written patch.

3.2 Specifying which calls are in TMC position

To reason about the stack usage of their programs, users must understand which calls are in tail-modulo-cons position. Informally, they are the calls placed under any composition of either tail-recursive or constructor contexts.

We can in fact give a simple formal description of this intuition, here for DATA_{LANG} in [Figure 8](#). A tail frame T is a single term-former with holes in tail-position. A constructor frame K is a single constructor term-former (we omit deterministic blocks, which do not occur in the source). A tail context T^* is an arbitrary composition of tail-frame, and a TMC context U^* is an arbitrary composition of tail frames and constructor frames.

If a source function can be decomposed in a TMC context U^* with source expressions in its holes, some of which are calls to TMC-transformed functions, then our relation admits a DPS transformation where all those function calls are tail-calls, and this transformation is reachable in our OCAML implementation, possibly by adding some annotations.

Note in particular that we do not optimize calls to the same function we are defining, direct calls to arbitrary other functions can be transformed, if those functions have been annotated to be TMC-transformed. This is analogous to how most functional languages support arbitrary *tail calls* and not just tail self-recursion. We seamlessly support mutually recursive functions, DPS calls into locally-bound functions, etc.

3.3 Design choices

3.3.1 Resolving non-determinism. The main question faced by an implementation is how to resolve the inherent non-determinism. We identified fairly different kinds of non-determinism.

Choices with an obviously better alternative. Some choices have an obviously better alternative, because they allow to strictly improve more programs. For example, some implementations of TMC limit themselves to calls that are immediately inside a constructor: they would optimize the recursive call to f in $\text{Cons}(x, f\ y)$ but not, for example, in $\text{Cons}(x1, \text{Cons}(x2, f\ y))$, or in $\text{Cons}(x, \text{if } p \text{ then } f\ y \text{ else } z)$.

In the terms of Section 3.2, some implementations only allow to optimize calling contexts of the form $T^*[K]$ or $T^*[K^*]$. We support a more general situation $U^* = (T|K)^*$.

Choices with a subtly better alternative. In some cases it is possible to put a bit more work in the choice heuristics, to generate slightly better code, in terms of performance or readability. For example, some natural way to simplify the implementation would result in more subterms being transformed in destination-passing-style, only to finally use the **DPSBase** rule without any DPS function call. The generated code is slightly less pleasant to read, and in our experience it can be noticeably slower.

Code readability is important in our context of an on-demand program transformation used by performance experts: those experts will often read the intermediate representations produced by the compiler to check that their performance assumptions hold.

Incomparable choices: force the user to decide. The remaining choices are between incomparable alternatives, that could each be better than the other depending on the specific program or programmer intent. Consider again our binary tree example, $\text{Node}(\text{map } f \text{ left}, \text{map } f \text{ right})$: we could make either the first or the second argument a tail-call.

Our policy in such cases is to let the user decide, by providing control over the transformation choices using `[@tailcall]` program annotations, and to force the user to decide by raising an error in case of ambiguity:

```
| Cifthenelse(cond, ifso, ifnot) →
   Cifthenelse(cond, map_tail f ifso, map_tail f ifnot)
```

```
Error: "[@tail_mod_cons]": this constructor application may be TMC-transformed
in several different ways. Please disambiguate by adding an explicit
"@tailcall" attribute to the call that should be made tail-recursive,
or a "[@tailcall false]" attribute on calls that should not be
transformed.
```

This approach would be incompatible with a view of the TMC transformation as an implicit optimization, applied whenever possible – in that case one would rather have the compiler make arbitrary choices rather than fail. We rather view TMC as a tool for expert users to better reason about program performance. It should be predictable and flexible (let the user express their intent). Failing on the lack of annotations is an acceptable way to drive user interaction.

3.3.2 First-order implementation. A notable limitation of the OCAML implementation is that it is first-order and non-modular in nature: only direct calls to known functions can be converted into DPS style, and the availability of a DPS variant is not exposed through module abstractions.

It would be possible to allow DPS calls through external modules or higher-order function parameters, by annotating interfaces and function arguments with a `[@tail_mod_cons]` annotation, and elaborating them into a pair of functions, the direct-style and the DPS version.

3.4 Constructor compression

The translation as we described it formally in Section 2.2 generates unpleasant (and slower) code when many constructors are nested before the recursive call. For example, consider this strange function: element in a list:

```
let rec dup = function [] → [] | x :: xs → x :: x :: dup xs
```

Such nested constructors are common in compiler codebases, for example a desugaring pass that transforms a single term-former into a composition of several simpler term-formers, and applies recursively to its subterms.

Following the TMC transformation naively, the DPS version would two location and performs two writes, leading to more complex code with worse constant factors. We introduced “constructor compression”, an optimization of the generated code that avoids creating intermediary destinations for nested constructors. Compare the naive translation of `dup`, on the left, and our compressed translation on the right:

<pre>let rec dup_dps dst ofs = function [] → dst.(ofs) ← [] x :: xs → let dst1 = x :: ■ in dst.(ofs) ← dst1; let dst2 = x :: ■ in dst1.(1) ← dst2; dup_dps dst2 1 xs</pre>	<pre>let rec dup_dps dst ofs = function [] → dst.(ofs) ← [] x :: xs → let dst2 = x :: ■ in dst1.(1) ← x :: dst2; dup_dps dst2 1 xs</pre>
--	--

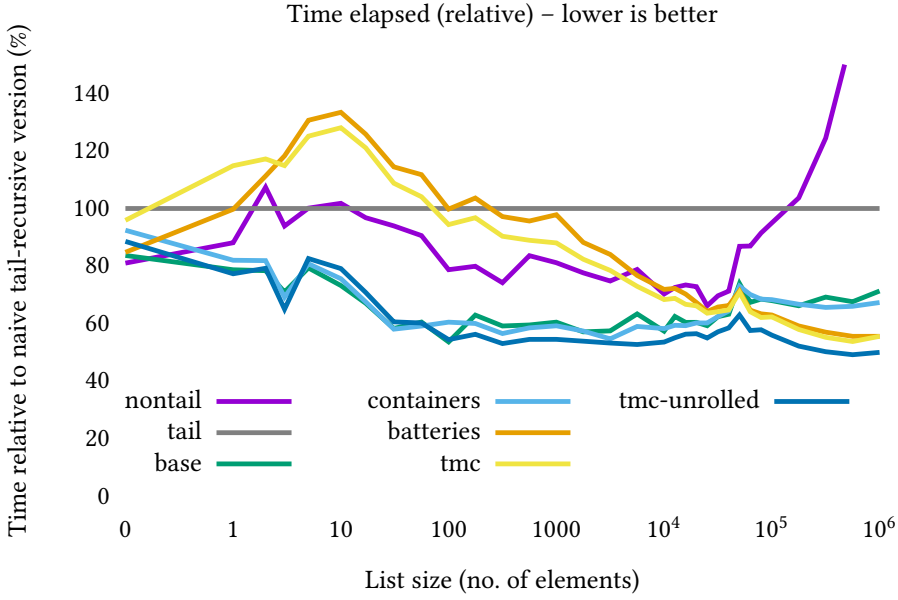
This is implemented by passing as an extra transformation parameter a stack of “delayed” constructor applications, that are in context and must be applied to the result of the subterm. When we encounter the final recursive call, we “reify” this stack: the last/innermost constructor in the stack becomes the new destination (`dst1` in the example above), and the rest of the stack is applied to the new destination when we write to the old destination. There are two subtleties:

- (1) `if p then e1 else e2` has two subterms which are transformed in DPS style, and naively passing the stack of delayed constructors to both subterms would duplicate code; instead of also reify the current stack when encountering such constructs.
- (2) This transformation may permute constructor applications after effectful subterms. If the constructor application contains effectful arguments, those must be `let`-bound at their original position to avoid changing the evaluation order.

3.5 Implementation

Implementing the TMC transform is not an obvious top-down or bottom-up traversal, because it relies on two informations flowing in opposite directions: the transformation is *possible* for subterms that are in tail-or-constructor position relative to the root of the function (top-down information), and it is *desirable* for subterms whose leaves contain calls to TMC-transformed functions (bottom-up information). A naive approach is to perform a recursive traversal that tracks the top-down context and, on each subterm, recursively traverses the whole subterm to check desirability; this has quadratic time complexity, which is best avoided in production compilers.

Instead of trying to transform each subterm in a single pass, our implementation computes for each subterm a “choice”, a summary of all the bottom-up information relevant to choose how to transform it – once we have the top-down information available. This includes (1) the direct-style transform of the subterm, (2) the DPS-style transform of the subterm, (3) metadata tracking whether

Fig. 9. `List.map` benchmark on OCAML 5.1

the transformation is beneficial (if a TMC-function call was found in tail-modulo-cons position), the list of TMC calls it contains (for error diagnostics), and whether some were explicitly requested by the user (for disambiguation). Computing transformed terms is not compositional, computing choices is compositional. We compute a choice for the whole function body (in one pass), from which we can directly extract the direct-style and DPS-style transformations of the function.

3.6 Evaluation: benchmarks

We measured the performance of `List.map` (`fun n → n + 1`) to validate our claims that the TMC transformation preserves program performance, and let us replace complex hand-optimized tail-recursive implementations.

The `List.map` versions in the graph are the following. We measure the code size (in lines) of each version, as a reasonable approximation of its implementation complexity.

nontail (5 lines of code) The naive, non-tail-recursive implementation.

tail (9 lines) The naive tail-recursive implementation, `List.rev (List.rev_map f xs)`.

base (78 lines) The implementation of Jane Street’s [Base](#) library (version 0.14.0). It is heavily hand-optimized to compensate for the costs of being tail-recursive.

containers (55 lines) Another standard-library extension by Simon Cruanes; it is the hand-optimized tail-recursive implementation we included in the Prologue.

batteries (29 lines) The implementation of the community-maintained [Batteries](#) library. It is actually written in destination-passing-style, using an unsafe encoding with `Obj.magic` to unsafely cast a mutable record into a list cell. (The trick comes from the older `Extlib` library, was introduced by Brian Hurt in 2003, and has a comment crediting Jacques Garrigue for the particular encoding used.)

tmc (5 lines) “Our” version, the last version of the Prologue: the result of applying our implementation of the TMC transformation to the simple, non-tail-recursive version.

tmc-unrolled (18 lines) The result of manually unrolling the **tmc** implementation three times, to be compare with **base** and **containers** that use manual unrolling as well.

The benchmarks reports the relative performance compared to the naive tail-recursive version as our baseline. They were run on OCAML 5.1 in July 2024, on a Linux machine with an AMD Ryzen processor fixed at a 3Ghz frequency, looping each measurement for 5s (a single `List.map` run takes between 7ns, for empty lists, and 89ms on lists with a million element).

Qualitatively we see that there are four groups:

- **tmc**, **batteries** perform very on large lists, but they are slower than the baseline on small lists.
- **nontail** performs better than **tmc**, **batteries** on list sizes up to 10^4 , and much worse on larger lists.
- **base**, **containers** perform noticeably better than **nontail** at all sizes, but worse than the TMC versions above size 10^4 .
- **tmc-unrolled** is the best option, it performs as well as **base** and **containers** before 10^4 , and as well as **tmc**, **batteries** afterwards.

Our interpretation of the result is that some unrolling makes a noticeable performance difference for such a short function: **tmc** is not good enough on smaller lists, but **tmc-unrolled** is the best-performing, despite much simpler than the **base** and **containers** version.

Asymptotics of nontail. The bad behavior of **nontail** on large lists comes from a quadratic behavior on very large call stacks, coming from a repeated scan of the call stack during minor collections. (The OCAML compiler and runtime could be tweaked to avoid this quadratic behavior, at the cost of some small constant overhead on function returns.)

4 SPECIFYING TMC

In this section, we gradually introduce aspects of our relational separation logic, by introducing our specifications for the direct-style and destination-passing-style transformations of [Section 2](#) in relational separation logic.

4.1 Direct transformation

Intuitively, the direct transformation $e_s \xrightarrow[\text{dir}]{\xi} e_t$ preserves the behaviours of the source expression e_s . Basically, e_s and e_t compute the same thing. Using *relational Hoare logic*, an assumed extension of standard Hoare logic relating two expressions, we would write:

$$\left\{ e_s \xrightarrow[\text{dir}]{\xi} e_t \right\} e_s \gtrsim e_t \{v_s, v_t. v_s \approx v_t\}$$

The informal meaning of this specification is that 1) e_t refines e_s in the sense that any behaviour (converging, diverging or stuck execution) of e_t is also a behaviour of e_s and 2) if e_t converges to value v_t , e_s also converges to some value v_s that is *similar* to v_s . We will formalize the notion of *behaviour* in [Section 6](#) and that of *similarity* later in this section. For the time being, the ready may assume similarity is just equality on values.

4.2 DPS transformation

The DPS transformation $(\ell, i, e_s) \xrightarrow[\text{dps}]{\xi} e_t$ is parameterized by a destination (ℓ, i) pointing to an uninitialized field of some block. Intuitively, e_t computes the same thing as e_s but writes it into the

$$\begin{array}{c}
\text{() } \approx \text{() } \quad i \approx i \quad t \approx t \quad b \approx b \quad \frac{\forall i \in \text{Index. } (\ell_s + i) \overset{\text{bij}}{\approx} (\ell_t + i)}{\ell_s \approx \ell_t} \quad \frac{f \in \text{dom}(p_s)}{@f \approx @f}
\end{array}$$

Fig. 10. Similarity in iProp

destination instead of returning it. Using *relational separation logic*, an assumed further extension of relational Hoare logic with the concepts of separation logic, we write:

$$\left\{ (\ell, i, e_s) \overset{\xi}{\underset{\text{dps}}{\rightsquigarrow}} e_t * (\ell + i) \mapsto_t \blacksquare \right\} e_s \gtrsim e_t \{v_s, \text{()}. \exists v_t. (\ell + i) \mapsto_t v_t * v_s \approx v_t\}$$

The *points-to assertion* $(\ell + i) \mapsto_t$ – expresses that the destination is uniquely owned by the transformed program. The specification requires e_t to fill it with some value that is similar to the returned source value. This captures destination-passing style in simple terms.

4.3 Heap bijection

Defining value similarity as just syntactic equality is not sufficient: corresponding source and target block allocations are not done in locksteps, so their location may differ. For example, consider the `map` function and its DPS transform from Section 1.2. In the source program, the `cons` cell `y :: @map (fn, xs)` is allocated after the recursive call. In the transformed program, the corresponding block is allocated before the call.

To deal with this, we introduce a *heap bijection* as in SIMULIRIS [Gäher, Sammler, Spies, Jung, Dang, Krebbers, Kang and Dreyer 2022]. Corresponding source and target locations – pointing to block fields in our semantics – are logically and permanently registered in this bijection. We can then refer to a registered pair of locations through the assertion $\ell_s \overset{\text{bij}}{\approx} \ell_t$. This mechanism is formalized by the **BIJINSERT** rule:

$$\text{BIJINSERT} \quad \frac{\ell_s \mapsto_s v_s \quad \ell_t \mapsto_t v_t \quad v_s \approx v_t}{\ell_s \overset{\text{bij}}{\approx} \ell_t}$$

We can formally define value similarity $v_s \approx v_t$ in Figure 10. It coincides with equality except on blocks, for which we require all fields to be registered in the bijection.

5 RELATIONAL SEPARATION LOGIC

In this section, we describe the inference rules of our relational program logic, presented in Figure 11. We omitted some congruence rules for brevity. The relation $e_s \gtrsim e_t \{\Phi\}$ relates a source expression e_s with a target expression e_t under a postcondition Φ . We extend it to support a precondition in the standard way:

$$\{P\} e_s \gtrsim e_t \{\Phi\} := \Box (P * e_s \gtrsim e_t \{\Phi\})$$

Compared to the specifications of Section 4, we introduced an additional protocol parameter X . We explain it together with the **RELPROTOCOL** rule in Section 6.

Language-independent rules. The following rules are independent of **DATALANG** and could be reused as is in further works.

RELPOST states that two expressions are related when they are in the relational postcondition.

RELSTUCK relates *strongly stuck* expressions. An expression is strongly stuck when it is stuck for any heap state.

RELPOST	RELSTUCK
$\frac{\Phi(v_s, v_t)}{v_s \succeq v_t \langle X \rangle \{ \Phi \}}$	$\frac{\text{strongly-stuck}_{p_s}(e_s) \quad \text{strongly-stuck}_{p_t}(e_t)}{e_s \succeq e_t \langle X \rangle \{ \Phi \}}$
RELBIND	
$\frac{e_s \succeq e_t \langle X \rangle \{ \lambda(v_s, v_t). E_s[v_s] \succeq E_t[v_t] \langle X \rangle \{ \Phi \} \}}{E_s[e_s] \succeq E_t[e_t] \langle X \rangle \{ \Phi \}}$	
RELSRCPURE	RELTGTPURE
$\frac{e_s \xrightarrow[pure]{p_s} e'_s \quad e'_s \succeq e_t \langle X \rangle \{ \Phi \}}{e_s \succeq e_t \langle X \rangle \{ \Phi \}}$	$\frac{e_t \xrightarrow[pure]{p_t} e'_t \quad e_s \succeq e'_t \langle X \rangle \{ \Phi \}}{e_s \succeq e_t \langle X \rangle \{ \Phi \}}$
RELSRCBLOCK1	RELSRCBLOCK2
$\frac{\text{let } x_1 = e_{s1} \text{ in } \text{let } x_2 = e_{s2} \text{ in } \succeq e_t \langle X \rangle \{ \Phi \} \quad [t, x_1, x_2]}{\{t, e_{s1}, e_{s2}\} \succeq e_t \langle X \rangle \{ \Phi \}}$	$\frac{\text{let } x_2 = e_{s2} \text{ in } \text{let } x_1 = e_{s1} \text{ in } \succeq e_t \langle X \rangle \{ \Phi \} \quad [t, x_1, x_2]}{\{t, e_{s1}, e_{s2}\} \succeq e_t \langle X \rangle \{ \Phi \}}$
RELTGTBLOCK	
$\frac{\text{let } x_1 = e_{t1} \text{ in } e_s \succeq \text{let } x_2 = e_{t2} \text{ in } \langle X \rangle \{ \Phi \} \quad [t, x_1, x_2]}{e_s \succeq \{t, e_{t1}, e_{t2}\} \langle X \rangle \{ \Phi \}}$	$\frac{\text{let } x_2 = e_{t2} \text{ in } \text{let } x_1 = e_{t1} \text{ in } \langle X \rangle \{ \Phi \} \quad [t, x_1, x_2]}{e_s \succeq \{t, e_{t1}, e_{t2}\} \langle X \rangle \{ \Phi \}}$
RELSRCBLOCKDET	RELTGTBLOCKDET
$\frac{\forall \ell_s. \ell_s \mapsto_s (t, v_{s1}, v_{s2}) \multimap \ell_s \succeq e_t \langle X \rangle \{ \Phi \} \quad [t, v_{s1}, v_{s2}] \succeq e_t \langle X \rangle \{ \Phi \}}{[t, v_{s1}, v_{s2}] \succeq e_t \langle X \rangle \{ \Phi \}}$	$\frac{\forall \ell_t. \ell_t \mapsto_t (t, v_{t1}, v_{t2}) \multimap e_s \succeq \ell_t \langle X \rangle \{ \Phi \} \quad e_s \succeq [t, v_{t1}, v_{t2}] \langle X \rangle \{ \Phi \}}{e_s \succeq [t, v_{t1}, v_{t2}] \langle X \rangle \{ \Phi \}}$
RELSRCLOAD	RELTGTLOAD
$\frac{(\ell_s + i) \mapsto_s v_s \quad (\ell_s + i) \mapsto_s v_s \multimap v_s \succeq e_t \langle X \rangle \{ \Phi \}}{\ell_s.(i) \succeq e_t \langle X \rangle \{ \Phi \}}$	$\frac{(\ell_t + i) \mapsto_t v_t \quad (\ell_s + i) \mapsto_t v_t \multimap e_s \succeq v_t \langle X \rangle \{ \Phi \}}{e_s \succeq \ell_t.(i) \langle X \rangle \{ \Phi \}}$
RELSRCSTORE	RELTGTSTORE
$\frac{(\ell_s + i) \mapsto_s v_s \quad (\ell_s + i) \mapsto_s v'_s \multimap () \succeq e_t \langle X \rangle \{ \Phi \}}{\ell_s.(i) \leftarrow v'_s \succeq e_t \langle X \rangle \{ \Phi \}}$	$\frac{(\ell_t + i) \mapsto_t v_t \quad (\ell_t + i) \mapsto_t v'_t \multimap e_s \succeq () \langle X \rangle \{ \Phi \}}{e_s \succeq \ell_t.(i) \leftarrow v'_t \langle X \rangle \{ \Phi \}}$
RELOAD	RELSTORE
$\frac{\ell_s \approx \ell_t \quad \forall v_s, v_t. v_s \approx v_t \multimap \Phi(v_s, v_t)}{\ell_s.(i) \succeq \ell_t.(i) \langle X \rangle \{ \Phi \}}$	$\frac{\ell_s \approx \ell_t \quad v_s \approx v_t \quad \Phi((), ())}{\ell_s.(i) \leftarrow v_s \succeq \ell_t.(i) \leftarrow v_t \langle X \rangle \{ \Phi \}}$
RELPROTOCOL	
$\frac{X(\Psi, e_s, e_t) \quad \forall e'_s, e'_t. \Psi(e'_s, e'_t) \multimap e'_s \succeq e'_t \langle X \rangle \{ \Phi \}}{e_s \succeq e_t \langle X \rangle \{ \Phi \}}$	

Fig. 11. Relational separation logic (excerpt)

$$\begin{aligned}
X_{\text{dir}}(\Psi, e_s, e_t) &:= \exists f, v_s, v_t. \\
&\quad f \in \text{dom}(p_s) * e_s = @f v_s * e_t = @f v_t * \\
&\quad v_s \approx v_t * \\
&\quad \forall w_s, w_t. w_s \approx w_t \multimap \Psi(w_s, w_t) \\
X_{\text{DPS}}(\Psi, e_s, e_t) &:= \exists f, f_{\text{dps}}, v_s, \ell_1, \ell_2, \ell, i, v_t. \\
&\quad f \in \text{dom}(p_s) * \xi[f] = f_{\text{dps}} * e_s = @f v_s * e_t = @f_{\text{dps}} \ell_1 * \\
&\quad (\ell_1 + 1) \mapsto_t (\ell_2, v_t) * (\ell_2 + 1) \mapsto_t (\ell, i) * (\ell + i) \mapsto \blacksquare * v_s \approx v_t \\
&\quad \forall w_s, w_t. (\ell + i) \mapsto w_t * w_s \approx w_t \multimap \Psi(w_s, ()) \\
X_{\text{TMC}} &:= X_{\text{dir}} \sqcup X_{\text{DPS}} = \lambda(\Psi, e_s, e_t). X_{\text{dir}}(\Psi, e_s, e_t) \vee X_{\text{DPS}}(\Psi, e_s, e_t)
\end{aligned}$$

Fig. 12. TMC protocol (X_{TMC})

RELBIND is a standard bind rule sequencing computations on both sides.

RELSRCPURE and **RELTGTPURE** let us take pure reduction steps in either the source or target. Pure steps (omitted for brevity) are the reduction steps that are deterministic and do not depend on the state.

Language-specific rules: non-determinism. $e_s \gtrsim e_t \langle X \rangle \{\Phi\}$ asserts that e_t refines e_s : any behavior of e_t is also a behavior of e_s . Consequently, non-determinism is treated differently in the source and target: we treat non-determinism as *angelic* in source reductions and *demonic* in target reductions.

Our operational semantics uses non-determinism in the reduction of constructors: $\{t, e_1, e_2\}$ reduces to $[t, x_1, x_2]$, where x_1 and x_2 are bound to e_1 and e_2 in some non-deterministic order. In the program logic, the user may *choose* an order for the source reduction, by using one of the rules **RELSRCBLOCK1** or **RELSRCBLOCK2**, and it has to prove that the expressions are related against *any* target order, by proving the two premises of the rule **RELTGTBLOCK**.

Language-specific rules: private locations. We can reason on points-to assertions — that we interpret as locations private to the source or target — in a standard way. From a deterministic constructor $[t, v_1, v_2]$, we can apply **RELSRCBLOCKDET** or **RELTGTBLOCKDET**, yielding a points-to assertion for the allocated block. The rules **RELSRCLOAD** and **RELTGTLOAD** let us load the pointed value while **RELSRCSTORE** and **RELTGTSTORE** let us update it with a new value.

Language-specific rules: locations in the bijection. Corresponding source and target locations registered in the bijection through **BIJINSERT** have given up their respective points-to assertions but can still be accessed using the rules **RELLOAD** and **RELSTORE**.

RELLOAD states that simultaneously loading from two corresponding blocks yields similar values.

RELSTORE let us simultaneously store similar values into the same field of two corresponding blocks.

Together, these two rules enforce the bijection invariant: the contents of corresponding blocks are always similar.

6 ABSTRACT PROTOCOLS

In [Section 8](#), we explain how our relation is defined coinductively and the first step of the proof essentially amounts to coinduction. To internalize the coinduction hypothesis into the program logic, we introduce an additional parameter X , a *protocol* [de Vilhena and Pottier 2021], which are general proof-state transformers of type

$$(\text{Expr} \rightarrow \text{Expr} \rightarrow \text{iProp}) \rightarrow \text{Expr} \rightarrow \text{Expr} \rightarrow \text{iProp}$$

Protocols are used in the logic via the **RELPROTOCOL** rule. A pair of expressions e_s and e_t is supported by the protocol when it relates them to a postcondition Ψ , capturing the possible results

of an abstract/axiomatic transition from e_s and e_t . To conclude that e_s and e_t are related, one must prove that any two e'_s and e'_t accepted by this postcondition Ψ remain related.

6.1 TMC protocols

In our correctness proof for the TMC transformation, we use a sepecific protocol X_{TMC} defined in Figure 12 by combining two sub-protocols X_{dir} and X_{DPS} for the direct-style and DPS-style functions. Our coinduction hypothesis assumes toplevel function calls to be compatible with the direct and DPS specifications that we want to prove, and allows to reason about recursive calls to those functions inside the function bodies we are trying to relate.

X_{dir} specifies the *direct calling convention* induced by the direct transformation. It requires e_s and e_t to be function calls to the same function with similar arguments. To apply it, users can choose any postcondition implied by value similarity. This rule is equivalent to the SIM-CALL rule of SIMULIRIS, and most useful protocols are formed by combining $i\text{Prop}_{\text{dir}}$ with other, more specialized protocols.

X_{DPS} specifies the *DPS calling convention* induced by the DPS transformation. It requires e_s to be a function call to a TMC-transformed function f and e_t to be a function call to the DPS transform of f . As in the DPS specification, ownership of the destination must be passed to the protocol. To apply it, users can choose any postcondition implied by the postcondition of the DPS specification, including the recovered ownership of the modified destination.

6.2 Other examples of protocols

Our program logic can be instantiated with other protocols to reason other program transformations. To demonstrate this generality, we have also verified an inlining and an accumulator-passing-style (APS) transformation — both included in our mechanization.

Inlining: Here, the relation $e_s \rightsquigarrow e_t$ allows e_t to recursively inline functions in e_s . As with TMC, it captures all possible inlining strategies.

This relation can be proved correct by using a fairly simple protocol (combined with X_{dir}) relating a source function and its body:

$$\begin{aligned} X_{\text{inline}}(\Psi, e_s, e_t) &:= \exists f, x, e'_s, e'_t, v_s, v_t. \\ &e_s = @f v_s * v_s \approx v_t * \\ &p_s[f] = (\text{rec } x = e'_s) * e'_s \rightsquigarrow e'_t * e_t = (\text{let } x = v_t \text{ in } e'_t) * \\ &\forall w_s, w_t. w_s \approx w_t \multimap \Psi(w_s, w_t) \end{aligned}$$

Accumulator-passing style. : The APS transformation is a variant of the TMC transformation where the contexts that are made tail-recursive are applications of associative arithmetic operators, typically of the form $(e + \square)$ or $e_1 + (e_2 \times \square)$. (See the discussion in Leijen and Lorenzen [2023].)

We defined an APS transformation, after extending DATA LANG with integers and arithmetic operations. We verify it with a protocol similar to X_{DPS} that allows calling the APS transform of a source function with an integer accumulator:

$$\begin{aligned} X_{\text{APS}}(\Psi, e_s, e_t) &:= \exists f, f_{\text{aps}}, v_s, v_{\text{acc}}, v_t. \\ &f \in \text{dom}(p_s) * \xi[f] = f_{\text{aps}} * \\ &v_s \approx v_t * e_s = @f v_s * e_t = @f_{\text{aps}}(v_{\text{acc}}, v_t) * \\ &\forall v'_s, e'_t. \\ &\text{match } v'_s \text{ with } n \Rightarrow e'_t = v_{\text{acc}} + n \mid _ \Rightarrow \text{strongly-stuck}_{p_t}(e'_t) \text{ end } \multimap \\ &\Psi(v'_s, e'_t) \end{aligned}$$

One subtlety is that our DATA LANG language is untyped, so arithmetic operations (here addition) may get stuck on non-integer values. If the function call $@f v_s$ in the source program returns a

$$\begin{aligned}
\text{wf}(\Gamma) &:= \forall x. \exists v_s, v_t. \Gamma(x) = (v_s, v_t) * v_s \approx v_t \\
\Gamma_s(x) &:= \Gamma(x)_1 \\
\Gamma_t(x) &:= \Gamma(x)_2 \\
e_s \geq e_t \{ \Phi \} &:= \forall \Gamma. \text{wf}(\Gamma) \multimap \Gamma_s(e_s) \geq \Gamma_t(e_t) \{ \Phi \} \\
\{ P \} e_s \geq e_t \{ \Phi \} &:= \Box (P \multimap e_s \geq e_t \{ \Phi \})
\end{aligned}$$

Fig. 13. Runtime relation

non-integer value, then the outer context $x_{acc} + \Box$ gets stuck. But in the transformed program, this failure happens inside the body of the APS-transformed function $@_{faps}$. To represent this failure case in our protocol, the postcondition Ψ relates a non-integer source return value v'_s with any strongly stuck expression e'_t in the target. This relies on the generality of our protocols being predicate transformers on expressions, not just values.

7 PROOF OF THE SPECIFICATION

In this section, we prove the specifications of [Section 4](#).

As mentioned in [Section 6](#), we instantiate our program logic with a specific protocol X_{TMC} defined in [Figure 12](#). We define a shorthand notation for this instantiation:

$$e_s \geq e_t \{ \Phi \} := e_s \geq e_t \langle X_{TMC} \rangle \{ \Phi \}$$

So far, we worked with *closed* expressions, that have no free variables. We need to generalize the specifications to *open* expressions that may have free variables, as is standard. To do so, we introduce a *runtime relation* $e_s \geq e_t \{ \Phi \}$ in [Figure 13](#). It requires $\Gamma_s(e_s)$ and $\Gamma_t(e_t)$ to be related for any *well-formed closing bisubstitution* $\Gamma \in \mathbb{X} \rightarrow \text{Val} \times \text{Val}$. In practice, Γ contains let-reduced binders and their substitute source and target values.

In addition, we will only consider *valid source expressions* — denoted by $\text{wf}(e_s)$ —, *i.e.* those that do not involve any location, deterministic block expressions, or undefined source function.

LEMMA 7.1 (SPECIFICATION OF DIRECT TRANSFORMATION).

$$\left\{ \text{wf}(e_s) * e_s \xrightarrow[\text{dir}]{\xi} e_t \right\} e_s \geq e_t \{ v_s, v_t. v_s \approx v_t \}$$

LEMMA 7.2 (SPECIFICATION OF DPS TRANSFORMATION).

$$\left\{ \text{wf}(e_s) * (\ell, i, e_s) \xrightarrow[\text{dps}]{\xi} e_t * (\ell + i) \mapsto_t \blacksquare \right\} e_s \geq e_t \{ v_s, (\cdot). \exists v_t. (\ell + i) \mapsto_t v_t * v_s \approx v_t \}$$

Both proofs proceed by induction over e_s and mutual induction over $e_s \xrightarrow[\text{dir}]{\xi} e_t$ and $(\ell, i, e_s) \xrightarrow[\text{dps}]{\xi} e_t$ — we refer to our mechanization for the details.

8 SIMULATION

So far, we assumed a program logic satisfying a set of reasoning rules. In the section, we prove that our rules are sound: they imply a *simulation* à la SIMULIRIS [[Gähler, Sammler, Spies, Jung, Dang, Krebbers, Kang and Dreyer 2022](#)]. This simulation comes with an *adequacy theorem* that allows to extract a *behavior refinement* in the meta-logic (Coq, without Iris), our final soundness theorem.

$$\begin{aligned}
\text{sim-body}_X &:= \lambda \text{sim}. \lambda \text{sim-inner}. \lambda (\Phi, e_s, e_t). \forall \sigma_s, \sigma_t. I(\sigma_s, \sigma_t) \multimap \Rightarrow \\
&\quad \left[\begin{array}{l}
\textcircled{1} \quad I(\sigma_s, \sigma_t) * \Phi(e_s, e_t) \\
\textcircled{2} \quad I(\sigma_s, \sigma_t) * \text{strongly-stuck}_{p_s}(e_s) * \text{strongly-stuck}_{p_t}(e_s) \\
\textcircled{3} \quad \exists e'_s, \sigma'_s. (e_s, \sigma_s) \xrightarrow{p_s^+} (e'_s, \sigma'_s) * I(\sigma'_s, \sigma_t) * \text{sim-inner}(\Phi, e'_s, e_t) \\
\textcircled{4} \quad \text{reducible}_{p_t}(e_t, \sigma_t) * \forall e'_t, \sigma'_t. (e_t, \sigma_t) \xrightarrow{p_t} (e'_t, \sigma'_t) \multimap \Rightarrow \\
\quad \vee \left[\begin{array}{l}
\textcircled{A} \quad I(\sigma_s, \sigma'_t) * \text{sim-inner}(\Phi, e_s, e'_t) \\
\textcircled{B} \quad \exists e'_s, \sigma'_s. (e_s, \sigma_s) \xrightarrow{p_s^+} (e'_s, \sigma'_s) * I(\sigma'_s, \sigma'_t) * \text{sim}(\Phi, e'_s, e'_t)
\end{array} \right] \\
\textcircled{5} \quad \exists E_s, e'_s, E_t, e'_t, \Psi. \\
\quad e_s = E_s[e'_s] * e_t = E_t[e'_t] * X(\Psi, e'_s, e'_t) * I(\sigma_s, \sigma_t) * \\
\quad \forall e''_s, e''_t. \Psi(e''_s, e''_t) \multimap \text{sim-inner}(\Phi, E_s[e''_s], E_t[e''_t])
\end{array} \right] \\
\text{sim-inner}_X &:= \lambda \text{sim}. \mu \text{sim-inner}. \text{sim-body}_X(\text{sim}, \text{sim-inner}) \\
\text{sim}_X &:= \nu \text{sim}. \text{sim-inner}_X(\text{sim})
\end{aligned}$$

$$e_s \geq e_t \langle X \rangle [\Phi] := \text{sim}_X(\Phi, e_s, e_t)$$

$$e_s \geq e_t \langle X \rangle \{\Phi\} := e_s \geq e_t \langle X \rangle [\lambda(e'_s, e'_t). \exists v_s, v_t. e'_s = v_s * e'_t = v_t * \Phi(v_s, v_t)]$$

Fig. 14. Simulation with protocol

8.1 Definition

Our simulation is defined in Figure 14. It is largely inspired by the SIMULIRIS simulation — simplified due to the absence of concurrency. The main difference lies in the protocol clause ⑤, which is a generalization of the function call clause of SIMULIRIS.

The definition consists of two nested fixpoints **sim** and **sim-inner**. **sim** is a greatest fixpoint (coinduction) allowing expressions to diverge (in a controlled way, see clause ④ⓑ). **sim-inner** is a least fixpoint (induction) allowing source and target stuttering (see clauses ③ and ④ⓐ). The *state interpretation* $I(\sigma_s, \sigma_t)$ intuitively materializes the invariant of the simulation, including the heap bijection (see Section 4); it is systematically maintained. We now review the six clauses.

① *Postcondition*. The simulation can stop when the postcondition is satisfied.

② *Stuck expressions*. The simulation can also stop on simultaneously stuck expressions.

③ *Target stuttering*. The source expression can angelically take some steps. This can only happen finitely many times in **sim-inner**. If we used **sim** instead, a silent loop in the source could be simulated by anything, breaking preservation of divergence.

④ⓐ *Source stuttering*. The target expression can demonically take one step. This can also only happen finitely many times in **sim-inner**. If we used **sim** instead, a silent loop in the target could simulate any source expression, breaking preservation of termination.

④ⓑ *Synchronization*. Alternatively, both expressions can simultaneously take one step. This can happen infinitely many times in **sim**. If we used **sim-inner** instead, we would be unable to relate divergent programs.

⑤ *Protocol application*. Finally, we can apply the protocol under evaluation contexts. We can choose any postcondition Ψ accepted by the protocol and assume it to prove the continuation.

$$\begin{array}{c}
() \sim () \quad i \sim i \quad t \sim t \quad b \sim b \quad \ell_s \sim \ell_t \quad @f \sim @f \\
\\
\frac{v_s \sim v_t}{\text{Conv}(v_s) \sqsupseteq \text{Conv}(v_t)} \quad \frac{e_s \notin \text{Val} \quad e_t \notin \text{Val}}{\text{Conv}(e_s) \sqsupseteq \text{Conv}(e_t)} \quad \text{Div} \sqsupseteq \text{Div} \\
\\
\text{behaviours}_p(e) := \{ \text{Conv}(e') \mid \exists \sigma. (e, \emptyset) \xrightarrow{p}^* (e', \sigma) \wedge \text{irreducible}_p(e', \sigma) \} \uplus \\
\{ \text{Div} \mid (e, \emptyset) \uparrow_p \} \\
e_s \sqsupseteq e_t := \forall b_t \in \text{behaviours}_{p_t}(e_t). \exists b_s \in \text{behaviours}_{p_s}(e_s). b_s \sqsupseteq b_t \\
p_s \sqsupseteq p_t := \forall f \in \text{dom}(p_s). v. \text{wf}(v) \implies @f v \sqsupseteq @f v
\end{array}$$

Fig. 15. Program refinement

8.2 Adequacy

Informally, our simulation is *adequate* in the sense that if e_s simulates e_t , e_t refines e_s , *i.e.* the behaviours of e_t are included in the behaviours of e_s . Crucially, this is only true for a *closed* simulation, *i.e.* not using the protocol — equivalently using the empty protocol \perp . Formally, we write:

LEMMA 8.1 (SIMULATION ADEQUACY). $(\vdash e_s \gtrsim e_t \langle \perp \rangle \{ \approx \}) \implies e_s \sqsupseteq e_t$

The notions of *behaviours* and *refinement* are defined in Figure 15. We consider not only converging behaviours (resulting in values or stuck expressions) but also diverging behaviours. Expression refinement $e_s \sqsupseteq e_t$ is *termination-preserving*: if e_s always terminates, so does e_t . Program refinement $p_s \sqsupseteq p_t$, also defined in Figure 15, requires any source function call in p_t to behave as in p_s .

Adequacy is proved with an empty protocol, but we combine it with a *simulation closure* result for any protocol X . This lemma says that X respects a certain admissibility condition, then program relations established using this protocol also hold with the empty protocol:

LEMMA 8.2 (SIMULATION CLOSURE). *For any protocol X , we have:*

$$\begin{aligned}
& \Box (\forall \Psi, e_s, e_t. X(\Psi, e_s, e_t) \multimap \text{sim-inner}_{\perp}(\lambda(_, e'_s, e'_t). e'_s \gtrsim e'_t \langle X \rangle [\Psi])(\perp, e_s, e_t)) \multimap \\
& e_s \gtrsim e_t \langle X \rangle [\Phi] \multimap e_s \gtrsim e_t \langle \perp \rangle [\Phi])
\end{aligned}$$

In simple terms, any expressions accepted by a valid protocol have to be able to take one simultaneous step after potential stuttering to return in the open simulation.

8.3 Transformation soundness

We can finally express the soundness of the TMC transformation: if program p_s is well-formed and transforms into program p_t , p_t refines p_s . A program is well-formed when its function definitions are well-formed and well-scoped.

THEOREM 8.3 (TRANSFORMATION SOUNDNESS). $\text{wf}(p_s) \wedge p_s \rightsquigarrow p_t \implies p_s \sqsupseteq p_t$

The meat of the proof is to use Lemma 8.2 to connect Lemma 7.1 and Lemma 8.1. For this we must prove that the TMC protocol is admissible. In the specific case of TMC, the admissibility condition can be reformulated in In other words, we need a way to close an open simulation. Obviously, this closure mechanism cannot work for any protocol; we have to constrain it a bit. Actually, for the TMC protocol, we can prove a slightly simpler condition that applies admissibility:

$$\Box \left(\forall \Psi, e_s, e_t. X(\Psi, e_s, e_t) \multimap \exists e'_s, e'_t. e_s \xrightarrow[\text{pure}]{p_s} e'_s * e_t \xrightarrow[\text{pure}]{p_t} e'_t * e'_s \gtrsim e'_t \langle X \rangle [\Psi] \right)$$

9 RELATED WORK

9.1 TMC support in compilers

Tail-recursion modulo cons was well-known in the Lisp community as early as the 1970s. For example the REMREC system [Risch 1973] would automatically transform recursive functions into loops, and supports modulo-cons tail recursion. It also supports tail-recursion modulo associative arithmetic operators, which is outside the scope of our work, but supported by the GCC compiler for example. The TMC fragment is precisely described (in prose) in Friedman and Wise [1975].

In the Prolog community it is a common pattern to implement destination-passing style through unification variables; in particular “difference lists” are a common representation of lists with a final hole. Unification variables are first-class values, in particular they can be passed as function arguments. This makes it easy to write the destination-passing-style equivalent of a context of the form `List.append li □`, as the difference list `(List.append li X, X)`. In constrast, we only support direct constructor applications. However, this expressivity comes at a performance cost, and there is no static checking that the data is fully initialized at the end of computation.

Independently of our work, Koka has implemented TMC starting in August 2020² [Leijen and Lorenzen 2023]. An interesting problem they had to solve, which does not occur in OCAML, is how to support TMC in presence of non-linear contibuations. Our correctness argument for TMC relies on the fact that the destination is uniquely owned, and written exactly once; this property may not hold in programs that use multishot continuations (`call/cc`, `let/cc`, `delim/cc`). The standard Koka runtime uses its reference-counting machinery to determine that a destination is not uniquely-owned anymore, and stores extra metadata in partially-initialized blocks to be able to copy them on-demand in this case. Its Javascript backend instead reverts to a CPS transformation when non-linear control flow is detected.

9.2 Reasoning about destination-passing-style

In general, if we think of non-tail recursive functions as having an “evaluation context” left for after the recursive call, then the techniques to turn classes of calls into tail-calls correspond to different reified representations of non-tail contexts, as long as they support efficient composition and hole-plugging. TMC comes from representing data-construction contexts as the partial data itself, with hole-plugging by mutation. Associative-operator transformations represent the context $1 + (4 + \square)$ as the number 5 directly. (Sometimes it suffices to keep around an abstraction of the context; this is a key idea in John Clements’ work on stack-based security in presence of tail calls.)

Minamide [1998] gives a “functional” interface to destination-passing-style program, by presenting a partial data-constructor composition `Foo(x, Bar(□))` as a use-once, linear-typed function `linfun h → Foo(x, Bar(h))`. Those special linear functions remain implemented as partial data, but they expose a referentially-transparent interface to the programmer, restricted by a linear type discline. This is a beautiful way to represent destination-passing style, orthogonal to our work: users of Minamide’s system would still have to write the transformed version by hand, and we could implement a transformation into destination-passing style expressed in his system. Mezzo [Balabonski, Pottier and Protzenko 2016] supports a more general-purpose type system based on separation logic, which can directly express uniquely-owned partially-initialized data, and its implicit transformation into immutable, duplicable results. (See the `List` module of the Mezzo standard library, and in particular `cell`, `freeze` and `append` in destination-passing-style).

Leijen and Lorenzen [2023] provide a (pen-and-paper) correctness argument for TMC, or in fact a family of approaches based on optimized representations of classes of non-tail contexts, in the style of program calculation. The clarity of their exposition is remarkable. The implementation

²<https://github.com/koka-lang/koka/commit/f6a343d31f486ea5edd44798dca7bca52d7b450c>

they prove correct, which corresponds to the approach described in [Minamide \[1998\]](#), results in a slightly different code generation where recursive calls to `map_dps` are passed *both* the beginning of the list and the destination to be written at its end. The beginning of the list remains constant over all recursive calls, and it is known to the caller, so our DPS version does not propagate it. We were inspired by the generality of their presentation and verified that our proof technique can also be applied to some over TMC variants, by extending our mechanized development with a correctness proof for an accumulator-passing-style transformation.

9.3 Relational reasoning in separation logic

Defining a program logic to capture unary program properties is a typical usage of IRIS; relational properties are rarer. [Tassarotti, Jung and Harper \[2017\]](#) use (a linear variant of) IRIS to prove the correctness of a program transformation that implements communication channels using shared references. There is a good discussion of related work in ReLoC Reloaded [[Frumin, Krebbers and Birkedal 2021](#)].

A relational program logic can be justified in IRIS by interpreting it as a unary relation on the target program, typically involving the `wp` predicate of the base language. This approach is inspired by CaReSL [Turon, Dreyer and Birkedal \[2013\]](#). We follow a more direct, traditional approach of interpreting the program logic as a (binary) simulation relation (defined in the IRIS meta-logic) which is shown to be adequate, to imply a refinement between the program behaviors (denotations).

It is in fact surprisingly difficult to define simulations in IRIS, if we expect it to be adequate (to correspond to the usual notion of simulation outside the IRIS world). This is due to meta-theoretical difficulties around the “later” modality which led to the Transfinite IRIS variant [Spies, Gäher, Gratzner, Tassarotti, Krebbers, Dreyer and Birkedal \[2021\]](#). We started by defining simulations in Transfinite IRIS, but later moved to the SIMULIRIS approach [[Gäher, Sammler, Spies, Jung, Dang, Krebbers, Kang and Dreyer 2022](#)], where simulations are defined in standard IRIS without using the “later” modality, using coinduction instead (via its impredicative encoding).

As a minor technical point of comparison to SIMULIRIS, our definition of behaviors (denotations) includes non-termination, successfully evaluating to a value, but also failing with an error, and refinement preserves all three kind of behaviors. There are no undefined behaviors.

9.4 Protocols

The function-call rule of SIMULIRIS only relates calls to the same function, so it is unsuitable for program transformations that also transform function definitions. We parametrize our program logic and notion of simulation on a *protocol* X , an arbitrary predicate transformer injected into the relation. This approach is reminiscent of the *axiomatic semantics* proposed [Wang, Cuellar and Chlipala \[2014\]](#) to reason about foreign function calls. In the IRIS community, we were directly inspired the *protocols de Vilhena and Pottier [2021]*, and this approach was also reused recently, in a unary setting, by [Guéneau, Hostert, Spies, Sammler, Birkedal and Dreyer \[2023\]](#). Our notion of protocol is slightly more general than in those two works, as it can relate arbitrary expressions in evaluation position (not just function calls), and “return” after an axiomatic transition with arbitrary expressions (not just values). We use this extra generality to reason about accumulator-passing-style transformation in presence of ill-typed programs – see [Section 6](#).

REFERENCES

- Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2016. The Design and Formalization of Mezzo, a Permission-Based Programming Language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 38, 4 (Aug. 2016), 94. <https://doi.org/10.1145/2837022>
- Paulo Emílio de Vilhena and François Pottier. 2021. A separation logic for effect handlers. In *POPL*. <http://cambium.inria.fr/~fpottier/publis/de-vilhena-pottier-sleh.pdf>
- Daniel P. Friedman and David S. Wise. 1975. *Unwinding stylized recursions into iterations*. Technical Report 19. Computer Science Department, Indiana University, Bloomington. <https://legacy.cs.indiana.edu/ftp/techreports/TR19.pdf>
- Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Logical Methods in Computer Science* Volume 17, Issue 3 (July 2021). [https://doi.org/10.46298/lmcs-17\(3:9\)2021](https://doi.org/10.46298/lmcs-17(3:9)2021)
- 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. (2022).
- Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. In *OOPSLA*. <https://inria.hal.science/hal-04203298>
- Daan Leijen and Anton Lorenzen. 2023. Tail Recursion Modulo Context: An Equational Approach. *Proc. ACM Program. Lang.* 7, POPL (2023), 1152–1181. <https://doi.org/10.1145/3571233>
- Yasuhiko Minamide. 1998. A functional representation of data structures with a hole. In *POPL*. <https://sv.c.titech.ac.jp/minamide/papers/hole.popl98.pdf>
- Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95. <https://doi.org/10.1145/3211968>
- Tore Risch. 1973. *REMREC – A Program for Automatic Recursion Removal in Lisp*. Technical Report DLU73/24. Dept. of Computer Science, Uppsala University. <http://user.it.uu.se/~torer/publ/remrec.pdf>
- Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9236)*, Christian Urban and Xingyuan Zhang (Eds.). Springer, 359–374. https://doi.org/10.1007/978-3-319-22102-1_24
- Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In *PLDI*. <https://iris-project.org/transfinite-iris/>
- Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *ESOP*. <https://arxiv.org/abs/1701.05888>
- Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. <https://people.mpi-sws.org/~dreyer/papers/caresl/paper.pdf>
- Peng Wang, Santiago Cuellar, and Adam Chlipala. 2014. Compiler verification meets cross-language linking via data abstraction. In *OOPSLA*. <http://adam.chlipala.net/papers/CitoOOPSLA14/>

A EXTENDED REPUBLICATION

A preliminary, work-in-progress version of this work was published at a national conference – all talks given in a non-English language. This makes the present submission to a PACMPL-published conference a partial republication.

This is unusual, so we decided to write this appendix section to explain this choice. Our explanation is carefully worded to maintain anonymity. (Note: there is overlap between the authors of both presentations, but they are not equal sets, so guessing which previous version we are talking about would be only partially de-anonymizing.)

A.1 The letter of the law.

We believe that partial republication fully satisfies the [SIGPLAN Republication Policy](#), which we quote below in full (with our emphasis):

Prior Publication of Closely Related Work. If a closely related paper was previously accepted at a conference or a workshop with proceedings of any kind, the original paper must be cited, its relationship to the current paper explained, and the program chair must be explicitly informed. The second paper will be judged on the additional value of its publication in the new venue. Such value might come from the **wider audience for the new venue** or from subjecting the work to a **higher standard of review**. Program committees should consider factors such as the following in deciding whether to publish the second paper:

- The call for papers for the first venue clearly states that publication in the venue is not intended to preclude later publication.
- The original proceedings are not easily accessible to the SIGPLAN community.
- The first venue **targeted a geographically limited audience**.

The present submission will be subject to higher standards (the acceptance rate of the national venue is in the 80-100% range). If accepted, it would reach a wider audience. The past submission targeted a geographically limited audience. (If we remember correctly, everyone attending the conference that year was working in the same country it was located in.)

A.2 The spirit of the law.

We believe that a partial republication mirrors the established practice of publishing a first version of a paper in conference proceedings, and then an extended version in a journal. Here the situation is different but similar, we published a work-in-progress presentation of our work in a minor/national conference, and we are submitting a full presentation of the same work (and additional contributions) to a major/international conference-journal hybrid.

Common/accepted conference-to-journal republication typically expect “enhanced versions” with, according to the Journal of Functional Programming guidelines for example, “should contain additional discussion, examples, or proofs”. We believe that our new presentation of our work is well above this bar, as it includes major new scientific contributions.

A.3 Comparing the two versions.

The work-in-progress paper was written in October 2020, after we implemented TMC for OCaml, but before it was integrated into the OCaml compiler. It focuses on the implementation in OCaml, and does not contain any correctness argument for the program transformation – a major missing piece for a high-quality publication in an international conference.

The majority of the presentation in the present paper is new. The following parts are reused from the previous paper – integrated by the same author who originally wrote them:

- Two thirds of the introduction (the first two pages).
- [Section 3.1](#) on examples of TRMC functions in OCAML (1.5 pages).

We thus estimate the reused content at about 3.5 pages, the rest of the content was written from scratch for the present submission.

The major scientific difference between the two papers is the correctness proof in Iris, which is a significant contribution of its own. To our knowledge, it is the first mechanized proof of correctness for the tail-modulo-constructor program transformation, and the first verification of a program transformation (for all inputs) on top of SIMULIRIS. This constitutes the majority of the content of the current presentation.

More minor differences include an analysis of the adoption of TMC in OCAML libraries post-upstreaming, and a comparison with the Koka work [[Leijen and Lorenzen 2023](#)] which was developed simultaneously with ours.

A.4 Motivating our choice.

An alternative that we of course considered would be to present only the correctness proof for TMC in the present paper, without reusing any contributions and presentation from the work-in-progress, implementation-focused paper.

This approach would however result in a “small” publication about the (non-trivial) implementation work in the OCAML compiler, and a more prestigious international submission for the correctness proof alone. It would deprive from symbolic/academic reward the contributors who lead the implementation and upstreaming work, without which none of the present work would exist in the first place. (This approach would also weaken national conferences, by making researchers think twice about submitting implementation-focused work in progress.)

This is in fact a common dynamic in programming-language research that combines delicate implementation work with delicate meta-theory work. The latter is more easily identified as challenging research, easier to evaluate for quality and novelty, and tends to gather more scientific credit. The implementation work also typically comes before, and it is natural to do the verification work separately, with a different (sometimes entirely disjoint) group of contributors. We are trying to avoid this dynamic by grouping the two sorts of contributions together in the same publication.

B MORE ON TMC IN OCAML

B.1 Alternatives

We mention other approaches than TMC to solve the problem of overflowing the call stack, and explain why they were less suitable for OCAML. We also discuss a significant implementation change between OCAML 4 and OCAML 5.

B.1.1 Doing a CPS transformation. Instead of a program transformation in *destination-passing* style, we could perform a more general program transformation that can make more functions tail-recursive, for example a generic *continuation-passing* style (CPS) transformation.

We have three arguments for implementing the TMC transformation:

- The TMC transformation generates more efficient code, using mutation instead of function calls. On the OCAML runtime, the difference is a large constant factor.³
- The CPS transformation can be expressed at the source level, and can be made reasonably nice-looking using some monadic-binding syntactic sugar. TMC can only be done by the compiler, or using safety-breaking features.

B.1.2 Lazy data. Lazy (call-by-need) languages will also often avoid running into stack overflows: as soon as a lazy datastructure is returned, which is the default, functions such as `map` will return immediately, with recursive calls frozen in a lazy thunk, waiting to be evaluated on-demand as the user traverses the result structure. User still need to worry about tail-recursivity for their strict functions; strict functions are often preferred when writing performant code.

B.1.3 Unlimiting the system stack. Some operating systems can provide an unlimited system stack; such as `ulimit -s unlimited` on Linux systems – the system stack is then resized on-demand. Frustratingly, unlimited stacks are not available on all systems, and not the default on any system in wide use. Convincing all users to setup their system in a non-standard way would be *much* harder than performing a program transformation or accepting the CPS overhead for some programs.

B.1.4 Using another stack. Using the native system stack is a choice of the OCAML 4 implementation. Some other implementations of functional languages, such as SML/NJ, use a different stack (the OCAML bytecode interpreter does this), or directly allocate stack frames on their GC-managed heap. This approach can make “stack overflow” go away completely, and it also makes it very simple to implement stack-capture control operators, such as continuations, or other stack operations such as continuation marks.

On the other hand, using the native stack brings compatibility benefits (coherent stack traces for mixed OCAML +C programs), and seems to improve the performance of function calls (on benchmarks that are only testing function calls and return, such as Ackermann or the naive Fibonacci, OCAML can be 4x, 5x faster than SML/NJ.)

OCAML 5. OCAML 5.0.0 was released in December 2022, about a year after we landed our TMC implementation in the OCAML 4 compiler. OCAML 5 uses a different runtime to support multicore programming, and a different calling convention to support algebraic effects. In particular, it only uses the system stack for C calls, and a “cactus stack” for OCAML calls.

OCAML 5 manages its own stack, but the implementors still decided to have a stack limit. It is sensibly higher by default (1Gio at the time of writing) than the system stack (8Mio), so many “small” can now use non-tail-recursive functions without fears of stack overflows, but overflows remain possible and likely in practice on large inputs. The reason to keep a (user-settable) limit

³On a toy benchmark with large-sized lists, the CPS version is 100% slower and has 130% more allocations than the non-tail-recursive version.

for the OCAML call stack is usability in the face of buggy programs. When programmers write recursive functions, they occasionally go through incorrect versions with a faulty base case that fail to terminate. In this case, we want the system to fail quickly with an error, instead of remaining silent for a few minutes, crashing once all the machine memory has been consumed.

We were curious about whether users would still see a need for TMC in OCAML 5, or whether they would stop using the feature in practice. The feedback we got from expert users is that stack overflows is still an issue they worry about. We observe that TMC adoption in OCAML codebases keep growing after the transition to OCAML 5.

B.2 Implementation history

We first proposed adding TMC as an optional program transformation to the OCAML compiler in May 2015.⁴ The proposal was received favorably, but it never received an in-depth review and a detailed performance evaluation and remained unmerged for years.

We restarted the implementation in 2020, resulting in a new pull July 2020 request with a modified implementation, and a careful performance evaluation⁵. The new implementation put a larger focus on producing readable code, giving more control to the user through annotations. It also removed an optimization of the previous implementation that would specialize the DPS version, generating a distinct definition for each block offset. The new design was carefully reviewed by Basile Clément and Pierre Chambart, and was finally merged in November 2021, available to users with OCAML 4.14, released in March 2022.

The TMC transformation is that it adds extra parameters to functions (two parameters, the block and the offset). At the time the OCAML compiler had a limitation on some supported architectures, where it would not optimize tail calls above a certain number of arguments (enough that they cannot be all passed by registers), breaking the tail-call promises of TMC on those systems. Xavier Leroy implemented a change to the OCAML calling convention for those architectures in May 2021,⁶ which was motivated by the TMC work.

B.3 Implementation: an applicative functor

The computation of choices described in Section 3.5 can be expressed elegantly. Instead of a monomorphic type `Choice.t` that represents a choice of how to transform a term, we use a polymorphic type `'a Choice.t` that represents how to transform zero, one or several subterms that occur in the source; for example transforming two subterms in the same context relies on a choice of type `(lambda * lambda) Choice.t`, where `lambda` is the type of terms in the intermediate representation we are working on. This `'a Choice.t` type can be equipped with an applicative functor interface:

⁴URL removed for anonymity.

⁵URL removed for anonymity.

⁶<https://github.com/ocaml/ocaml/pull/10595>

```

1471 module Choice : sig
1472   type 'a t = {
1473     dps : 'a Dps.t;
1474     direct : unit → 'a;
1475     tmc_calls : tmc_call_info list;
1476     benefits_from_dps : bool;
1477     explicit_tailcall_request : bool;
1478   }
1479
1480   (* construct a choice from an arbitrary term *)
1481   val lambda : lambda → lambda t
1482
1483   (* applicative functor interface *)
1484   val unit : unit t
1485   val map : ('a → 'b) → 'a t → 'b t
1486   val pair : 'a t * 'b t → ('a * 'b) t
1487
1488   (* extract the direct-style and DPS transforms *)
1489   val direct : lambda t → lambda
1490   val dps :
1491     lambda t → tail:bool → dst:offset dst →
1492     lambda
1493 end

```

With this interface, the easy cases of the transformation can be expressed compactly:

```

1488 let rec choice ctx ~tail t =
1489   match t with
1490   [...]
1491   | Lifthenelse (l1, l2, l3) →
1492     let l1 = traverse ctx l1 in
1493     let+ l2 = choice ctx ~tail l2
1494     and+ l3 = choice ctx ~tail l3
1495     in Lifthenelse (l1, l2, l3)
1496   [...]

```

The binding operators `let+`, `and+` are standard syntax for applicative-like structures in OCAML: `let+` desugars to `Choice.map` and `and+` desugars to `Choice.pair`. `traverse` performs a direct-style translation, and `traverse_letrec` may also transform `let`-bound functions marked with `[@tail_mod_constr]` and add them to the local transformation environment.

B.4 Evaluation: adoption

The TMC transformation was first made available to users in OCAML 4.14.0, released in March 2022. At the time there were no users of the feature. Notably, the OCAML standard library had intentionally not been modified to start using it: before the release, we did not want the `stdlib` codebase to depend on a not-available-yet feature, and after the release we decided not to push for adoption (we may be biased about the benefits/importance of TMC), and let other contributors propose its use, after doing their own evaluation of performance impact and code-clarity benefits.

Over time, `[@tail_mod_cons]` was adopted in a few places in the OCAML standard library, either to make some functions tail-recursive that previously were not, or simplify some complex tail-recursive implementations. As of spring 2024, the feature is now used in `Hashtbl.find_all` and in `List` module (`init`, `map`, `mapi`, `map2`, `find_all`, `filteri`, `filter_map`, `take`, `take_while`, `of_seq`). `init` and the `map*` functions, which are the most heavily used, were unrolled once – the minimal amount observed to preserve the non-tailrec performance for small lists. Other functions are written in the simplest possible way. (All `stdlib` uses so far build lists rather than another datatype.)

We performed a systematic search for `[@tail_mod_cons]` usage on November 2023, and we found that it was also used

- in a few utility modules (general-purpose functions designed to extend or replace the standard library), notably in Jane Street’s base library.
- in the middle of domain-specific user code, to build lists, in 14 different projects
- in the middle of user code, to build other types than lists, in three projects:
 - in RedPRL⁷, it is used to build snoc lists
 - in a project named PL-reading-group⁸, it is used to build a GADT of difference lists
 - in acutis⁹, in a group of mutually-recursive functions that builds a complex AST structure used options and nested records

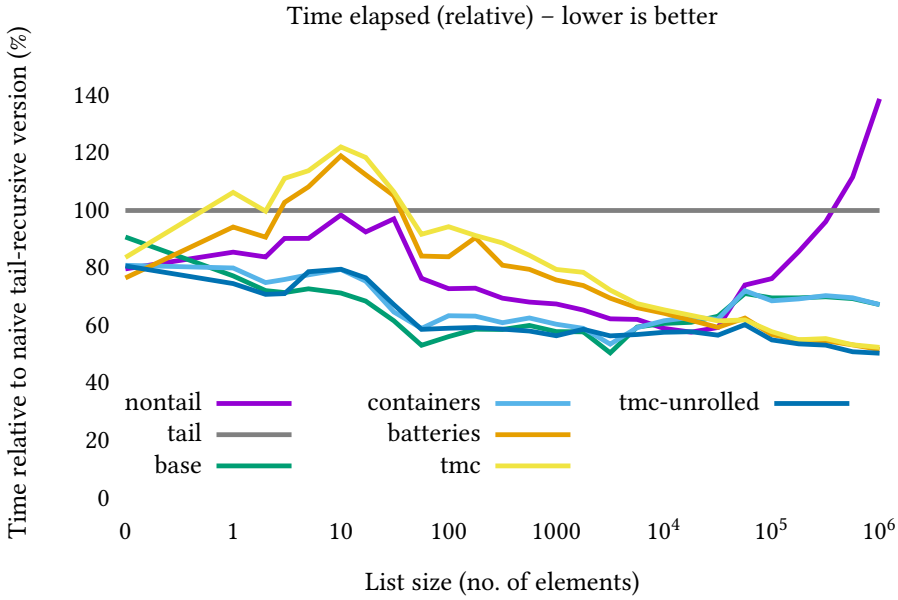
We also found three cases where the annotation is (in our opinion) misused: in two cases, the function is already tail-recursive, so there is no need for the annotation, and in a third case¹⁰ we consider that the use is gratuitous and can be replaced by already-available standard library functions.

⁷<https://github.com/RedPRL/ocaml-bwd/blob/fbf496b29532085b38073eaa62ba3d22ac619d5d/src/BwdNoLabels.ml#L60>

⁸<https://github.com/Skyb0rg007/PL-Reading-Group/blob/7002ae35ba69fb9010e5049eee88ab475bc79ec3/effects/lib/free/tseq.ml#L44>

⁹<https://github.com/johnridesabike/acutis/blob/4113fb516d9c5dd6f2dbfd9657f52c5ae7a5dcae/lib/matching.ml#L161>

¹⁰https://github.com/gridbugs/llama/blob/96d1c96c31ff136f465b5f37c981fff591bac6fd/src/midi/byte_array_parser.ml#L50

Fig. 16. `List.map` benchmark on OCAML 4.14

C BENCHMARK RESULTS ON OCAML 4

The benchmark results for OCAML 4 are given in Figure 16. The results are very close to Figure 9, and in particular the qualitative analysis of the results is mostly unchanged. (We used `ulimit -s unlimited` to disable the system stack limit, to run the non-tail-recursive version on large lists.)