Internship: A formal proof of correctness of Tail-Recursion Modulo Constructor in the Iris logic

Léo Stefanesco, François Pottier, Gabriel Scherer
INRIA Paris and Saclay

Current version: February 15, 2021

Abstract

Tail-recursion is a well known optimization in functional languages which reuses the current stack frame when the last instruction of a function is a recursive call. The `map` function on lists is a canonical example of a function whose natural and elegant implementation is *not* tail-recursive, as the recursive call is an argument to a constructor. *Tail-recursion modulo constructor* (TRMC) was introduced in the 70's (Friedman and Wise, 1974; Warren, 1980) to optimize functions in which the final operations are constructor applications and a recursive call.

For instance, the `map` function is transformed into a *destination-passing-style* function which constructs a partial result list, gradually adding new elements by mutation of the tail of the partial list, until it reaches the end of the input list.

TRMC has been implemented in Lisp and Prolog compilers, but never in a production ML complier. There is a prototype for the OCaml compiler developed by Frédéric Bour and Gabriel Scherer.

We would like to prove formally the correctness of this transformation. Reasoning on the transformation is delicate as the destination-passing-style function uses mutable state internally, following a unique-ownership discipline.

We plan to use the Iris logic, which is embedded in Coq, to prove a contextual refinement between the source function and its optimized version. Iris is a higher-order separation logic, a natural choice to reason about uniquely-owned mutable state.

1 Practicalities

1.1 Prerequisites

The main prerequisite for this internship is a good familiarity with Coq, or possibly another proof assistant such as Isabelle, Agda or Lean, as this internship entails a medium-sized development in Coq.

1.2 Work

The first step would be to implement a simplified version of the ocaml optimization pass in Coq, on a simple imperative functional language. The core of internship would be to investigate how to encode the mutation and later the freezing of the data-structure constructed by the function which is being optimized and to prove that the source program and the target program are related by a suitable logical relation, see below. The proof would be done in the Iris Proof Mode, an implementation of an interactive proof mode for a separation logic embedded in Coq.

1.3 Environment

Given the current sanitary conditions, the internships would be mostly remote. The intern will still get the opportunity to meet the advisors in-person, at least once at week, probably at the INRIA center in Paris.

2 Scientific background

2.1 Tail-recursion modulo constructor

The natural and non tail-recursive implementation of `map`

```
let rec map f = function
| [] -> []
| x :: xs -> f x :: map f x s
```

is transformed into the following pair of functions

```
let rec map' f = function
| [] -> []
| x :: xs ->
  let dst = f x :: Hole in
  map'_dps dst f xs;
  result

and map'_dps dst f = function
| [] ->
  dst.i <- []
| x :: xs ->
  let next_dst = f x :: Hole in
  dst.1 <- next;
  map_dps next_dst f xs (* tail-call *)
```

In the code above, `Hole` denotes an undefined value which will be overwritten before the function terminates. The `cons` cells of the list we are building is seen as a pair whose second component is mutable (`dst.1 <- ...`). This transformation can be done today by hand using the unsafe `Obj` module of Ocaml to cast lists to mutable lists with the same memory representation. This is used in the `Batteries` library for instance. See (Scherer, 2020) for more details on the implementation of TRMC in Ocaml.

The goal of this internship is to prove that `map'` is a *contextual refinement* of `map`. This means that any observable behavior of `map'` was already an observable behavior of `map`. Formally, this is stated as:

for all contexts `K`, if `K[map']` terminates, so does `K[map]`.

Another desirable property is *termination preservation*: `map'` terminates on more inputs than `map`. Otherwise any non-terminating implementation of `map'` would be considered valid.

2.2 The Iris logic

Iris (https://iris-project.org/) is a higher-order separation logic implemented in the Coq proof assistant. In other words it is a Hoare logic with a *separating conjunction* $A * B$ which expresses that the formulas A and B denote disjoint resources (typically the memory), this means that resources described by A can be relied upon and modified without worrying about invalidating invariants in other parts of the program.

This Hoare logic is built using the *base logic* of Iris, which defines a higher order logic with a separating conjunction and a guarded recursion operator with which one can define Hoare triples.

Using this machinery, we can define *binary logical relations* such that related programs contextually refine each others (Krebbers et al., 2017). We will use such a logical relation to prove the correctness of the TRMC optimization by showing that the source program and its transformation.

References

Daniel Friedman and David Wise. Technical report TR19: Unwinding structured recursions into iterations. Technical report, Indiana University, 1974.

D.H.D. Warren. DAI research report. Technical report, University of Edinburgh, 1980.

Gabriel Scherer. Pull request: TRMC, reloaded, 2020. URL https://github.com/ocaml/ocaml/pull/9760.

Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *POPL*, 2017.