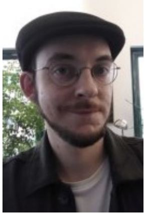
Tail Modulo Cons,
OCaml,
and Relational Separation Logic





Clément Allain

Frédéric Bour

Basile Clément

François Pottier

Gabriel Scherer

map: natural implementation

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

```
# List.init 250_000 (fun _ → ())
  |> map Fun.id
  |> ignore
  ;;
Stack overflow during evaluation (looping recursion?).
```

map: accumulator-passing style

```
let rec map_aps acc f xs =
  match xs with
  | [] →
      List.rev acc
  | x :: xs →
      let y = f x in
      map_aps (y :: acc) f xs
```

```
let map xs =
  map_aps [] f xs
```

```
# List.init 250_000 (fun _ → ())
  |> map Fun.id
  |> ignore
  ;;
- : unit = ()
```

map: destination-passing style

```
let rec map_dps dst f xs =
  match xs with
  | [] →
      set_field dst 1 []
  | x :: xs →
      let y = f x in
      let dst' = y :: _ in
      set_field dst 1 dst' ;
      map_dps dst' f xs
```

```
let map f xs =
  match xs with
  | [] →
      []
  | x :: xs →
      let y = f x in
      let dst = y :: _ in
      map_dps dst f xs ;
      dst
```

```
# List.init 250_000 (fun _ → ())
  |> map Fun.id
  |> ignore
  ;;
- : unit = ()
```

map: Tail Modulo Constructor (TMC)

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

```
# List.init 250_000 (fun _ -> ())
  |> map Fun.id
  |> ignore
  ;;
- : unit = ()
```

TMC transformation

- **Safe:** performed by the OCaml compiler.
- **Explicit:** `[@tail_mod_cons]` annotation.

- **Generality:**
  - Works on any algebraic data type (lists, trees, *etc.*).
  - Supports mutually recursive functions.

- **Implementation details:** see the paper.
- **Performance:** see benchmarks in the paper.
- **Feature adoption:** see survey in the paper.

- **Soundness:** formally verified in Coq/Rocq in an simplified setting ...

DataLang: syntax

$$
\begin{array}{llll}
\mathrm{Index} & \ni & i & ::= 0 \mid 1 \mid 2 \\
\mathrm{Tag} & \ni & t & \\
\mathbb{B} & \ni & b & \\
\mathbb{L} & \ni & \ell & \\
\mathbb{F} & \ni & f & \\
\mathbb{X} & \ni & x, y & \\
\mathrm{Val} & \ni & v, w & ::= () \mid i \mid t \mid b \mid \ell \mid @f \\
\mathrm{Expr} & \ni & e & ::= v \mid x \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid e_1\ \overline{e_2} \\
& & & \quad \mid e_1 = e_2 \mid \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \\
& & & \quad \mid \{\, t\,,\, e_1\,,\, e_2 \,\} \\
& & & \quad \mid e_1\,.\,(e_2) \mid e_1\,.\,(e_2) \leftarrow e_3 \\
\mathrm{Def} & \ni & d & ::= \mathbf{fun}\ \overline{x} \rightarrow e \\
\mathrm{Prog} & \ni & p & := \mathbb{F} \xrightarrow{\mathrm{fin}} \mathrm{Def} \\
\mathrm{State} & \ni & \sigma & := \mathbb{L} \xrightarrow{\mathrm{fin}} \mathrm{Val} \\
\mathrm{Config} & \ni & \rho & := \mathrm{Expr} \times \mathrm{State}
\end{array}
$$

DataLang: map

```
map := fun f xs →
  match xs with
  | [] →
      []
  | x :: xs →
      let y = f x in
      y :: @map f xs
```

DATALANG: map (transformed)

```
map_dps := fun dst idx f xs →
   match xs with
   | [] →
        dst.(idx) ← []
   | x :: xs →
        let y = f x in
        let dst' = y :: ■ in
        dst.(idx) ← dst' ;
        @map_dps dst' 2 f xs
```

```
map_dir := fun f xs →
   match xs with
   | [] →
        []
   | x :: xs →
        let y = f x in
        let dst = y :: ■ in
        @map_dps dst 2 f xs ;
        dst
```

TMC transformation

$$e_s \overset{\xi}{\underset{\mathrm{dir}}{\rightsquigarrow}} e_t \qquad d_s \overset{\xi}{\underset{\mathrm{dir}}{\rightsquigarrow}} d_t$$

$$(e_{dst}, e_{idx}, e_s) \overset{\xi}{\underset{\mathrm{dps}}{\rightsquigarrow}} e_t \qquad d_s \overset{\xi}{\underset{\mathrm{dps}}{\rightsquigarrow}} d_t$$

$$p_s \rightsquigarrow p_t$$

Transformation soundness

$p_s \rightsquigarrow p_t$ program p_s transforms into program p_t

$\Downarrow$

$p_s \sqsupseteq p_t$ program p_t refines program p_s
(termination-preserving behavioral refinement)

Transformation soundness

Termination-preserving behavioral refinement

$$
\begin{aligned}
p_s \sqsupseteq p_t \quad &:= \quad \forall\, f \in \mathrm{dom}(p_s), v_s, v_t. \\
&\qquad \mathrm{wf}(v_s) \wedge v_s \sim v_t \implies \\
&\qquad @f\ \ v_s \sqsupseteq @f\ \ v_t \\[1em]
e_s \sqsupseteq e_t \quad &:= \quad \forall\, b_t \in \mathrm{behaviours}_{p_t}(e_t). \\
&\qquad \exists\, b_s \in \mathrm{behaviours}_{p_s}(e_s).\ b_s \sqsupseteq b_t \\[1em]
\mathrm{behaviours}_p(e) \quad &:= \quad \{\mathbf{Conv}(e') \mid \ldots\} \uplus \{\mathbf{Div} \mid (e, \emptyset) \Uparrow_p\}
\end{aligned}
$$

Transformation soundness

$p_s \rightsquigarrow p_t$ program p_s transforms into program p_t

$\Downarrow$

$p_s \sqsupseteq p_t$ program p_t refines program p_s
(termination-preserving behavioral refinement)

Transformation soundness

$p_s \rightsquigarrow p_t$	program p_s transforms into program p_t
$\Downarrow$	
$p_s \gtrsim p_t$	program p_t simulates program p_s (*relational separation logic*, $\textsc{Simuliris}$)
$\Downarrow$	
$p_s \sqsupseteq p_t$	program p_t refines program p_s (termination-preserving behavioral refinement)

Transformation soundness

Relational separation logic

REL-PURE
$$\frac{e_s \xrightarrow[\text{pure}]{p_s} e_s' \qquad e_t \xrightarrow[\text{pure}]{p_t} e_t' \qquad e_s' \gtrsim e_t'\ [\Phi]}{e_s \gtrsim e_t\ [\Phi]}$$

REL-SOURCE-LOAD
$$\frac{(\ell + i) \mapsto_s v_s \qquad (\ell + i) \mapsto_s v_s \twoheadrightarrow v_s \gtrsim e_t\ [\Phi]}{\ell\,.\,(i) \gtrsim e_t\ [\Phi]}$$

REL-TARGET-LOAD
$$\frac{(\ell + i) \mapsto_t v_t \qquad (\ell + i) \mapsto_t v_t \twoheadrightarrow e_s \gtrsim v_t\ [\Phi]}{e_s \gtrsim \ell\,.\,(i)\ [\Phi]}$$

Transformation soundness

$p_s \rightsquigarrow p_t$ program p_s transforms into program p_t

Abstract protocols
(calling conventions)

$$\frac{\text{REL-PROTOCOL} \\ X(e_s, e_t, \Psi) \qquad \forall\, e_s', e_t'.\ \Psi(e_s', e_t') \twoheadrightarrow e_s' \gtrsim e_t' \ \langle X \rangle\ [\Phi]}{e_s \gtrsim e_t \ \langle X \rangle\ [\Phi]}$$

$p_s \sqsupseteq p_t$ program p_t refines program p_s
(termination-preserving behavioral refinement)

Transformation soundness

$p_s \rightsquigarrow p_t$ program p_s transforms into program p_t

$\Downarrow$

$p_s \gtrsim p_t$ program p_t simulates program p_s
(*relational separation logic*, Simuliris)

$\Downarrow$

$p_s \sqsupseteq p_t$ program p_t refines program p_s
(termination-preserving behavioral refinement)

Specification in separation logic

$$\frac{\{???\}}{\texttt{@map}\ v_s \ \gtrsim\ \texttt{@map\_dir}\ v_t}\ \{???\}$$

$$\frac{\{???\}}{\texttt{@map}\ v_s \ \gtrsim\ \texttt{@map\_dps}\ \ell\ i\ v_t}\ \{???\}$$

Direct transformation

$$\frac{\frac{\{v_s \approx v_t\}}{\texttt{@map}\ v_s \ \gtrsim\ \texttt{@map\_dir}\ v_t}}{\{w_s, w_t.\ w_s \approx w_t\}}$$

$$\textsc{rel-dir}\ (\textsc{Simuliris})\quad \frac{f \in \mathrm{dom}(p_s) \qquad v_s \approx v_t \qquad \forall\, w_s, w_t.\ w_s \approx w_t \mathrel{-\!\!*} \Phi(w_s, w_t)}{\texttt{@}f\ \ v_s \gtrsim \texttt{@}f\ \ v_t\ [\Phi]}$$

DPS transformation

$$\dfrac{\{v_s \approx v_t * (\ell + i) \mapsto_t \blacksquare\}}{\dfrac{\texttt{@map}\ v_s \gtrsim \texttt{@map\_dps}\ \ell\ i\ v_t}{\{w_s, ().\, \exists\, w_t.\, w_s \approx w_t * (\ell + i) \mapsto_t w_t\}}}$$

REL-DPS

$$\dfrac{\begin{array}{c} \xi[f] = f_{dps} \\ \overline{v_s} \approx \overline{v_t} \\ \ell \mapsto_t \overline{v} \\ \forall\, w_s, w_t.\, w_s \approx w_t \mathrel{-\!\!*} \ell \mapsto_t \overline{v}[i \mapsto w_t] \mathrel{-\!\!*} \Phi(w_s, ()) \end{array}}{\texttt{@}f\ \overline{v_s} \gtrsim \texttt{@}f_{dps}\ \ell\ i\ \overline{v_t}\ [\Phi]}$$

REL-PROTOCOL

$$\dfrac{\mathrm{X}(e_s, e_t, \Psi) \qquad \forall\, e_s', e_t'.\, \Psi(e_s', e_t') \mathrel{-\!\!*} e_s' \gtrsim e_t' \langle \mathrm{X} \rangle\ [\Phi]}{e_s \gtrsim e_t \langle \mathrm{X} \rangle\ [\Phi]}$$

Conclusion

- Implementation of the TMC transformation in the OCaml compiler.
- Mechanized soundness proof using *relational separation logic*.
- *Abstract protocols* to support different calling conventions: APS, inlining.

Thank you for your attention!