Tail-Recursion Modulo Constructor

Frédéric Bour

Oct 27, 2014

Plan

1. TRMC ?
   - List.map
   - Manual rewriting
   - Implementation details
2. Tweaking the Garbage-Collector

List.map

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

- simple, elegant, useful, . . .
- but not tail-recursive: list constructed when returning.

List.map

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

- simple, elegant, useful, . . .
- but not tail-recursive: list constructed when returning.

Data accumulated on the stack is proportional to the length of the list.

Quest for a tail-recursive List.map

Such implementations exist, for instance in Core.
The resulting code is much more complex and unnatural.

Quest for a tail-recursive List.map

Such implementations exist, for instance in Core.
The resulting code is much more complex and unnatural.

Could the compiler do this job?

Tail-recursion modulo constructor.

- 0001538: feature wish: tail recursion modulo cons
- turn any constructor application with a recursive-call into a tail-recursive call

Quest for a tail-recursive List.map

Such implementations exist, for instance in Core.
The resulting code is much more complex and unnatural.

Could the compiler do this job?

Tail-recursion modulo constructor.

- 0001538: feature wish: tail recursion modulo cons
- turn any constructor application with a recursive-call into a tail-recursive call

A candidate:

```
f x :: map f xs
```

Intuition

- Construct the resulting list before the recursive call
- Put a placeholder for the not-yet-computed tail of the list
- Rewrite the function to take this result value as an argument
- Change all return points to mutate this block instead of returning the value

Rewriting manually

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



;;
```

Rewriting manually

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

```
and map_1 result f = function
| [] -> result.1 <- []
| x :: xs -> let next = f x :: _ in
             result.1 <- next;
             map_1 next f xs (* tail call! *)
;;
```

Rewriting manually

```
let rec map f = function
| [] -> []
| x :: xs -> let result = f x :: _ in
             map_1 result f xs;
             result

and map_1 result f = function
| [] -> result.1 <- []
| x :: xs -> let next = f x :: _ in
             result.1 <- next;
             map_1 next f xs (* tail call! *)
;;
```

Implementation details

- Done at lambda-level rather than surface language.
- Code is duplicated for each position that can be mutated.
- Rewrite only what appears syntactically as a TRMC-call (not following let-bindings, etc).
- Impact on the semantic:
    - actual evaluation order of arguments is affected (TRMC evaluated last), this is allowed in OCaml
    - programs that would fail with a stack overflow might now succeed... or consume all available memory.

What breaks

- Potential multicore GCs. Fields non-mutable at the typed-level are now the results of some mutations
- DelimCC, Hansei. . .

Sketch of the rewriting pass

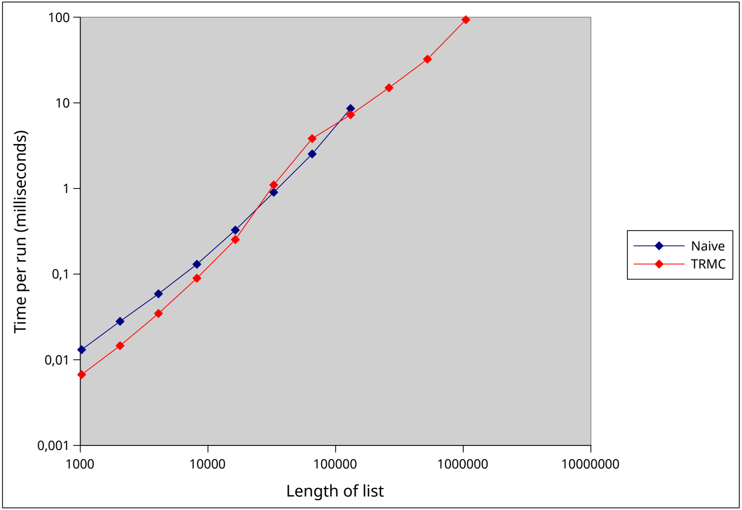
1. Traverse lambda-code,
2. For each recursive let-bindings $\{f, g, \ldots\}$,
3. For each function f,
4. For each leaf that is a TRMC-call:
   - identify field index i
   - generate a function f_i
   - replace call by a placeholder
   - bind the value constructed to a variable
   - call f_i with the original arguments & the fresh variable
   - return the variable

Generating derived functions

For generating f_i:

1. Start from f definition
2. Add an additional parameter r
3. For each leaf in tail-position:
    - if it is a tail-rec call to g, rewrite to g_i
    - if it is a trmc call to g,
        - identify field index i'
        - replace call by a placeholder
        - bind the value constructed to a variable
        - store value in ith field of r
        - call $g_{i'}$, the call is in tail-position
    - otherwise, directly store the result of the computation in ith field of r

Performance



Plan

Reminder about OCaml-GC

- Heap is split in a fixed size minor heap and a growing major heap.
- Allocations are done in minor-heap.

Reminder about OCaml-GC

- Heap is split in a fixed size minor heap and a growing major heap.
- Allocations are done in minor-heap.
- When minor heap is full, it is collected independently.
  This is correct if there is no reference from major heap to minor heap
- This can't always be guaranteed, so OCaml keeps track of all mutations which might break this invariant: that's the write barrier, candidates are stored in the *remembered set*.

Reminder about OCaml-GC

- Heap is split in a fixed size minor heap and a growing major heap.
- Allocations are done in minor-heap.
- When minor heap is full, it is collected independently.
  This is correct if there is no reference from major heap to minor heap
- This can't always be guaranteed, so OCaml keeps track of all mutations which might break this invariant: that's the write barrier, candidates are stored in the *remembered set*.
- Collection is done starting from roots $\cup$ this set.

Observations about TRMC-mutations

- Mutation will happen (ignoring exceptions)
- Mutation will happen only once (ignoring delim-cc)

Observations about TRMC-mutations

- Mutation will happen (ignoring exceptions)
- Mutation will happen only once (ignoring delim-cc)
- A reference from major-heap to minor-heap will be created only if a minor-collection was done since the beginning of TRMC recursion.

Special handling of TRMC-mutations

Idea:

- use a distinguished value t for the placeholders
- never go through the write barrier during TRMC-mutation
- during collection, if a field being moved to major-heap is equal to t, place the field in the remembered set.

Performance improvements