Mocaml - a multi level meta programming extension to OCaml

Kristoffer Kortbaek

2024/06/21

Abstract

This work introduces *mocaml*, a small syntax extension to OCaml adding meta-programming to a small subset of the language (arithmetic expressions, branches, and function application) through multi-level programming features for handwriting program generators.

Similar work to this has already been done, and [5] implements a two-level functional language with templates `<>` and escapes `~` and [6] implements a multi-stage ML inspired language with arbitrary stages. MetaOCaml is an OCaml dialect that also implements multi-stage programming to the langauge. Instead of going with two-level annotations, this work is inspired by [1] and adds multi level operations. This approach still allows the user to stage the computation into multiple stages. The implementation of mocaml uses quasi quotations to annotate OCaml programs with multi-level operations, and a syntax rewriter then rewrites the OCaml parsetree.

Keywords: multi-stage programming, multi-level programming, program generators.

Contents

1 Introduction

There have been multiple implementations of meta programming in the ML family of languages. Notable implementations are [6] and [3] that both add two-level annotations with escapes, `~`, and templates, `<>` to Standard ML and OCaml respectively. Brackets or templates, `<>`, enclose code to be generated and escapes `~` mark a hole in the template to insert code [3]. One of the notable features of MetaML and Meta-OCaml is that the languages enables the programmer write type checked program generators [3].

[1] on the other hand introduces a novel multi-level specialization to a lisp like language. Glück and Jørgensen introduce the notion of multi-level generating extensions, where program generation of an `n` input program can be staged into `n` stages, with the result of the `n-1` stages producing a multi-level generating extension. They also provide a novel binding time analysis (BTA) that given the binding time of the program input finds an optimal multi-level annotation of the *MetaScheme* language.

The goal of this work is to enable the user to hand-write program generators. This is done by combining multi-level annotations similar to those in [1] to a subset of *OCaml* while preserving the type safe guarantees of OCaml similar to [3] and [6]. In other words, the introduction of multi-level generating extensions to the language should not make well typed programs go wrong if the equivalently non-annotated source program would not go wrong. This will further be discussed in a later section. The extended subset of OCaml is called *mocaml*. The target language after specialization is still OCaml and the program generator is also written in OCaml.

The focus of this study is not in binding time analysis, but instead handwriting program generators in a multi-level language. Therefore, the multi-level constructs of mocaml must be manually annotated with binding time information. Additionally, the arguments of a mocaml program is assumed to have binding time $1, 2, ..., n$ in the order they are listed for an n input program.

2 Generating Extensions and Program Generators

In [1] the process of multi-level generating extensions is demonstrated. This will quickly be summarized as to later how handwriting program generators in mocaml differs from PE. For a program `p` written in language `L` then let $[\![\texttt{p}]\!]_{\texttt{L}}$ `in` denote the application of `L` program `p` to its input `in`. The process of multi-level generating extensions for an `n`-inpit source program `p` is:

$$\left.\begin{aligned}
\texttt{p}-\texttt{mgen}_0 &= [\![\texttt{mcogen}]\!]_{\texttt{L}}\texttt{p}\ '0...n-1' \\
\texttt{p}-\texttt{mgen}_1 &= [\![\texttt{p}-\texttt{mgen}_0]\!]_{\texttt{L}}\texttt{in}_0 \\
&\vdots \\
\texttt{p}-\texttt{res}_1 &= [\![\texttt{p}-\texttt{mgen}_{n-2}]\!]_{\texttt{L}}\texttt{in}_{n-2} \\
\texttt{p}-\texttt{res}_1 &= [\![\texttt{p}-\texttt{mgen}_{n-2}]\!]_{\texttt{L}}\texttt{in}_{n-2} \\
\texttt{out}_1 &= [\![\texttt{p}-\texttt{res}_{n-1}]\!]_{\texttt{L}}\texttt{in}_{n-1}
\end{aligned}\right\} n \text{ stages}$$

The multilevel program generator `mcogen`[1] is first specialized w.r.t. the `L` program `p` and a multi-level annotation of the inputs with binding time (bt) classification $t_0, .., t_{n-1}$ to produce a multi-level *generating extension* p-mgen$_0$. The generating extension is then specialized in n stages with an input and each staging produces a new generating extension until a final output is produced.

Program generation and specialization in mocaml is slightly different from the traditional generating extensions in partial evaluation. mocaml is used to hand-write OCaml program generators and the user must manually annotate the multi-level program with binding time information to stage the program. This is in contrast to BTA which, as it was noted in [6], can be seen as an automatic staging technique. Manual annotation might in some situation be preferred since the evaluation order can be controlled [6]. Next, the user must invoke a special `run` operation that takes a multilevel n input mocaml program `p` and an input $\texttt{in}_i$. As a result, a new annotated multilevel program is produced that might be specialized again with the remaining inputs. Without going into the details of mocaml, writing program generators looks as follows:

```
let res1 = run p in_0
let res2 = run res1 in_1
...
let out = run res_{n-1} in_{n-1}
```

This looks similar to the multi-level specialization with multi-level generating extensions. Again, to summarize the difference, the annotations are done manually and the output of each stage is a residual multi-level program that can be specialized further. Hand writing different program generators then depends on how the `run` operation is used and with different inputs. Multi-level PE would on the other hand produce a multi-level generating extension at each stage. Additionally, in mocaml the output of any of the n is a multi-level residual program which also be run since it is just a generated program.

[1]The program generator is also for historical reasons called a compiler generator, hence the `cogen` name. This is because `p` would be an interpreter.

3 Generating Programs

This section is dedicated so gently introduce the multi-level annotations of mocaml and later demonstrate how the mocaml specializer generates OCaml programs from an annotated program.

3.1 Multi Level OCaml Syntax Extensions

To facilitate multi-level features in OCaml, source programs must now use special multi-level operations. These are implemented as a special syntax extensions to the language. In OCaml syntax extensions are written with the syntax `[%%...]` and `[%...]`. Everything within the braces can then be easily interpreted by a syntax rewriter. Below example demonstrates what these syntax extensions look like:

```
(* Normal single-stage add program *)
let add a b = a + b
(* multi-level add program *)
[%%ml let add a b = [%add 2
          [%lift 1 1 a] (*a has bt 1 and is liftet to bt 1+1*)
          [%lift 2 b]]]
```

As the above example demonstrates, multi-level programs are first decorated with the special `[%%ml ...]` syntax extension. This tells the syntax rewriter this should get treated as a multi-level program. Within the `add` function, a syntax extension `[%add t e1 e2]` is used. This means that add is a special multi-level code-generating addition function. Similarly, `lift` is also a syntax extension that, and the meaning of both operators is explained in the subsequent sections.

3.2 Multi Level OCaml

The abstract syntax of mocaml is similar to *MetaScheme* presented in [1]. Each multi-level construct of mocaml corresponds to an equivalent OCaml expression, but with an associated binding time $t \geq 0$ as an additional argument. In the above add program, the multi-level add operator has binding time `t=2`. The abstract syntax of `mocaml` is:

$$\boxed{\begin{array}{l} p \in Program; x \in Variable; d \in Definition; c \in Constant; \\ e \in Constant; s, t \in BindingTime \end{array}}$$

$p := d_1, ..., d_m$

$d :=$ `[%%let f x1 ...  x2 = e]`

  | `let p_mgen = [%run f e]`

$e := c \mid x \mid$ `if b then e1 else e2`

  | `[%add t e1 e2]` | `[%sub t e1 e2]` | `[%mul t e1 e2]`

  | `[%div t e1 e2]` | `[%app t f e]` | `[%lift s t e]` | `[%lift t e]`

$b :=$ `true` | `false` | `e1 < e2`

A multi-level mocaml program is an annotated of OCaml program using the above syntax extensions. Except for the `if-then-else`, an annotation denotes the operator to use (application, lift or any of the arithmetic operators) and the binding time for that operator. Conceptually, this corresponds to the `(_ 'op (es))` from [1].

The operator `[%lift s t e]` is equivalent to the definition in [1] and it coerces an expression of binding time `t` to an expression of binding time `s+t`. Similarly, `[%lift t e]` is a shorthand for `s=0`. Currently, there is also a limitation that `app` can only be used to make recursive calls and not call arbitrary functions.

3.3 Program representation

The `[%run f e]` ocaml syntax extension gets the ParseTree[2] for `f`, and lifts it to a more manageable DSL that can handle specialization and code generation. This was inspired by the approach to two-level annotations by [6] and [3]. Below are the type definitions for the mocaml DSL:

[2]This is a tree sturcuture generated by the OCaml compiler after parsing and before type checking.

```
type ml_val = Val of int | Ident of string
and ml_expr = { (* Wrapper to store binding time info *)
  v: ml_val;
  t: int;
}

type ml_cond = Leq of ml_ops * ml_ops
             | Bool of bool

and ml_binop = Add of ml_ops * ml_ops
             | Sub of ml_ops * ml_ops
             | Mul of ml_ops * ml_ops
             | Div of ml_ops * ml_ops
and ml_ops = Binop of int * ml_binop
           | Expr of ml_expr
           | Lift of int * ml_ops
           | Fun of string * ml_ops
           | IfElse of ml_cond * ml_ops * ml_ops
           | App of int * string * (ml_ops list)
```

Listing 1: Type signature for mocaml operations and expressions. These correspond to the abstract syntax of mocaml.

All multi-level annotation gets converted to multi-level operation. This is called an `ml_ops` in the DSL. The leaf nodes of a multi-level operation is the `ml_expr` record. This stores the binding time `t` of the variable together with its value (either a constant if its value is known at the current bt, or otherwise a variable identifier). These types correspond to the definitions of the abstract syntax of mocaml. Since the `ml_expr` stores the binding time `t`, there is still a need to store the bt `s`. For this reason, an expression like `[%lift 1 2 3]` can be compactly represented as a `Lift (1, {t=2, v=(Val 3))}`. This way we can conveniently discern the `s` and `t` components of the binding time.

After lifting the annotation to the above types, program generation and specialization to the current input $\mathtt{in}_i$ takes place.

3.4 Program Specilization

For a first example of hand writing program generators in mocaml consider the below listing 1. The multi level program `mul_ml` is declared in the top left, and then specialized with respect to an input using the `run` primitive. The generated residual program is shown at the bottom in figure 1c. For reference, a semantically equivalent OCaml program is shown at the top right. The mocaml program is an annotated OCaml program where each operation has associated binding time information. The multi-level program looks similar to the plain OCaml program - the same structure of `if-then-else` is used and the same operations are also used by the mocaml program.

```
[%%ml let mul_ml n m =
if [%lift 1 n] < [%lift 1 1]
then [%lift 2 0]
else
  [%add 2
      [%lift 2 m]
      [%app 2
          (mul_ml
              [%sub 1
                [%lift 1 n]
                [%lift 1 1]
              ]
              [%lift 2 m])]]
]
```

(a) Multi-level program

```
let rec mul n m =
if n < 1
then 0
else m + mul (n - 1) m
```

(b) OCaml source program

```
let res m = [%add 1
  [%lift 1 m]
  [%add 1
    [%lift 1 m]
    [%add 1
      [%lift 1 m]
      [%lift 1 0]]]]
```

(c) Generated residual program with n=3

```
let res2 = 6
```

(d) Second specialization with $m = 2$.

Figure 1: Multi-level specialization of a multiplication program.

The specializer works similar to the library described in [1]. For instance, when an operation like `add` or `sub` is hit by the sepcializer, it is checked whether the binding time is $t = 1$, in which case the underlying OCaml implementation is used to calculate the result. Otherwise, the binding time is simply decreased by one. As a consequence of this, the residual program in figure 1c has the bounds check and recursive calls have eliminated by the specialization. For large inputs, specialization of recursive functions might lead to code explosion. Also note that after the first specialization, all annotation have binding time $t = 1$. This situation is similar to traditional offline two-level partial evaluation where (in this case) all inputs are static and a result can be calculated. When annotations have bt > 1 they can be treated as *dynamic* and execution has to be delayed to a later stage. The resulting program after a second specialization is shown in figure 1d.

4 Semantics of mocaml

4.1 Type Chcking

OCaml is a strongly typed language with the property that well-typed OCaml programs do not go wrong as described in [4]. This is one of the main motivations for adding multi-level annotations to OCaml - handwritten program generators are still well-typed. This however only dictates that the OCaml types are consistent. It must also be ensured that the binding-time annotations of a mocaml program are consistent.

4.1.1 Binding Time Types

The same definitions of binding-time values (bt-values) and binding-time types (bt-types) of [1] will be used to describe the type checking of binding-time types in mocaml. Like in [1], the different base type, denoted B^t with binding type value t, are not distinguished between, since we do not care for them when type checking binding-time types. It is left to a later stage of the OCaml type checker to type check the OCaml base types. A big difference from [1] is that mocaml requires the programmer to manually annotate all multi-level construct with binding time information. The approach to binding-time type checking for mocaml is then done as an on-the-fly approach during specialization using the below rules. This catches any malformed binding time annotations and a suitable error message is given. The rules for type checking closely follows those in [1] and they are given below:

$$\text{Con}\frac{}{\Gamma \vdash c : B^0} \quad \text{Var}\frac{}{\Gamma \vdash x : \tau}\Gamma(x) = \tau$$

$$\text{Lift}\frac{\Gamma \vdash e : B^t}{\Gamma \vdash \texttt{[\%lift s t e]} : B^{s+t}} \quad \text{Plus}\frac{\Gamma \vdash e_1 : B^t \quad \Gamma \vdash e_2 : B^t}{\Gamma \vdash \texttt{[\%plus t e1 e2]} : B^t}$$

$$\text{If}\frac{\Gamma \vdash e_0 : B^t \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e2 : \tau} \|\tau\| \geq t$$

$$\text{App}\frac{\Gamma \vdash \texttt{fn} : \tau_1, .., \tau_n \rightarrow^t \tau' \quad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{[\%app t fn } e_1, .., e_n] : \tau'}$$

Most of the rules are already explained in [1] and wont be repeated. It can be noted that there is not an `Op` rule like in [1], but only a `plus` rule (the rules for subtraction, multiplication and division are analogous). The reason being that mocaml only supports a finite number of multi-level operations due to limitations of the implementation language OCaml. Future work could look into supporting more or arbitrary operators. Notably among the rules is the rule for application. It tells us that after t specializations the result of the application will become available. This was also seen in the two-level program in figure 1 where the result of the application was fully known after two specializations.

4.1.2 OCaml Type Checking

One of the main motivations for mocaml was the ability to generate type safe programs. The multi-level annotations for mocaml are built as syntax extensions to OCaml and the program generator will therefore run before the OCaml type checker. This means that any generated programs that successfully type checks by the OCaml type checker will be as type safe as any handwritten OCaml source program.

4.2 Call Unfolding

It was shown in figure 1 that specialization of mocaml programs can unfold recursive calls. However, as it is noted in [2], without a strategy for call unfolding, specialization might lead to infinite call unfolding. The call unfolding strategy used in mocaml is similar to the strategy in *Scheme0* as stated in [2]. That is, if we have a static branch (which means the binding time is $bt \leq 1$) then call unfolding optimization takes place. As long as there is no *static* infinite recursion, then this leads to finite call unfolding. In the example of figure 1, the branch is static (it has $bt = 1$) and as a result the loop is unfolded three times by exchanging the recursive call to `mul_ml` with the specialization of the body of `mul_ml`.

In case a recursive call takes place in a branch with $bt > 1$, then the branch is still dynamic and it will be unknown when a base case for the recursion to bottom out will occur. Therefore, specialization will leave the recursive call. To demonstrate this consider the below figure 2:

```
[%%ml let mul_dynamic_if n m =
if [%lift 2 m] < [%lift 2 1]
then [%lift 2 0]
else
  [%add 2
      [%lift 2 n]
      [%app 2
          (mul_dynamic_if
              [%lift 1 n]
              [%sub 2
                 [%lift 2 m]
                 [%lift 2 1]])]]
]
```

(a) mocaml program before specialization

```
let rec mul_dynamic_if m =
if [%lift 1 m] < [%lift 1 1]
then [%lift 1 0]
else
  [%add 1
    [%lift 1 7]
    [%app 1
       mul_ml_dynamic_if
      [%sub 1
        [%lift 1 m]
        [%lift 1 1]]]]
```

(b) mocaml program after specialization with n=7

Figure 2: Specialization of recursive function with dynamic branch.

This program is similar to figure 1, but the branch is now dynamic. As a result, there is still a recursive call in the residual program. It can be observed that the remaining recursive call is missing an argument compared to the original function definition. This is because the specializer has replaced each occurrence of the original argument n with the value $n = 7$. As a last note about call unfolding, the remaining branch of the residual program is now static, and additional specialization would therefore lead to further call unfolding. The specializer would in fact finish the entire multiplication calculation.

5 Editor Support

One of the benefits that mocaml is implemented as a syntax extension on top of OCaml and only modifies the AST is that existing OCaml tooling works. Emacs has great OCaml support and lets the programmer query a server (Merlin) about type information under the cursor. In the below listing, the the mocaml program from figure 1a is specialized for an input. The generated program `mul_ml'` is returned by Merlin to have the type `int -> int` as expected.

```
let mul_ml' = [%run mul_ml 7]
(* gives back the type: int -> int *)
```

Figure 3: A specialized program will be treated as any hand written program. Editor tools usch as Merlin will thereby still work.

There is no easy way to see the contents of a generated program. Therefore, the program generator will as a side effect print the generated program. Syntax errors and type errors are currently also poorly supported. An error will be shown with syntax or type error information, but with no information about where in the annotated mocaml program.

6 Conclusion and Future Work

To conclude the work of mocaml we will look at a final example and evaluate the design of mocaml. The below figure 4 shows a program for calculating a scaled sum $\sum_a^b sa$. The program has three inputs. As a result, the the number of lift operations and bt annotattions start to clutter the program. Future work could start by implementing a multi-level binding time annotation to also enable the support for automatically annotated programs.

```
[%%ml let sum_scale s a b =
if [%lift 3 b] < [%lift 1 2 a]
then [%lift 3 0]
else [%add 3
    [%mul 3 [%lift 2 1 s] [%lift 3 b]]
    [%app 3 (sum_scale
               [%lift 1 s]
               [%lift 2 a]
               [%sub 3 [%lift 3 1] [%lift 3 b]])]
]
]
```

Figure 4: 3-level program calculating the sum $\sum_a^b sa$

The sum_scale program can easily be used to write a program generator that as produces a residual program for calculating the sum $\sum_{i=0}^n i$ as follows:

```
let sum = [%run sum_scale]
let sum_to n = [%run sum 0]
```

Figure 5: Hand-written program generator to calculate the sum from 0...n

Although the example is fairly contrived, it still highlights some important aspects about the design of mocaml such as code reuse. The single multi-level sum_scale program can be used to generate many different programs by specializing different argument. For instance, by just specifying the scale parameter, we get a program that can compute any sum $sum_{i=a}^{b} i$. It can be imagined that if more OCaml features were added to mocaml, a more complex multi-level program could be written and more specialized programs could be generated - thus enabling more code reuse.
Although it has not been tested, specialized programs should also perform better. Some computation can be precomputed, and some calls might even be fully evaluated. Function application is however limited to recursion and future work should look into the ability to call arbitrary mocaml functions.

To conclude, this work has introduced mocaml. A syntax extension to small subset OCaml with multi-level features enabling meta-programming features such as writing simple program generators. Generated programs are type safe and checked by the OCaml type checker. Since mocaml is a small layer on top of OCaml, error reporting such as type mismatch is still provided as user feedback.

References

[1] Robert Glück and John Jørgensen. Multi-level specialization (extended abstract). In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation. Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 326–337. Springer-Verlag, 1999.

[2] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. Chapter 5.5: Call unfolding on the fly. In *Partial Evaluation and Automatic Program Generation*, pages 118–122. Prentice Hall, 1993.

[3] Oleg Kiselyov and Yukiyoshi Kameyama. Metaocaml: Theory and implementation. *arXiv preprint arXiv:2309.08207*, 2023.

[4] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[5] Tim Sheard. Type safe abstractions using program generators. *Technical Report*, CSE-95-013, 1995.

[6] Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Science Direct*, 248:211–242, 2000.

A Code

A.1 mocaml Syntax and Types

```
open Ppxlib

let pp_expression = Pprinter.pp_expression

type ml_defs = { name : string; expr : expression } [@@deriving show]

(* Primitive types that each expression can take. *)
type ml_val = Val of int | Ident of string [@@deriving show, eq]

(* Wrapper to store binding time info *)
and ml_expr = { v : ml_val; t : int } [@@deriving show]

(* Supported Ops *)
type ml_cond = Leq of ml_ops * ml_ops | Bool of bool [@@deriving show]

and ml_binop =
  | Add of ml_ops * ml_ops
  | Sub of ml_ops * ml_ops
  | Mul of ml_ops * ml_ops
  | Div of ml_ops * ml_ops
[@@deriving show]

and ml_ops =
  | Binop of int * ml_binop
  | Expr of ml_expr
  | Lift of int * ml_ops
  | Fun of string * ml_ops
  | IfElse of ml_cond * ml_ops * ml_ops
  | App of int * string * ml_ops list
[@@deriving show]

module Errors = struct
  let invalid_binding_times ~e1 ~e2 ~e1' ~e2' =
    Printf.sprintf
      {|ICE. Binding times did not match. Before:
%s
%s
After:
%s
%s
 |}
      (show_ml_ops e1) (show_ml_ops e2) (show_ml_ops e1') (show_ml_ops e2')
end

module E = Algaeff.State.Make (struct
  type t = (string * int) list
end)

let eval_leaf = function
```

```
  | Val v -> Some v
  | Ident ident ->
      print_endline @@ "eval leaf ident: " ^ ident;
      List.assoc_opt ident @@ E.get ()

let var_name = function
  | Ppat_var ident -> ident.txt
  | _ -> failwith "Only normal functions fun x -> body can be specialized"

let rec bt_of_ops op =
  match op with
  | Binop (t, _binop) -> t
  | Expr { v = _; t } -> t
  | Lift (s, op) -> s + bt_of_ops op
  | Fun (_, body) -> bt_of_ops body
  | App (t, _, _) -> t
  | _ -> failwith @@ "Cannot find bt for this expression: " ^ show_ml_ops op

let construct_binop = function
  | Add (e1, e2) -> (e1, e2, fun (e1', e2') -> Add (e1', e2'))
  | Sub (e1, e2) -> (e1, e2, fun (e1', e2') -> Sub (e1', e2'))
  | Div (e1, e2) -> (e1, e2, fun (e1', e2') -> Div (e1', e2'))
  | Mul (e1, e2) -> (e1, e2, fun (e1', e2') -> Mul (e1', e2'))

let bt_of_pexp_desc = function
  | Pexp_constant (Pconst_integer (s, _)) -> int_of_string s
  | _ -> failwith "bt must be a constant"

let create_ml_expr ?(t = 0) (expr : expression) =
  match expr.pexp_desc with
  | Pexp_constant (Pconst_integer (i, _)) ->
      let v = Val (int_of_string i) in
      Some { v; t }
  | Pexp_ident { txt = Lident var; loc = _loc } -> Some { v = Ident var; t }
  | _ -> None
```

A.2 Specializer

```
open Multi_level_ops
open! Pprinter
open Ppxlib
open! Base
open! Stdio

module S = Algaeff.State.Make (struct
  type t = ml_ops
end)

(* TODO: This module should be about done.
   The only thing left to check is that all the bt rules fir
   type checking is implemented. *)

let lift_binop t e1 e2 ~traverse ~binop =
  let _ = traverse e1 in
  let e1' = S.get () in
  let _ = traverse e2 in
  let e2' = S.get () in
  let e = Binop (bt_of_pexp_desc t.pexp_desc, binop (e1', e2')) in
  S.set e

let decrease_bt e v = if e.t > 0 then { v; t = e.t - 1 } else { v; t = 0 }

let rec fun_body fn =
  match fn with Fun (_, body) -> fun_body body | body -> body

let rec replace ~ident ~with_val in_op =
  let ( let* ) = Result.( >>= ) in
  let ( >>= ) = Result.( >>= ) in

  let module R = Algaeff.Reader.Make (Bool) in
  let rec eval = function
    | Binop (1, binop) -> eval_binop binop
    | Lift (s, e) when s <= 1 -> eval e
    | Expr e when e.t <= 1 -> (
        match e.v with
        | Val v -> Ok v
        | ident' when equal_ml_val ident' ident -> (
            match with_val with Ident _ -> Error "Impossible" | Val v -> Ok v)
        | _ -> Error ("Cannot evaluate identifiers: " ^ show_ml_val e.v))
    | e -> Error ("Cannot eval expr of bt > 1: " ^ show_ml_ops e)
  and eval_binop binop =
    let eval_binop' e1 e2 (oper : int -> int -> int) =
      let* v1 = eval e1 in
      let* v2 = eval e2 in
      Ok (oper v1 v2)
    in
    match binop with
    | Add (e1, e2) -> eval_binop' e1 e2 Int.( + )
    | Sub (e1, e2) -> eval_binop' e1 e2 Int.( - )
    | Mul (e1, e2) -> eval_binop' e1 e2 Int.( * )
    | Div (e1, e2) -> eval_binop' e1 e2 Int.( / )
  in

  let rec go_cond cond =
    match cond with
    | Leq (e1, e2) ->
        let* e1' = go e1 in
        let* e2' = go e2 in
        if bt_of_ops e1' = bt_of_ops e2' && bt_of_ops e1' = 0 then
          let* v1 = eval e1' in
          let* v2 = eval e2' in
          Ok (Bool (v1 < v2))
        else Ok (Leq (e1', e2'))
    | b -> Ok b
  and go op =
    match op with
    | Binop (1, binop) ->
        let e1, e2, make_binop = construct_binop binop in
        let* e1' = go e1 in
        let* e2' = go e2 in
        let* v = eval @@ Binop (1, make_binop (e1', e2')) in
        Ok (Expr { v = Val v; t = 0 })
    | Binop (t, binop) ->
        let e1, e2, make_binop = construct_binop binop in
        let* e1' = go e1 in
        let* e2' = go e2 in
        if bt_of_ops e1' = bt_of_ops e2' then
          Ok (Binop (t - 1, make_binop (e1', e2')))
        else
          Error (Multi_level_ops.Errors.invalid_binding_times ~e1 ~e2 ~e1' ~e2')
    | Expr e when equal_ml_val e.v ident -> Ok (Expr (decrease_bt e with_val))
    | Expr e -> Ok (Expr (decrease_bt e e.v))
    | Fun (a, body) ->
        let* body' = go body in
        Ok (Fun (a, body'))
    (* Release the inner value. s specializations has occured. *)
    | Lift (1, e) when bt_of_ops e = 0 ->
        let* e' = go e in
        Ok e'
    | Lift (s, e) ->
        (* In this case, if t=0 and s=0 the expression is just evaluated as is *)
        let t = bt_of_ops e in
        let* e' = go e in
        if t = 0 then Ok (Lift (s - 1, e')) else Ok (Lift (s, e'))
    | IfElse (cond, e_then, e_else) when bt_of_ops e_then = bt_of_ops e_else
      -> (
        let* cond' = go_cond cond in
        match cond' with
        | Bool b ->
            R.scope (fun _ -> true) @@ fun () ->
            if b then go e_then else go e_else
        | _ ->
            let* e_then' = go e_then in
            let* e_else' = go e_else in
            Ok (IfElse (cond', e_then', e_else')))
    | IfElse _ -> Error "Branches must have the same binding times"
    (* Test of the first argument got smaller.
       Only also works for static tests also. *)
    | App (t, fn, arg :: args) -> (
        (* print_endline @@ show_ml_ops op; *)
        match bt_of_ops arg with
        | 1 ->
            let* v = eval arg >>= fun v -> Result.return @@ Val v in
            (* TODO: Need infinite recusion? *)
            let body = fun_body in_op in
            let is_static_if = R.read () in
            let* args' = Result.all @@ List.map ~f:go args in
            if is_static_if then replace ~ident ~with_val:v body
            else Result.return @@ App (t - 1, fn, args')
            (* TODO: Safe to remove specialized arg? *)
        | _ ->
            Result.fail
            @@ "First argument to multi-level function should of bt=1.\nExpr: "
            ^ Multi_level_ops.show_ml_ops op)
    | App (_, _, []) -> Result.fail "App must take at least 1 argument"
  in
  R.run ~env:false @@ fun () -> go in_op

let specialize (to_specialize : expression) (arg : expression) : expression =
  let lift v ident =
    object (self)
      inherit Ast_traverse.iter as _super

      (* Traverse expressions that we try to specialize.
         The expression will either be a function, where we try to specialize
         one of the arguments, or a multilevel expression. The later happens if there
         is only one argument left to sepcialize.*)
      method! expression expr =
        match expr with
        | [%expr fun [%p? p] -> [%e? rest]] ->
            self#expression rest;
            let body = S.get () in
            S.set @@ Fun (var_name p.ppat_desc, body)
        | [%expr [%add [%e? t] [%e? e1] [%e? e2]]] ->
            lift_binop t e1 e2 ~traverse:self#expression
              ~binop:(fun (e1', e2') -> Add (e1', e2'))
        | [%expr [%sub [%e? t] [%e? e1] [%e? e2]]] ->
            lift_binop t e1 e2 ~traverse:self#expression
              ~binop:(fun (e1', e2') -> Sub (e1', e2'))
        | [%expr [%mul [%e? t] [%e? e1] [%e? e2]]] ->
            lift_binop t e1 e2 ~traverse:self#expression
              ~binop:(fun (e1', e2') -> Mul (e1', e2'))
        | [%expr [%div [%e? t] [%e? e1] [%e? e2]]] ->
            lift_binop t e1 e2 ~traverse:self#expression
              ~binop:(fun (e1', e2') -> Div (e1', e2'))
        | [%expr [%lift [%e? t] [%e? e]]] ->
            let bt = bt_of_pexp_desc t.pexp_desc in
            (* Try to lift either constant or ident.
               In case that fails, recursively lift the sub expression*)
            let e' =
              match create_ml_expr e ~t:bt with
              | Some expr -> Expr expr
              | None ->
                  _super#expression e;
                  S.get () (* TODO: super or self *)
            in
            S.set e'
        | [%expr [%lift [%e? s] [%e? t] [%e? e]]] ->
            let bt = bt_of_pexp_desc t.pexp_desc in
            let s' = bt_of_pexp_desc s.pexp_desc in
            let e' =
              match create_ml_expr e ~t:bt with
              | Some expr -> Expr expr
              | None ->
                  self#expression e;
                  S.get ()
            in
            S.set (Lift (s', e'))
        | [%expr if [%e? e1] < [%e? e2] then [%e? b1] else [%e? b2]] ->
            (* TODO: will not generate correctly if e1 or e2 are not lift operations *
            self#expression e1;
            let e1' = S.get () in
            self#expression e2;
            let e2' = S.get () in
            self#expression b1;
            let e_then = S.get () in
            self#expression b2;
            let e_else = S.get () in
            let cond = Leq (e1', e2') in
            S.set (IfElse (cond, e_then, e_else))
        (* The last case can be any OCaml expression. However thse might
           still possibly contain ML ops.*)
        | [%expr [%app [%e? t] [%e? fn_app]]] -> (
            match fn_app.pexp_desc with
            | Pexp_apply
                ({ pexp_desc = Pexp_ident { txt = Lident fname; _ }; _ }, args)
              ->
                let bt = bt_of_pexp_desc t.pexp_desc
                and args_op =
                  List.map args ~f:(fun (_, expr) ->
                      self#expression expr;
                      S.get ())
                in
                S.set @@ App (bt, fname, args_op)
            | _ -> failwith "Expected function application: [%app t fn (args)]."
            )
        | _ ->
            failwith @@ "Expression not implemented: "
            ^ Pprintast.string_of_expression expr
    end
```

```
  in
  let arg_ml_expr =
    match create_ml_expr arg with
    | Some expr -> expr
    | None -> failwith "You can only specialize with a constant"
  in

  let loc = to_specialize.pexp_loc in
  (* Specialize a: fun args -> body.
     NOTE: Body might itself be a fun.*)
  let specialize_fun arg body =
    match arg.ppat_desc with
    | Ppat_var ident_loc -> (
        let lift_body = lift arg_ml_expr.v (Ident ident_loc.txt) in
        lift_body#expression body;
        let specialization =
          replace ~ident:(Ident ident_loc.txt) ~with_val:arg_ml_expr.v
            (S.get ())
        in
        match specialization with
        | Ok specialization -> Codegen.cogen ~loc specialization
        | Error msg -> Codegen.fail_with ~loc:body.pexp_loc msg)
    | _ -> Codegen.fail_with "invalid type" ~loc:arg.ppat_loc
  in
  (* Traverse the tree inside an effect handler to collect states *)
  S.run ~init:(Expr arg_ml_expr) (fun () ->
      (* Codegen.fail_with ~loc:to_specialize.pexp_loc "foo" *)
      print_endline "to specialize:";
      print_endline (Pprinter.show_exp to_specialize);
      match to_specialize with
      | [%expr
          let rec [%p? fn] = fun [%p? arg] -> [%e? body] in
          [%e? _]] ->
          specialize_fun arg body
      | [%expr fun [%p? arg] -> [%e? body]] -> specialize_fun arg body
      | _ ->
          Codegen.fail_with ~loc:to_specialize.pexp_loc
          @@ "wrong type: "
          ^ Pprinter.show_exp to_specialize)
```

A.3 Code Generator

```
open! Ppxlib
open! Pprinter

let fail_with text ~loc = Location.raise_errorf ~loc "%s" text

let gen_binop ~ctxt ~oper expr =
  let loc = Expansion_context.Extension.extension_point_loc ctxt in
  match expr with
  | [%expr [%e? t] [%e? e1] [%e? e2]] ->
      (* TODO: Consider binding times *)
      [%expr [%e oper] [%e e1] [%e e2]]
  | e ->
      let msg =
        Printf.sprintf "failed generating code for: %s" (Pprinter.show_exp e)
      in
      fail_with msg ~loc

let gen_add ~ctxt =
  let loc = Expansion_context.Extension.extension_point_loc ctxt in
  gen_binop ~ctxt ~oper:[%expr Int.add]

let gen_sub ~ctxt =
  let loc = Expansion_context.Extension.extension_point_loc ctxt in
  gen_binop ~ctxt ~oper:[%expr Int.sub]

let gen_div ~ctxt =
  let loc = Expansion_context.Extension.extension_point_loc ctxt in
  gen_binop ~ctxt ~oper:[%expr Int.div]

let gen_mul ~ctxt =
  let loc = Expansion_context.Extension.extension_point_loc ctxt in
  gen_binop ~ctxt ~oper:[%expr Int.mul]

let gen_lift ~ctxt expr =
  let loc = Expansion_context.Extension.extension_point_loc ctxt in
  match expr with
  | [%expr [%e? _t] [%e? e]] -> [%expr [%e e]]
  | [%expr [%e? _t] [%e? _s] [%e? e]] -> [%expr [%e e]]
  | _ -> fail_with ~loc "Invalid lift"

let gen_val ~loc leaf =
  let open Multi_level_ops in
  match leaf with
  | Val v -> Ast_builder.Default.eint v ~loc
  | Ident id -> Ast_builder.Default.evar id ~loc

let gen_app ~ctxt (fn_app : expression) =
  let loc = Expansion_context.Extension.extension_point_loc ctxt in
  match fn_app with
  | [%expr [%e? _t] [%e? fn_app]] -> fn_app
  (* match fn_app.pexp_desc with *)
  (* | Pexp_apply (fn, args) -> *)
  (*   let args' = List.map snd args in *)
  (*   Stdio.print_endline "Args;"; *)
  (*   Stdio.print_endline @@ [%show: expression list] args'; *)
  (*   Stdio.print_endline "fn;"; *)
  (*   Stdio.print_endline @@ Pprinter.show_exp fn; *)
  (*   Ast_builder.Default.pexp_apply ~loc fn args *)
  | _ ->
      fail_with ~loc @@ "Invalid function application generated: "
      ^ Pprinter.show_exp fn_app

(* TODO: We need to rename or "quote" the function name if there is an application
   and bind it to an inner recursive helper like:
   let sum =
     let rec sum' n = if n < 1 then 0 else n + sum' (n - 1) in
     sum'
*)

module S = Algaeff.State.Make (struct
  type t = string option
end)

let rec gen_code ~loc op =
  let open Multi_level_ops in
  match op with
  | Binop (t, binop) -> (
      let e1, e2, _ = construct_binop binop in
      let t' = Ast_builder.Default.eint t ~loc in
      let e1' = gen_code e1 ~loc in
      let e2' = gen_code e2 ~loc in
      match binop with
      | Add _ -> [%expr [%add [%e t'] [%e e1'] [%e e2']]]
      | Sub _ -> [%expr [%sub [%e t'] [%e e1'] [%e e2']]]
      | Div _ -> [%expr [%div [%e t'] [%e e1'] [%e e2']]]
      | Mul _ -> [%expr [%mul [%e t'] [%e e1'] [%e e2']]])
  | Expr e ->
      let t = Ast_builder.Default.eint e.t ~loc in
      let expr = gen_val ~loc e.v in
      [%expr [%lift [%e t] [%e expr]]]
  | Lift (s, e) ->
      let e =
        match e with
        | Expr e -> [%expr [%e gen_val e.v ~loc]]
        | _ -> gen_code e ~loc
      and t = Ast_builder.Default.eint ~loc @@ bt_of_ops e
      and s' = Ast_builder.Default.eint s ~loc in
      [%expr [%lift [%e s'] [%e t] [%e e]]]
  | Fun (a, body) ->
      let a' = Ast_builder.Default.ppat_var ~loc (Loc.make ~loc a) in
      let body' = gen_code body ~loc in
      [%expr fun [%p a'] -> [%e body']]
  | IfElse (cond, e_then, e_else) ->
      let cond' =
        match cond with
        | Leq (e1, e2) ->
            let e1' = gen_code e1 ~loc in
            let e2' = gen_code e2 ~loc in
```

```
          [%expr [%e e1'] < [%e e2']]
        | _ -> fail_with "unexpected conditional" ~loc
      in
      let e_then' = gen_code e_then ~loc in
      let e_else' = gen_code e_else ~loc in
      [%expr if [%e cond'] then [%e e_then'] else [%e e_else']]
  | App (t, fn, args) ->
      S.set (Some fn);
      let fn_exp = Ast_builder.Default.evar ~loc fn in
      let arg_labels =
        List.map (fun arg -> (Nolabel, gen_code arg ~loc)) args
      in
      let t' = Ast_builder.Default.eint ~loc t in
      let fn_app = Ast_builder.Default.pexp_apply fn_exp arg_labels ~loc in
      [%expr [%app [%e t'] [%e fn_app]]]

let cogen ~loc op =
  S.run ~init:None @@ fun () ->
  let code = gen_code ~loc op and has_app = S.get () in
  match has_app with
  | Some fname ->
      let fn_pat = Ast_builder.Default.pvar ~loc fname
      and fn_expr = Ast_builder.Default.evar ~loc fname in
      [%expr
        let rec [%p fn_pat] = [%e code] in
        [%e fn_expr]]
  | None -> code
```

A.4 Syntax Rewriter

```
open Ppxlib
open Multi_level_ops

let map_structure (structure : structure) =
  (* TODO: maybe inpalce updates. We already track these effects *)
  let module E = Algaeff.State.Make (struct
    type t = ml_defs list
  end) in
  let ( let* ) = Option.bind in
  let mapper =
    object
      inherit Ast_traverse.map as super

      method! structure_item stri =
        let loc = stri.pstr_loc in
        match stri with
        | [%stri [%%ml let [%p? decl] = [%e? expr]]] -> (
            match decl.ppat_desc with
            | Ppat_var loc' ->
                (* TODO: Here it could also make sense to traverse the function
                       and build ml_ops. *)
                    let fname = loc'.txt in
                    let def = { name = fname; expr } in
                    E.modify (fun defs -> def :: defs);
                    [%stri []]
                | _ -> Codegen.fail_with "invalid multi level declaration" ~loc)
            | [%stri let [%p? lhs] = [%run [%e? f] [%e? expr]]] -> (
                match f.pexp_desc with
                | Pexp_ident ident -> (
                    let fname =
                      match ident.txt with Lident fname -> fname | _ -> ""
                    in
                    let specialization =
                      let* ml_def =
                        List.find_opt
                          (fun def -> String.equal fname def.name)
                          (E.get ())
                      in
                      let specialization = Pe.specialize ml_def.expr expr in
                      let new_fun_decl =
                        { name = var_name lhs.ppat_desc; expr = specialization }
                      in
                      E.modify (fun defs -> new_fun_decl :: defs);
                      Option.some specialization
                    in
                    match specialization with
                    | Some expr -> [%stri let [%p lhs] = [%e expr]]
                    | None ->
                        Codegen.fail_with
                          ("Multi level decl for " ^ fname ^ " not found")
                          ~loc)
                | _ ->
                    Codegen.fail_with
                      "run not impl for more than 1 argument to specializer" ~loc)
            | _ -> super#structure_item stri
      end
    in
    let res = E.run ~init:[] (fun () -> mapper#structure structure) in
    print_endline "-----";
    print_endline @@ Pprinter.show_strct res;
    print_endline "-----";
    res

let run_file () =
  let position = Driver.Instrument.Before in
  let instrument = Driver.Instrument.make map_structure ~position in
  (* TODO: make it instrument and run before the context_free *)
  Driver.register_transformation "global" ~instrument

let create_binop_rule ext_name rewriter =
  Context_free.Rule.extension
  @@ Extension.V3.declare ext_name Extension.Context.expression
       Ast_pattern.(single_expr_payload __)
       rewriter

let () =
  run_file ();
  let rule_add = create_binop_rule "add" Codegen.gen_add in
  let rule_sub = create_binop_rule "sub" Codegen.gen_sub in
  let rule_div = create_binop_rule "div" Codegen.gen_div in
  let rule_mul = create_binop_rule "mul" Codegen.gen_mul in
  let rule_lift =
    Context_free.Rule.extension
    @@ Extension.V3.declare "lift" Extension.Context.expression
         Ast_pattern.(single_expr_payload __)
         Codegen.gen_lift
  in
  let rule_app =
    Context_free.Rule.extension
    @@ Extension.V3.declare "app" Extension.Context.expression
         Ast_pattern.(single_expr_payload __)
         Codegen.gen_app
  in
  Driver.register_transformation
    ~rules:[ rule_lift; rule_add; rule_sub; rule_div; rule_mul; rule_app ]
    "expression"
```