Mocaml - a multi level meta programming extension to OCaml

Kristoffer Kortbaek

2024/06/21

Abstract

This is an abstract. These are citations: [6][3][5][1]

Contents

1 Introduction

There have been multiple implementation 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 introduce 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 new program generator. 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 combine the work of multi-level program generation described by [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 hand-writing 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 the work by [1] the process of multi-level generating extensions is demonstrated. This will quickly be summarized as to demonstrate how mocaml differs from this approach. For a program `p` written in language `L` then let $[\![\mathtt{p}]\!]_{\mathtt{L}}$ `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` 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 program generator 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 handwrite program generator with the user manually annotating the multi-level program with binding time information. 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]. For mocaml programs, this means that the user can stage the execution by annotating the multi-level program appropriately. 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 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. However, the annotations are done manually and the output of each stage is a residual multi-level program that can be specialized further by the mocaml program generator. 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 used as is since it is just a generated program.

3 Implementation

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. 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]
          [%lift 2 b]]]
```

As the above example demonstrates, multi-level programs are first decorated with the special `[%%ml ...]` syntax extension. This tells the multi-level compiler generator to treat the add program 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 multi-level extensions to OCaml is called *mocaml*, and the abstract syntax of the language is similar to *MetaScheme* presented in [1]. Each multi-level construct of mocaml has 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`

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

Unlike Lisp, OCaml does not provide the ability to treat code as data, and a generating extension can not be represented in code as a multi-level program and a library for code generation and specialization as it is done in [1]. However, like in [1], mocaml still only modifies the AST of OCaml. The `[%run f e]` ocaml syntax extension gets the ParseTree[1] 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:

[1]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.

As it can be seen, the types correspond to the definitions of the abstract syntax of mocaml. The only base type allowed for mocaml is currently integers. The `ml_expr` type is used to store binding time info of leaf nodes such as constants and argument identifiers. This way, an expression like `[%lift 2 3]` can be compactly represented as an `ml_expr` with `v=Val 3` and binding time $t = 2$. It was briefly mentioned that specialization and code generation works on the DSL defined by the above types. Program generation itself happens in two stages:

- When the `[%run ml_func arg]` annotation is used, the syntax tree of the `ml_func` is lifted into the above types and specialization takes place.
- Then, code generation is conducted on the specialization. It is still possibly to further specialize the result given more arguments.

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. For reference, a semantically equivalent OCaml program is shown at the top right. The mocaml program is decorated with multi level operations and binding time information, but otherwise looks almost identical to the 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.

As it was previously mentioned in listing 1, `run` lifts a multi-level mocaml program into a OCaml DSL, and specialization then works on the DSL. 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 been eliminated by the specialization. For large inputs, specialization of recursive functions might lead to code explosion. Also note that if the residual program is specialized once more, then all constructs have binding time $t = 1$, and we have a situation similar to traditional offline two-level partial evaluation where (in this case) all inputs are static and a result can be calculated. 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. For this reason, both aspect of well-typed mocaml programs will be discussed in this section.

4.1.1 Binding Time

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 ocaml 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:

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

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

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

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

Most of the rules are already explained in [1] and wont be repeated. Also note 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.

4.3 Editor Support

What of the benefits that the mocaml program generator only modifies the OCaml 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 specialized 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 telling the programmer that there is a syntax or type error in the multi level program, but with no information about where.

5 Results and Future Work

This work has demonstrated mocaml, a small multi-level extension to OCaml for writing program generators. In mocaml, expressions can be dynamic until a certain stage has been reach, and afterwords the expression becomes static and it can be partially evaluated. A call unfolding strategy was also presented. It guarantees that recursive calls are only unfolded if there is a guarantee that

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.