Mocaml - a multi level meta programming extension to OCaml

Kristoffer Kortbaek

2024/06/21

Abstract

This is an abstract. These are citations: [5][2][4][1]

Contents

1 Introduction

There have been multiple implementation of meta programming in the ML family of languages. Notable implementations are [5] and [2] 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 [2]. One of the notable features of MetaML and Meta-OCaml is that the languages enables the programmer write type checked program generators [2].

[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, 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 [2]. 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 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 Implementation

2.1 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.

2.2 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:

$p \in Program; x \in Variable; d \in Definition; c \in Constant;$
$e \in Constant; s, t \in BindingTime$

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

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

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

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

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

| `[%div t e1 e2]` | `[%app 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.

2.3 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. Below the types defining the DSL is given:

[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 * expression * (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 definition of the abstract syntax of mocaml. As mentioned, specialization and code generation works on these types.

2.4 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.

3 mocaml Semantics

3.1 Type Chcking

OCaml is a strongly typed language with the property that well-typed OCaml programs do not go wrong as described in [3]. This is one of the main motivation 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 is consistent. Both aspect of well-typed mocaml programs will be discussed in this section.

3.1.1 Binding Time

Since there is no multi-level binding time analysis for mocaml programs, it is up to the programmer to add the correct binding time value to each multi-level annotation. During

3.2 Call Unfolding

Using recursion in mocaml can cause problems with infinite call unfolding during specialization. For the multiplication program shown in figure 1a, the specialization simply unfolded the recursive call $n = 3$ times. This succeeded because the condition has binding time $t = 1$ and the specializer can thereby detect when the branch eventually becomes `true` and recursion stops. However, if the branch had instead conditioned on $m < 1$, then the binding time would be $t = 2$, and the `if-then-else` expression would still be dynamic at the current stage. As a consequence, if the specializer tries to unroll the recursion, it will run into infinite recursion.

4 Results and Future Work

5 Conclusion

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

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

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

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