Safe Pattern Generation for Multi-Stage Programming

Ethan Range
University of Cambridge
ethan@ethanrange.com

Jeremy Yallop
University of Cambridge
jeremy.yallop@cl.cam.ac.uk

Abstract

Multi-stage programming (MSP) is a useful way of generating optimised programs, but existing MSP systems have limited expressiveness, with no support for type-safe generation of patterns whose shape is statically unknown (i.e. determined only when code generators are run, not when they are written). This work introduces a design for typed generation of arbitrary patterns, and shows how to use the design to generate statically unknown patterns.

1. Introduction

Multi-stage programming (MSP) is a metaprogramming paradigm that enables generation of programs guaranteed to be well-typed and well-scoped. MSP typically uses two constructs: a quotation `.<e>.` of type `'a code` denotes a code fragment of an expression `e` of type `'a`, while a splice `.~c` of type `'a` inserts a fragment `c` of type `'a code` into a larger fragment. These constructs support the definition of generators that can specialize an algorithm to a given input, such as specialisation of x^n to a given n (Ershov 1977).

One specialisation candidate is the recursive list `map` function:

```
let rec map (f : 'a -> 'b) (l : 'a list) : 'b list =
    match l with
        | [] -> []
        | x :: xs -> let y = f x in y :: map f xs
```

which may be unrolled an arbitrary number of recursive steps to improve performance (Appel 1992). For example, unrolling 2 steps (with a final case for lists shorter than the unrolling depth) gives:

```
let rec map (f : 'a -> 'b) (l : 'a list) : 'b list =
    match l with
        | [] -> []
        | x1 :: x2 :: xs ->
            let y1 = f x1 in
            let y2 = f x2 in
            y1 :: y2 :: map f xs
        | x :: xs -> let y = f x in y :: map f xs
```

The optimal unrolling depth depends on how the function is used, making unrolling an ideal candidate for specialisation by staging.

Unfortunately, existing MSP systems do not support safe generation of statically unknown patterns like n-ary `cons`. Some, e.g. Template Haskell (Sheard and Peyton Jones 2002), support unsafe generation of arbitrary patterns, while others, e.g. MetaOCaml (Kiselyov 2014), support safe generation only of statically known patterns.

2. Type-safe pattern generation

This work introduces a typed embedded domain-specific language (DSL) for generation of statically unknown patterns, implemented as a MetaOCaml library. The DSL represents patterns using a type constructor `pat`, analogous to the `code` type constructor that represents quoted expressions. However, while the expression typing judgment $\Gamma \vdash \mathsf{e} : \tau$ associates a type τ with each expression (under some context Γ), the pattern typing judgment $\Gamma \vdash \mathsf{p} : \langle \Gamma', \tau \rangle$ associates both a type and a bound variable context Γ' with each

OCaml	**DSL**	`pat` **type**
`_`	`__`	`('a, 'r, 'r) pat`
`int` literal	`int` n	`int -> (int, 'r, 'r) pat`
Identifier	`var`	`('a, 'a code -> 'r, 'r) pat`
`[]`	`empty`	`('a list, 'r, 'r) pat`
If `x : ('a, 'f, 'g) pat` *and* `xs : ('a list, 'g, 'r) pat` *then*		
`x :: xs`	`x >:: xs`	`('a list, 'f, 'r) pat`

Figure 1. Selected term constructors from the pattern DSL

pattern. For example, the judgement associates the type `'a list` and the context `x :'a, xs :'a list` with the pattern `x :: xs`.

The `pat` type in the DSL is therefore parametrised by both Γ' and τ. In contrast to languages based on CMTT (Nanevski et al. 2008) whose type systems directly support contexts, representing context types in MetaML-family languages like MetaOCaml requires careful encoding to support operations like concatenation. Lindley's (2008) *difference types* meet this requirement[1]: a context type is encoded as a function type `'f` whose parameter types correspond to the context variables and whose return type `'r` can be instantiated to the type of an expression in which those variables are bound. Adding one more parameter `'a` for the input type of a pattern gives the following form for `pat`:

```
type ('a, 'f, 'r) pat
```

Figure 1 shows DSL term constructors for selected OCaml patterns — wildcard, constants, variables, nil and cons — with their types. An OCaml pattern can thus be represented in the DSL:

```
x :: (3 :: _)  ⇝  var >:: (int 3 >:: __)
```

with `'a` as `int list`, `'f` as `'int code -> 'c code` and `'r` as `'c code`. Invalid patterns are rejected as desired:

```
x :: 3         ⇝  var >:: int 3
```

```
Error: Type int is not compatible with type 'b list
```

Constructing a pattern-matching case requires both a DSL description of a pattern and a right-hand side (RHS) expression. This RHS can be defined with a function of the `'f` type described above. The `(=>)` operator combines a pattern and an RHS:

```
val (=>) : ('a, 'f, 'r code) pat -> 'f -> ('a, 'r) case
```

corresponding to the typing rule:

$$\frac{\Gamma \vdash \mathsf{p} : \langle \Gamma', \tau \rangle \qquad \Gamma \cup \Gamma' \vdash \mathsf{e} : \tau'}{\Gamma \vdash \mathsf{p} \Rightarrow \mathsf{e} : \mathrm{case}(\tau, \tau')}$$

[1] Fridlender and Indrika (2000) and others apply similar techniques

The RHS takes the form `fun r0 r1 ... rn -> .<e>.`. The arguments `r0 ... rn` are quotations of the identifiers representing variables bound in the pattern, and `e` is the expression for the case right-hand side, which can contain the pattern variable identifiers spliced in. With this approach, the pattern-matching case:

```
| x :: xs -> (x, xs)
```

may be constructed as:

```
var >:: var => fun x xs -> .< (.~x, .~xs) >.
```

Combining a pattern and an RHS produces a value of type `('a,'r) case`. The type parameters here are similar to those in `pat`, but `'f` is omitted, as it is no longer required for validation, just as Γ' does not appear in the conclusion of the rule.

This representation of a case RHS as a function is closely related to Rhiger's (2009) design for first-class patterns, approaching the problem of pattern generation as a translation from the function representation of a pattern to the built-in representation of a pattern in some target language.

Finally, the `match_` combinator generates an OCaml `match` expression from a list of cases that share an input and output type:

```
val match_ : 'a code -> ('a, 'b) case list -> 'b code
```

3. Statically unknown pattern generation

The constructors in Figure 1 give different types to patterns that bind different numbers of variables: in the `cons` patterns previously introduced, as the number of pattern variables bound varies with n, the RHS function type, `'f`, also varies. This variation in the types does not prevent generation of statically unknown patterns, but to prevent exposure of the varying `'f` type, the pattern and the RHS expression must be generated simultaneously.

An RHS expression for an n-ary `cons` pattern can be generated by an inductive definition, with a base case, handling the final `'b list` in a `cons` pattern, and an inductive case, that combines an identifier of a value of type `'b` and the existing RHS expression definition. For example, shown below are base and inductive cases for a summation RHS:

```
let base (xs : int list code) = .<0>.
let ind (v : int code) (acc : int code) = .<.~v + .~acc>.
(* .<x1 + (x2 + (... + (xn + 0)))>. *)
```

When generating this pattern and RHS expression, the intermediate results must be wrapped to prevent exposure of the `'f` type. To both allow the use of the `(=>)` operator to construct a `case` from the unwrapped pattern and expression, and allow application of an inductive definition, the wrapper uses a continuation style:

```
type ('a, 'r) pwrap = Pat : ('a list, 'f, 'r code) pat
                        * (('r code -> 'r code) -> 'f)
                        -> ('a, 'r) pwrap
```

Each application of the inductive step is represented as a modification of the result, of type `'r code -> 'r code`. This approach avoids the need to manipulate the existential `'f` type within the wrapper directly while allowing the function of type `'f` required to be retrieved by applying the continuation to the identity function. With this approach, a combinator to produce n-ary `cons` patterns from an inductive definition can be defined:

```
let gen_n_cons (m : int)
               (base : 'a list code -> 'r code)
               (ind : 'a code -> 'r code -> 'r code)
             : ('a list, 'r) case =
  let rec loop n : ('a, 'r) pwrap =
    if n = 0
    then Pat (var, fun k xs -> k (base xs))
    else let Pat (p, k) = loop (n - 1) in
      Pat (var :: p, fun c x -> k (compose (ind x) c))
  in let Pat (p, c) = loop m in p => (c Fun.id)
```

This combinator enables defining the desired generator for unrolled map functions:

```
let gen_unrolled_map (n : int) =
  .<let rec map f l = .~(match .<l>. [
    empty       => .<[]>.;
    gen_n_cons n (fun xs -> .<map f .~xs>.)
                 (fun x acc ->
                    .<let y = f .~x in y :: .~acc>.);
    var >:: var => .<fun x xs ->
                      let y = f x in y :: map f xs>.
]) in map>.
```

4. Ensuring well-scopedness

The type system of the DSL ensures that generated code is well-typed. Ensuring that code is also well-scoped requires addressing two additional issues in the implementation.

4.1 Pattern variable naming

Generating identifiers for pattern variables requires care to avoid collisions with variables occurring free in the RHS expression, since collisions may cause incorrect scope inclusion, type errors, or incorrect semantics (Kiselyov 2014). Identifier generation is managed with a global counter, ensuring unique pattern variable names. As MetaOCaml uniquely renames variables in generated code, and as the unique naming strategy used in this work differs from MetaOCaml's strategy, collisions cannot occur with non-pattern variable identifiers. Integration of this library into MetaOCaml would allow for unification of these two unique naming strategies.

4.2 Scope extrusion

Scope extrusion occurs when a variable in a code quotation escapes the scope of its binding (Pickering et al. 2019). MetaOCaml prevents scope extrusion by tracking free variables and virtual bindings in code quotations, and raising run-time exceptions when extrusion is detected. For pattern generation, preventing scope extrusion requires two distinct tasks. First, any variable used in an RHS expression must be checked for extrusion. This check is implemented as part of the `match_` code generator.

The second task is to update MetaOCaml's record of free variables with the identifiers generated for pattern variables. This ensures that scope extrusion of variables bound in a generated pattern can be detected. This is not currently implemented in this work, however. While relatively simple to achieve, updating the free variable record would require integration of this library into MetaOCaml, due to reliance on internal, unexposed state within MetaOCaml.

5. Status and future work

This work provides a design for type-safe generation of arbitrary patterns. To remedy the safety caveat discussed above, integration of this library into MetaOCaml is a natural avenue of future work, but the approach is sufficiently general to also be applicable to other MSP systems such as Template Haskell (Sheard and Peyton Jones 2002) or MacoCaml (Xie et al. 2023). The implementation currently supports only a subset of OCaml pattern types, but the type system provides an extensible platform for implementation of alternative pattern types, although challenges remain around GADT patterns, polymorphic variant patterns, and disjunctive patterns with variables bound in differing orders between disjuncts.

References

A. W. Appel. Unrolling recursions saves space. Technical Report CS-TR-363-92, Princeton University, March 1992. URL https://www.cs.princeton.edu/techreports/1992/363.ps.gz.

A. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, 1977. ISSN 0020-0190. . URL https://www.sciencedirect.com/science/article/pii/0020019077900783.

D. Fridlender and M. Indrika. Do we need dependent types? *J. Funct. Program.*, 10(4):409–415, 2000. . URL https://doi.org/10.1017/s0956796800003658.

O. Kiselyov. The design and implementation of BER MetaOCaml. In M. Codish and E. Sumii, editors, *Functional and Logic Programming*, pages 86–102, Cham, 2014. Springer International Publishing. ISBN 978-3-319-07151-0. . URL https://doi.org/10.1007/978-3-319-07151-0_6.

S. Lindley. Many holes in Hindley-Milner. In E. Sumii, editor, *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*, pages 59–68. ACM, 2008. . URL https://doi.org/10.1145/1411304.1411313.

A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3):23:1–23:49, 2008. . URL https://doi.org/10.1145/1352582.1352591.

M. Pickering, N. Wu, and C. Kiss. Multi-stage programs in context. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 71–84, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368131. . URL https://doi.org/10.1145/3331545.3342597.

M. Rhiger. Type-safe pattern combinators. *Journal of Functional Programming*, 19(2):145–156, 2009. . URL https://doi.org/10.1017/S0956796808007089.

T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, Dec 2002. ISSN 0362-1340. . URL https://doi.org/10.1145/636517.636528.

N. Xie, L. White, O. Nicole, and J. Yallop. MacoCaml: Staging Composable and Compilable Macros. *Proc. ACM Program. Lang.*, 7(ICFP), Aug 2023. . URL https://doi.org/10.1145/3607851.