UNIVERSITY OF CAMBRIDGE
Department of Computer Science and Technology

Safe Pattern Generation for Multi-Stage Programming

Ethan Range

St Edmund's College

June 2024

Submitted in partial fulfilment of the requirements for the
Master of Philosophy in Advanced Computer Science

Total page count: 48

Main chapters (excluding front-matter, references and appendix): 44 pages (pp 1–44)

Main chapters word count: 14914

Methodology used to generate that word count:

```
$ make wordcount
gs -q -dSAFER -sDEVICE=txtwrite -o - \
   -dFirstPage=7 -dLastPage=50 output/report-submission.pdf | \
egrep '[A-Za-z]{3}' | wc -w
14914

$ make wc_breakdown
========================
Total:            14914
Intro:              942
Background:        2576
Related Work:      1718
Implementation:    5429
Evaluation:        3270
Conclusion:         979
========================
```

Declaration

Abstract

Safe Pattern Generation for Multi-Stage Programming

Multi-stage programming is a metaprogramming paradigm, allowing for the generation of programs guaranteed to be well-typed and well-scoped. This generation may be used to produce optimised versions of algorithms, such as performing unrolling of a list-mapping function. This function, like many in functional programming languages, makes use of pattern-matching expressions, however, current multi-stage programming languages do not support the type-safe generation of arbitrary pattern-matching expressions.

This work introduces a domain-specific language (DSL) for the compositional description and generation of pattern-matching expressions, an associated type system to restrict generation to valid, type-safe code, and combinators for the construction of statically unknown patterns using this DSL. This work is implemented as a library built on top of the MetaOCaml extension of OCaml.

The pattern system presented provides mechanisms to generate a broad class of pattern-matching expressions, is extensible to support many other pattern types, and provides a simple, ergonomic interface for use, with concise, informative error messages. The approach taken is also sufficiently general to be applicable to alternative multi-stage programming systems.

Acknowledgements

I owe a great deal of thanks to the many people who have supported me throughout this dissertation.

I must firstly thank my supervisor, Jeremy Yallop, for his guidance throughout this project, his enlightening feedback and comments, and his general mentorship that I have been so fortunate to benefit from for the past eight months.

I also owe many thanks to my family and friends, who have provided the support which has enabled me to complete this work. Special mention is deserved for the college friends with whom I spent so many hours in silent, studious company.

Finally, I would like to thank Oleg Kiselyov, for his extensive work and documentation on MetaOCaml, which has assisted so much of my learning in this project.

Contents

Chapter 1

Introduction

1.1 Multi-stage programming for optimisation

Multi-stage programming is a form of metaprogramming, in which the compilation pipeline of a program is broken up into multiple distinct stages. Sections of program code may be stored as expressions within the source language, manipulated, and then spliced back into the program, with the type system and multi-stage compilation process ensuring that this programmatically generated code is well-typed and well-staged. Such code generation has a wide array of applications, from automated boiler-plate generation to code injection for testing and profiling, however, one significant area of use is in program optimisation by rewriting.

The classical example is that of computing the power function x^n, originally introduced by Ershov [1] in the context of partial evaluation. A simple recursive function, derived from the recursive definition of x^n:

$$x^n = \begin{cases} 1 & n = 0 \\ (x^{\frac{n}{2}})^2 & n \geq 1 \text{ and } n \text{ is even} \\ x * x^{n-1} & n \geq 1 \text{ and } n \text{ is odd} \end{cases}$$

is given in OCaml by:

```
let rec power (n : int) (x : int) : int =
    if n = 0 then 1
    else if n mod 2 = 0 then
        let sqr = power (n / 2) x in sqr * sqr
    else x * power (n - 1) x
```

However, for a given value of `n`, the performance of this computation can be significantly improved with multi-stage programming [2]. It is possible to write a function that will generate a power function specialised to a particular n. For example, for $n = 5$, the following function could be generated:

```
let power5 (x : int) = x * (let y = x * x in y * y)
```

1.2 List-unrolling optimisation

Generating optimised versions of programs with metaprogramming can provide significant performance gains in a safe manner [3, 4]. The power example is, however, a rather simple example. In functional languages such as OCaml, Haskell or Scala, pattern matching is an integral part of many algorithms, with one such example being the higher-order `map` function:

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

Such a map function can be optimised by unrolling [5], in which the target list is processed in chunks for more efficient cache usage. For example, unrolling to chunks of 4 list elements gives:

```
let rec map (f : 'a -> 'b): 'a list -> 'b list = function
    | [] -> []
    | x1 :: x2 :: x3 :: x4 :: xs ->
        let y1 = f x1 in
        let y2 = f x2 in
        let y3 = f x3 in
        let y4 = f x4 in
        y1 :: y2 :: y3 :: y4 :: map f xs
    | x :: xs -> let y = f x in y :: map f xs
```

Due to the different situations a map function may be used in, it would be ideal to allow such an unrolling to be generated for an arbitrary integer n, to enable an optimal unrolling size to be selected. It is here however that a significant problem exists; to the author's knowledge, no existing multi-stage programming language provides type-safe generation of arbitrary patterns, including popular examples such as Template Haskell [6] and BER MetaOCaml [7, 8, 9]. This prevents the generation of statically unknown patterns such as those required for list unrolling.

1.3 Contributions

The aim of this project was therefore to extend an existing metaprogramming system, to enable the type-safe generation of statically unknown patterns. In order to achieve this, firstly, a deeply embedded domain-specific language (DSL) for the compositional description of patterns was designed and implemented, with an associated type system to restrict pattern-matching expression construction to valid and compatible patterns. The type system ensures that any source code generated from this DSL is valid and well-typed. In addition, a family of combinators was defined, with which patterns such as the n-ary list conses, detailed above, can be constructed from run-time values.

As a preview of the results achievable with this typed pattern generation, it now becomes possible to generate the `map` function above. The generator code:

```
let gen_unrolled_map (n : int) = .<let rec map f = .~(function_ [
    []          => .<[]>.;
    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>.
```

yields the following generated code for $n = 4$:

```
let gen : (('a -> 'b) -> 'a list -> 'b list) code =
    .<let rec map_3 f_4 = function
        | [] -> []
        | r0::r1::r2::r3::r4 ->
            let y_16 = f_4 r0 in y_16 ::
        (let y_15 = f_4 r1 in y_15 ::
        (let y_14 = f_4 r2 in y_14 ::
        (let y_13 = f_4 r3 in y_13 :: (map_3 f_4 r4))))
        | r0::r1 -> let y_7 = f_4 r0 in y_7 :: (map_3 f_4 r1)
    in map_3>.
```

which closely resembles the hand-unrolled example given above, differing only on the scope of the let bindings and the variable names, appended with numbers to avoid name collisions.

The remainder of this dissertation is structured as follows. Chapter 2, presents the requisite background material for this dissertation, with metaprogramming, multi-stage programming, the MetaOCaml system extended in this work, and pattern-matching semantics covered. Chapter 3 surveys and evaluates existing approaches to pattern generation, and reviews related work in first-class function representations and module generation. Chapter 4 and Chapter 5 present and discuss the design and implementation of the pattern DSL, the associated type system and the pattern combinators. Chapter 6 evaluates the expressivity, safety and usability of the pattern generation system. Finally, Chapter 7 summarises this dissertation and considers possible extensions and future work.

Chapter 2

Background

This chapter provides a general introduction to the broad themes of this dissertation. The field of metaprogramming, and specifically the multi-stage programming (MSP) variety, is outlined (Section 2.1), with the various flavours and styles of MSP covered. The characteristics and syntax of the MetaOCaml extension for OCaml, the host system for this work, are presented (Section 2.2), and the power function generation example, described in Chapter 1, is elaborated in full depth, with discussion on implementation styles. Finally, a refresher on the semantics and typing rules of pattern-matching expressions in OCaml is also included (Section 2.3).

2.1 Metaprogramming and multi-stage programming

2.1.1 Metaprogramming

Metaprogramming, at the most fundamental level, is the programming paradigm in which fragments of code can be treated as data within the source language. The construction, manipulation and eventual use of these program fragments provides quite profound control to the programmer. In the simplest example, this could be achieved by the representation of program fragments as strings, with the provision of some interpreting function to run these strings of code. Given two program fragments, `"2"` and `"+"`, a new fragment could be produced by concatenation, `"2 + 2"`. Evaluating this fragment with an interpreting function, `run`, would yield `4`. With such a simplistic system, problems become immediately apparent. The program `"2 + +"` can be generated, which is not syntactically valid.

An improved version of metaprogramming could use data types in the host language to represent code. For example, in OCaml:

```
type code = Int of int | Add of code * code
```

This ensures that only syntactically valid programs may be stated. The first example is given by `Add (Int 2, Int 2)`, while `"2 + +"` is unrepresentable. This use of data types cannot however ensure that programs are well-typed [10]. As an example, if the data type was extended with a `Char of char` variant, then the program `Add (Int 2, Char 'a')` would be a syntactically valid, but ill-typed fragment. As a further issue, care must be taken to avoid variable name collisions or accidental capture of variables in generated programs, as this responsibility is left to the programmer. A variant of metaprogramming, multi-stage programming, provides a framework within which generated programs can be guaranteed to be well-typed and free of accidental captures.

2.1.2 Multi-stage Programming

In multi-stage programming, two key annotations are introduced to the base language:

- **Quotation** - for some expression of the base language `e`, a quotation of `e`, denoted `.<e>.`, represents a code fragment of the expression.
- **Splice** - for some code fragment `c`, a splice of `c`, denoted `.~c`, represents the underlying expression represented within the code fragment.

A multi-stage program divides the evaluation of a program into numerous stages, which may be denoted by integers. The 0 stage represents evaluation at run-time, with earlier stages representing compile-time evaluation and later stages representing presently unevaluated stages. An expression is evaluated at a particular stage, given by the number of quotations it is within scope of, minus the number of splices it is under.

There are two main forms of multi-stage programming, namely untyped and typed MSP. These two forms are outlined below, considering code fragments of expressions.

Untyped multi-stage programming

In both untyped and typed MSP, quoted code fragments are represented by some `code` type. The untyped approach, taken in the implementation of Template Haskell [6], represents the quotation `.<e>.`, for some expression `e` and type `a` with `e : a`, with the type `exp code`. As this quotation type does not reflect the type of the original expression, it cannot guarantee type safety. For example, the following are both quoted expressions in OCaml:

```
let x : exp code = .<2>.
let f : exp code = .<fun x -> x>.
```

However only the first of the two applications given below is well-typed:

```
let valid : exp code = .<.~f .~x>.
let invalid : exp code = .<.~x .~x>.
```

With the untyped approach, an error for the second example would only be raised at the time at which the expressions are spliced, which would lead to a run-time exception. There are, however, advantages to this approach; the type of function generated needs not be known in advance of generation, which allows for greater flexibility in terms of program generation.

Typed multi-stage programming

An alternative approach to typing multi-stage programming is that of typed MSP, which is found in MetaOCaml [7, 8, 9] and Typed Template Haskell [11, 12]. Now for an expression `e : a`, the quotation `.<e>.` has type `a code`. Returning to the example given above, this typing now disallows the erroneous case. The attempt to apply `.~f : a -> a` to `.~x : int` succeeds, however applying `int` to `int` yields a type error at compile-time. From this typing, stronger guarantees about the type safety of the generated code may be recovered, however, this comes at the cost of the aforementioned flexibility.

Run-time and compile-time multi-stage programming

Another significant design choice for an MSP system is that of run-time versus compile-time code generation, which refers to the stages at which code may be generated. In a compile-time metaprogramming language, such as Template Haskell, code may only be generated during compilation. This contrasts with a system such as MetaOCaml, in which code may be generated at run-time, possibly with statically unknown values as input, and then incorporated into the already running program. This can also be used to emulate compile-time generation, by writing generated code to files and compiling them as part of compilation.

For this dissertation, the work will build upon the BER MetaOCaml extension for OCaml, which exclusively supports run-time code generation. As a result, it is this variant of MSP which will be predominantly considered.

Scope extrusion

One issue that must also be considered for the guaranteed safety of code generated with multi-stage programming is that of scope extrusion. Scope extrusion occurs when a variable in a code quotation escapes the scope of its binding [13]. This usually occurs when multi-stage programming is used with side effects, such as mutability. An example of this is the following OCaml program:

```
let r = ref .<0>.
let _ = .<fun r0 -> .~(r := .<r0>.; .<0>.)>.
let extrusion = .<fun _ -> .~(!r)>.
```

In this example, a mutable reference to a code quotation is stored as `r`. An effectful operation is then performed, creating a new bound variable `r0`, and storing a quotation of this variable in the reference. The extrusion then occurs when this reference is accessed, and spliced into the code, outside of the binding of `r0`. The variable now appears in a quotation without a binding, and attempting to run this code will result in an unbound variable exception.

Preventing scope extrusion is an essential component of ensuring the safety of generated code. At best, it can result in unbound variable run-time exceptions, as seen in the above example. More insidiously, the escaped variable may be accidentally captured by another binding, leading to generated code with incorrect semantics [7].

2.2 The MetaOCaml library

2.2.1 Syntax and typing

As stated above, the contributions of this dissertation will extend the BER MetaOCaml N114 variant of the OCaml compiler, which provides typed, run-time multi-stage programming for OCaml 4.14.1. This extension provides many of the features detailed above:

- Expressions such as `e : a` may be quoted to give new code expressions of `.<e>. : a code`
- Code expressions may be spliced back into the program with the splice of `c : a code` giving `.~c : a`

In addition to these previously covered features, MetaOCaml also provides a number of `run` functions, to allow for the compilation and execution of generated code. These differing functions correspond to variations of compilation, such as native or bytecode compilation. The implementation of this work uses native code compilation, and so will make use of the `Runnative` library, which provides the following function for execution:

```
Runnative.run : 'a code -> 'a
```

As MetaOCaml provides only run-time code generation, the restrictions of OCaml's type system place some relevant restrictions on the types of code that may be generated. For example, for an `'a code` generating function taking some dynamic value, such as an integer, the resulting `'a` could not be dependent on the inputted value; this would be an example of general dependent typing, as the type would be dependent on a value. This restriction is not however fatal. Through constructions such as existential type wrappers, these dependent types may be shielded until a uniform result type can be reached. This restriction, and the tricks required to safely bypass it, represent a core part of this dissertation.

2.2.2 Power example revisited

With this description of the syntax and characteristics of MetaOCaml, it is now possible to revisit the power function example presented in Chapter 1. The original function had type `power : int -> int -> int`, however in order to produce a staged version of this function, specialised to a particular value of `n`, this type signature must be modified. It is valuable at this point to consider two possible approaches to generating a function, termed the "open code" style and the "closed code" style.

When generating the specialised function, which should be a function taking an `x : int`, and raising it to the `n`th power, perhaps the most natural type signature would be an `(int -> int) code`, the so-called "closed" style. It is in fact possible to proceed with this approach. Each case of the original recursive algorithm may be replaced by a quote of a function taking an `int`, and ultimately producing an `int`, as demonstrated below.

```
let rec cpow (n : int) : (int -> int) code =
    if n = 0 then .<fun _ -> 1>.
    else if n mod 2 = 0 then
                  .<fun x -> let y = .~(cpow (n / 2)) x in y * y>.
    else          .<fun x -> x * .~(cpow (n - 1)) x>.
```

Generating this code, specialised to $n = 5$, and splicing it back into the program shows that this generator does in fact yield the desired function:

```
let power5 = Runnative.run (cpow 5);;
let result = power5 2 (* 32 *)
```

However, as a result of using this "closed" code style, the generated code is not particularly elegant. Applying a function to a value within a quotation yields an unreduced function application:

```
let f : (int -> int) code = .<fun x -> x>.
let y : int = 2
let app : int code = .<.~f y>. (* .<(fun x_9 -> x_9) 2>. *)
```

As a result, the generated code for the above example is littered with these redexes:

```
.<fun x_1 -> x_1 * ((fun x_2 ->
    let y_6 = (fun x_3 ->
        let y_5 = (fun x_4 -> x_4 * ((fun _ -> 1) x_4)) x_3 in
        y_5 * y_5) x_2 in
    y_6 * y_6) x_1)>.
```

While it would be reasonable to expect the OCaml compiler to reduce most, if not all, of these redexes, this is not guaranteed. The generated code is also difficult for the programmer to read and debug. An alternative approach which may instead be used is the “open” code style. Instead of generating an `(int -> int) code`, the specialised function generator will instead be of type `int -> int code -> int code`. The key difference here is that the `x` parameter is bound outside of the quotation, hence the application occurs outside of the quotation which prevents it from appearing in the generated coach. Applying this technique to the power function generator gives:

```
let rec opow (n : int) (x : int code) : int code =
    if n = 0 then                .<1>.
    else if n mod 2 = 0 then .<let y = .~(opow (n / 2) x) in y * y>.
    else                         .<.~x * .~(opow (n - 1) x)>.
```

To retrieve the desired `(int -> int) code` from this generator, a quotation is constructed that takes an integer input and applies the `int code -> int code` function, then retrieves the integer result, inserting quotations and splices as required. This generated program also gives the desired semantics:

```
let power5_code = .<fun x -> .~(opow 5 .<x>.)>.;;
let power5 = Runnative.run power5_code;;
let result = power5 2 (* 32 *)
```

Due to the function application occurring outside of the quotations, the generated code is now significantly more concise and readable:

```
.<fun x_1 ->
    x_1 * (let y_3 = let y_2 = x_1 * 1 in y_2 * y_2 in y_3 * y_3)>.
```

As a final step to achieve the desired output as given in Chapter 1, more concise cases for $n = 1$ and $n = 2$ can be provided in the generator, as:

```
if n = 1 then x
if n = 2 then .<.~x * .~x>.
```

This yields a generated code snippet for $n = 5$ of:

```
.<fun x_1 -> x_1 * (let y_2 = x_1 * x_1 in y_2 * y_2)>.
```

as desired.

2.3 Pattern matching

Pattern matching is a programming language feature whereby some input value is checked against a selection of patterns - structural descriptions of an expression. Where a value matches a pattern, the value is often deconstructed into its constituent parts, with which execution may proceed.

In OCaml, two constructions in which pattern matching may be used are `match` expressions and `function` expressions. Each of these expression types has a list of `pattern ->`

expr, where pattern is a pattern constructed from the primitives detailed below, and expr is some expression. Particularly relevant to this work is the requirement that both all patterns, as well as all expressions, must have compatible types. For example, it is a type error to have a function expression with differing return types:

```
let f = function
    | x -> 1
    | y -> true
```

```
Error: This expression has type bool but an expression was expected of
type int
```

In addition, it is a type error to have patterns matching different types, such as a constant int and a constant bool:

```
let f = function
    | 1    -> 0
    | true  -> 0
```

```
Error: This pattern matches values of type bool but a pattern was
expected which matches values of type int
```

OCaml provides a rich class of pattern constructs, however three broad classes are most relevant for the purposes of this dissertation.

Constant patterns

Constant patterns pattern match against some particular value. For example, the pattern 5 is matched exactly when the value compared against it is the int value 5. Such a pattern does not bind any value, and the expression associated with it is evaluated alone.

Variable patterns

Variable patterns take the form of a variable identifier such as x, and match against any value. When proceeding with the evaluation of the associated expression, the matched value is bound to the variable identifier and becomes available during the evaluation. Of importance later is the linearity of variable patterns; a variable may not be bound and rebound in a pattern, as in:

```
let f = function
    | x, x -> true
    | x, y -> false
```

```
Error: Variable x is bound several times in this matching
```

Compound patterns

Finally, compound patterns are any construction which combines existing patterns to produce a new pattern, with examples being tuple patterns, or the cons pattern demonstrated below.

```
let f = function
    | []      -> 0
    | x :: xs -> 1
```

If variable patterns form part of the constituent patterns, the variables in each are bound to the matched value within the expression. As above, there cannot be collisions of variable identifiers.

Chapter 3

Related work

In this chapter, the current state of pattern generation in existing multi-stage programming systems is explored. A list summation function is introduced as a minimal example for statically unknown pattern generation, and approaches to generating this function are covered. The untyped MSP case is considered first, with a demonstration of the expressivity of such a system in generating patterns, however without any guarantee of safety (Section 3.1). The pattern generation system of MetaOCaml is presented (Section 3.2), offering safety, but restricting generation to statically known patterns. The wider field of representing patterns as first-class language constructs is surveyed, and the topic's relevance to pattern generation is discussed (Section 3.3). Finally, the related topic of staged generation of non-expression constructs is considered (Section 3.4).

3.1 The untyped case

The concept of arbitrary pattern generation is not entirely unexplored; mechanisms for their generation in an untyped manner have featured in several metaprogramming systems, notably including (untyped) Template Haskell [6]. In order to provide a minimal example for a generated function with dynamic patterns, the `sum_n` function may be introduced. This function, of type `int list -> int`, produces the summation of a list of integers of length exactly `n` and raises a run-time exception for any other input. In Haskell, the function `sum_3` may be given as:

```haskell
sum_3 :: List Int -> Int
sum_3 [x1, x2, x3] = x1 + x2 + x3
sum_3 _            = error "Incorrect list length"
```

Defining a generator for such a function is indeed possible in Template Haskell. The two pattern match clauses can be built up individually, with the `listP` combinator used to generate an n-ary list pattern, and an appropriate expression of summations built by concatenating a list of variables with the `+` operator in a fold:

```
-- Pre: n >= 1
gen_sum_n :: Int -> Q Dec
gen_sum_n n = funD name [cl1, cl2]
  where
  name = mkName $ "sum_" ++ show n
  cl1  = do xxs@(x : xs) <- replicateM n (newName "x")
            let pattern  = listP (map varP xxs)
            let ex = foldl (\e n -> [| $e + $(varE n)|]) (varE x) xs
            clause [pattern] (normalB ex) []
  cl2  = clause [wildP] (normalB [|error "Incorrect list length"|]) []
```

This yields the generated code desired:

```
sum_3 [x_0, x_1, x_2] = (x_0 GHC.Num.+ x_1) GHC.Num.+ x_2
sum_3 _ = GHC.Err.error "Incorrect list length"
```

However, this broad expressivity comes at the cost of nearly any safety guarantees about the generated code. Should a simple mistake be made in the generator, for example passing `xs` to the `listP` combinator instead of `xxs`, the generated code could include unbound variables:

```
sum_3 [x_0, x_1] = (x_2 GHC.Num.+ x_0) GHC.Num.+ x_1
sum_3 _ = GHC.Err.error "Incorrect list length"
```

This generated code would result in an exception being raised at splice time. An ideal system would enable such expressivity while maintaining guarantees that any generated code will be well-typed.

3.2 MetaOCaml pattern generation

The MetaOCaml library does provide one mechanism for typed pattern-matching expression generation, namely the "first-class pattern matching" introduced in BER MetaOCaml N104 [14]. Taking advantage of OCaml's ability to represent pattern match clauses as function literals, MetaOCaml provides a `make_match` combinator, which combines a set of quoted function literals to form a pattern-matching expression:

```
val make_match : 'a code -> ('a -> 'r) pat_code list -> 'r code
```

Type safety is ensured by the `match` scrutinee expression, of type `'a`, matching each pattern clause input type, and the `match` expression result type, `'r`, matching each of the pattern clause return types. Additionally, the OCaml type checker is modified to include a `[@metaocaml.functionliteral]` annotation, which validates at compile-time that the pattern clause is indeed a direct function literal. This system does allow for the generation of a `sum_n` function for a specific `n`, such as $n = 3$:

```
let gen_sum_3 = .<let sum_3 l = .~(make_match .<l>. [
    .<fun [x1; x2; x3] -> x1 + x2 + x3>. [@metaocaml.functionliteral];
    .<fun _ -> raise @@ Invalid_argument "Incorrect list length">.
                                         [@metaocaml.functionliteral];
    ]) in sum_3>.
```

which generates the code:

```
.<let sum_3_7 l_3 = match l_3 with
| x1_4::x2_5::x3_6::[] -> (x1_4 + x2_5) + x3_6
| _ -> Stdlib.raise (Stdlib.Invalid_argument "Incorrect list length")
in sum_3_7>.
```

However, this approach is inherently limited in expressivity. With no type-safe method with which to build up a pattern, and therefore a function literal, it is not possible to generate arbitrary patterns at run-time. Consequently, it is not possible to define some generator which produces a `sum_n` function for a dynamic input `n`. Additionally, the `functionliteral` type annotation required on each element is both syntactically cumbersome and only serves to obfuscate the underlying semantics of the generated function.

3.3 First-class patterns

While there is little existing progress on the topic of safe, arbitrary pattern generation, a closely related topic is that of first-class pattern representations, and more generally, the ability to abstract over patterns. The first-class representation approach to pattern matching, in which the patterns themselves are first-class constructs in the language, has a relatively established history. Tullsen proposed a pattern system for Haskell with patterns represented as functions of `a -> Maybe b` and pattern matching achieved with a monadic approach [15]. Closely related is the notion of Scala's extractor objects [16], which represent pattern matching as an `unapply` method that attempts to deconstruct the input object, and returns an `Option` of the object components corresponding to the success or failure of matching.

Rhiger introduced a Haskell library for encoding patterns and pattern matching as functions [17]. This library is based on a selection of combinators, which construct functions that take as input a match scrutinee, and a function representing the right-hand side expression of a matching case. This right-hand side function accepts the variables bound in a pattern as input and returns the result of the pattern match. With some additions and extensions to these core combinators, such as for failure handling, in Rhiger's pattern system the Haskell pattern:

```
(x, (2, y))
```

may be represented by the combinators:

```
pattern = pair var (pair (cst 2) var)
```

Passing this pattern a scrutinee, such as `(1, (2, 3))`, and a function binding the pattern variables, such as `\x y -> x + y`, results in pattern matching:

```
x :: Int
x = match (1, (2, 3)) $ pattern ->> (\x y -> x + y) -- 4
```

Should the pattern match fail, a run-time exception is raised.

This approach to pattern matching has core concepts very relevant to the problem of safe pattern generation. In order to enable safe pattern generation, it must be possible for the programmer of a metaprogramming system to safely specify patterns and expressions. In order to enable greater expressivity than the limited form of generation seen in MetaOCaml, these pattern and expression representations should be compositional, to facilitate the construction of more complex, compound patterns. Rhiger's pattern combinators allow such a compositional specification, and so the problem of first-class pattern generation can be viewed as a translation from this representation to pattern representation in the host language.

This method of pattern function construction makes use of heterogeneous sequences to represent a flattened sequence of variable bindings for the pattern. This representation is in turn built from the continuation-style numerals described by Fridlender and Indrika [18] for implementing a generalised `zipWithN` function, which was inspired by Danvy's approach to a `printf` implementation in StandardML [19]. A more rigorous description of this technique is presented by Yang [20].

Rhiger's pattern combinators do however exhibit a notable downside. Their continuation style implementation results in incredibly complex types, such that even a simple mistake can yield rather verbose and difficult-to-decipher error messages. As a first example, the above pattern has the enormous type of:

```
pattern :: (  (t1 -> t2 -> t3)
           -> (t7 -> t8 -> t1)
           -> (t7, (t8, t2))
           -> t3,
              (a1, (Int, t))
           -> ((a1, (t, t9)) -> t5)
           -> (() -> t5)
           -> t9 -> t5
           )
pattern = pair var (pair (cst 2) var)
```

Then, passing a scrutinee with the incorrect bracketing associativity to the pattern defined above, such as `((1, 2), 3)`, results in an unclear error message that gives little hint as to the source of the error, instead pointing to the pattern expression:

```
errorMsg :: Int
errorMsg = match ((1, 2), 3) $ pattern ->> (\x y -> x + y)
```

```
error:
   • Couldn't match expected type ‘Int’ with actual type ‘(a0, b0)’
   • In the first argument of ‘(+)’, namely ‘x’
     In the expression: x + y
     In the second argument of ‘(->>)’, namely ‘( x y -> x + y)’
```

There does however exist a reimplementation of Rhiger's pattern combinators [21] using type families [22], offering a nearly identical interface with much more palatable types. In addition, match failure handling is implemented with the `Maybe` monad instead of Rhiger's `fail` and `catch` combinators.

Finally, a similar approach of representing patterns and pattern matching as functions is taken by Atkey et al. [23], in defining combinators to introduce pattern matching for a domain-specific language.

3.4 Generation of alternate constructs

While MetaOCaml presently only allows for the generation of expressions, there exists some work on the generation of other language constructs. In allowing the generation of mutually recursive programs, Yallop and Kiselyov approach the issue of generating lists of bindings in MetaOCaml's expression-only quotation system, both with compiler magic [24] and in plain OCaml [25].

Even more relevant to the concept of pattern generation is work on ML module generation. Inoue et al. [26] present the concept of module generation for code optimisation, introducing two extensions to MetaOCaml's staging to allow for type-safe generation. This work is built upon by Watanabe et al. [27], who proposes a language extending MetaOCaml, and a translation back to plain MetaOCaml, for module generation. This language and translation approach was subsequently refined, optimised and extended by Sato et al. [28, 29]. It is an observation made by Inoue et al., however, that is perhaps most relevant to this work. Using a first-class representation of modules as functions, generation of a restricted subset of modules may be implemented using MetaOCaml without modification. This suggests a similar approach may be possible for pattern generation, in conjunction with the first-class pattern work described in Section 3.3.

Chapter 4

A pattern generation system

In the following two chapters, the design and implementation of a system to enable the type-safe generation of statically unknown patterns at run-time is presented.

Chapter 4 introduces the core system for pattern generation. Firstly, the necessary components for such a system are identified and described (Section 4.1). A type system representation for patterns is then introduced, with the description of a domain-specific language to instantiate the pattern type (Section 4.2). A representation for clauses of a pattern-matching expression is introduced (Section 4.3), and the method for the generation of these pattern cases is described. Finally, the step of generating well-typed, well-scoped OCaml code from these first-class representations is outlined (Section 4.4), covering both the basic generation of the pattern-matching constructs, as well as techniques to prevent scope extrusion.

In Chapter 5, the pattern generation described in Chapter 4 is used for the generation of statically unknown patterns. This starts with a discussion on the restrictions on the types of patterns which can be dynamically generated within the constraints of OCaml's type system (Section 5.1). Considering these limitations, an approach to constructing pattern-matching expressions through inductive code definitions is described (Section 5.2). Several candidate implementation strategies to construct these pattern-matching expressions are covered (Section 5.3), ultimately leading to a general, type-safe mechanism. Finally, successful, type-safe generation of the `sum_n` function introduced in Section 3.1 is demonstrated (Section 5.4).

4.1 Pattern system overview

In order to facilitate the safe generation of pattern-matching expressions, several key components are required. The first of these is some type-system representation of a pattern itself. As covered in Section 2.3, there are a number of restrictions placed on the clauses in a pattern-matching expression, and so any pattern type must be expressive enough to at least allow compatibility checking of the pattern input type, and to present any variable binding information for the pattern expression.

When this pattern representation is combined with a function representing the pattern match right-hand side, the pattern clause, or "case", produced must also be represented, maintaining enough information to enable the compatibility of case input and output types to be checked. Additionally, some mechanism is required to substitute the pattern variables into the function, to produce the final right-hand side expression.

Finally, some operation to consume these abstract representations of pattern match cases, and to produce concrete OCaml code, is required. In producing this final generated code, a number of considerations must be made to ensure the type safety of the output. Care must be taken with naming the variable patterns produced, to guard against collisions with both other pattern variables and other free variables. Consideration must also be given to ensuring that any cases of scope extrusion, in which variables escape the scope of their binding in generated code, can be caught by MetaOCaml's extrusion prevention system.

4.2 Pattern representation

Given the requirements described above, patterns are represented in the manner outlined below.

4.2.1 A first-class pattern type

A pattern is represented as the following type:

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

The three type parameters, `'a`, `'f` and `'r` represent the following types:

- `'a` - The first type parameter is the type of the pattern input, for example, `int * bool` for the pattern `(2, true)`. The exposure of this type enables the patterns within a pattern match to be checked for input compatibility.
- `'f` - The second type parameter represents the function type required for generating the right-hand side of the case. The curried function type takes as input the variables bound by the pattern and returns a `code` of the return type of the pattern case.
- `'r` - The final parameter is a `code` of the return type of the pattern case in which the pattern features. This type is technically independent of the pattern - for example, a case with pattern `p` could return either an `int` or a `char`.

As noted, the return type of the case, `'r`, or equivalently the return type of the function of type `'f`, is not actually a property of the pattern itself. The type is however included to enable the safe construction of compound patterns, as described below. This inclusion does introduce some restrictions, however, due to OCaml's let-polymorphism [30]. For example, the same let-bound pattern cannot be reused in cases with different return types, as the `'r` type is fixed by the first usage.

Given below are several examples of native OCaml pattern match cases, and the corresponding type the representation of their patterns would be assigned:

OCaml Case	pat type
`| 2 -> 'a'`	`(int, char code, char code) pat`
`| (2, true) -> false`	`(int * bool, bool code, bool code) pat`
`| x -> 1`	`('a, 'a code -> int code, int code) pat`
`| (x, y) -> 0`	`('a * 'b, 'a code -> 'b code -> int code, int code) pat`

4.2.2 Pattern combinator DSL

In order to instantiate values of the pattern type, a deeply embedded DSL for pattern definition can be specified. To do this in OCaml, the pattern type can be made into a GADT. This approach allows for the DSL to be type-checked by OCaml's type checker itself, although prevents the representation of some pattern types, as discussed in Section 7.2. The DSL is implemented by the following:

```
type (_, _, _) pat =
  Any : ('a, 'r, 'r) pat
| Int : int -> (int, 'r, 'r) pat
| Var : ('a, 'a code -> 'r, 'r) pat
| EmptyList : ('a list, 'r, 'r) pat
| Pair : ('a, 'k, 'j) pat * ('b, 'j, 'r) pat -> ('a * 'b, 'k, 'r) pat
| Cons : ('a, 'k, 'j) pat * ('a list, 'j, 'r) pat
      -> ('a list, 'k, 'r) pat
```

The terms of this pattern DSL can be broadly categorised into the three classes of pattern identified in Section 2.3.

Constant patterns

The `Any`, `Int` and `EmptyList` patterns, representing OCaml's wildcard, `int` constant and `[]` constant patterns respectively, are all examples of constant patterns. The input type reflects the semantics of the pattern; any input for a wildcard, an `int` for the `int` constant or a list type for the empty list constant. As constant patterns do not introduce new bindings into the case's right-hand side, the expected function type is precisely that of the return type. The GADT constructor only differs for the `int` constant, as the specific `int` being matched for is provided as input.

Variable patterns

Following the semantics of a variable pattern, the input type for a `Var` pattern is unconstrained, as is, naturally, the return type of whatever right-hand side it is combined with. The most important component of the variable pattern is the prepended binding of an `'a code` argument to the required right-hand side function.

This type may seem counter-intuitive, as the value being bound to the variable is of type `'a`, not `'a code`. This is in fact a manifestation of the "open" code generation style, discussed in Section 2.2. Instead of requiring an input function of type, say, `('a -> 'b -> 'r) code`, to express the body of the pattern expression, the requirement for a function of type `'a code -> 'b code -> 'r code` is built up. This prevents the generation of redexes in the final generated code.

Compound patterns

Finally, `Pair` and `Cons` are examples of compound patterns, representing tuple patterns and list patterns respectively. Both `Pair` and `Cons` combine two patterns to produce a new pattern.

`Pair` is the more general of the two constructs. The argument patterns are entirely unconstrained, with any input, output or binding types allowed for either. The resulting pattern has an input type of the product of the two argument input types, `'a * 'b`. To sequence the binding functions, the result type of the first argument function, after application of the `'a code`s, must match the type of the second argument function. This occurs precisely when the return type of the first binding function is the same as the second binding function type. This requirement is encoded in the GADT as shown. Finally, the return type of the produced pattern is now the return type of the second binding function.

The `Cons` combinator is nearly identical to `Pair`, with one crucial restriction. The right-hand side of a cons operation must be of type `'a list` if the left-hand side is of type `'a`, and the result of this operation is an `'a list`. This constraint is therefore encoded in the type.

4.3 Case representation and generation

With a typed representation of patterns and a language with which to construct them, it now becomes necessary to define a single case of a pattern match, consisting of a pattern, and a right-hand side expression with any pattern variables bound. To achieve this, a case is represented by the following type:

```
type ('a, 'r) case = Parsetree.pattern * 'r code
```

In this representation, the `'a` and `'r` type parameters serve exactly the same purpose as in the pattern representation, namely representing the input and output type of the case. This input type is no longer reflected in the underlying pattern representation, however, as it is represented by the untyped primitive used in the OCaml Parsetree. The expression body is represented as a code fragment of the return type, with any pattern-bound variables replaced with the variable identifiers from the pattern.

In order to produce a case from a pattern and a function representing the body, the `(=>)` operation is introduced:

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

As shown by the type signature, if a function handling the appropriate bindings, as called for by the pattern's `'f` type, is provided, a case with input and output types identical to those of the pattern can be produced. This case creation involves two primary steps; pattern generation and code expression generation.

4.3.1 OCaml pattern generation

Actual construction of the untyped pattern representation is relatively trivial with the availability of the `Ast_helper` library, which provides methods to construct Parsetree representations of native OCaml patterns. As a result, pattern construction becomes a simple exercise in recursively traversing the DSL tree that has been built up to produce a representative pattern. The only area of this generation that requires significant care is the selection of pattern variable names, to prevent name collisions or unintentional scope inclusion.

In order to achieve this, a global counter is defined:

```
let patvar_count : int ref = ref 0
```

Then to name a variable, the counter is incremented and its value is appended to a common prefix, `r`. This produces variable names of `r0`, `r1`, `r2` ... `rn`. The global nature of this counter, as opposed to a more functional approach, prevents collisions between different case generations, allowing for the safe nesting of generated pattern-matching expressions. This approach to collision-free naming is not completely infallible, however, as it may be possible for an `rn` variable to occur free in the expression body already. This issue, possible solutions, and the reasoning for their absence from this implementation are discussed in Section 6.3.

4.3.2 Pattern expression generation

As described in Section 4.2, with "open"-style generation, each variable in a pattern prepends another input of type `'a code`, for some `'a`, to the binding function. Therefore, the function `f` of type `'f` provided to `(=>)` has a type of the form:

```
f : 'a code -> 'b code -> 'c code -> ... -> 'r code
```

As each of these `code` arguments required is just the `code` representation of the variable identifier bound in the pattern, and as the `'r code` returned by `f` is precisely the code expression required to construct the case, generation appears as simple as applying `f` to the n variable identifiers. However, defining such an `apply` function for arbitrary n would result in a function with a type dependent on the value n. Instead, to safely type this application, the type-level encoding of n, contained within the type of the pattern, must be used. By using the concept of continuation numerals [18, 17], a function of the type `'f -> 'r code` can be constructed in a recursive traversal of the DSL tree, as follows.

For any non-binding pattern, such as the constant patterns, the `'f -> 'r code` function is simply the identity, as `'f` is `'r code`. For variable patterns, the `one` numeral may be used, which takes an input and a continuation, and calls the continuation with this input:

```
let one (v : 'v) : ('v -> 'k) -> 'k = fun k -> k v
```

By partially applying the `one` combinator to the variable identifier, an `('a code -> 'r code) -> 'r code` function is obtained. This type matches the `'f -> 'r code` type required, given that for a variable pattern, `'f` is `'a code -> 'r code`.

Finally, for a compound pattern, the `'f -> 'r code` function is produced by composing the constructed functions for each of the constituent patterns. For a compound pattern of type `(_, 'i, 'k) pat`, the constituent patterns are of type `(_, 'i, 'j) pat` and `(_, 'j, 'k) pat`, for some `'j`. The required function, of type `'i -> 'k`, is produced by composing the constructed functions `'i -> 'j` and `'j -> 'k`.

Implementing this function construction recursively gives:

```
let nil : 'i -> 'i = fun k -> k
let one (v : 'v) : ('v -> 'k) -> 'k = fun k -> k v
let comp (p : 'i -> 'j) (q : 'j -> 'k) : 'i -> 'k = fun k -> q (p k)

let rec mk_exp_gen : type a f r . (a, f, r) pat -> (f -> r) = function
    | Any          -> nil
    | EmptyList    -> nil
    | Int _        -> nil
    | Var          -> incr patvar_count;
                      let name = "r" ^ string_of_int !patvar_count in
                      one (mk_expr_ident name)
    | Pair(l, r)   -> let lf = mk_exp_gen l in
                      let rf = mk_exp_gen r in comp lf rf
    | Cons(x, xs) -> let lf = mk_exp_gen x in
                      let rf = mk_exp_gen xs in comp lf rf
```

The required `'r code` can then be retrieved by applying this constructed `'f -> 'r code` function to the `f : 'f` passed to `(=>)`. The `case` can then be constructed as desired.

4.4 Code generation

With a first-class representation of pattern-matching cases achieved, the final step in pattern generation is to produce valid OCaml code. This process has two core components; the basic generation of the pattern constructs themselves from the case representations, and the implementation of methods for detecting scope extrusion.

4.4.1 Basic generation

In order to construct the concrete `code` for a desired pattern-matching expression, two code generation combinators are introduced, exploiting the similarities between `function` and `match` expressions:

```
val function_ : ('a, 'r) case list -> ('a -> 'r) code

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

Each takes a list of cases, while in the case of the `match` expression generator, the scrutinee of the match must also be provided. The `function` expression generator naturally produces the quotation of a function, while the `match` expression produced is simply an expression of the output type.

In order to retrieve the primitive code representation, and free and bound variable records for the code quotation passed to these generators, the underlying representation of an `'a code` in MetaOCaml must be accessed. The low-level representation of an `'a code` is given by:

```
type +'a code = private code_repr
type code_repr = Code of flvars * Parsetree.expression
```

Therefore a `code_repr` can be retrieved from the `_ code` arguments through the subtyping operator, `(:>)`. It is at this point however that a design decision for this library poses a barrier. As this work was completed as a library on top of MetaOCaml, and as MetaOCaml's interface does not expose the internal representation of a `code_repr`, it appears impossible to deconstruct this type. As a workaround, an identical type to `code_repr` is defined in this library, and functions for the coercion between these two types are defined. This workaround will cease to be required after integration of this work into MetaOCaml however, as discussed in Section 6.3.

With access to this underlying representation, the Parsetree representation of each code block can be accessed. The underlying pattern clause representation, `Parsetree.case`, can be constructed from the pattern stored in the `case` and the expression from the `code` representation. Once again using the `Ast_helper` library, the desired `match` and `function` expressions can be constructed from these `Parsetree.case`s.

4.4.2 Scope extrusion

In order to ensure the safety of generated code, MetaOCaml prevents scope extrusions through a system of tracking free variables and virtual bindings in code quotations and raising run-time exceptions where scope extrusion is detected [7]. To facilitate this tracking, prior to any code generation, any variables appearing in the generated code must be validated, to ensure that they have not escaped their binding. This can be achieved by the internal `validate_vars` family of functions provided by MetaOCaml. In particular, `validate_vars_list` takes as input a list of `code_repr`s and returns a list of the underlying `Parsetree.expression`s and a single combined representation of the free variables and virtual bindings, of type `flvars`. The final `code_repr` can be produced by the combination of the validated `flvars` and the constructed `Parsetree.expression`, then coerced to the `'r code` required for the return type.

The same issue faced with the underlying `code_repr` type is also encountered here; the required `validate_*` functions of MetaOCaml are not exposed in the interface. Once again, this can be worked around; as the functions are pure, their definitions can be copied into the pattern library and used equivalently, however, this approach cannot be used where functions have internal state. This issue will again cease to exist upon integration into MetaOCaml.

Chapter 5

Statically unknown pattern generation

The combination of the pattern type system, DSL, case construction and code generation, described in Chapter 4, provides a foundation with which a wide-ranging class of patterns can be safely expressed and generated. However, this system provides only equivalent expressivity to the existing MetaOCaml implementation of pattern generation [14]. To extend beyond the existing limitations, it would be useful to allow for the generation of statically unknown patterns, such as the n-ary cons patterns required for the optimisations described in Section 1.2. This chapter covers the possible patterns that can be safely generated (Section 5.1), the approaches (Section 5.2) and techniques (Section 5.3) required for their generation, and the applications of these patterns (Section 5.4).

5.1 The dependent type restriction

The class of pattern-matching expressions containing statically unknown patterns that can be safely generated is inherently limited by the restrictions imposed by OCaml's type system. This restriction arises as OCaml's type system does not generally support dependent types; where types within a program are dependent on a program value. Although generalised algebraic data types (GADTs) can offer apparently similar behaviour, they only support types indexed by other types, and not the value-indexed types of true dependent typing. In the context of pattern generation, this limitation has a profound impact. For a code-generating function taking an input, the type of the generated code cannot be indexed by the input value, although may be indexed by the input type.

An example of a function that this restriction prevents generation of is a non-recursive function to sum n-nested, right-associated tuples. As an example, for $n = 3$, such a function could be defined as:

```
let tup_sum_3 = function
    | (x1, (x2, x3)) -> x1 + x2 + x3
    | _              -> raise @@ Invalid_argument "Invalid input tuple"
```

However, for this family of sum functions, the type signature of a function contains the nested tuple type accepted as input:

n	`tup_sum_n` type
1	`int -> int`
2	`int * int -> int`
3	`int * (int * int) -> int`
4	`int * (int * (int * int)) -> int`

As a result, the type of a generator of such a family of functions, taking an input `n : int`, would be `int -> (<nested tuple type> -> int) code`. This generator type would therefore be indexed by the value of the input `int`, and hence is not representable within the OCaml type system.

This restriction does not however spell the end for generating any patterns depending on a value. A possible approach is to simultaneously generate a pattern and the desired input for a pattern, then combine them with the `match_` code generation combinator from Section 4.4. The type of `match_`:

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

ensures that the `'a` type, which depends on a value, does not reach the final type of the generated code, or generator, avoiding violation of the restriction.

Not all compound patterns are as restrictive as the tuple pattern, however. Considering the `sum_n` function introduced in Section 3.1, a promising pattern emerges:

n	`sum_n` type
0	`int`
1	`int -> int`
2	`int list -> int`
3	`int list -> int`

For $n \geq 2$, the type of the `sum_n` function is constant, and so a generator for this function could conceivably be constructed, without violating the dependent type restriction. Challenges remain however in precisely how to define such a generator. While the desired pattern can be constructed using the previously-defined pattern DSL, with `Var` and Cons, also required is a code expression accepting the bindings introduced by such a pattern. For a strategy to generate such an expression, the concept of an inductive code definition can be introduced.

5.2 Inductive code definitions

As described in Section 4.3, for some pattern of type `('a, 'f, 'r) pat`, in order to produce an `('a, 'r) case` from this pattern, a function of type `'f` must be provided. For `sum_n`, while the type parameters of the `case` produced are constant for $n \geq 2$, as `'a` and `'r` are constant, the `'f` type for each specialised function will vary, due to the prepended binding argument for each variable pattern:

n	`'f` **type**
0	`int code`
1	`int code -> int code`
2	`int code -> int code -> int code`
3	`int code -> int code -> int code -> int code`

Therefore, to construct a suitable expression to be associated with each of the generated patterns, an approach to generating functions of this form is required. Considering the pattern in the table above, one possible method would be construction by an inductive definition. By providing a base case, for $n = 0$, of type `int code`, and an inductive case, combining an existing result of type `int code` and a newly bound variable of type `int code`, an appropriate function could be generated for any value of n.

Generalising this approach further, the base case must give a base value of type `'r code`, to provide a result of the correct type when $n = 0$. The inductive case is a function of type `'a code -> 'r code -> 'r code`, as a newly bound variable on the left-hand side of a cons pattern is of type `'a`, and the correct return value must be preserved.

Applying this approach to the `sum_n` example, the right-hand generated must be of the form `v1 + v2 + v3 + ...`. As such a summation can be defined recursively, this approach may be used to generate the desired function:

```
let base : int code = .<0>.
let ind (v : int code) (acc : int code) : int code = .<.~v + .~acc>.
```

Applying these cases manually produces a function of the desired type, while substituting in values to the function yields the desired summation code:

```
let sum_binds = fun x y z -> ind x (ind y (ind z base))
let sum_code = sum_binds .<1>. .<2>. .<3>. (* .<1 + (2 + (3 + 0))>. *)
```

While expressing pattern-matching bodies in this way does somewhat obscure their semantics, this technique provides a method with which a broad class of expressions may be generated. Parallels can be drawn with the higher-order `fold` function found in many functional programming languages. For example, consider the type of OCaml's `List.fold_right` function:

```
val fold_right : ('q -> 'p -> 'p) -> 'q list -> 'p -> 'p
```

Taking `'q` as `'a code` and `'p` as `'r code`, the types of the base and inductive cases appear as the `fold` argument types. This hints at the correspondences between `fold_right` and inductive definitions, and the significant expressive power available with this approach [31].

5.3 Pattern-matching expression generation

With both a pattern DSL enabling recursive construction of patterns, and a technique to generate compatible expressions for these patterns, it now becomes possible to attempt the generation of statically unknown pattern-matching expressions. Given the inductive nature of the definitions provided for each component, it appears most natural to implement this generation as a recursive function.

5.3.1 Direct recursion approach

A first attempt may proceed as follows. A combinator to construct a variable pattern, `var`, a combinator to join two patterns into a list pattern, `(>::)`, and a combinator for an empty list constant pattern, `empty`, are defined. A recursive function could then return an `empty` pattern as a base case, and cons a `var` pattern onto the pattern from the recursive call in the inductive case:

```
let rec gen_pattern : int -> (int list, 'f, int code) pat = fun n ->
    if n = 0
    then empty
    else let p = gen_pattern (n - 1) in var >:: p
```

There is an inherent issue with such a function, however. As shown in Section 5.2, the type of `'f` varies depending on the value of the input `n`. The return type of `gen_pattern` is therefore dependent on a value, violating the dependent type restriction (Section 5.1). Indeed, this function does not type check; the type of `'f` is inferred as `int code` from the base case, which is incompatible with the return type of the inductive case. In order, to generate such a pattern, the `'f` type parameter of the pattern must be shielded from the top level.

5.3.2 Existential wrapper approach

A method with which the `'f` type can be shielded is the use of an existential type to wrap the pattern. This places the constraint that a type of `'f` must exist, however, the precise definition of this type is not exposed. Such a pattern wrapper may be specified as:

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

gen_pattern may then be rewritten, wrapping the pattern before returning it, and unwrapping the recursive result through pattern matching:

```
let rec gen_pattern : int -> (int, int) wrap = fun n ->
    if n = 0
    then Pat empty
    else let Pat p = gen_pattern (n - 1) in Pat (var >:: p)
```

Such an approach successfully hides the type, allowing the function to compile. However, another issue arises when attempting to produce a case from this wrapped pattern:

```
let case : (int list, int) case = match gen_pattern 2 with
    | Pat p -> p => fun x y -> .<.~x + .~y>.

Error: This expression should not be a function, the expected type is
       $Pat_'f
```

Due to the opaque nature of the existential type for 'f of the wrapper, even though the binding function provided is correct for the pattern contained within the wrapper, this cannot be deduced by the type checker. It is therefore required that the pattern and the expression are built together, and both stored within the wrapper, to ensure that their compatibility is witnessed by the wrapper. A new wrapper for this purpose may be defined:

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

Then using this wrapper, the pattern and expression can be constructed simultaneously:

```
let gen_pattern (n : int)
                (base : int code)
                (ind : int code -> int code -> int code):
                (int, int) pewrap =
    let rec loop (n : int) : (int, int) pewrap =
        if n = 0
        then Pat (empty, base)
        else let Pat (p, e) = loop (n - 1) in
             Pat (var >:: p, fun x -> ind x e)
    in loop n
```

However, despite the modification to the wrapper type, the opaqueness of the existential wrapper type once again becomes an issue. As the e from the recursive call is retrieved from the wrapper, it once again has the opaque type of $Pat_'f. It is therefore incompatible with the inductive code function. Worse still, as no information is known about this type, it is impossible to make safe use of e, preventing any sort of inductive construction.

Thankfully, there still remains an alternative strategy.

5.3.3 Continuation-style approach

In order to avoid the need to directly modify any value of the existential `'f` type, a continuation-based approach can instead be used. Instead of applying the changes to the right-hand side expression during recursion, the changes can instead be represented as continuations of type `'r code -> 'r code`, and composed together to produce a final modifying function, from which the `'f` function is produced. Once again, a wrapper may be defined to store such types:

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

Then to define the generation algorithm, in the base case, the continuation is simply taken as an argument, and applied to the base argument. For the recursive case, a continuation `c` and an `'a code` are taken as input, before the continuation from the recursive call, `k`, is applied to the composition of the inductive code generation step, partially applied to the `'a code`, and the input continuation `c`. By unwrapping through pattern matching, and applying the continuation to the identity function to retrieve the required `'f` function, the case can be constructed as desired:

```
let gen_pattern (n : int)
                (base : int code)
                (ind : int code -> int code -> int code):
                (int list, int) case =
    let rec loop (n : int) : (int, int) patwrap =
        if n = 0
        then Pat (empty, fun k -> k base)
        else let Pat (p, k) = loop (n - 1)in
             Pat (var >:: p, fun c x -> k (fun o -> c (ind x o)))
    in match loop n with
        | Pat (p, k) -> p => (k Fun.id)
```

Through this continuation-style approach, the type-safe pattern DSL generation of Chapter 4 can be used to generate statically unknown patterns, without sacrificing safety.

5.4 `sum_n` function generation

In the style of the continuation-based pattern generation, detailed in Section 5.3, two list pattern generation combinators are defined in this library:

```
val gen_exactly_n_cons : int -> 'r code
                -> ('a code -> 'r code -> 'r code) -> ('a list, 'r) case

val gen_n_cons : int -> ('a list code -> 'r code)
                -> ('a code -> 'r code -> 'r code) -> ('a list, 'r) case
```

The first is simply the `gen_pattern` function presented in Section 5.3, with a more general type signature. The latter is very similar, however produces pattern matches of the form `r0 :: r1 :: ... :: rtail` as opposed to `r0 :: r1 :: ... :: []`. The base case is thus a `var`, and the base value binds the corresponding `'a list`.

With these combinators, as well as the `__` combinator representing a wildcard pattern, the `sum_n` function, specialised to a dynamic value `n`, may finally be defined:

```
let gen_sum_n (n : int) = function_ [
    gen_exactly_n_cons n .<0>. (fun x acc -> .<.~x + .~acc>.);

    __ => let error_msg = Format.sprintf
            "Sum function only accepts a list of length %d" n in
          .<raise @@ Invalid_argument error_msg>.
  ]
```

Such a generator produces, for $n = 3$, the code:

```
.<function
    | r6::r7::r8::[] -> r6 + (r7 + (r8 + 0))
    | _ -> Stdlib.raise (Stdlib.Invalid_argument
            "Sum function only accepts a list of length 3")>.
```

Which achieves the desired semantics of the `sum_n` function:

```
let sum_3 = Runnative.run (gen_sum_n 3)
let () = print_int (sum_3 [1; 2;])

Fatal error: exception Invalid_argument('Sum function only accepts a
list of length 3')

let () = print_int (sum_3 [1; 2; 3]) (* 6 *)
let () = print_int (sum_3 [1; 2; 3; 4])

Fatal error: exception Invalid_argument('Sum function only accepts a
list of length 3')
```

It is with the demonstration of this example that the primary goal of this dissertation is achieved.

Chapter 6

Evaluation

This chapter explores the expressivity, usability and safety of the pattern generation system, outlined and implemented in Chapter 4 and Chapter 5. Firstly, Section 6.1 expands upon the class of pattern-matching expressions able to be generated, introducing recursive and nested pattern-matching expressions. Section 6.2 considers the ergonomics of the library, with particular attention given to type error message quality. Finally, Section 6.3 reviews the safety of the library. Instances in which OCaml's type system is bypassed are individually considered, and the issues of unique identifier assignment and scope extrusion avoidance are discussed.

6.1 Generation expressivity

As discussed in Chapter 5, and particularly Section 5.2, the pattern system allows a significantly larger class of pattern-matching expressions to be dynamically generated than existing, type-safe pattern generation systems. The library also provides the flexibility for two additional pattern generation techniques, namely nested and recursive pattern matching.

6.1.1 Nested pattern generation

Due to the use of a global reference for the unique naming of variable pattern identifiers, it is possible to guarantee that pattern variables will not collide with one another. This allows for a generated pattern-matching expression to be spliced in as the right-hand side expression of another pattern case. For example, the following generator may be defined:

```
let nested_example : (int -> int -> int) code = function_ [
    var => fun x -> (function_ [var => fun y -> x])
];;
```

Here, although the use of the outer pattern variable occurs in the expression of an inner pattern variable, the semantics of the program are correctly preserved in the generated code, with the two pattern variables assigned differing identifiers:

```
.<function | r6 -> (function | r7 -> r6)>.
```

In practice, programs with nested pattern-matching expressions are usually better expressed with nested patterns and guards, however preventing accidental scope inclusion is an essential component of safe-code generation.

6.1.2 Recursive pattern-matching expressions

Many of the algorithms making use of pattern matching, such as those described in Section 1.2, use recursive calls in their computation. The pattern library provides adequate expressivity such that variables bound outside the pattern case may occur free within pattern expressions, enabling the generation of such algorithms. This can be achieved by splicing the generated pattern-matching expression into a let-rec binding and using the let-bound identifier within the pattern expression. For example, the generator for a function to compute the length of a list may be defined as:

```
let gen_len : ('a list -> int) code =
    .<let rec len l = .~(match_ .<l>. [
        empty       => .< 0 >. ;
        var >:: var => fun _ xs -> .<1 + len .~xs>.
    ]) in len>.
```

Which produces the desired code of:

```
.<let rec len_11 l_12 = match l_12 with
    | [] -> 0
    | r8::r9 -> 1 + (len_11 r9)
in len_11>.
```

There exist two points of note with this generator. Firstly, as MetaOCaml at present only provides facilities for the type-safe generation of expressions, the declaration `let rec len l = <match>` cannot be generated directly. Instead, this declaration must be let bound, and an expression of this declared function must be produced. The declaration can then be made when the generated code is run, as shown:

```
let len = Runnative.run gen_len
let () = print_int (len [1; 2; 3]) (* 3 *)
```

As a second point, the length function may be more idiomatically written as a `function` expression, instead of a match with an explicit scrutinee. Generating such a function is however not possible in this case, due to OCaml's restrictions on the right-hand side of let-rec bindings; code splices are not a permitted right-hand side:

```
let gen_len : ('a list -> int) code = .<let rec len = .~(function_ [
    empty         => .< 0 >. ;
    var >:: var => fun _ xs -> .<1 + len .~xs>.
  ]) in len>.
```

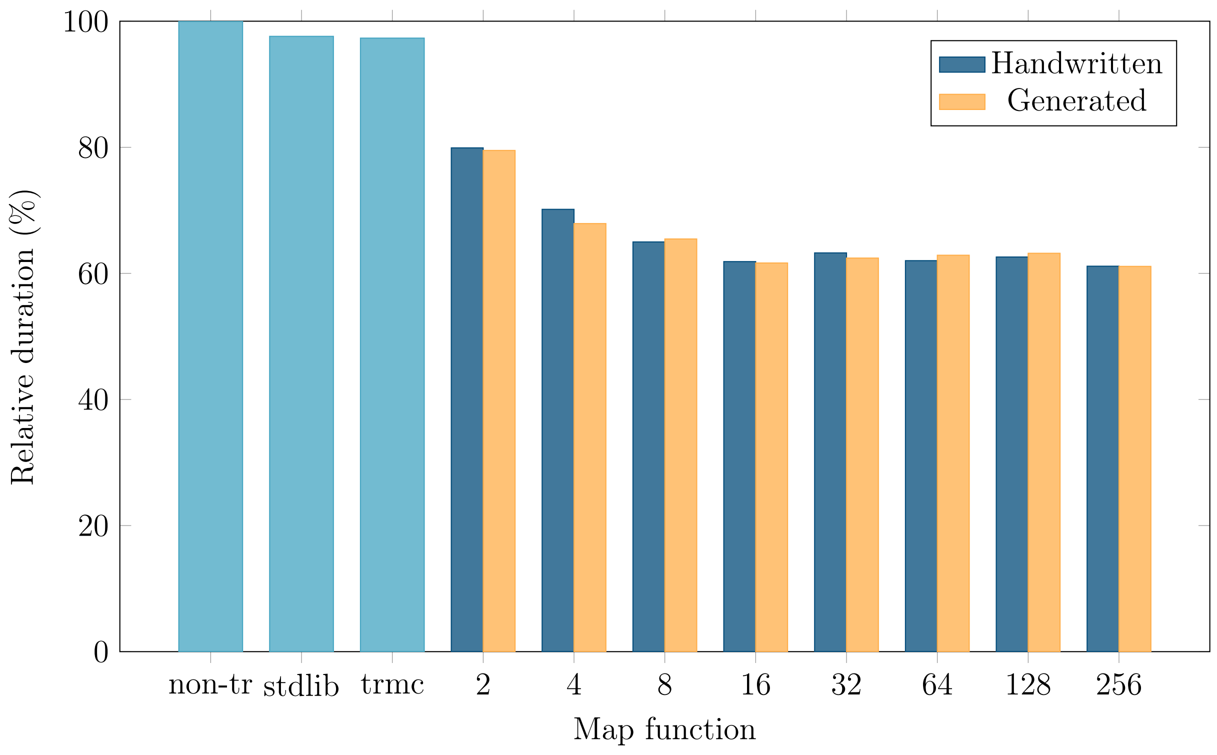
```
Error: This kind of expression is not allowed as right-hand side of
       `let rec'
```

The `match` approach succeeds, however, as the right-hand side is instead a function from the arguments of the let-rec to the code splice, which is a permitted type.

With this generation of recursive pattern-matching expressions now achievable, a generator for the motivating example of list map unrolling, introduced in Section 1.2, can now be defined with the concepts covered:

```
let gen_unrolled_map (n : int) = .<let rec map f = .~(function_ [
    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>.
```

This enables precisely the type of empirical specialisation desired. Performing a benchmark for the duration of mapping the `succ` function to a list of 100,000 integers, using a simple non-tail recursive map, the standard library `List.map`, a 2-unrolled map with tail recursion modulo cons [32], and a selection of both handwritten and generated non-tail recursive maps unrolled to powers of two yields the following results:



From this graph, two results are immediately apparent. Firstly, the performance of the generated and handwritten unrolled maps is nearly identical, which is to be expected given their near-identical syntax. Secondly, it is clear that unrolling provides a performance improvement, however plateaus at around a 16-cell unrolling on an Apple M1 Max CPU.

6.2 Library ergonomics

When considering the ergonomics of the produced library, the final result is very promising. In order to minimise the syntactic burden of specialising an existing function, it is preferable that a staged version of a function should be as similar to a standard implementation as possible. Below, three versions of the same summation function on a list of length 3 are considered; a plain OCaml implementation, a staged generator with MetaOCaml's pattern generation, and a staged generator with this work:

```
let sum3_plain = function
    | [x1, x2, x3] -> x1 + x2 + x3
    | _            -> failwith "Input must be length 3"

let sum3_metaocaml = .<let sum3 l = .~(make_match .<l>. [
    .<fun [x1; x2; x3] -> x1 + x2 + x3>. [@metaocaml.functionliteral];
    .<fun _ -> failwith "Input must be length 3">.
                                         [@metaocaml.functionliteral];
]) in sum3>.

let sum3_fcp = function_ [
    var >:: (var >:: (var >:: empty)) => (fun x1 x2 x3 ->
                                          .<.~x1 + .~x2 + .~x3>.);
    __                => .<failwith "Input must be length 3">.
]
```

All three versions have broadly the same structure, however, the original MetaOCaml implementation has a number of syntactic burdens, including the inability to define `function` expressions, and a syntactically dense `functionliteral` annotation for each case. The version with first-class patterns has some immediate disadvantages as well; the pattern syntax varies somewhat from OCaml's, while the pattern expression variables must be spliced in before use. The pattern syntax can be improved, however; exposing the underlying GADT tags, and using OCaml's list overloading syntax for GADTs allows for a more concise and familiar syntax:

```
let sum3_fcp = function_ [
    [var; var; var] => (fun x1 x2 x3 -> .<.~x1 + .~x2 + .~x3>.);
    __                => .<failwith "Input must be length 3">.]
```

Thus, writing generators for pattern-matching expressions is now scarcely more complex than writing the expression itself. With the drop-in usage of statically unknown pattern generation combinators also possible, the usability of this library is broadly good. However, a significant part of writing pattern generators is debugging incorrect definitions, and therefore helpful error messages have a significant influence on ergonomics.

6.2.1 Error messages

A significant criticism of Rhiger's first-class pattern matching [17], and indeed one levelled by Kiselyov in support of MetaOCaml's pattern generation approach [14], is the complexity of the pattern types, and the resulting incomprehensible error messages. This pattern generation library offers a significant improvement in this regard; the complexity of continuation-based types is abstracted from the user behind the `pat` and `case` types, providing a simple interface. Mistakes such as incompatible case patterns or return types, like those described in Section 2.3, produce concise, informative error messages:

```
let incomp_ret = function_ [
    var => (fun _ -> .<1>.);
    var => (fun _ -> .<true>.)
]

Error: This expression has type ('a, bool) case
       but an expression was expected of type ('a, int) case
       Type bool is not compatible with type int

let incomp_pat = function_ [
    int 2 => .<1>.;
    int 2 ** int 2 => .<1>.
]

Error: This expression has type (int * int, int) case
       but an expression was expected of type (int, int) case
       Type int * int is not compatible with type int
```

Similarly pleasing error messages are produced for `match` expression generation, and while MetaOCaml's repurposing of OCaml's existing pattern system yields more specific error messages, equivalent debugging information is provided:

```
let incomp_scr = match_ .<true>. [
    int 2 => .<1>.;
]
```

```
Error: This expression has type (int, int) case
       but an expression was expected of type (bool, 'a) case
       Type int is not compatible with type bool
```

```
let incomp_scr_metaocaml = make_match .<true>. [
    .<fun 2 -> 1>. [@metaocaml.functionliteral]
]
```

```
Error: This pattern matches values of type int
       but a pattern was expected which matches values of type bool
```

Error messages derived from inferred types are also pleasant:

```
let erroneous_use = match_ .<true>. [
    var => fun x -> .<.~x + 1>.
]
```

```
Error: This expression has type (int, int) case
       but an expression was expected of type (bool, 'a) case
       Type int is not compatible with type bool
```

The pattern generation system does introduce new opportunities for errors, such as providing too few or too many binding arguments to variable patterns or misusing the bound `code` arguments, for example by forgetting to splice them. The errors produced for these mistakes are once again helpful, however:

```
let too_many_binds = function_ [
    int 2 => fun x -> x
]
```

```
Error: This expression should not be a function, the expected type is
       'a Codelib.code
```

```
let too_few_binds = function_ [
    var => .<2>.
]
```

```
Error: This expression has type 'a code
       but an expression was expected of type 'b code -> 'c code
```

```
let forgotten_splice = function_ [
    [var; var] => fun x y -> .<x + y>.
]
```

```
Error: This expression has type 'a code
       but an expression was expected of type int
```

The type signatures of the statically unknown pattern generation combinators ensure that the clean, precise error messages produced by the core pattern system are propagated to their usage. This includes both misuse of combinator outputs:

```
let incompatible_combinators = function_ [
    gen_exactly_n_cons 2 .<true>. (fun x y -> .<.~x && .~y>.);
    gen_exactly_n_cons 3 .<0>. (fun x y -> .<.~x + .~y>.)
]
```

```
Error: This expression has type (int list, int) case
       but an expression was expected of type (bool list, bool) case
       Type int is not compatible with type bool
```

As well as misuse of the combinators themselves:

```
let incomp_basecase = function_ [
    gen_n_cons 2 (fun _ -> .<true>.) (fun x y -> .<.~x + .~y>.);
]
```

```
Error: This expression has type bool code
       but an expression was expected of type int code
       Type bool is not compatible with type int
```

```
let incomp_inductive = function_ [
   gen_n_cons 2 (fun _ -> .<true>.) (fun x y -> .<.~y  .~x>.);
]
```

```
Error: This expression has type bool
       This is not a function; it cannot be applied.
```

With this, it is therefore shown that a best-of-both-worlds for type-safe pattern generation does indeed exist, offering the expressivity of arbitrary pattern matching, with the concise, helpful error messages of native pattern-matching expressions.

6.3 Safety guarantees

The primary contribution of this work is to provide a type-safe system for pattern generation. There are however several points of discussion to be raised on this matter. Firstly, in order to produce typed values from low-level, untyped AST manipulation, some instances of unsafe coercion are required. These unsafe coercions are minimal and can be listed and justified. In addition, for the assurance that all generated code is safe, issues of scoping, including the prevention of scope extrusion and accidental scope inclusion, must be combatted.

6.3.1 Unsafe coercions

Within the safe subset of the pattern generation library, there are 3 locations in which OCaml's unsafe coercion function, `Obj.magic`, is required. The first of these uses is the coercion discussed in Section 4.4, in which a duplicate of the underlying, and unexposed, type of an `'a code` in MetaOCaml, a `code_repr`, is defined within the library. Unsafe coercion is then performed between the two identical types. This coercion is, in fact, safe however; due to the identical type definitions, compiled with the same compiler and options, the underlying memory representations of the two types are the same, and so coercion between them may be performed safely. As also previously mentioned, this workaround will cease to be required upon integration of this work into MetaOCaml.

The second such case is the unsafe casting of `code_repr`s to `'a code`s in the code generation functions of `match_` and `function_`. When generating code, the output results are the raw Parsetree.expression, representing the AST of the code, and a representation of the free variables and virtual bindings featured within the expression. These two values may be combined with the `Code` constructor into `code_repr`, which is a supertype of the desired `'a code`, however, to obtain the desired `'a code`, unsafe coercion is required. This is the primary purpose of the pattern type system described in this work; the `'a` type is determined by the provided pattern and pattern expression, ensuring that the `code_repr` does indeed correspond to a valid `'a code`, thus ensuring that the coercion is safe.

The final instance occurs in the construction of the Parsetree.pattern and the body of the pattern expression. With a fresh pattern variable identifier generated, the string of the name is unsafely coerced into a code quotation of a free occurrence of the identifier, of type `'a code`. This allows the variable to be passed to the pattern expression binding function, substituting the identifier into any occurrences of the pattern variable in the expression body. This is safe due to the simultaneous generation of the Parsetree.pattern. The fresh identifier is generated and then used for both the pattern as well as the binding argument associated with the pattern in the pattern expression. Thus the free code quotation is actually bound under the pattern and is coerced to the type derived from the pattern, guaranteeing the safety of this coercion.

6.3.2 Pattern variable uniqueness

An important component of generating safe pattern-matching code is selecting unique identifiers for pattern variables. This is especially important as if a collision occurs, accidental scope inclusion may occur, resulting in a program with semantics potentially differing from those specified by the generator. An example of this could occur with a generator such as the following:

```
let scope_capture : (int -> int) code = .<let x0 = 1 in .~(function_ [
    var => fun _ -> .<x0>.
])>.
```

If no renaming of the variables occurred, and the identifier `x0` was also generated for the pattern variable, the following code would be produced:

```
.<let x0 = 1 in function | x0 -> x0>.
```

The return value of this function would now be the input value, as opposed to the constant value of `1` intended by the generator.

The unique variable identifier generation present in this work ensures that pattern variables never collide with each other, however, it is not guaranteed that the generated identifiers do not collide with any of the free variables in the code quotation for the right-hand side of the pattern. To achieve this, the internal MetaOCaml function `get_fresh_name` could be used. This function takes as input a variable name prefix and a representation of free variables and virtual bindings and produces a unique variable name, which also does not appear in the free variables heap or list of virtually bound variables. Thus, it may be used to generate a pattern variable identifier which is guaranteed to avoid collision and accidental scope inclusion. Unfortunately, this generation function is not exposed by MetaOCaml, and the previously used workaround of duplicating unexposed functions within the library is also unusable, as the unique generator relies on inaccessible internal state.

This issue is, however, a minor obstacle; upon integration of this work with MetaOCaml, the internal state will be accessible, and unique generation can be achieved. This may require slight modifications to the `case` type representation and code generation, however, to delay pattern variable identifier assignment until code generation, when the free variables of the pattern expressions are known. Ultimately, however, this is seemingly never an issue in practice. All variables bound within a code quotation, such as the `x0` defined in the example above, are uniquely renamed in a manner that never collides with the unique naming strategy used for pattern variables. Thus, free variables cannot be accidentally included in the scope of a pattern variable binding. The code actually produced by `scope_capture` is as follows:

```
.<let x0_18 = 1 in function | r16 -> x0_18>.
```

This generated code differentiates the pattern variable and let-bound variable correctly, as denoted in the generator. Thus any generated code is indeed safe and exhibits the specified semantics.

6.3.3 Scope extrusion

The issue of preventing scope extrusion, described in Section 2.1, is an integral part of ensuring the safety of generated code. Failing to do so may result in unbound variable errors at splice time, or more insidiously, incorrect semantics due to unintended scope inclusion [14]. As outlined in Section 4.4, MetaOCaml detects scope extrusion as a dynamic check, by validating that all variables are within their scope before any use of them. More specifically, for every code quotation of type `'a code`, MetaOCaml stores a heap of the identifiers of free variables within the code quotation and a list of virtual let-bindings in effect during the evaluation of the code expression. These free variables are dynamically bound within some scope. Scope extrusion is then defined as any free variable present in the heap, which is not within its dynamically bound scope, at any stage of evaluation [7]. Validation is then simply traversing the heap, and verifying that each variable identifier is within its dynamically bound scope.

As also discussed in Section 4.4, this work performs the validation required to prevent scope extrusion before every use of a variable in code generation. However, as new variables are introduced during pattern generation, these variables must also be added to the free variable heap, and dynamically bound within the body, to ensure any extrusion of them can be detected by the validation step. In the current version of the library, this is not implemented. Once again, this is due to the requirement to use internal, unexposed MetaOCaml functions relying on internal state. The heap storing free variable identifiers relies on priority values, with new insertions into the heap generating a priority from the internal `prio_counter`. It is therefore not possible to correctly insert new free variables into the heap without the step of integrating this work into MetaOCaml.

This does mean that it is possible to cause undetected scope extrusion of pattern variables when generating code, such as performing let-insertion with `genlet` and the default locus [33]:

```
let genlet_extrusion = function_ [
    var => fun x -> let y = genlet ~name:"y" .<.~x + 1>. in .<.~y>.
]

(* .<let y_20 = r17 + 1 in function | r17 -> y_20>. *)
```

This issue is relatively simply resolved upon integration of this work into MetaOCaml, by adding the generated pattern variable identifiers to the free variable heap immediately after construction, and dynamically binding them in the pattern expression.

Chapter 7

Conclusion and future work

In this chapter, the work constituting this dissertation is summarised, with the motivation, goals, achievements and their implementations reviewed, and conclusions drawn from this (Section 7.1). A range of potential future avenues of work in this area are also discussed, covering topics from extending the pattern system to implementation into other multi-stage programming systems and formalisation (Section 7.2).

7.1 Summary and conclusion

This dissertation introduces a system enabling the type-safe generation of pattern-matching expressions with statically unknown patterns in multi-stage programming. The motivating problem for this arose from a desire to allow the specialisation of list algorithms, such as `map`, using multi-stage programming. In order to achieve this, the aim of the project was to design and implement a system allowing for list cons patterns of an arbitrary length to be generated, parametrised by a run-time value.

In creating this system, two primary contributions were made. The first is an embedded domain-specific language allowing for the description of patterns, and associated machinery, enabling both the typing of these patterns and the cases containing them, and the generation of code from this DSL. The second contribution is a family of combinators, facilitating the construction of patterns derived from statically unknown values.

The result of this was very successful; the implemented pattern system allows for the safe construction of the list cons patterns desired to specialise list mapping. More generally, the pattern system provides a way to generate a broad class of pattern-matching expressions, is extensible to many other pattern types, and provides a clean and ergonomic interface for use, with clear, informative error messages. The two minor caveats for safety present are both easily remedied by tighter integration to MetaOCaml. The pattern system is also suitably general such that this work is applicable to other multi-stage programming systems.

7.2 Future work

This work provides a solid foundation, from which a number of directions for continued work may be pursued. A selection of these possibilities are described below:

Implementation into MetaOCaml

As discussed in Section 4.4 and Section 6.3, in order to remove the remaining safety caveats of this work, the library must be integrated into the MetaOCaml system itself. Doing so would allow access to the internal state required for unique pattern identifier generation and scope extrusion detection. This work should be relatively trivial, as the implementation of this work is structured so as to mirror the structure of MetaOCaml, enabling easy integration.

Support for a broader range of patterns

At present, only a subset of the patterns available in native OCaml pattern matching are able to be generated with this work. While many of the missing patterns can be implemented relatively easily, there exist a number of pattern types whose generation poses significant challenges. The current GADT-based implementation of the pattern DSL inherently restricts the types that can be represented. Although moving away from this approach would require sacrificing features such as the GADT list syntax overloading, introduced in Section 6.2, it would be necessary to support more complex patterns, such as GADT patterns, polymorphic variant patterns, and polymorphic record patterns. Also problematic would be disjunctive patterns, in which the same variable patterns appear in different orders in separate disjunct patterns.

However, despite these challenges, the outlook for extending pattern support is not entirely bleak. Rhiger shows that representing patterns with functions actually allows for a broader class of pattern types than supported by Haskell [17], suggesting that the approach taken in this work is sufficiently expressive to represent the aforementioned patterns. Given the expressivity of the first-class patterns approach, it may also be possible to allow the generation of pattern types not found in native OCaml, such as the `view` patterns described by Wadler [34].

Implementation into other multi-stage programming systems

As a result of the general approach to typing patterns, pattern-matching cases and pattern-matching expressions taken in this work, the techniques described here are not only compatible with the MetaOCaml system on which this library is implemented. A possible avenue for further work would be to consider the implementation of type-safe pattern generation into other metaprogramming systems, such as Template Haskell [6] or MacoCaml [35]. This would likely require significant modification to the code generation combinators; MacoCaml, for example, targets OCaml's `lambda` representation instead of the Parsetree AST representation targeted by MetaOCaml. The underlying pattern type system, however, could remain the same.

Relaxation of the dependent type restriction

The restriction on generating code with a type dependent on values, discussed in Section 5.1, prevents the safe generation of many pattern types, such as nested tuple patterns. One possible solution, particularly if working with a compile-time metaprogramming systems such as the aforementioned Template Haskell or MacoCaml, would be to define generation combinators that accept a type-level input, such as those seen in defining a generic `zipWithN` [18]. The output type of generators could then vary, dependent on a type instead of a value. Statically known values could be promoted to a type representation, to provide an elegant interface while enabling a far broader class of patterns to be safely generated.

Rigourisation and formalisation

While the safety of this work has been reviewed and justified in Section 6.3, a more rigorous treatment of the pattern type system described was outside the scope of this dissertation. A formal description of the typing rules for pattern and code generation combinators would allow for greater confidence in their safety. Such a system could also integrate with the formal theory of MetaOCaml [8], and extend this to validate the detection of scope extrusion of pattern variables. With such rigour, the idea of formalising this work in a proof assistant such as Coq or Agda could be explored.

Pattern generation fuzzing

To improve the reliability of this library without the significant labour that would be associated with formalising it, the concept of fuzzing could be applied. In fuzzing, random program inputs are generated and fed to a program, with any unintended behaviour, such as crashes, detected and recorded. Such testing of the pattern system may allow for the detection of bugs in areas such as scope extrusion detection, although would not suffice to guarantee correctness.

Bibliography

[1] A.P. Ershov. "On the partial computation principle". In: *Information Processing Letters* 6.2 (1977), pp. 38–41. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(77)90078-3. URL: https://www.sciencedirect.com/science/article/pii/0020019077900783.

[2] Oleg Kiselyov. *Reconciling Abstraction with High Performance: A MetaOCaml approach.* 2018, pp. 18–20. DOI: 10.1561/2500000038.

[3] Jacques Carette and Oleg Kiselyov. "Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code". In: *Science of Computer Programming* 76.5 (2011). Special Issue on Generative Programming and Component Engineering (Selected Papers from GPCE 2004/2005), pp. 349–375. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2008.09.008. URL: https://www.sciencedirect.com/science/article/pii/S016764230800110X.

[4] Oleg Kiselyov and Walid Taha. "Relating FFTW and split-radix". In: *Proceedings of the First International Conference on Embedded Software and Systems*. ICESS'04. Hangzhou, China: Springer-Verlag, 2004, pp. 488–493. ISBN: 3540281282. DOI: 10.1007/11535409_71. URL: https://doi.org/10.1007/11535409_71.

[5] Andrew W Appel. *Unrolling recursions saves space.* Tech. rep. CS-TR-363-92. Princeton University, Mar. 1992. URL: https://www.cs.princeton.edu/techreports/1992/363.ps.gz.

[6] Tim Sheard and Simon Peyton Jones. "Template meta-programming for Haskell". In: *SIGPLAN Not.* 37.12 (Dec. 2002), pp. 60–75. ISSN: 0362-1340. DOI: 10.1145/636517.636528. URL: https://doi.org/10.1145/636517.636528.

[7] Oleg Kiselyov. "The Design and Implementation of BER MetaOCaml". In: *Functional and Logic Programming.* Ed. by Michael Codish and Eijiro Sumii. Cham: Springer International Publishing, 2014, pp. 86–102. ISBN: 978-3-319-07151-0. DOI: 10.1007/978-3-319-07151-0_6.

[8] Oleg Kiselyov. *MetaOCaml Theory and Implementation.* 2023. arXiv: 2309.08207 [cs.PL].

[9] Oleg Kiselyov. "MetaOCaml: Ten Years Later". In: *Functional and Logic Programming.* Ed. by Jeremy Gibbons and Dale Miller. Singapore: Springer Nature Singapore, 2024, pp. 219–236. ISBN: 978-981-97-2300-3. DOI: 10.1007/978-981-97-2300-3_12.

[10] Walid Taha. “A Gentle Introduction to Multi-stage Programming”. In: *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*. Ed. by Christian Lengauer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 30–50. ISBN: 978-3-540-25935-0. DOI: `10.1007/978-3-540-25935-0_3`.

[11] Matthew Pickering, Andres Löh, and Nicolas Wu. *A Specification for Typed Template Haskell*. 2021. arXiv: `2112.03653 [cs.PL]`.

[12] Ningning Xie et al. “Staging with class: a specification for typed template Haskell”. In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: `10.1145/3498723`. URL: `https://doi.org/10.1145/3498723`.

[13] Matthew Pickering, Nicolas Wu, and Csongor Kiss. “Multi-stage programs in context”. In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. Berlin, Germany: Association for Computing Machinery, 2019, pp. 71–84. ISBN: 9781450368131. DOI: `10.1145/3331545.3342597`. URL: `https://doi.org/10.1145/3331545.3342597`.

[14] Oleg Kiselyov. *MetaOCaml – An OCaml dialect for multi-stage programming*. Accessed: 2024-05-22. Dec. 2023. URL: `https://okmij.org/ftp/ML/MetaOCaml.html`.

[15] Mark Tullsen. “First Class Patterns”. In: *Proceedings of the Second International Workshop on Practical Aspects of Declarative Languages*. PADL ’00. Berlin, Heidelberg: Springer-Verlag, 2000, pp. 1–15. ISBN: 3540669922. URL: `https://dl.acm.org/doi/10.5555/645770.667771`.

[16] Burak Emir, Martin Odersky, and John Williams. “Matching Objects with Patterns”. In: *ECOOP 2007 – Object-Oriented Programming*. Ed. by Erik Ernst. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 273–298. ISBN: 978-3-540-73589-2. DOI: `10.1007/978-3-540-73589-2_14`.

[17] Morten Rhiger. “Type-safe pattern combinators”. In: *Journal of Functional Programming* 19.2 (2009), pp. 145–156. DOI: `10.1017/S0956796808007089`.

[18] Daniel Fridlender and Mia Indrika. “Do we need dependent types?” In: *Journal of Functional Programming* 10.4 (2000), pp. 409–415. DOI: `10.1017/S0956796800003658`.

[19] Olivier Danvy. “Functional unparsing”. In: *Journal of Functional Programming* 8.6 (1998), pp. 621–625. DOI: `10.1017/S0956796898003104`.

[20] Zhe Yang. “Encoding types in ML-like languages”. In: *Theoretical Computer Science* 315.1 (2004). Mathematical Foundations of Programming Semantics, pp. 151–190. ISSN: 0304-3975. DOI: `10.1016/j.tcs.2003.11.017`. URL: `https://www.sciencedirect.com/science/article/pii/S0304397503006212`.

[21] Brent Yorgey Reiner Pope. *first-class-patterns*. Version 0.3.2.5. May 22, 2024. URL: `https://hackage.haskell.org/package/first-class-patterns`.

[22] Tom Schrijvers et al. “Towards open type functions for Haskell”. In: *Proceedings of the 19th International Symposium on Implementation and Application of Functional*

Languages. Ed. by Olaf Chitil. Freiburg, Germany: Computing Laboratory, University of Kent, Sept. 2007, pp. 233–251. URL: `https://proglang.informatik.uni-freiburg.de/IFL2007/proceedings.pdf`.

[23] Robert Atkey, Sam Lindley, and Jeremy Yallop. “Unembedding domain-specific languages”. In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*. Haskell ’09. Edinburgh, Scotland: Association for Computing Machinery, 2009, pp. 37–48. ISBN: 9781605585086. DOI: `10.1145/1596638.1596644`. URL: `https://doi.org/10.1145/1596638.1596644`.

[24] Jeremy Yallop and Oleg Kiselyov. “Generating mutually recursive definitions”. In: *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM 2019. Cascais, Portugal: Association for Computing Machinery, 2019, pp. 75–81. ISBN: 9781450362269. DOI: `10.1145/3294032.3294078`. URL: `https://doi.org/10.1145/3294032.3294078`.

[25] Oleg Kiselyov and Jeremy Yallop. “let (rec) insertion without Effects, Lights or Magic”. In: *CoRR* abs/2201.00495 (2022). arXiv: `2201.00495`. URL: `https://arxiv.org/abs/2201.00495`.

[26] Jun Inoue, Oleg Kiselyov, and Yukiyoshi Kameyama. “Staging beyond terms: prospects and challenges”. In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM ’16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pp. 103–108. ISBN: 9781450340977. DOI: `10.1145/2847538.2847548`. URL: `https://doi.org/10.1145/2847538.2847548`.

[27] Takahisa Watanabe and Yukiyoshi Kameyama. “Program generation for ML modules (short paper)”. In: *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM ’18. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 60–66. ISBN: 9781450355872. DOI: `10.1145/3162072`. URL: `https://doi.org/10.1145/3162072`.

[28] Yuhi Sato, Yukiyoshi Kameyama, and Takahisa Watanabe. “Module generation without regret”. In: *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM 2020. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 1–13. ISBN: 9781450370967. DOI: `10.1145/3372884.3373160`. URL: `https://doi.org/10.1145/3372884.3373160`.

[29] Yuhi Sato and Yukiyoshi Kameyama. “Type-safe generation of modules in applicative and generative styles”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2021. Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 184–196. ISBN: 9781450391122. DOI: `10.1145/3486609.3487209`. URL: `https://doi.org/10.1145/3486609.3487209`.

[30] Anil Madhavapeddy and Yaron Minsky. *Real World OCaml: Functional Programming for the Masses*. 2nd ed. Cambridge University Press, 2022, pp. 160–161. DOI: 10.1017/9781009129220.

[31] Graham Hutton. "A tutorial on the universality and expressiveness of fold". In: *Journal of Functional Programming* 9.4 (1999), pp. 355–372. DOI: 10.1017/S0956796899003500.

[32] Frédéric Bour, Basile Clément, and Gabriel Scherer. *Tail Modulo Cons*. 2021. arXiv: 2102.09823 [cs.PL].

[33] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. "Shifting the stage: staging with delimited control". In: *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM '09. Savannah, GA, USA: Association for Computing Machinery, 2009, pp. 111–120. ISBN: 9781605583273. DOI: 10.1145/1480945.1480962. URL: https://doi.org/10.1145/1480945.1480962.

[34] P. Wadler. "Views: a way for pattern matching to cohabit with data abstraction". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. Munich, West Germany: Association for Computing Machinery, 1987, pp. 307–313. ISBN: 0897912152. DOI: 10.1145/41625.41653. URL: https://doi.org/10.1145/41625.41653.

[35] Ningning Xie et al. "MacoCaml: Staging Composable and Compilable Macros". In: *Proc. ACM Program. Lang.* 7.ICFP (Aug. 2023). DOI: 10.1145/3607851. URL: https://doi.org/10.1145/3607851.