The Ultimate Conditional Syntax

Lionel Parreaux
The Hong Kong University of Science and Technology
parreaux@cse.ust.hk

Abstract

ML-language dialects and related typically support expressive pattern-matching syntaxes which allow programmers to write concise, expressive, and type-safe code to manipulate algebraic data types. Many features have been proposed to enhance the expressiveness of these pattern-matching syntaxes, such as pattern bindings, pattern alternatives (aka disjunction), pattern conjunction, view patterns, pattern guards, 'if-let' patterns, multi-way if-expressions, etc.

In this extended abstract, we propose a new framework for expressing pattern-matching code in a way that is both more expressive and (we argue) more readable than previous alternatives. In particular, our syntax subsumes many proposed extensions to ML pattern matching by allowing parallel and nested matches interleaved with computations and intermediate bindings. This is achieved through a form of nested multi-way if-expressions, an expression-splitting mechanism to factor common conditional prefixes, and a technique we call *conditional pattern flowing*.

We present many examples of this new syntax in the setting of MLscript, a new ML-family programming language that is being developed at the Hong Kong University of Science and Technology.

1 Introduction

Consider the following excerpt from an OCaml pattern matching expression (which could contain lots of cases):

```
match e with
  ...
  | Lit value when Map.mem value mapped →
      Map.find value mapped
  ...
```

where `mapped` is a mapping from literal values to results, `Map.mem` returns whether a given key is in the map and `Map.find` accesses the corresponding key, throwing an exception if the key is not found.

This code suffers from "Boolean blindness", which is when a program branches based on a Boolean value and when the corresponding branches implicitly depend on the value of that Boolean. It is problematic because `Map.find` is a partial function that is only safe to call when accessing a key that is known to be in the map. Since the check that the key is indeed in the map is not explicitly connected to the call to `Map.find`, a seemingly-innocuous refactoring and other changes could silently break the implicit assumption, resulting in runtime failure, for example if we added `|| value == 0` to the `when` clause.

It would be better to use the `Map.find_opt` function instead, which returns an optional value that is `None` when the key is not found. This would also be more efficient, as it would only require a single access to the map. However, due to the limitations of pattern matching in OCaml, there is no *local*, *non-disruptive* way of changing our program to using `find_opt` instead of `find`. If the program above had other `Lit` cases listed below, for example as in:

```
...
| Lit value when Map.mem value mapped →
    Map.find value mapped
...
| Lit value | Add(0, value) | Add(value, 0) →
    print_int value ; process value
...
```

then we would need to completely change the structure of our pattern matching expression, for example by writing:

```
let helper value =
  print_int value ; process value in
match e with
  ...
  | Lit value →
    match Map.find_opt value mapped with
      | Some result → result
      | None → helper value
  ...
  | Add(0, value) | Add(value, 0) →
      helper value
  ...
```

In MLscript, a new ML dialect currently being developed[1] at the Hong Kong University of Science and Technology, the following is what one would write instead:

```
if e is
  ...
  Lit(value) and Map.find_opt(value) is Some(result)
    then Some(result)
  ...
  Lit(value) | Add(0, value) | Add(value, 0)
    then print_int(value); Some(value)
  ...
```

which shows that `Map.find_opt` can now be used without disrupting the existing flow of pattern matching.

In the code above, we rely on several fundamental features of MLscript's conditional syntax, which we facetiously dub the *Ultimate Conditional Syntax* (UCS):

[1] The features shown in this paper are in active development in an unstable branch of the repository at https://github.com/hkust-taco/mlscript/.

Pattern flowing. First, conditional Boolean expressions can bind pattern variables through the special '`is`' and '`as`' operator forms, in a way that is reminiscent of *flow-typing*, also called *occurrence-typing* [Castagna et al. 2021; Pearce 2013; Tobin-Hochstadt and Felleisen 2008].

'`Expr is Pattern`' tests whether `Expr` matches `Pattern`, and if so it binds the corresponding pattern variables in the logical continuation of this expression. A logical continuation can be a `then`, for example in '`if x is Some(x) then x + 1`', but it can also be an `and`, among other things, as in '`if x is Some(x) and x > 0 then ...`'. Moreover, the sets of pattern variables bound in the two sides of an `or` are checked to be the same and are given corresponding least-upper-bound types, as in '`if x is Some(v) or y is Left(v) then v ...`'.

'`Expr as Pattern`' asserts that `Expr` matches `Pattern`, used to destructure expressions unconditionally, and '`Pattern as Pattern`' is a form of conjunctive pattern, a generalization of OCaml's '`as`' and Haskell's '@'.

A nested multi-way 'if' with interleaved 'let'. Second, the structure of if-then-else expressions is akin to a *nested* version of what Haskell calls "*multi-way if-expressions*"[2] but in which bindings can be interleaved. Haskell's basic version of multi-way if is essentially as follows, in MLscript syntax:

```
if
  A then ...
  B then ...
  C then ...
  ...
```

We generalize this to follow the structure exemplified below:

```
if
  let P0 = ...
  A and
      let P1 = ...
      B and
          let P2 = ...
          C then ...
          let P3 = ...
          D then ...
          else ...
      let P4 = ...
      E then ...
      let P5 = ...
      F then ...
      else ...
```

where `A`, `B`, `C`, `D`, `E`, and `F` are arbitrary boolean expressions which may bind patterns and each `P`*n* is a let-bound pattern, which may also introduce bindings of its own (or might just be used for its side effects, as in `let () = print_string "got so far in the match!"`).

Note that MLscript is indentation-sensitive and uses the so-called "off-side rule",[3] which is why we do not need to the `in` keyword after each `let` and also why we do not need a '`|`' operator to separate match cases (similar to Haskell).

Conditional splits. Third and finally, an MLscript conditional expression may be *split* in rather arbitrary way, leading up to one or more conditional branches. For example, all of the following splits are legal:

```
if x ==
  0 then "zero"
  1 then "unit"
  else "?"

if x
  == 0 then "null"
  > 0  then "positive"
  < 0  then "negative"

if pred of 0, 1, 2,
  3, 4 then "A"
  5, 6 then "B"
else "C"
```

(In MLscript, '`f(a, b, c)`' may be written '`f of a, b, c`' where '`of`' is right-associative and can serve the purpose of Haskell's '$' dollar-sign operator).

From the above examples, it becomes clear that the splits on the '`is`' operator seen previously are merely a special case of MLscript's general operator-application splitting rules:

```
if foo(u, v, w) is
  Some(x) and x is
    Left(_)  then "left-defined"
    Right(_) then "right-defined"
  None       then "undefined"
```

While conditional splitting seems quite atypical at first, it is not ambiguous and is thus not hard to parse. Our parser uses a straightforward recursive descent implementation (for better recovery and error messages). When parsing sub-expressions, we return an `Either Term ThenElseBlock`, where the latter stands for a block of `let`, `then`, and an optional `else` clause *or* an expression that ends with such nested blocks. When we find a `ThenElseBlock` instead of a `Term`, depending on where the sub-expression was parsed, we either yield an error (if this position does not accept then-else blocks) or we propagate the then-else block out to the outer expression, until we finally reach an enclosing `if`.

Splitting function definitions. As a small syntactic sugar feature, MLscript also support splitting patterns at function definition sites, which recovers the ability to define functions by successive equations like in SML or Haskell:

```
fun foo of
  C1(a, b, ...) = e1
  C2(c, d, ...) = e2
  C3(e, f, ...) = e3
  ...
```

[2] http://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/multiway_if.html

[3] There is nothing in our approach that fundamentally relies on indentation. One could use separators such as braces and commas or semicolons to split operators, as in `if x { > 0 then "positive", == 0 then "null", else "?" }`. The indentation syntax is simply the approach we found the cleanest and easiest to read when expressing conditional splits.

```
  _              = en
```

As a concrete example, consider the '`zip_with`' function below, which zips two lists together using a function `f`, returning `None` when the lists have mismatched lengths:

```
fun zip_with(f, xs, ys) =
  if  xs is x :: xs
  and ys is y :: ys
  and zip_with(f, xs, ys) is Some(tail)
  then Some of f(x, y) :: tail
  else None
```

This function can alternatively be written as follows:

```
fun zip_with of f,
  x :: xs, y :: ys
      when zip_with(f, xs, ys) is Some(tail)
        = Some of f(x, y) :: tail
  _, _  = None
```

Exhaustiveness Checking. MLscript's UCS is meant to be a fully type-safe alternative to traditional conditional structures. Therefore, it is crucial to check for the exhaustiveness of conditional expressions, preventing non-exhaustive-patterns crashes at runtime.

Boolean conditionals of the forms `Expr and Expr`, `Expr or Expr`, `not Expr`, `Expr is Pattern`, and `Expr as Pattern` are interpreted specially. Other conditional expressions `Expr` are treated as shorthands for `Expr is true`, and are identified up to syntactic alpha equivalence — and possibly, in the future, a form of normalization identifying expressions like `x > 2`, `2 < x`, and `not (2 >= x)`, for example.[4] Then, the compiler transforms the whole conditional expression by unnesting patterns and handling each matched operand individually, in its order of appearance, making sure that all cases are covered in the process.

MLscript supports a form of subtyping known as *algebraic subtyping* [Dolan 2017; Parreaux 2020], which admits principal principal type inference at the cost of some approximations in the meaning of type constructors. In particular, in MLscript the union of tuple types $(\tau_1, \tau_2) \mid (\tau_2, \tau_3)$ and the tuple type with union components $(\tau_1 \mid \tau_2, \tau_2 \mid \tau_3)$ are equivalent. Therefore, the conditional expression below:

```
if x is
  (0, 0) then true
  (1, 1) then false
```

is not considered exhaustive and is rejected by the compiler, which complains that patterns `(0, 1)` and `(1, 0)` are not handled. In a different language, such as TypeScript (which does not feature global type inference, a fundamental tradeoff), such a conditional expression could be typed exhaustively by assuming $(0, 0) \mid (1, 1)$ for the type of `x`, which unlike in MLscript is not equivalent to $(0 \mid 1, 0 \mid 1)$.

[4] We could naturally also imagine the use of an SMT solver to decide exhaustiveness in the presence of decidable theories like linear integer arithmetic.

```
tp(1) match
  case Ref(r) => glb(r, tp(2))
  case tp1 =>
    tp(2) match
      case Ref(r) => glb(tp1, r)
      case _ =>
        val tp1_p = process(tp1)
        if tp1_p != tp1 then glb(tp1_p, tp2)
        else
          val tp2_p = process(tp2)
          if tp2_p != tp2 then glb(tp1, tp2_p)
          else if ...
            ...
```

```
if
  tp(1) as tp1 is Ref(r) then glb(r, tp(2))
  tp(2) as tp2 is Ref(r) then glb(tp1, r)
  let tp1_p = process(tp1)
  tp1_p != tp1           then glb(tp1_p, tp2)
  let tp2_p = process(tp2)
  tp2_p != tp2           then glb(tp1, tp2_p)
  ...
  ...
```

```
if name.startsWith("_") then
  name.tailOption match
    case Some(namePostfix)
      if namePostfix.forall(_.isDigit)
      => namePostfix.toIntOption match
        case Some(index)
          if index <= arity && index > 0
          => Right((index, name))
        case _ => Left(name)
    case _ => Left(name)
else Left(name)
```

```
if name.startsWith("_")
  and name.tailOption is Some(namePostfix)
  and namePostfix.forall(..isDigit)
  and namePostfix.toIntOption is Some(index)
  and index <= arity and index > 0
  then Right of (index, name)
else Left(name)
```

Figure 1. Examples of rightward drift in Scala 3 (left) and how the UCS eliminates the problem in MLscript (right).

2 Relation With Existing Approaches

Previous techniques. **Views** [Burton et al. 1996; Wadler 1987], such as GHC Haskell's *view patterns* implementation,[5] allow passing values being pattern-matched through functions and matching on the result, which offers some welcome flexibility. Their main limitation is that they cannot be shared easily between patterns — while they have a heuristic for avoiding repeated work, it is quite limited (it does not always apply) and in any case still incurs textual repetition. **Active patterns** [Erwig 1997] like in F# and Scala's extractors serve a similar purpose as view patterns and let programmers factor custom matching logic in the clothes of a normal pattern. I actually believe that the need for these abstractions is mainly due to the pattern matching syntax being too restrictive. Once we allow the interleaving of normal computations within matching conditionals and performing cascading matches intermediate on derived values, as in MLscript with the UCS, the need for view patterns and extractors diminishes.[6] **Haskell's pattern guards** are actually strictly more powerful than the alternatives above, in that they can execute arbitrary logic during pattern matching and match on the resulting intermediate values. However, they are still not quite as expressive as the UCS, because the matching structure is still *one-level*: different pattern branches cannot share these intermediate values and matches, unlike the UCS, which may *nest* sub-matches, resulting in several `then` clauses instead of just one.[7] Moreover, we argue that the UCS is easier to parse visually and often reads almost like English, which makes it nicer to learn and to use (though this last argument is merely a subjective assessment). Figure 2 shows a comparison between the two approaches on a non-trivial example. Mcbride and Mckinna [2004] later investigated the use of views and guards in dependent type theory, showing that they become even more interesting therein. **OCaml's pattern alternatives** let us use disjunctions of sub-patterns which may contain pattern variables, a quite powerful capability, which is covered by MLscript's use of the `or` and `is` operators. **Rust's and Swift's 'if-let' forms**[8][9] let programmers write '`if`' statements that perform matching at the same time and bind the extracted values within the `true` branch. (Moreover, Swift allows more than one match in '`if let`', unlike Rust.) It is easy to see that this is a very specific special case of the UCS. In Rust and Swift, this syntax also work for imperative loops; we could allow something similar same in MLscript, as in the following:

```
let mut cur = xs
while cur is x :: xs and x > 0 do
  cur ← xs
  print_int(x)
```

A **Scala 3 proposed extension to pattern matching**[10] would allow nesting pattern matches in pattern guards, fixing the limitation of Haskell's pattern guards. Still, this syntax does not allow the binding of intermediate values in between several cases, like is possible with MLscript's UCS. Many **Lisp dialects like Racket** have supported expressive conditional and pattern-matching structures, such as the `cond` and `match` macros, in the context of a dynamically-typed language. It should be easy to show that the UCS subsumes these constructs in a statically-typed context. Other **Haskell pattern matching extensions** were proposed by Servadei [2018], among which nonlinear pattern variables, which we could also consider as an extension in MLscript. **Occurrence typing** was introduced by Tobin-Hochstadt and Felleisen [2008] for Typed Scheme and later incorporated in TypeScript and Flow, where it is known as *flow typing*. This approach allows the types of variables to be locally refined based on path conditions encountered in the program. The connection between this and our approach is in how they interacts with conditionals and boolean operators to thread through the information that is discovered during the test. The difference is that in our approach, conditional tests may perform proper pattern matching with `is` and `as`, binding new names in the process, which are then available in the corresponding parts of the conditional expression.

Fixing the right drift problem. This problem often affects languages with pattern matching but no way of testing several conditions in parallel. Consider the left-hand-sides of Figure 1 and how the nesting of matches and conditionals on unrelated operands creates a shift to the right. The right-hand side of Figure 1 shows the much more concise and non-drifting MLscript versions of the same code.

3 Conclusions and Future Work

We have only began to prototype the UCS feature of MLscript and will report back once we get more experience implementing and using it. Our plan is to use MLscript with the UCS in a compilers course in the near future and see how students respond to it, which should provide us with precious feedback. We expect that students unfamiliar with pattern matching and functional programming may be less intimidated by the UCS since it is a logical extension to the usual if-then-else expression, reads almost like English, and has obvious meaning in many cases, unlike some classical ML-style pattern matching features.

[5] https://gitlab.haskell.org/ghc/ghc/-/wikis/view-patterns
[6] We may still consider adding custom active patterns/extractors to MLscript in the future, simply for the (small) extra convenience they may provide.
[7] "No, you can't. We all want it, but nobody can come up with a sensible syntax." https://stackoverflow.com/a/34168666/1518588
[8] https://doc.rust-lang.org/rust-by-example/flow_control/if_let.html
[9] https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html
[10] https://github.com/lampepfl/dotty-feature-requests/issues/301

Haskell (with -XViewPatterns):

```
case e of
  Var x
    | Some tmp ← get context x
    , Some res ← case tmp of
        IntVal v  → Some $ Left v
        BoolVal v → Some $ Right v
        _ → None
    → res
  Lit (IntVal v)  → Left v
  Lit (BoolVal v) → Right v
```

Scala 3 (proposed extension):

```
e match
  case Var(x) if context.get(x) match
    case Some(IntVal(v))  => Left(v)
    case Some(BoolVal(v)) => Right(v)
  case Lit(IntVal(v))  => Left(v)
  case Lit(BoolVal(v)) => Right(v)
  ...
```

MLscript:

```
if e is
  Var(x) and context.get(x) is
    Some(IntVal(v))  then Left(v)
    Some(BoolVal(v)) then Right(v)
  Lit(IntVal(v))  then Left(v)
  Lit(BoolVal(v)) then Right(v)
  ...
```

MLscript (after desugaring):

```
if
  e is Var(x) and
    let tmp0 = context.get(x)
    tmp0 is Some(IntVal(v))  then Left(v)
    tmp0 is Some(BoolVal(v)) then Right(v)
  e is Lit(IntVal(v))  then Left(v)
  e is Lit(BoolVal(v)) then Right(v)
  ...
```

Figure 2. More complex example in Haskell, Scala 3 (with a proposed extension), and MLscript version with its desugaring.

References

Warren Burton, Erik Meijer, Patrick Sansom, Simon Thompson, and Philip Wadler. 1996. Views: An extension to Haskell pattern matching. ↪ page 4

Giuseppe Castagna, Victor Lanvin, Mickaël Laurent, and Kim Nguyen. 2021. Revisiting Occurrence Typing. arXiv:1907.05590 [cs.PL] ↪ page 2

Stephen Dolan. 2017. *Algebraic subtyping*. Ph. D. Dissertation. ↪ page 3

Martin Erwig. 1997. Active patterns. In *Implementation of Functional Languages*, Werner Kluge (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–40. ↪ page 4

Conor Mcbride and James Mckinna. 2004. The view from the left. *Journal of Functional Programming* 14, 1 (2004), 69–111. https://doi.org/10.1017/S0956796803004829 ↪ page 4

Lionel Parreaux. 2020. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 124 (Aug. 2020), 28 pages. https://doi.org/10.1145/3409006 ↪ page 3

David J. Pearce. 2013. Sound and Complete Flow Typing with Unions, Intersections and Negations. In *Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer, Berlin, Heidelberg, 335–354. https://doi.org/10.1007/978-3-642-35873-9_21 ↪ page 2

Giacomo Servadei. 2018. Toward a more expressive pattern matching in Haskell. (2018). ↪ page 4

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 395–406. https://doi.org/10.1145/1328438.1328486 ↪ pages 2 and 4

P. Wadler. 1987. 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* (Munich, West Germany) *(POPL '87)*. Association for Computing Machinery, New York, NY, USA, 307–313. https://doi.org/10.1145/41625.41653 ↪ page 4

A Final Transformed Code from Figure 2

Consider the MLscript expression in Figure 2 and its first desugaring into a form without conditional operator splits.

The next step is to perform a *conditional-to-pattern-matching* transformation, where all patterns are unnested and all operands are matched exhaustively one after the other (possibly raising exhaustiveness errors in the process). We end up with the program below, which is expressed in a form with no 'if'-expression nesting that is thus isomorphic to traditional core representations of pattern matching:

```
if e is
  Var(x) then
    let tmp0 = context.get(x)
    if tmp0 is
      Some(tmp1) then
        if tmp1 is
          IntVal(v)  then Left(v)
          BoolVal(v) then Right(v)
          ... // if there are more Val cases
      else ... // from the other Var cases
  Lit(tmp2) then
    if tmp2 is
      IntVal(v)  then Left(v)
      BoolVal(v) then Right(v)
      ...
  ...
```

Notice that in the final representation above, we have to handle the other `Var` and `Val` cases which are not shown in the original example (omitted by the `...`). If these were not to be found, the conditional expression would be deemed non-exhaustive.