A Core Language for Extended Pattern Matching and Binding Boolean Expressions

ARTHUR CHARGUÉRAUD and YANNI LEFKI, Inria & ICube lab, CNRS, Université de Strasbourg, France

Functional programming languages include various pattern matching features, such as guarded patterns, matching by custom predicate, active patterns, synonymous patterns, etc. Besides, several languages include mechanisms for binding names as part of a boolean expression that appears in either an if-statement, a while-loop condition, or a pattern guard. These names may be bound either with a simple let-binding or via a test performed using pattern-matching. All these features are useful in practice, yet it appears that no mainstream language supports them all at once. In this work, we present a core language that consists of a small number of constructs that suffice to encode and combine all the desired features of pattern matching and binding boolean expressions. Thereby, we hope to consolidate existing knowledge on the topics of pattern matching and generalized forms of boolean expressions, through a streamlined presentation. We expect it to be useful not only for pedagogical purposes, but also potentially for simplifying the work of compiler developers.

1 INTRODUCTION

Pattern-matching has been popularized by ML languages. Existing functional languages include a number of extensions beyond plain pattern-matching over data constructors. There are also interesting interactions when pattern matching is used as part of a boolean test. In such case, one may wish to export bound names in the *then* branch.

This paper addresses the challenge of presenting a language that supports all these extensions at once. What we seek for is not to consider the union of several languages, but instead to identify a core language in which all of these interesting features can be encoded. In this work, we leave aside the questions about checking pattern-matching exhaustivity (a.k.a. completeness). This question is certainly interesting, yet in the presence of *when*-clauses or *view*-functions, checking exhaustivity is beyond reach without recourse to an advanced program logic.

Our work is closely related to the work by Cheng and Parreaux [2024], who also aim to give a unified presentation of generalized conditionals with pattern matching. We believe that our formalism is slightly simpler, and that it makes it corresponds to a conservative extension of the conditional and standard pattern-matching constructs.

We begin by reviewing the *pattern-matching* and *boolean-binding-expressions* features. We then present a set of core constructs, as well as encodings for derived constructs. We present evaluation rules as well as ML-style typing rules for the core language, then establish type preservation. Moerover, we establish derived evaluation rules and derived typing rules for the derived constructs, thereby showing that our encodings satisfy the expected properties.

2 SURVEY

Traditional pattern-matching. The basic idea of the pattern-matching construct is to test one value against a set of patterns, for deciding which continuation to follow. Patterns can test for constants, data constructors, or records. Patterns may bind variables, and may include subpatterns recursively. *Wildcard*-patterns match any value, without binding a name. *Or*-patterns allows testing if a value satisfies one of several patterns, in which case all these patterns must export the same set of variables. Here is an example using OCaml syntax:

Authors' address: Arthur Charguéraud, arthur.chargueraud@inria.fr; Yanni Lefki, yanni.lefki@inria.fr, Inria & ICube lab, CNRS, Université de Strasbourg, France.

```
match l with
  | ((Some x, None) | (None, Some x)) :: t -> (Some x, Some x) :: t
  | _ -> l
```

Dual to *or-patterns* are *and*-patterns, a.k.a. *intersection*-patterns. They allow testing if the value satisfies several patterns at once. Intersection patterns become particularly useful in the presence of more advanced features in patterns such as guarded patterns. They may also be used to encode *alias-patterns*, written "`p as x`" in OCaml.

When-clauses, a.k.a. guards. Most functional programming languages allow the branches of a pattern-matching to be guarded by a *when-clause*. A branch is only considered if the pattern is satisfied and if the boolean condition in the when-clause evaluates to *true*. For example:

```
match l with
  | Cons(Some(x), t) when x > 0 -> t
  | _ -> false
```

A more general approach allows *when-clauses* to appear in depth, inside any subpattern. This possibility can be useful in combination of *or-patterns*, as argued in a recent RFC for Rust [Rust 2025a]. Here is an example using Rust syntax:

```
match user.subscription_plan() {
     (Plan::Regular if user.credit() >= 100)
   | (Plan::Premium if user.credit() >= 80) => // continuation
```

The authors of this RFC point out that such guard patterns have previously appeared in particular in the Unison language [Unison 2025], in Wolfram's language [Wolfram 2025], as well as in the E-language (under the name *such-that* pattern) [ELanguage 2025].

Predicate patterns. Instead of binding a variable then testing whether this variable satisfies a boolean predicate as part of a guard, one may wish to directly include in the grammar of patterns the possibility of satisfying a boolean predicate. Here is an example in C#, where "`<= 10`" is a boolean predicate testing if a number is no greater than `10`, and where "`and`" denotes pattern intersection.

```
temperature switch {
  (<= 10) => "cold",
  (> 10 and <= 25) => "warm",
  (> 25) => "hot"};
```

Views. Pattern-matching on algebraic data types is very effective for implementing a data structure. However, for modular programming, the internal representations as an algebraic data type should not be exposed. The mechanism of *views* has been introduced to reconcile *pattern matching* with *data abstraction*. Here is an example of views in Haskell syntax.

```
type Bag -- any bag implementation, e.g., a list or a tree
data BagView = Empty | Add int Bag -- type used for iterating over bags
view :: Bag -> BagView
size :: Bag -> Integer
size (view -> Empty) = 0
size (view -> Add x b) = 1 + size b
```

This mechanism has been described by Okasaki in the context of Standard-ML [1998], it has been implemented in Haskell [Licata and Peyton Jones 2025], it appears under the name of *extractor* in Scala [Scala 2025] and under the name *active pattern* in F# [Syme et al. 2007]. Views can be also be

t, f, g	:=	x	variable
		$\lambda(x_1, ..., x_n).t$	function
		$f\,(t_1, ..., t_n)$	application
		$\textbf{switch}^{\text{trm}}\, c_1 \mid ... \mid c_n$	branching, as part of a term
c	:=	$\textbf{case}\ b\ \textbf{then}\ t$	branch of a $\textbf{switch}^{\text{trm}}$
V	:=	a set of variables	
b	:=	$t\ \textbf{is}\ p$	match against a pattern
		$\textbf{switch}^{\text{bbe}}\, d_1 \mid ... \mid d_n$	branching, as part of a BBE
		$\textbf{restrict}\ V\ b$	filter exported bindings
d	:=	$\textbf{case}\ b_1\ \textbf{then}\ b_2$	branch of a $\textbf{switch}^{\text{bbe}}$
		$\textbf{restrict}\ V\ d$	filter exported bindings
p	:=	$x^?$	pattern variable
		$p_1 \mid p_2$	pattern disjunction
		$p_1\ \&\ p_2$	pattern intersection
		$C\ (p_1, ..., p_n)$	constructor pattern
		$p\ \textbf{when}\ b$	guarded pattern
		$\textbf{restrict}\ V\ p$	filter exported bindings

Fig. 1. Grammar of our core language

encoded by means of the more general notion of *first-class patterns* [Tullsen 2000][Jay and Kesner 2009]. An example implementation of first-class patterns is the one provided in OCaml by means of the source code preprocessor named `Ast_pattern` [ppxlib 2025].

Binding boolean expressions. SSReflect [Gonthier and Le Roux 2009] is an extension to the Rocq proof assistant that has made use, since the early 2000s, of the construct "**if** t_0 **is** p **then** t_1 **else** t_2", as a syntactic sugar for "**match** t_0 **with** $p \Rightarrow t_1 \mid \_ \Rightarrow t_2$". The SSReflect manual indicates that this construct could previously be found in ML variants such as the ρ-calculus [Cirstea and Kirchner 2001] or the pattern calculus [Jay 2004].

A more general possibility is to perform on-the-fly *pattern-matching* as part of boolean conditions has been recently added to the Rust language [Rust 2025b]. The idea is that in an if-statement, one may exploit pattern matching to implement part of the test, and the variables obtained from the pattern matching can be exported into the *then*-branch of the if-statement. In the Rust example below, the variables `fn_name`, `aft_name` and `args_str` are bound in the *then*-branch.

```
if let Some((fn_name, aft_name)) = s.split_once("(")
  && !fn_name.is_empty()
  && is_legal_ident(fn_name)
  && let Some((args_str, "")) = aft_name.rsplit_once(")") {
   return fn_name + args_str;
}
```

3 SYNTAX

Figure 1 presents the grammar of our core language, which consists of λ-calculus with n-ary functions, extended with a *switch* construct. We let t range over terms. The meta-variables f and g also range over terms that represent functions. This switch construct is made of a list of branches,

Derived terms

let $p = t_1$ **in** t_2	$\equiv$	**switch**$^{\text{trm}}$ **case** $(t_1$ **is** $p)$ **then** t_2	
if$^{\text{trm}}$ b **then** t_1 **else** t_2	$\equiv$	**switch**$^{\text{trm}}$ **case** b **then** t_1 | **case** true **then** t_2	

Derived BBEs

t	$\equiv$	t **is** true	where t : bool
if$^{\text{bbe}}$ b_0 **then** b_1 **else** b_2	$\equiv$	**switch**$^{\text{bbe}}$ **case** b_0 **then** b_1 | **case** true **then** b_2	
b_1 **and** b_2	$\equiv$	**if**$^{\text{bbe}}$ b_1 **then** b_2 **else** false	
b_1 **or** b_2	$\equiv$	**if**$^{\text{bbe}}$ b_1 **then** true **else** b_2	
not b	$\equiv$	**if**$^{\text{bbe}}$ b **then** false **else** true	

Derived patterns

p **as** $x^?$	$\equiv$	p & $(x^?)$	
_	$\equiv$	$x^?$	for a fresh x
g	$\equiv$	$x^?$ **when** $g\,(x)$ where $g : (T \to \text{bool})$	for a fresh x
$f\,(p_1, ..., p_n)$	$\equiv$	$x^?$ **when** $f\,(x)$ **is** Some $(p_1, ..., p_n)$	for a fresh x

Fig. 2. Encoding of derived language constructs in term of core constructs

where each branch consists of a *binding-boolean-expression* (BBE) and a continuation. A BBE is an expression that evaluates to a boolean and, in case the result is true, binds a number of variables.

In the core language, there are only three constructs for introducing BBEs. Firstly, the construct t **is** p tests whether the result of t satisfies the pattern p, and if so, binds the appropriate pattern variables. Secondly, a switch can be viewed as a BBE if all its continuations consist of BBEs. Such switch constructs are used in particular to encode conjunctions and disjunctions of BBEs, as detailed further. Thirdly, the construct **restrict** V b can be used to reduce the set of bindings that are exported by a BBE. We have found the need to introduce the **restrict** construct to enable the set up of a semantics that does not depend on types. We present a motivating example at the end of the present section.

Remark: in the current presentation, we have deliberately chosen to separate the grammar of BBEs from the grammar of terms, to improve clarity, at the expense of preventing a few possible factorizations in the evaluation and typing rules.

The grammar of patterns includes the traditional constructs, namely variables, disjunction, intersection, testing against a constructor. In addition, the grammar of patterns includes the guard construct, written p **when** b. Such a guarded pattern exports all the variables bound by the pattern p as well as all the variables bound by the binding boolean expression b. One specificity of our design is that for an intersection pattern p_1 & p_2, the variables bound by p_1 scope over p_2. In particular, the variables bound by p_1 may appear as parts of the pattern guards that appear inside p_2.

Figure 2 presents our encodings for the remaining constructs. The constructs **let** $p = t_1$ **in** t_2, and **if** b **then** t_1 **else** t_2 can be encoded using a switch. Any boolean expression t can be viewed as a binding-boolean-expression that binds no variable. The logical combinators b_1 **and** b_2, as well as b_1 **or** b_2 and **not** b can all be encoded using conditionals, which themselves are encoded using switches. An *alias-pattern* of the form p **as** $x^?$ can be encoded using a pattern intersection. The *wildcard-pattern* can be encoded using a pattern variable that does not occur anywhere else in the program. A boolean predicate g, i.e. a function of type $T \to$ bool, can be viewed as a pattern filtering values of type T. Finally, the *view-pattern* takes the form $f\,(p_1, ..., p_n)$, where f is a user-defined function and p_i are sub-patterns. If each pattern p_i has type T_i, the function f must have type $T \to (T_1, ..., T_n)$ option. When testing a value v against the view pattern $f\,(p_1, ..., p_n)$, there are two

v	:=	$\lambda(x_1, ..., x_n).t$	λ-abstraction
	|	$C\ (v_1, ..., v_n)$	constructor (includes booleans, integers, options, tuples)
M	:=	map from variable to values	
r	:=	Mismatch | Match M	result of evaluating a BBE

Fig. 3. Entities involved in the statement of the semantics

cases. If $f(v)$ produces None, then the matching fails. Otherwise, if $f(v)$ produces Some $(v_1, ..., v_n)$, then the matching succeeds if and only if each of the values v_i satisfy the corresponding patterns p_i.

Let us now come back on the need for the **restrict** construct. Consider the pattern shown below. In this pattern, the or-pattern on the right-hand side of the conjunction produces in all cases a binding on x, while the pattern on the left-hand side of the conjunction produces a binding on y. Hence, overall, we expect the pattern to produce bindings for x and y.

$$\big(\big((x^?, y^?)\ \textbf{when}\ x > 0\ \textbf{and}\ y > 0\big)\ \mid\ (x^?, \_)\ \textbf{when}\ x > 1\big)\ \ \&\ \ (y^?, \_)$$

Consider now the evaluation of this pattern against the pair $(2, 1)$ using a naive interpreter. The evaluation of the sub-pattern "$(x^?, y^?)$ **when** $x > 0$ **and** $y > 0$" succeeds, resulting in a map $\{x \mapsto 2,\ y \mapsto 1\}$. Because the left-branch of the or-pattern has already succeeded, the sub-pattern "$(x^?, \_)$ **when** $x > 1$" is not evaluated at all. Then, the evaluation of the sub-pattern $(y^?, \_)$ succeeds, resulting in a map $\{y \mapsto 2\}$. The problem that now arises is that an interpreter lacks sufficient information for properly merging $\{x \mapsto 2,\ y \mapsto 1\}$ with $\{y \mapsto 2\}$ and output the expected result $\{x \mapsto 2,\ y \mapsto 2\}$. By introducing a **restrict** on the name x around the or-pattern, as shown below, we remedy the situation: the interpreter now only needs to compute the *disjoint* union of $\{x \mapsto 2\}$ and $\{y \mapsto 2\}$.

$$\big(\textbf{restrict}\ \{x\}\ \big(\big((x^?, y^?)\ \textbf{when}\ x > 0\ \textbf{and}\ y > 0\big)\ \mid\ (x^?, \_)\ \textbf{when}\ x > 1\big)\big)\ \ \&\ \ (y^?, \_)$$

In summary, the **restrict** construct can be used to ensure that all the branches of or-patterns (or of switches) produce the same set of bindings, and its presence is needed to provide formal evaluation rules that depend only on the grammar and not on the types. In practice, the programmer needs not insert **restrict** constructs explicitly. Indeed, they can be easily elaborated by the typechecker. Concretely, the typechecker could compute the set of bindings exported by each branch, compute the intersection of these sets, then wrap each of the branches with a **restrict** to filter the names that belong to the intersection. We leave to future work the formalization of this elaboration phase.

The **restrict** construct may also be useful for the encodings. The 3 derived pattern constructions that appears at the bottom of Figure 2 rely on the introduction of a fresh variable x. It may be more satisfying, in particular with respect to error messages, to make sure that the generated name x does not escape the local scope. To that end, it suffices to wrap the encoded pattern into a **restrict** construct. For example the boolean predicate g viewed as a pattern could be encoded as $\textbf{restrict}\,\varnothing\,(x^?\ \textbf{when}\ g\,(x))$.

4 SEMANTICS

Figure 3 presents the grammar entities involved for semantics. We let v range over values, which include functions and data constructors. We let M denote a set of bindings from variables to values, arising from a successful pattern-matching or BBE evaluation. We let r denote the result of evaluating a matching or a BBE. Such a result is either Mismatch, or of the form Match M for a map M describing the bindings obtained.

The operation $\mathrm{Subst}(M, t)$ substitutes a set of bindings into a term t. Likewise, $\mathrm{Subst}(M, b)$ substitutes a set of bindings into a BBE b, and $\mathrm{Subst}(M, p)$ substitutes a set of bindings into a pattern p. Substituting inside patterns is required because terms and BBEs can recursively occur inside patterns.

For simplicity, in the statements of the evaluation rules, we omit details about the threading of the mutable store. That said, the reader should keep in mind that the evaluation of guards and of view functions may involve arbitrary side-effects. Hence, the evaluation order is critical. In particular, certain rewriting rules and certain pattern compilation schemes that could be devised might be correct only under specific purity assumptions.

There are 3 evaluation judgments. The judgment $t \Downarrow_{\mathsf{trm}} v$ asserts that the term t evaluates to the value v. The judgment $b \Downarrow_{\mathsf{bbe}} r$ asserts that the BBE b evaluates to a result r. The judgment $v \triangleright p \Downarrow_{\mathsf{pat}} r$ asserts that the matching of the value v against the pattern p evaluates to a result r. The notation $M_1 \# M_2$ expresses that the two maps M_1 and M_2 have disjoint domains, i.e. $\mathrm{dom}(M_1) \cap \mathrm{dom}(M_2) = \varnothing$. The notation $M_{|V}$ denotes the restriction of the map M to the bindings on the names listed in the set V. By extension, $r_{|V}$ denotes the restriction of the bindings exported by a result. Formally, $\mathsf{Mismatch}_{|V} = \mathsf{Mismatch}$ and $(\mathsf{Match}\ M)_{|V} = \mathsf{Match}\ (M_{|V})$.

Figure 4 presents the type-agnostic evaluation rules for the core language. These rules consist of definitions. Figure 5 presents the evaluation rules for the derived constructs. These rules consist of lemmas, proved correct with respect to the encodings presented in Figure 2.

5 TYPING RULES

We consider simple types, with n-ary arrow types and type constructors. Figure 6 presents the typing entities. A typing environment is written E. A binding boolean expression (BBE) admits as type a map from variables to types, written B. This map describes the types of the variables exported by the BBE in case it evaluates to true. The bindings in such a map B are unordered. Certain typing rules extend a context E with the set of bindings from a map B. The result, written "E, B", can be obtained by placing the bindings form B in an arbitrary order.

There are 3 typing judgments. The judgment $E \vdash_{\mathsf{trm}} t : T$ asserts that, in the environment E, the term t admits the type T. The judgment $E \vdash_{\mathsf{bbe}} b \rightsquigarrow B$ asserts that the BBE b, in case it evaluates to true, binds a set of variables whose names and types are described by the map B. The judgment $E \vdash_{\mathsf{pat}} p : T \rightsquigarrow B$ asserts that the pattern p filters values of type T and, in case of success, binds a set of variables whose names and types are described by the map B.

Figure 7 presents the typing rules for the core language. The definitions are straightforward and follow the usual patterns, except for the typechecking a branch of our $\mathbf{switch}^{\mathsf{bbe}}$ construct. Consider a branch $\mathbf{case}\ b_1\ \mathbf{then}\ b_2$. How should we define the map B that describes the bindings exported by the branch in case the evaluation of this branch succeeds? In general, B should be defined as the union of the bindings exported by b_1 and b_2 (rule Typ-bbe-case-1). However, there is a particular case that we need to handle differently, in particular for the purpose of typechecking as we expect the intersection pattern construct $b_1\ \mathbf{and}\ b_2$. Recall that this construct is encoded as "$\mathbf{switch}^{\mathsf{bbe}}\ \mathbf{case}\ b_1\ \mathbf{then}\ b_2 \mid \mathbf{case}$ true $\mathbf{then}$ false". Consider a branch of the form "$\mathbf{case}\ b'\ \mathbf{then}$ false". The evaluation of such a branch *never* succeeds. Hence, it is sound to consider that this branch admits as type B, for any map B, as captured by the rule Typ-bbe-case-2.

Figure 8 presents *derived* typing rules for the derived constructs. Here again, those derived rules consist of lemmas proved correct with respect to the encodings presented in Figure 2. Observe, in particular, how the lemma Typ-bbe-and captures that the set of bindings exported by $b_1\ \mathbf{and}\ b_2$ corresponds to the union of the set of bindings coming from b_1 and b_2, and that the bindings of b_1 scope inside b_2.

$$\textsc{Eval-trm-val}\quad \frac{}{v \Downarrow_{\text{trm}} v}$$

$$\textsc{Eval-trm-beta}\quad \frac{\forall i.\quad t_i \Downarrow_{\text{trm}} v_i \qquad \text{Subst}(\{x_1 \mapsto v_1, ..., x_n \mapsto v_n\}, t) \Downarrow_{\text{trm}} v}{(\lambda(x_1, ..., x_n).t)\ (t_1, ..., t_n) \Downarrow_{\text{trm}} v}$$

$$\textsc{Eval-trm-switch-1}\quad \frac{b_1 \Downarrow_{\text{bbe}} \text{Mismatch} \qquad \textbf{switch}^{\text{trm}}\ c_2 \mid ... \mid c_n \Downarrow_{\text{trm}} v}{\textbf{switch}^{\text{trm}}\ (\textbf{case}\ b_1\ \textbf{then}\ t_1) \mid c_2 \mid ... \mid c_n \Downarrow_{\text{trm}} v}$$

$$\textsc{Eval-trm-switch-2}\quad \frac{b_1 \Downarrow_{\text{bbe}} \text{Match}\ M_1 \qquad \text{Subst}(M_1, t_1) \Downarrow_{\text{trm}} v}{\textbf{switch}^{\text{trm}}\ (\textbf{case}\ b_1\ \textbf{then}\ t_1) \mid c_2 \mid ... \mid c_n \Downarrow_{\text{trm}} v}$$

$$\textsc{Eval-bbe-is}\quad \frac{t \Downarrow_{\text{trm}} v \qquad v \triangleright p \Downarrow_{\text{pat}} r}{t\ \textbf{is}\ p \Downarrow_{\text{bbe}} r}$$

$$\textsc{Eval-bbe-switch-1}\quad \frac{d_1 \Downarrow_{\text{bbe}} \text{Mismatch} \qquad \textbf{switch}^{\text{bbe}}\ d_2 \mid ... \mid d_n \Downarrow_{\text{bbe}} r}{\textbf{switch}^{\text{bbe}} d_1 \mid ... \mid d_n \Downarrow_{\text{bbe}} r}$$

$$\textsc{Eval-bbe-switch-2}\quad \frac{d_1 \Downarrow_{\text{bbe}} \text{Match}\ M_1}{\textbf{switch}^{\text{bbe}} d_1 \mid ... \mid d_n \Downarrow_{\text{bbe}} \text{Match}\ M_1}$$

$$\textsc{Eval-bbe-restrict}\quad \frac{b \Downarrow_{\text{bbe}} r}{\textbf{restrict}\ V\ b \Downarrow_{\text{bbe}} r_{|V}}$$

$$\textsc{Eval-bbe-case-1}\quad \frac{b_1 \Downarrow_{\text{bbe}} \text{Mismatch}}{\textbf{case}\ b_1\ \textbf{then}\ b_2 \Downarrow_{\text{bbe}} \text{Mismatch}}$$

$$\textsc{Eval-bbe-case-2}\quad \frac{b_1 \Downarrow_{\text{bbe}} \text{Match}\ M_1 \qquad \text{Subst}(M_1, b_2) \Downarrow_{\text{bbe}} \text{Mismatch}}{\textbf{case}\ b_1\ \textbf{then}\ b_2 \Downarrow_{\text{bbe}} \text{Mismatch}}$$

$$\textsc{Eval-bbe-case-3}\quad \frac{b_1 \Downarrow_{\text{bbe}} \text{Match}\ M_1 \qquad \text{Subst}(M_1, b_2) \Downarrow_{\text{bbe}} \text{Match}\ M_2 \qquad M_1 \# M_2}{\textbf{case}\ b_1\ \textbf{then}\ b_2 \Downarrow_{\text{bbe}} \text{Match}\ (M_1 \uplus M_2)}$$

$$\textsc{Eval-bbe-case-restrict}\quad \frac{d \Downarrow_{\text{bbe}} r}{\textbf{restrict}\ V\ d \Downarrow_{\text{bbe}} r_{|V}}$$

$$\textsc{Eval-pat-var}\quad \frac{}{v \triangleright x^{?} \Downarrow_{\text{pat}} \text{Match}\ \{x \mapsto v\}}$$

$$\textsc{Eval-pat-or-1}\quad \frac{v \triangleright p_1 \Downarrow_{\text{pat}} \text{Match}\ M_1}{v \triangleright p_1 \mid p_2 \Downarrow_{\text{pat}} \text{Match}\ M_1}$$

$$\textsc{Eval-pat-or-2}\quad \frac{v \triangleright p_1 \Downarrow_{\text{pat}} \text{Mismatch} \qquad v \triangleright p_2 \Downarrow_{\text{pat}} r}{v \triangleright p_1 \mid p_2 \Downarrow_{\text{pat}} r}$$

$$\textsc{Eval-pat-and-1}\quad \frac{v \triangleright p_1 \Downarrow_{\text{pat}} \text{Mismatch}}{v \triangleright p_1\ \&\ p_2 \Downarrow_{\text{pat}} \text{Mismatch}}$$

$$\textsc{Eval-pat-and-2}\quad \frac{v \triangleright p_1 \Downarrow_{\text{pat}} \text{Match}\ M_1 \qquad v \triangleright \text{Subst}(M_1, p_2) \Downarrow_{\text{pat}} \text{Mismatch}}{v \triangleright p_1\ \&\ p_2 \Downarrow_{\text{pat}} \text{Mismatch}}$$

$$\textsc{Eval-pat-and-3}\quad \frac{v \triangleright p_1 \Downarrow_{\text{pat}} \text{Match}\ M_1 \qquad v \triangleright \text{Subst}(M_1, p_2) \Downarrow_{\text{pat}} \text{Match}\ M_2 \qquad M_1 \# M_2}{v \triangleright p_1\ \&\ p_2 \Downarrow_{\text{pat}} \text{Match}\ (M_1 \uplus M_2)}$$

$$\textsc{Eval-pat-cstr-1}\quad \frac{v \text{ not of the form } C(v_1, ..., v_n)}{v \triangleright C\ (p_1, ..., p_n) \Downarrow_{\text{pat}} \text{Mismatch}}$$

$$\textsc{Eval-pat-cstr-2}\quad \frac{v = C(v_1, ..., v_n) \qquad \exists i.\ v_i \triangleright p_i \Downarrow_{\text{pat}} \text{Mismatch}}{v \triangleright C\ (p_1, ..., p_n) \Downarrow_{\text{pat}} \text{Mismatch}}$$

$$\textsc{Eval-pat-cstr-3}\quad \frac{v = C(v_1, ..., v_n) \qquad \forall i.\quad v_i \triangleright p_i \Downarrow_{\text{pat}} \text{Match}\ M_i \qquad \forall i \neq j.\ M_i \# M_j}{v \triangleright C\ (p_1, ..., p_n) \Downarrow_{\text{pat}} \text{Match}\ (\bigcup_i M_i)}$$

$$\textsc{Eval-pat-when-1}\quad \frac{v \triangleright p \Downarrow_{\text{pat}} \text{Mismatch}}{v \triangleright p\ \textbf{when}\ b \Downarrow_{\text{pat}} \text{Mismatch}}$$

$$\textsc{Eval-pat-when-2}\quad \frac{v \triangleright p \Downarrow_{\text{pat}} \text{Match}\ M \qquad \text{Subst}(M, b) \Downarrow_{\text{bbe}} \text{Mismatch}}{v \triangleright p\ \textbf{when}\ b \Downarrow_{\text{pat}} \text{Mismatch}}$$

$$\textsc{Eval-pat-when-3}\quad \frac{v \triangleright p \Downarrow_{\text{pat}} \text{Match}\ M_1 \qquad \text{Subst}(M_1, b) \Downarrow_{\text{bbe}} \text{Match}\ M_2 \qquad M_1 \# M_2}{v \triangleright p\ \textbf{when}\ b \Downarrow_{\text{pat}} \text{Match}\ (M_1 \uplus M_2)}$$

$$\textsc{Eval-pat-restrict}\quad \frac{v \triangleright p \Downarrow_{\text{pat}} r}{v \triangleright \textbf{restrict}\ V\ p \Downarrow_{\text{pat}} r_{|V}}$$

Fig. 4. Semantics of core constructs

$$\textsc{Eval-trm-let}\quad \frac{t_1 \Downarrow_{\text{trm}} v_1 \qquad v_1 \triangleright p \Downarrow_{\text{pat}} \text{Match } M \qquad \text{Subst}(M, t_2) \Downarrow_{\text{trm}} v_2}{\textbf{let } p = t_1 \textbf{ in } t_2 \ \Downarrow_{\text{trm}}\ v_2}$$

$$\textsc{Eval-trm-if-1}\quad \frac{b \Downarrow_{\text{bbe}} \text{Mismatch} \qquad t_2 \Downarrow_{\text{trm}} v}{\textbf{if}^{\text{trm}}\ b \textbf{ then } t_1 \textbf{ else } t_2 \ \Downarrow_{\text{trm}}\ v}$$

$$\textsc{Eval-trm-if-2}\quad \frac{b \Downarrow_{\text{bbe}} \text{Match } M \qquad \text{Subst}(M, t_1) \Downarrow_{\text{trm}} v}{\textbf{if}^{\text{trm}}\ b \textbf{ then } t_1 \textbf{ else } t_2 \ \Downarrow_{\text{trm}}\ v}$$

$$\textsc{Eval-bbe-trm-1}\quad \frac{t \Downarrow_{\text{trm}} \text{false}}{t \Downarrow_{\text{bbe}} \text{Mismatch}}$$

$$\textsc{Eval-bbe-trm-2}\quad \frac{t \Downarrow_{\text{trm}} \text{true}}{t \Downarrow_{\text{bbe}} \text{Match } \varnothing}$$

$$\textsc{Eval-bbe-if-1}\quad \frac{b_0 \Downarrow_{\text{bbe}} \text{Mismatch} \qquad b_2 \Downarrow_{\text{bbe}} r}{\textbf{if}^{\text{bbe}}\ b_0 \textbf{ then } b_1 \textbf{ else } b_2 \ \Downarrow_{\text{bbe}}\ r}$$

$$\textsc{Eval-bbe-if-2}\quad \frac{b_0 \Downarrow_{\text{bbe}} \text{Match } M_0 \qquad \text{Subst}(M_0, b_1) \Downarrow_{\text{bbe}} \text{Mismatch}}{\textbf{if}^{\text{bbe}}\ b_0 \textbf{ then } b_1 \textbf{ else } b_2 \ \Downarrow_{\text{bbe}}\ \text{Mismatch}}$$

$$\textsc{Eval-bbe-if-3}\quad \frac{b_0 \Downarrow_{\text{bbe}} \text{Match } M_0 \qquad \text{Subst}(M_0, b_1) \Downarrow_{\text{bbe}} \text{Match } M_1 \qquad M_0 \# M_1}{\textbf{if}^{\text{bbe}}\ b_0 \textbf{ then } b_1 \textbf{ else } b_2 \ \Downarrow_{\text{bbe}}\ \text{Match } (M_0 \uplus M_1)}$$

$$\textsc{Eval-bbe-and-1}\quad \frac{b_1 \Downarrow_{\text{bbe}} \text{Mismatch}}{b_1 \textbf{ and } b_2 \Downarrow_{\text{bbe}} \text{Mismatch}}$$

$$\textsc{Eval-bbe-and-2}\quad \frac{b_1 \Downarrow_{\text{bbe}} \text{Match } M_1 \qquad \text{Subst}(M_1, b_2) \Downarrow_{\text{bbe}} \text{Mismatch}}{b_1 \textbf{ and } b_2 \Downarrow_{\text{bbe}} \text{Mismatch}}$$

$$\textsc{Eval-bbe-and-3}\quad \frac{b_1 \Downarrow_{\text{bbe}} \text{Match } M_1 \qquad \text{Subst}(M_1, b_2) \Downarrow_{\text{bbe}} \text{Match } M_2 \qquad M_1 \# M_2}{b_1 \textbf{ and } b_2 \Downarrow_{\text{bbe}} \text{Match } (M_1 \uplus M_2)}$$

$$\textsc{Eval-bbe-or-1}\quad \frac{b_1 \Downarrow_{\text{bbe}} \text{Match } M_1}{b_1 \textbf{ or } b_2 \Downarrow_{\text{bbe}} \text{Match } M_1}$$

$$\textsc{Eval-bbe-or-2}\quad \frac{b_1 \Downarrow_{\text{bbe}} \text{Mismatch} \qquad b_2 \Downarrow_{\text{bbe}} r}{b_1 \textbf{ or } b_2 \Downarrow_{\text{bbe}} r}$$

$$\textsc{Eval-bbe-not-1}\quad \frac{b \Downarrow_{\text{bbe}} \text{Mismatch}}{\textbf{not } b \Downarrow_{\text{bbe}} \text{Match } \varnothing}$$

$$\textsc{Eval-bbe-not-2}\quad \frac{b \Downarrow_{\text{bbe}} \text{Match } M}{\textbf{not } b \Downarrow_{\text{bbe}} \text{Mismatch}}$$

$$\textsc{Eval-pat-as-1}\quad \frac{v \triangleright p \Downarrow_{\text{pat}} \text{Mismatch}}{v \triangleright p \textbf{ as } x^? \ \Downarrow_{\text{pat}}\ \text{Mismatch}}$$

$$\textsc{Eval-pat-as-2}\quad \frac{v \triangleright p \Downarrow_{\text{pat}} \text{Match } M \qquad x \notin \text{dom}(M)}{v \triangleright p \textbf{ as } x^? \ \Downarrow_{\text{pat}}\ \text{Match } (M \uplus \{x \mapsto v\})}$$

$$\textsc{Eval-pat-wild}\quad \frac{}{v \triangleright \_ \ \Downarrow_{\text{pat}}\ \text{Match } \varnothing}$$

$$\textsc{Eval-pat-view-1}\quad \frac{f(v) \Downarrow_{\text{trm}} \text{None}}{v \triangleright f(p_1, ..., p_n) \ \Downarrow_{\text{pat}}\ \text{Mismatch}}$$

$$\textsc{Eval-pat-view-2}\quad \frac{f(v) \Downarrow_{\text{trm}} \text{Some } (v_1, ..., v_n) \qquad \exists i.\ v_i \triangleright p_i \Downarrow_{\text{pat}} \text{Mismatch}}{v \triangleright f(p_1, ..., p_n) \ \Downarrow_{\text{pat}}\ \text{Mismatch}}$$

$$\textsc{Eval-pat-view-3}\quad \frac{f(v) \Downarrow_{\text{trm}} \text{Some } (v_1, ..., v_n) \qquad \forall i.\quad v_i \triangleright p_i \Downarrow_{\text{pat}} \text{Match } M_i \qquad \forall i \neq j.\ M_i \# M_j}{v \triangleright f(p_1, ..., p_n) \ \Downarrow_{\text{pat}}\ \text{Match } (\bigcup_i M_i)}$$

$$\textsc{Eval-pat-pred-1}\quad \frac{g(v) \Downarrow_{\text{trm}} \text{false}}{v \triangleright g \Downarrow_{\text{pat}} \text{Mismatch}}$$

$$\textsc{Eval-pat-pred-2}\quad \frac{g(v) \Downarrow_{\text{trm}} \text{true}}{v \triangleright g \Downarrow_{\text{pat}} \text{Match } \varnothing}$$

Fig. 5. Semantics of derived constructs

T	:=	$D\ (T_1, ..., T_n)$	type constructor (includes bool, int and "option T")
		$(T_1, ..., T_n)\ \to\ T_r$	n-ary arrow type (might also be viewed as a contructor)
E	:=	$\varnothing \mid E, x : T$	typing environment
B	:=	map from variables to types	typing of the variables bound by a BBE

Fig. 6. Typing entities

$$\textsc{Typ-trm-var}\ \frac{(x : T) \in E}{E \vdash_{\text{trm}} x : T} \qquad \textsc{Typ-trm-fun}\ \frac{E, x_1 : T_1, ..., x_n : T_n \vdash_{\text{trm}} t : T}{E \vdash_{\text{trm}} \lambda(x_1, ..., x_n).t : (T_1, ..., T_n)\ \to\ T}$$

$$\textsc{Typ-trm-app}\ \frac{E \vdash_{\text{trm}} f : (T_1, ..., T_n)\ \to\ T \qquad \forall i.\quad E \vdash_{\text{trm}} t_i : T_i}{E \vdash_{\text{trm}} f\ (t_1, ..., t_n) : T} \qquad \textsc{Typ-trm-switch}\ \frac{\forall i.\quad E \vdash_{\text{trm}} c_i : T}{E \vdash_{\text{trm}} (\mathbf{switch}^{\text{trm}} c_1 \mid ... \mid c_n) : T}$$

$$\textsc{Typ-trm-case}\ \frac{E \vdash_{\text{bbe}} b \rightsquigarrow B \qquad E, B \vdash_{\text{trm}} t : T}{E \vdash_{\text{trm}} (\mathbf{case}\ b\ \mathbf{then}\ t) : T} \qquad \textsc{Typ-bbe-is}\ \frac{E \vdash_{\text{trm}} t : T \qquad E \vdash_{\text{pat}} p : T \rightsquigarrow B}{E \vdash_{\text{bbe}} (t\ \mathbf{is}\ p)\ \rightsquigarrow\ B}$$

$$\textsc{Typ-bbe-switch}\ \frac{\forall i.\quad E \vdash_{\text{bbe}} d_i \rightsquigarrow B}{E \vdash_{\text{bbe}} (\mathbf{switch}^{\text{bbe}} d_1 \mid ... \mid d_n)\ \rightsquigarrow\ B} \qquad \textsc{Typ-bbe-restrict}\ \frac{E \vdash_{\text{bbe}} b \rightsquigarrow B \qquad V \subseteq \text{dom}(B)}{E \vdash_{\text{bbe}} (\mathbf{restrict}\ V\ b)\ \rightsquigarrow\ B_{|V}}$$

$$\textsc{Typ-bbe-case-1}\ \frac{E \vdash_{\text{bbe}} b_1 \rightsquigarrow B_1 \qquad E, B_1 \vdash_{\text{bbe}} b_2 \rightsquigarrow B_2 \qquad B_1 \# B_2}{E \vdash_{\text{bbe}} (\mathbf{case}\ b_1\ \mathbf{then}\ b_2)\ \rightsquigarrow\ B_1 \uplus B_2} \qquad \textsc{Typ-bbe-case-2}\ \frac{E \vdash_{\text{bbe}} b_1 \rightsquigarrow B_1}{E \vdash_{\text{bbe}} (\mathbf{case}\ b_1\ \mathbf{then}\ \text{false})\ \rightsquigarrow\ B}$$

$$\textsc{Typ-bbe-case-restrict}\ \frac{E \vdash_{\text{bbe}} d \rightsquigarrow B \qquad V \subseteq \text{dom}(B)}{E \vdash_{\text{bbe}} (\mathbf{restrict}\ V\ d)\ \rightsquigarrow\ B_{|V}} \qquad \textsc{Typ-pat-var}\ \frac{}{E \vdash_{\text{pat}} x^{?}\ :\ T\ \rightsquigarrow\ \{x : T\}}$$

$$\textsc{Typ-pat-or}\ \frac{E \vdash_{\text{pat}} p_1 : T \rightsquigarrow B \qquad E \vdash_{\text{pat}} p_2 : T \rightsquigarrow B}{E \vdash_{\text{pat}} (p_1 \mid p_2)\ :\ T\ \rightsquigarrow\ B} \qquad \textsc{Typ-pat-and}\ \frac{E \vdash_{\text{pat}} p_1 : T \rightsquigarrow B_1 \qquad E \vdash_{\text{pat}} p_2 : T \rightsquigarrow B_2 \qquad B_1 \# B_2}{E \vdash_{\text{pat}} (p_1\ \&\ p_2)\ :\ T\ \rightsquigarrow\ B_1 \uplus B_2}$$

$$\textsc{Typ-pat-cstr}\ \frac{E \vdash_{\text{trm}} C : (T_1, ..., T_n) \to T \qquad \forall i.\quad E \vdash_{\text{pat}} p_i : T_i \rightsquigarrow B_i \qquad \forall i \neq j.\ B_i \# B_j}{E \vdash_{\text{pat}} C\ (p_1, ..., p_n)\ :\ T\ \rightsquigarrow\ \bigcup_i B_i}$$

$$\textsc{Typ-pat-when}\ \frac{E \vdash_{\text{pat}} p : T \rightsquigarrow B_1 \qquad E, B_1 \vdash_{\text{bbe}} b \rightsquigarrow B_2 \qquad B_1 \# B_2}{E \vdash_{\text{pat}} (p\ \mathbf{when}\ b)\ :\ T\ \rightsquigarrow\ B_1 \uplus B_2} \qquad \textsc{Typ-pat-restrict}\ \frac{E \vdash_{\text{pat}} p : T \rightsquigarrow B \qquad V \subseteq \text{dom}(B)}{E \vdash_{\text{pat}} (\mathbf{restrict}\ V\ p)\ :\ T\ \rightsquigarrow\ B_{|V}}$$

Fig. 7. Typing rules for the core language

Remark: if we had tried to unify the grammar of BBEs and that of terms, we might have considered a single typing judgment for both. This judgment would be written $E \vdash_{\text{trm}} t : T$, where t could be a BBE, and where T could be a special type of the form boolbinds(B). In other words, our judgment

$$\textsc{Typ-trm-let}\quad \frac{E \vdash_{\mathsf{pat}} p : T_1 \rightsquigarrow B \qquad E \vdash_{\mathsf{trm}} t_1 : T_1 \qquad E, B \vdash_{\mathsf{trm}} t_2 : T_2}{E \vdash_{\mathsf{trm}} \mathbf{let}\ p = t_1\ \mathbf{in}\ t_2 : T_2}$$

$$\textsc{Typ-trm-if}\quad \frac{E \vdash_{\mathsf{bbe}} b \rightsquigarrow B \qquad E, B \vdash_{\mathsf{trm}} t_1 : T \qquad E \vdash_{\mathsf{trm}} t_2 : T}{E \vdash_{\mathsf{trm}} \mathbf{if}^{\mathsf{trm}}\ b\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : T}$$

$$\textsc{Typ-bbe-trm}\quad \frac{E \vdash_{\mathsf{trm}} t : \mathsf{bool}}{E \vdash_{\mathsf{bbe}} t \rightsquigarrow \varnothing}$$

$$\textsc{Typ-bbe-if}\quad \frac{E \vdash_{\mathsf{bbe}} b_0 \rightsquigarrow B_0 \qquad E, B_0 \vdash_{\mathsf{bbe}} b_1 \rightsquigarrow B_1 \qquad B_0 \# B_1 \qquad B = B_0 \uplus B_1 \qquad E \vdash_{\mathsf{bbe}} b_2 \rightsquigarrow B}{E \vdash_{\mathsf{bbe}} \mathbf{if}^{\mathsf{trm}}\ b_0\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 \rightsquigarrow B}$$

$$\textsc{Typ-bbe-and}\quad \frac{E \vdash_{\mathsf{bbe}} b_1 \rightsquigarrow B_1 \qquad E, B_1 \vdash_{\mathsf{bbe}} b_2 \rightsquigarrow B_2 \qquad B_1 \# B_2}{E \vdash_{\mathsf{bbe}} b_1\ \mathbf{and}\ b_2 \rightsquigarrow B_1 \uplus B_2}$$

$$\textsc{Typ-bbe-or}\quad \frac{E \vdash_{\mathsf{bbe}} b_1 \rightsquigarrow B \qquad E \vdash_{\mathsf{bbe}} b_2 \rightsquigarrow B}{E \vdash_{\mathsf{bbe}} b_1\ \mathbf{or}\ b_2 \rightsquigarrow B}$$

$$\textsc{Typ-bbe-not}\quad \frac{}{E \vdash_{\mathsf{bbe}} \mathbf{not}\ b \rightsquigarrow \varnothing}$$

$$\textsc{Typ-pat-as}\quad \frac{E \vdash_{\mathsf{pat}} p : T \rightsquigarrow B \qquad x \notin \mathrm{dom}(B)}{E \vdash_{\mathsf{pat}} p\ \mathbf{as}\ x^{?} : T \rightsquigarrow B \uplus \{x : T\}}$$

$$\textsc{Typ-pat-wild}\quad \frac{}{E \vdash_{\mathsf{pat}} \_ : T \rightsquigarrow \varnothing}$$

$$\textsc{Typ-pat-pred}\quad \frac{E \vdash_{\mathsf{trm}} g : T \to \mathsf{bool}}{E \vdash_{\mathsf{pat}} g : T \rightsquigarrow \varnothing}$$

$$\textsc{Typ-pat-view}\quad \frac{E \vdash_{\mathsf{trm}} f : T \to (T_1, ..., T_n)\ \mathsf{option} \qquad \forall i.\quad E \vdash_{\mathsf{pat}} p_i : T_i \rightsquigarrow B_i \qquad \forall i \neq j.\ B_i \# B_j}{E \vdash_{\mathsf{pat}} f(p_1, ..., p_n) : T \rightsquigarrow \bigcup_i B_i}$$

Fig. 8. Typing rules for the derived constructs

$E \vdash_{\mathsf{bbe}} b \rightsquigarrow B$ would be encoded as $E \vdash_{\mathsf{trm}} b : \mathsf{boolbinds(B)}$. By collapsing BBEs and terms also in the evaluation rules would bring us closer to the presentation used by *first-class* patterns.

6 TYPE SOUNDNESS

For a type system, one usually establishes *preservation* and *progress*. Establishing the progress property for pattern-matching crucially depends on exhaustivity. Yet, as said earlier, this property is hard to obtain in the presence of arbitrary guards and view functions. Hence, we will here focus solely on type preservation, which asserts that if a term t has type T, and if t evaluates to a value v, then v has type T.

Because the grammars of terms, of BBEs, and of patterns are mutually recursive, we need to also establish, as part of a proof by mutual induction, preservation statements for BBEs and for patterns. The preservation statement for BBEs is as follows. If a BBE b exports a set of bindings B (described as a map from variables to types), and if b evaluates positively to a result of the form Match M (where M maps variables to values), then this set of result M contains exactly the entries advertised by B, at the appropriate types. Formally, we write $\vdash_{\mathsf{map}} M : B$ this relation, which asserts that if B binds a name x to a type T, then M binds the same name x to a value of type T.

$$\vdash_{\mathsf{map}} M : B \quad := \quad \mathrm{dom}(M) = \mathrm{dom}(B) \ \wedge\ \forall\,(x, T) \in B.\ \exists v.\ (x, v) \in M \ \wedge\ \vdash_{\mathsf{trm}} v : T$$

The preservation statement for patterns is as follows. If a pattern p filters values of type T and exports a set of bindings B, and if v is a value of type T, and if the evaluation of the pattern p against the value v evaluates positively to a result of the form Match M, then this set of result M contains the entries advertised by B, again in the sense that $\vdash_{\mathsf{map}} M : B$ holds.

In the formal statements shown below, we omit the empty typing environments, e.g., we write $\vdash_{\mathsf{trm}} t : T$ for $\varnothing \vdash_{\mathsf{trm}} t : T$.

Theorem 1 (Type Preservation). *The following statements hold:*

(1) $(t \Downarrow_{trm} v) \wedge (\vdash_{trm} t : T) \Rightarrow (\vdash_{trm} v : T)$
(2) $(b \Downarrow_{bbe} \mathit{Match}\, M) \wedge (\vdash_{bbe} b \rightsquigarrow B) \Rightarrow (\vdash_{map} M : B)$
(3) $(v \triangleright p \Downarrow_{pat} \mathit{Match}\, M) \wedge (\vdash_{pat} p : T \rightsquigarrow B) \wedge (\vdash_{trm} v : T) \Rightarrow (\vdash_{map} M : B)$

We have carried out the proof on paper. Our proof goes by induction on the evaluation judgments. The proof depends on the following substitution lemmas.

Lemma 1 (Substitution lemmas). *Assume* $\vdash_{map} M : B$. *Then, the following implications hold:*

(1) $(E, B \vdash_{trm} t : T) \Rightarrow (E \vdash_{trm} \mathit{Subst}(M, t) : T)$
(2) $(E, B \vdash_{bbe} b \rightsquigarrow B') \Rightarrow (E \vdash_{bbe} \mathit{Subst}(M, b) \rightsquigarrow B')$
(3) $(E, B \vdash_{pat} p : T \rightsquigarrow B') \Rightarrow (E \vdash_{pat} \mathit{Subst}(M, p) : T \rightsquigarrow B')$

We look forward to formalizing our definitions and proofs using Rocq.

REFERENCES

Luyu Cheng and Lionel Parreaux. 2024. The Ultimate Conditional Syntax. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 306 (Oct. 2024), 30 pages. https://doi.org/10.1145/3689746

H Cirstea and K Kirchner. 2001. The rewriting calculus - part I. *Logic Journal of the IGPL* 9, 3 (2001), 339–375. https://doi.org/10.1093/jigpal/9.3.339

ELanguage. 2025. E-Language, Pattern Grammar. http://www.erights.org/elang/index.html

Georges Gonthier and Stéphane Le Roux. 2009. An Ssreflect Tutorial. https://rocq-prover.org/doc/V8.9.1/refman/proof-engine/ssreflect-proof-language.html#gallina-extensions

Barry Jay and Delia Kesner. 2009. First-class patterns. *J. Funct. Program.* 19, 2 (March 2009), 191–225.

C. Barry Jay. 2004. The pattern calculus. *ACM Trans. Program. Lang. Syst.* 26, 6 (Nov. 2004), 911–937. https://doi.org/10.1145/1034774.1034775

Daniel Licata and Simon Peyton Jones. 2025. View patterns: lightweight views for Haskell. https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/view_patterns.html

Chris Okasaki. 1998. Views for standard ML. In *SIGPLAN Workshop on ML*. Citeseer, 14–23. https://www.cs.tufts.edu/~nr/cs257/archive/chris-okasaki/views.pdf

ppxlib. 2025. The Ast-pattern Module. https://ocaml-ppx.github.io/ppxlib/ppxlib/Ppxlib/Ast_pattern/index.html

Rust. 2025a. Guard Patterns. https://rust-lang.github.io/rfcs//3637-guard-patterns.html

Rust. 2025b. If let chains. https://rust-lang.github.io/rfcs/2497-if-let-chains.html

Scala. 2025. Extractors Objects. https://docs.scala-lang.org/fr/tour/extractor-objects.html

Don Syme, Gregory Neverov, and James Margetson. 2007. Extensible pattern matching via a lightweight language extension. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 29–40. https://doi.org/10.1145/1291151.1291159

Mark Tullsen. 2000. First Class Patterns. In *Proceedings of the Second International Workshop on Practical Aspects of Declarative Languages (PADL '00)*. Springer-Verlag, Berlin, Heidelberg, 1–15.

Unison. 2025. Guard Patterns. https://www.unison-lang.org/docs/fundamentals/control-flow/pattern-matching/#guard-patterns

Wolfram. 2025. Conditions. https://reference.wolfram.com/language/ref/Condition.html