Functional Pearl: Binding Boolean Expressions and Extended Pattern Matching

ARTHUR CHARGUÉRAUD and YANNI LEFKI, Inria & ICube lab, CNRS, University of Strasbourg, France

Functional programming languages include a variety of pattern matching features, including guarded patterns, active patterns (views), predicate patterns, etc. Certain languages also include mechanisms for binding names as part of the boolean expressions that appear in if-statement, while-conditions, or pattern guards. These features are all very useful, yet no mainstream language supports them all at once. In this work, we present a language that unifies all these constructs. Unlike in Cheng and Parreaux' proposal [2024], our typing and evaluation rules are stated directly on the user-level syntactic constructs (that is, not through an encoding), faithfully following the standard presentation style; moreover, we do cover pattern disjunction. This paper is intended for language and compiler designers, as well as for pedagogical purposes. It is focused on expressiveness and conciseness, leaving out discussions about exhaustiveness checking and compiler optimizations.

1 INTRODUCTION

High-level history of pattern matching. Pattern matching on data structures is a decisive feature of programming languages, that originates in Edinburgh ML (1973-1975) [Milner 1978]. This feature particularly shines for manipulating tree-based data structures, in particular abstract syntax trees (ASTs). Pattern matching has been popularized by languages such as Caml [Leroy and Mauny 1993] (developed since 1985), Standard ML [Milner 1984], Miranda [Turner 1985] and Haskell (released 1990) [Hudak et al. 1992]. It has been subsequently adopted in Scheme (via macros) [Wright and Cartwright 1994], Scala [Odersky et al. 2004], F# (released 2005) [Syme et al. 2007a], and Rust [Matsakis and Klock 2014], in particular.

Improving expressiveness with views. ML patterns allows the programmer to match a value against explicit data constructors. In their original form, however, they did not support pattern matching against an abstract data type, whose implementation is hidden to the client. To reconcile pattern matching with data abstraction, Wadler [1987] introduced the concept of *views*. One application of views is the possibility to define the inverse of a *smart constructor*, which is a helper function for building a data type. For example, if an AST smart constructor named `trm_and` is defined as `fun t1 t2 -> trm_if t1 t2 (trm_bool false)`, then one can define a *view* for matching terms of this form inside patterns. If we name this view `trm_and` (in a different namespace), then we can write, e.g.:

```
match t with | trm_and t1 (trm_and t2 t3)-> cont.
```

Early implementations of views include Miranda [Turner 1985], Scheme [Wright and Cartwright 1994], Standard ML [Okasaki 1998], and Haskell [Licata and Peyton Jones 2007].

Views are available under the name *extractor* in Scala [2025], and *active pattern* [Erwig 1996] in F# [Syme et al. 2007b].[1]

Improving expressiveness with guards. The language KRC introduces *guards* [Turner 1981], which essentially correspond to OCaml's *when* clauses. The Miranda language [Turner 1986] extended the syntax of guards. In 1995, Caml v0.7 introduces the *when* syntax in pattern matching. Haskell [Licata and Peyton Jones 2007] (and previously a GHC extension) generalizes when-clauses with *pattern-guards*, which take the form `pat1 | pat2 <- exp2`. Consider, in a hypothetical syntax: `match exp1 with | (pat1 | pat2 <- exp2)-> cont`. The continuation `cont` is executed if `exp1` matches `pat1` and `exp2` matches `pat2`. The bindings from `pat1` scope into `exp2`, and both the bindings from `pat1` and `pat2` scope into the continuation `cont`. Likewise, the Successor-ML project (2001-2005) presents a *pattern-with* construct, written `pat1 with pat2 = exp2`.

Improving expressiveness of conditionals. The construct `if exp is pat then .. else ..` provides an convenient syntax for testing one specific pattern. A typical example is `if (e is Some x)then f x else g`. which appears in Rocq's SSReflect extension [Gonthier and Le Roux 2009]. 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 Rust RFC Rust [2025] introduces the *if-let* construct, which goes further in allowing to chain pattern matching tests inside conditionals, in the form:

```
if let pat1 = exp1 && let pat2 = exp2 then t1 else t2
```

where the variables bound by `pat1` are available in `exp2`, and where variables bound both by `pat1` and `pat2` are available in the then-branch `t1`. Note that `exp1` and `exp2` can evaluate arbitrary expressions, like with Haskell guards. The authors of the Rust RFC point out that such patterns have previously appeared in particular in the Unison language [Unison 2025], in Wolfram's language [2025], as well as in the E-language under the name *such-that* pattern [2025]; (these citations point to the reference manuals). The language C# supports the syntax `if exp1 is pat1 && exp2 is pat2`, yet, as of 2025, the bound variables from `pat1` do not scope into `exp2`.

In the general form, `if b then t1 else t2`, we call `b` a *binding-boolean-expression* (BBE). Beyond if-statements, it would also make sense to exploit binding-boolean-expressions in conditions of while-loops. For example, to iteratively pop and process the elements of a queue `q`, one could write: `while pop_opt q is Some x do f x done`. We are not aware of existing language supporting this feature.

Improving expressiveness with backtracking. When pushed to the extreme, the ability to interleave pattern matching evaluation with arbitrary computations, before deciding whether a branch should be entered, shares similarities with a form of *backtracking pattern matching*. Such backtracking mechanisms play a central role in languages from Prolog family (1972). Backtracking pattern matching can also be useful to implement tactics in

[1] Views can be also be encoded by means of the more general notion of *first-class patterns* [Jay and Kesner 2009; Tullsen 2000]. 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].

interactive theorem provers, as with Rocq's Ltac [Delahaye 2000]. We are not aware of languages that includes standard ML-style pattern matching augmented with an explicit language construct for aborting the execution of a branch, and moving on to the next available pattern.

Improving readability with nonlinear patterns. Another feature inherited from Prolog-style languages is the ability to have multiple occurences of same variable in a pattern. For example, a rewrite rule in Ltac might be: `match t with trm_add ?x ?x -> trm_mul 2 x`, where pattern variables are materialized using a question mark. In ML, when only linear patterns are available, one must write the pattern in the form `trm_add x y when x = y`, which severely impedes readability when scaling up to larger examples. Readability can be improved by means of syntactic sugar, as done for example by Servadei [2017] in a Haskell extension. A limitation of Prolog-style nonlinear patterns is that they do not make explicit which comparison function should be used to compare the occurrences; invoking the default structural equality is not always appropriate. A possibility is to write the pattern `trm_add ?x (eq x)`, where `?x` denotes a pattern variable bound in the rest of the pattern, and where `eq x` denotes a partial application of a specific equality operator (e.g. `=`, `==`, `Set.eq`, etc.).[2] This possibility is technically expressible in the pattern language of Cheng and Parreaux [2024], however the authors do not advertise for this possibility.

Towards a unified presentation. All the aforementioned features are very useful in practice. Yet, no mainstream programming language supports them all at once. A first attempt towards a unifying presentation of pattern matching and conditionals has been proposed by Cheng and Parreaux [2024]. They present a surface language that is based on the notion of *splits*. A split consists of a series of branches of the same kind, interspersed with let-bindings, and optionally ended with a default term. Splits appear in the grammars of terms, of patterns, and of operators (including "`is`" and arithmetic operators), which are mutually recursive. The authors define a core language and explain how to translate most existing features into that core language. They equip this core language with a type system. They also provide a compilation scheme towards a lower-level language without any backtracking construct.

Contribution. The present paper aim to go further in three directions.

(1) We present evaluation rules and typing rules stated directly on the user-level language. We argue that programmers greatly benefit from understanding the semantics and the type errors directly in terms of the constructs that they write.
(2) Our user-level language is described by three simple mutually-recursive grammars: one for terms, one for patterns, and one for *binding-boolean-expressions*. We believe that it is conceptually simpler than Cheng and Parreaux's grammar based on *splits*, and closer to the standard presentation of patterns in ML languages.

[2] On the one hand, one might complain that in `trm_add ?x (eq x)` breaks the visual symmetry between the two arguments of `trm_add`. On the other hand, one might argue that, operationally speaking, the intent is perfectly clear: the first `x` is unconditionally "acquired", whereas the second `x` is "compared" against the first.

(3) We include support for pattern disjunction, and for an explicit backtracking construct, called **next**, from within the continuation of a branch. These two features were absent from Cheng and Parreaux [2024].

Like Cheng and Parreaux, we present a compilation scheme for eliminating the pattern language extensions into a tiny core language, easing the task for a compiler implementor who may wish to integrate the proposed features. We compile our more general construct **next** using standard exceptions.

We developed an OCaml-based prototype implementation for typechecking and compiling code written in our unifying language.[3] Our tool accepts an extended OCaml syntax as input—technically, valid OCaml syntax with attributes and ad-hoc functions. It can either perform full typechecking, or only apply a lightweight check to verify that every variable occurrence has a corresponding binding in scope. It can produce either text-based, conventional OCaml syntax, or an OCaml parse-tree.

Overall, our prototype can be used in either of two ways:

- either as a standalone tool that parses extended OCaml syntax, performs full typechecking, then produce a text-based standard OCaml source code;
- or as a PPX extension for OCaml that accepts extended OCaml syntax, performs lightweight scope verification, then eliminates the extended language constructs, before the OCaml compiler proceeds with its standard compilation chain.

Either way, at the expense of a slightly verbose syntax, an OCaml programmer can readily experiment with our extended grammar of pattern matching and binding-boolean-expressions. Although the prototype implementation is not the central contribution of this paper, the development of the prototype confirmed that our typing and compilation rules are straightforward to implement.

Contents of this paper.

- In Section 2, we begin with a presentation of all the core pattern matching features except backtracking features. We present syntax, typing rules, and evaluation rules.
- In Section 3, we present a construct called **next** to support explicit backtracking. We show how to integrate it in the language in a similar way as exit-blocks and its variants: **exit**, **return**, **break**, and **continue**. For the backtracking extensions, we also present syntax, typing rules, and evaluation rules.
- In Section 4, we present a *minimal language* in which all other features can be encoded. This core language is aimed for compiler implementers, who can exploit the encoding to minimize the number of constructs to handle.
- In Section 5, we show how to compile our minimal language, which can express all features, into a Core-ML language for which we assume traditional conditionals, pattern matching for testing only a single constructor at a time, and catchable exceptions.
- In the appendix, we present a preservation-and-progress type soundness proof.

[3]Our prototype does not currently include compilation schemes for **break**, **continue** and **return**. We have planned to support these features, which are orthogonal to the main focus of the paper, in the near future.

Beyond the scope of this paper. This paper does not cover the question of how to integrate optimizations in the process of compiling pattern matching. The reference work of Maranget [2007] in OCaml remains applicable for simple patterns. However, as soon as patterns involve the evaluation of arbitrary expressions, a fine-grained effect analysis would be required to ensure that modifications to the evaluation order are semantics-preserving. Such analyses might be feasible, e.g., by means of exploiting Rust-style types or a Separation Logic, but are far beyond the scope of the present paper.

Likewise, this paper does not cover the question of checking exhaustiveness of patterns. Maranget's work [2007] provides an efficient procedure for traditional patterns without any guard. Recent work [Moser et al. 2025] shows how to check exhaustiveness in the presence of guards that consist of simple boolean expressions, by leveraging SMT solvers. For similar simple guards, the CFML verification framework [Charguéraud 2011] for OCaml code supports interactive proofs of exhaustiveness in Rocq. To handle the general case of arbitrary guards, one would need (1) to equip every function involved in the guard with a formal specification, and (2) to leverage a sufficiently expressive program logic, adapted to an expressive language of patterns such as the one presented in this paper. We leave these investigations to future work.

2 A LANGUAGE WITH BINDING BOOLEAN EXPRESSIONS

In this section, we introduce our λ-calculus with binding boolean expressions (BBEs) and extended patterns, without incorporating backtracking features at this stage. We first present the syntax and typing rules, then describe the evaluation rules.

2.1 Syntax

Figure 1 presents the grammar of our language. Terms, written t, include: variables, numbers, primitive functions, n-ary function definitions, n-ary function calls, let-bindings that may bind a pattern, conditional constructs for testing a BBE, while loops guarded by a BBE, as well as the *switch* and *match* constructs. Data constructors are viewed as primitive functions. They include *tt* (the unit value), true and false, as well as None and Some v. The **switch** corresponds to a cascade of if-statements, but with its branches presented in a symmetric way. Each branch is of the form **case** b **then** t, where b is a BBE and t a continuation. The **match** construct correspond to the usual ML pattern matching, and is merely syntactic sugar for a **switch** exploiting the **is** construct.

$$\begin{array}{ll} & \textbf{match}\ t_0\ \textbf{with}\ |\ p_1 \to t_1\ |\ \ldots\ |\ p_n \to t_n \\ \equiv & \textbf{let}\ x = t_0\ \textbf{in}\ \ \textbf{switch}\ |\ \textbf{case}\ (x\ \textbf{is}\ p_1)\ \textbf{then}\ t_1\ |\ \ldots\ |\ \textbf{case}\ (x\ \textbf{is}\ p_n)\ \textbf{then}\ t_n \end{array}$$

The *binding-boolean-expressions* (BBE) include: boolean terms, tests of the form "t **is** p", as well as conjunction, disjunction and negation operators. Interestingly, in a conjunction "b_1 **and** b_2", the variables bound by b_1 scope into b_2. As discussed further, our typing rules require the pattern variables appearing in b_1 to be disjoint from those appearing in b_2. A disjunction b_1 **or** b_2 binds the variables that are bound both in b_1 and b_2. A negation **not** b performs a test but binds no variable.

Terms

$t :=$	x	variable
	n	integer
	π	primitive function
	$\lambda(x_1, ..., x_n).\ t$	function definition
	$t_0\ (t_1, ..., t_n)$	function application
	let $p = t_1$ **in** t_2	let-binding
	if b_0 **then** t_1 **else** t_2	conditional
	while b **do** t **done**	loop
	switch $\mid$ **case** b_1 **then** $t_1 \mid ... \mid$ **case** b_n **then** t_n	symmetric syntax for cascaded ifs
	match t_0 **with** $\mid p_1 \to t_1 \mid ... \mid p_n \to t_n$	particular case of **switch**

Binding boolean expressions (BBE)

$b :=$	t	term of boolean type
	t **is** p	matching against a pattern
	b_1 **and** b_2	BBE conjunction
	b_1 **or** b_2	BBE disjunction
	not b	BBE negation

Patterns

$p :=$	$x^?$	pattern variable
	$\_$	wildcard pattern
	n	integer pattern
	p_1 & p_2	pattern intersection
	$p_1 \mid p_2$	pattern disjunction
	$\neg\ p$	pattern negation
	p **as** $x^?$	alias pattern
	p **when** b	guarded pattern
	g	predicate pattern
	$t_0\ (p_1, ..., p_n)$	view pattern

Fig. 1. Syntax

The grammar of patterns includes the traditional constructs, namely variables written $x^?$, wildcard patterns, numbers, *alias-pattern* written p **as** $x^?$, as well as intersection, disjunction, and negation, We next describe the three additional pattern constructs.

The *guarded pattern*, written p **when** b, is satisfied by a value v if v matches p and the BBE b evaluates to true. Importantly, the variables bound by p scope inside b. The whole pattern p **when** b binds the variables from both p and b. (Here again, typing rules enforce disjointness of the pattern variables.) Compared with OCaml's when-clauses, our guarded patterns can appear arbitrarily deep inside patterns, moreover the guard is not just a boolean expression, but a *binding* boolean expression that may bind variables.

$$
\begin{array}{lcll}
T & := & \mid\ (T_1, ..., T_n) \to T_r & \text{type of a } n\text{-ary function} \\
 & & \mid\ D\ (T_1, ..., T_n) & \text{type constructor (e.g., unit, bool, int and option)} \\
E, \Sigma & := & \mid\ \varnothing \mid\ E, x : T & \text{typing environment} \\
B & := & \text{map from variables to types, giving the types of the bindings of a BBE} &
\end{array}
$$

Fig. 2. Typing entities

The *predicate pattern*, written g, where g is a function of type $T \to$ bool, filters values of type T. For example, a partially applied equality `(>= 0)` is a predicate pattern filtering nonnegative integers. A predicate pattern binds no variable.

Last, the *view pattern* takes the form $t_0\ (p_1, ..., p_n)$, where t_0 is a function and p_i are subpatterns. The term t_0 has type "$T \to (T_1, ..., T_n)$ option", where T_i denotes the type of the pattern p_i. A value v satisfies the view pattern $t_0\ (p_1, ..., p_n)$ if and only if $f(v)$ evaluates to a result of the form Some $(v_1, ..., v_n)$, where each v_i satisfies the subpattern p_i. Note that each subpattern p_i scope in the subpatterns to its right—recall from the introduction the example "trm_and $(x^?, (= x))$".

2.2 Typing Rules

Figure 2 presents the grammar of types and typing environments. The n-ary function type is written $(T_1, ..., T_n) \to T_r$. Other types, such as unit, bool, int and option, are all viewed as type constructors of the form $D\ (T_1, ..., T_n)$. Typing environment, written E, associates variables to types. The global typing environment Σ gives the types of primitive functions, in particular of data constructors.

We let the metavariable B denote a description of the types of the variables bound by a pattern or by a binding-boolean-expression. A map B can be viewed as a typing environment by fixing an arbitrary order. In particular, certain typing rules, presented next, involve contexts of the form "E, B".

There are 3 typing judgments. The judgment $E \vdash_{\text{trm}} t : T$ asserts that, in the environment E, the term t admits the type T. The judgment $E \vdash_{\text{bbe}} b \rightsquigarrow B$ 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 $E \vdash_{\text{pat}} p : T \rightsquigarrow B$ 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.

Typing of terms. Figure 3 presents the typing rules for terms. The rule TYP-TRM-LET illustrates well the interplay of the judgments. Consider a term **let** $p = t_1$ **in** t_2 typechecked in an environment E. The pattern p is typed with the judgment $E \vdash_{\text{pat}} p : T_1 \rightsquigarrow B$, where T_1 corresponds to the type of t_1. The continuation t_2 is typechecked in an environment "E, B", which corresponds to E extended with all the bindings exported by p.

The specificity of binding-boolean-expressions appears in the rule TYP-TRM-IF. Consider a term **if** b **then** t_1 **else** t_2 in an environment E. Its condition is typechecked as $E \vdash_{\text{bbe}} b \rightsquigarrow B$. The bindings from B are then involved in the typechecking of t_1—but not in that of t_2.

$$\textsc{typ-trm-var}\ \frac{(x : T) \in E}{E \vdash_{\text{trm}} x : T} \qquad \textsc{typ-trm-int}\ \frac{}{E \vdash_{\text{trm}} n : \text{int}} \qquad \textsc{typ-trm-prim}\ \frac{(\pi : T) \in \Sigma}{E \vdash_{\text{trm}} \pi : T}$$

$$\textsc{typ-trm-fun}\ \frac{x_1 : T_1, ..., x_n : T_n \vdash_{\text{trm}} t : T}{E \vdash_{\text{trm}} \lambda(x_1, ..., x_n).\, t : (T_1, ..., T_n) \ \to\ T} \qquad \textsc{typ-trm-app}\ \frac{E \vdash_{\text{trm}} t_0 : (T_1, ..., T_n) \ \to\ T \qquad \forall i.\quad E \vdash_{\text{trm}} t_i : T_i}{E \vdash_{\text{trm}} t_0\ (t_1, ..., t_n) : T}$$

$$\textsc{typ-trm-let}\ \frac{E \vdash_{\text{pat}} p : T_1 \rightsquigarrow B \qquad E \vdash_{\text{trm}} t_1 : T_1 \qquad E, B \vdash_{\text{trm}} t_2 : T_2}{E \vdash_{\text{trm}} \textbf{let}\ p = t_1\ \textbf{in}\ t_2 : T_2} \qquad \textsc{typ-trm-if}\ \frac{E \vdash_{\text{bbe}} b \rightsquigarrow B \qquad E, B \vdash_{\text{trm}} t_1 : T \qquad E \vdash_{\text{trm}} t_2 : T}{E \vdash_{\text{trm}} \textbf{if}\ b\ \textbf{then}\ t_1\ \textbf{else}\ t_2 : T}$$

$$\textsc{typ-trm-while}\ \frac{E \vdash_{\text{bbe}} b \rightsquigarrow B \qquad E, B \vdash_{\text{trm}} t : \text{unit}}{E \vdash_{\text{trm}} \textbf{while}\ b\ \textbf{do}\ t\ \textbf{done} : \text{unit}}$$

$$\textsc{typ-trm-switch}\ \frac{\forall i \in [1, n].\quad E \vdash_{\text{bbe}} b_i \rightsquigarrow B_i \ \ \wedge\ \ E, B_i \vdash_{\text{trm}} t_i : T}{E \vdash_{\text{trm}} \textbf{switch}\ |\ (\textbf{case}\ b_1\ \textbf{then}\ t_1)\ |\ ...\ |\ (\textbf{case}\ b_n\ \textbf{then}\ t_n) : T}$$

$$\textsc{typ-trm-match}\ \frac{E \vdash_{\text{trm}} t_0 : T_0 \qquad \forall i \in [1, n].\quad E \vdash_{\text{pat}} p_i : T_0 \rightsquigarrow B_i \ \ \wedge\ \ E, B_i \vdash_{\text{trm}} t_i : T}{E \vdash_{\text{trm}} \textbf{match}\ t_0\ \textbf{with}\ |\ p_1 \to t_1\ |\ ...\ |\ p_n \to t_n : T}$$

Fig. 3. Typing rules for terms

Likewise, in TYP-TRM-WHILE, the bindings exported by the loop condition scope inside the loop body.

The rules for **switch** and **match** typecheck each of the branches independently. Note that a **switch** or a **match** with zero branches admits any type; depending on how the semantics is formalized, the evaluation of **switch** or a **match** with no matching clause can either lead to a stuck term or to the raising of an exception.

Typing of BBEs. Figure 4 presents the typing rules for BBEs. TYP-BBE-TRM asserts that any term of type bool can be used where a BBE is expected. TYP-BBE-IS simply asserts that in t **is** p, the type of t should correspond to the type of p, and the exported bindings are those of p. The binding rules for BBE conjunction, disjunction and negation were described earlier on when presenting the syntax of BBEs. The role of the premise $\text{pv}(b_1) \cap \text{pv}(b_2) = \varnothing$ is explained near the end of the present section.

Figure 5 presents the typing rules for patterns. The scopes of bindings are as described when presenting the grammar. The rule TYP-PAT-VIEW typechecks a view pattern $t_0\ (p_1, ..., p_n)$ in an environment E, where t_0 has type $T \to (T_1, ..., T_n)$ option. Each p_i has type T_i and is typechecked in the environment E extended with the bindings exported by the subpatterns p_j that precedes it.

$$\textsc{typ-bbe-trm}\quad \frac{E \vdash_{\text{trm}} t : \text{bool}}{E \vdash_{\text{bbe}} t \rightsquigarrow \varnothing} \qquad \textsc{typ-bbe-is}\quad \frac{E \vdash_{\text{trm}} t : T \qquad E \vdash_{\text{pat}} p : T \rightsquigarrow B}{E \vdash_{\text{bbe}} (t \textbf{ is } p) \rightsquigarrow B}$$

$$\textsc{typ-bbe-and}\quad \frac{E \vdash_{\text{bbe}} b_1 \rightsquigarrow B_1 \qquad E, B_1 \vdash_{\text{bbe}} b_2 \rightsquigarrow B_2 \qquad \text{pv}(b_1) \cap \text{pv}(b_2) = \varnothing}{E \vdash_{\text{bbe}} b_1 \textbf{ and } b_2 \rightsquigarrow B_1 \uplus B_2}$$

$$\textsc{typ-bbe-or}\quad \frac{E \vdash_{\text{bbe}} b_1 \rightsquigarrow B_1 \qquad E \vdash_{\text{bbe}} b_2 \rightsquigarrow B_2}{E \vdash_{\text{bbe}} b_1 \textbf{ or } b_2 \rightsquigarrow B_1 \cap B_2} \qquad \textsc{typ-bbe-not}\quad \frac{E \vdash_{\text{bbe}} b \rightsquigarrow B}{E \vdash_{\text{bbe}} \textbf{not } b \rightsquigarrow \varnothing}$$

Fig. 4. Typing rules for BBEs

Freshness conditions for pattern variables. We let $\text{pv}(e)$ denotes the set of pattern variables syntactically occurring inside an entity e, but not recursively inside terms, in other words, $\text{pv}(t) = \varnothing$, and for BBE and patterns if $x^?$ occurs in e, then x belongs to $\text{pv}(e)$.

The formal definition is given in the appendix, Figure 25. As we prove in Section C, if a BBE b has type B, and if the evaluation of b produces a map M relating variables to values, then we have: $\text{dom}(B) \subseteq \text{dom}(M) \subseteq \text{pv}(b)$.

The typing rule for a BBE conjunction p_1 **and** p_2 includes the premise $\text{pv}(p_1) \cap \text{pv}(p_2) = \varnothing$. To see why this premise plays a crucial role, consider the pattern shown below, which is of the form $(p_{11} \mid p_{12})$ & p_2, with the pattern variable y bound on both sides of the conjunction—precisely the situation ruled out by the premise $\text{pv}(p_1) \cap \text{pv}(p_2) = \varnothing$.

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

The typing rule for pattern disjunction involves an intersection. Hence, from a typing perspective, the pattern $\big((x^?, y^?) \textbf{ when } y > 1\big) \mid (x^?, 0)$ binds the variable x but not the variable y. A difficulty arises from the fact that, from an execution perspective, this same disjunction pattern may export bindings for both x and y.

To see why, consider the evaluation of this pattern against the pair $(2, 5)$ using a naive interpreter. The evaluation of the sub-pattern "$(x^?, y^?)$ **when** $y > 1$" succeeds, resulting in a map $\{x \mapsto 2,\ y \mapsto 5\}$. Because the left-branch of the or-pattern succeeds, the sub-pattern "$(x^?, 0)$" is not evaluated at all. Then, the evaluation of the sub-pattern $(y^?, \_)$ succeeds, resulting in a map $\{y \mapsto 2\}$. At this point, a naive interpreter lacks sufficient information to merge $\{x \mapsto 2,\ y \mapsto 5\}$ with $\{y \mapsto 2\}$ and output the expected result $\{x \mapsto 2,\ y \mapsto 2\}$.

One possibility is to instrument the input programs to materialize additional operations whose purpose would be to filter out the bindings that should not be exported outside a disjunction. Another possibility is to syntactically restrict the input programs to ensure that the evaluation of pattern conjunctions never leads to conflicts on variable names. In this paper, we followed the second approach, which avoids complications in the compilation process, and prevents the programmer from writing confusing patterns.

$$\textsc{typ-pat-var}\quad \frac{}{E \vdash_{\text{pat}} x^{?} \;:\; T \;\rightsquigarrow\; \{x : T\}} \qquad \textsc{typ-pat-wild}\quad \frac{}{E \vdash_{\text{pat}} \_ \;:\; T \;\rightsquigarrow\; \varnothing} \qquad \textsc{typ-pat-int}\quad \frac{}{E \vdash_{\text{pat}} n \;:\; \mathsf{int} \;\rightsquigarrow\; \varnothing}$$

$$\textsc{typ-pat-and}\quad \frac{E \vdash_{\text{pat}} p_1 : T \rightsquigarrow B_1 \qquad E, B_1 \vdash_{\text{pat}} p_2 : T \rightsquigarrow B_2 \qquad \text{pv}(p_1) \cap \text{pv}(p_2) = \varnothing}{E \vdash_{\text{pat}} (p_1 \;\&\; p_2) \;:\; T \;\rightsquigarrow\; B_1 \uplus B_2}$$

$$\textsc{typ-pat-or}\quad \frac{E \vdash_{\text{pat}} p_1 : T \rightsquigarrow B_1 \qquad E \vdash_{\text{pat}} p_2 : T \rightsquigarrow B_2}{E \vdash_{\text{pat}} (p_1 \mid p_2) \;:\; T \;\rightsquigarrow\; B_1 \cap B_2}$$

$$\textsc{typ-pat-not}\quad \frac{E \vdash_{\text{pat}} p : T \rightsquigarrow B}{E \vdash_{\text{pat}} \neg\, p \;:\; T \;\rightsquigarrow\; \varnothing} \qquad \textsc{typ-pat-as}\quad \frac{E \vdash_{\text{pat}} p : T \rightsquigarrow B \qquad x \notin \text{pv}(p)}{E \vdash_{\text{pat}} p \;\mathbf{as}\; x^{?} \;:\; T \;\rightsquigarrow\; B \uplus \{x : T\}}$$

$$\textsc{typ-pat-when}\quad \frac{E \vdash_{\text{pat}} p : T \rightsquigarrow B_1 \qquad E, B_1 \vdash_{\text{bbe}} b \rightsquigarrow B_2 \qquad \text{pv}(p) \cap \text{pv}(b) = \varnothing}{E \vdash_{\text{pat}} p \;\mathbf{when}\; b \;:\; T \;\rightsquigarrow\; B_1 \uplus B_2} \qquad \textsc{typ-pat-pred}\quad \frac{E \vdash_{\text{trm}} g : T \to \mathsf{bool}}{E \vdash_{\text{pat}} g \;:\; T \;\rightsquigarrow\; \varnothing}$$

$$\textsc{typ-pat-view}\quad \frac{E \vdash_{\text{trm}} t_0 : T \to (T_1, ..., T_n)\ \mathsf{option} \qquad \forall i \in [1, n].\quad E, (\biguplus_{1 \le j < i} B_j) \vdash_{\text{pat}} p_i : T_i \rightsquigarrow B_i \qquad \forall i, j \in [1, n].\ i \neq j \Rightarrow \text{pv}(p_i) \cap \text{pv}(p_j) = \varnothing}{E \vdash_{\text{pat}} t_0\ (p_1, ..., p_n) \;:\; T \;\rightsquigarrow\; \biguplus_{1 \le i \le n} B_i}$$

Fig. 5. Typing rules for patterns

v	:=	$\bar{\pi}$	unapplied primitive function
	|	$\bar{n}$	integer
	|	$\bar{C}\ (v_1, ..., v_n)$	constructor (includes booleans, options, tuples)
	|	$\bar{\lambda}(x_1, ..., x_n).t$	function closure
M	:=	finite map from variable to values	
r	:=	Match M	positive result of a BBE, mapping variables to values
	|	Mismatch	negative result of a BBE

Fig. 6. Grammar of results involved in the semantics

2.3 Semantics

Figure 6 presents the grammar entities involved for semantics. We let v range over values, which include numbers, primitive functions, function closures, and data constructors. To distinguish the value constructs from the term constructs, we add an overbar. For example, we write $\bar{C}\ (v_1, ..., v_n)$ for a value constructor. We let M denote a map from variables to values. Such a map corresponds to the result of a *successful* pattern-matching or BBE evaluation. We let r denote the result of evaluating a matching or a BBE. Such a result

$$\textsc{eval-trm-int}\quad \frac{}{n \Downarrow_{\text{trm}} \bar{n}} \qquad \textsc{eval-trm-fun}\quad \frac{}{\lambda(x_1, ..., x_n).\, t \Downarrow_{\text{trm}} \bar{\lambda}(x_1, ..., x_n).t}$$

$$\textsc{eval-trm-beta}\quad \frac{\big(\forall i \in [1,n].\quad t_i \Downarrow_{\text{trm}} v_i\big) \qquad \text{Subst}(\{x_1 \mapsto v_1, ..., x_n \mapsto v_n\}, t) \Downarrow_{\text{trm}} v}{(\lambda(x_1, ..., x_n).\, t)\ (t_1, ..., t_n)\ \Downarrow_{\text{trm}}\ v}$$

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

$$\textsc{eval-trm-if-1}\quad \frac{b \Downarrow_{\text{bbe}} \text{Match } M \qquad \text{Subst}(M, t_1) \Downarrow_{\text{trm}} v}{\textbf{if } b \textbf{ then } t_1 \textbf{ else } t_2\ \Downarrow_{\text{trm}}\ v}$$

$$\textsc{eval-trm-if-2}\quad \frac{b \Downarrow_{\text{bbe}} \text{Mismatch} \qquad t_2 \Downarrow_{\text{trm}} v}{\textbf{if } b \textbf{ then } t_1 \textbf{ else } t_2\ \Downarrow_{\text{trm}}\ v}$$

$$\textsc{eval-trm-while-1}\quad \frac{b \Downarrow_{\text{bbe}} \text{Match } M \qquad (\text{Subst}(M, t);\ \textbf{while } b \textbf{ do } t \textbf{ done}) \Downarrow_{\text{trm}} v}{\textbf{while } b \textbf{ do } t \textbf{ done}\ \Downarrow_{\text{trm}}\ v}$$

$$\textsc{eval-trm-while-2}\quad \frac{b \Downarrow_{\text{bbe}} \text{Mismatch}}{\textbf{while } b \textbf{ do } t \textbf{ done}\ \Downarrow_{\text{trm}}\ tt}$$

$$\textsc{eval-trm-switch}\quad \frac{\forall i \in [1,k).\, b_i \Downarrow_{\text{bbe}} \text{Mismatch} \qquad b_k \Downarrow_{\text{bbe}} \text{Match } M \qquad \text{Subst}(M, t_k) \Downarrow_{\text{trm}} v}{\textbf{switch } |\ (\textbf{case } b_1 \textbf{ then } t_1)\ |\ ...\ |\ (\textbf{case } b_n \textbf{ then } t_n) \Downarrow_{\text{trm}} v}$$

$$\textsc{eval-trm-match}\quad \frac{t_0 \Downarrow_{\text{trm}} v_0 \qquad \textbf{match } v_0 \textbf{ with } |\ p_1 \to t_1\ |\ ...\ |\ p_n \to t_n \Downarrow_{\text{trm}} v}{\textbf{match } t_0 \textbf{ with } |\ p_1 \to t_1\ |\ ...\ |\ p_n \to t_n\ \Downarrow_{\text{trm}}\ v}$$

$$\textsc{eval-trm-match-val}\quad \frac{\forall i \in [1,k).\, v_0 \triangleright p_i \Downarrow_{\text{pat}} \text{Mismatch} \qquad v_0 \triangleright p_k \Downarrow_{\text{pat}} \text{Match } M \qquad \text{Subst}(M, t_k) \Downarrow_{\text{trm}} v}{\textbf{match } v_0 \textbf{ with } |\ p_1 \to t_1\ |\ ...\ |\ p_n \to t_n\ \Downarrow_{\text{trm}}\ v}$$

Fig. 7. Semantics of terms

is either of the form Match M, or Mismatch. The notation $M_1 \# M_2$ expresses that the two maps M_1 and M_2 have disjoint domains.

$$M_1 \# M_2 \quad := \quad \text{dom}(M_1) \cap \text{dom}(M_2) = \varnothing$$

The evaluation rules involve the operation $\text{Subst}(M, e)$, which substitutes all the bindings from M into the entity e. Such an entity can be a term, a BBE or a pattern.

To simplify the statements of the evaluation rules, we omit details about the threading of the mutable store. We give further on an example rule that explicits the store.

There are 3 evaluation judgments. The judgment $t \Downarrow_{\text{trm}} v$ asserts that the term t evaluates to the value v. The judgment $b \Downarrow_{\text{bbe}} r$ asserts that the BBE b evaluates to a result r. The judgment $v \triangleright p \Downarrow_{\text{pat}} r$ asserts that the matching of the value v against the pattern p evaluates to a result r.

Semantics of terms. Figure 7 presents the evaluation rules for terms. Let us comment on a few rules. The rule EVAL-TRM-LET illustrates well the scoping of pattern variables. Consider

$$\textsc{eval-bbe-trm-1}\ \frac{t \Downarrow_{\text{trm}} \text{false}}{t \Downarrow_{\text{bbe}} \text{Mismatch}} \qquad \textsc{eval-bbe-trm-2}\ \frac{t \Downarrow_{\text{trm}} \text{true}}{t \Downarrow_{\text{bbe}} \text{Match } \varnothing} \qquad \textsc{eval-bbe-is}\ \frac{t \Downarrow_{\text{trm}} v \qquad v \triangleright p \Downarrow_{\text{pat}} r}{t\ \textbf{is}\ p\ \Downarrow_{\text{bbe}}\ r}$$

$$\textsc{eval-bbe-and-1}\ \frac{b_1 \Downarrow_{\text{bbe}} \text{Mismatch}}{b_1\ \textbf{and}\ b_2 \Downarrow_{\text{bbe}} \text{Mismatch}} \qquad \textsc{eval-bbe-and-2}\ \frac{b_1 \Downarrow_{\text{bbe}} \text{Match } M_1 \qquad \text{Subst}(M_1, b_2) \Downarrow_{\text{bbe}} \text{Mismatch}}{b_1\ \textbf{and}\ b_2 \Downarrow_{\text{bbe}} \text{Mismatch}}$$

$$\textsc{eval-bbe-and-3}\ \frac{b_1 \Downarrow_{\text{bbe}} \text{Match } M_1 \qquad \text{Subst}(M_1, b_2) \Downarrow_{\text{bbe}} \text{Match } M_2 \qquad M_1 \# M_2}{b_1\ \textbf{and}\ b_2 \Downarrow_{\text{bbe}} \text{Match } (M_1 \uplus M_2)} \qquad \textsc{eval-bbe-or-1}\ \frac{b_1 \Downarrow_{\text{bbe}} \text{Match } M_1}{b_1\ \textbf{or}\ b_2 \Downarrow_{\text{bbe}} \text{Match } M_1}$$

$$\textsc{eval-bbe-or-2}\ \frac{b_1 \Downarrow_{\text{bbe}} \text{Mismatch} \qquad b_2 \Downarrow_{\text{bbe}} \text{Match } M_2}{b_1\ \textbf{or}\ b_2 \Downarrow_{\text{bbe}} \text{Match } M_2} \qquad \textsc{eval-bbe-or-3}\ \frac{b_1 \Downarrow_{\text{bbe}} \text{Mismatch} \qquad b_2 \Downarrow_{\text{bbe}} \text{Mismatch}}{b_1\ \textbf{or}\ b_2 \Downarrow_{\text{bbe}} \text{Mismatch}}$$

$$\textsc{eval-bbe-not-1}\ \frac{b \Downarrow_{\text{bbe}} \text{Mismatch}}{\textbf{not}\ b \Downarrow_{\text{bbe}} \text{Match } \varnothing} \qquad \textsc{eval-bbe-not-2}\ \frac{b \Downarrow_{\text{bbe}} \text{Match } M}{\textbf{not}\ b \Downarrow_{\text{bbe}} \text{Mismatch}}$$

Fig. 8. Semantics of binding-boolean-expressions

let $p = t_1$ **in** t_2. The subterm t_1 produces a value v_1. This value v_1 is tested against the pattern p. This matching is written "$v \triangleright p$". If the matching is successful, it yields a result Match M. The bindings of M are then substituted into t_2. If the matching is unsuccessful, the evaluation is stuck.

Consider now a conditional "**if** b **then** t_1 **else** t_2". Suppose that the evaluation of b produces Match M. In this case, the bindings of M are substituted into t_1 (EVAL-TRM-IF-1). Otherwise, if b produces Mismatch, the evaluation proceeds with t_2 (EVAL-TRM-IF-2).

Likewise, for a while loop (rules EVAL-TRM-WHILE-1 and EVAL-TRM-WHILE-2). The bindings produced by the condition b are substituted into the body t.

Semantics of BBEs and patterns. Figures 8 and 9 present the evaluation rules for BBEs and patterns. A crucial aspect of the semantics are the premises of the form $M_1 \# M_2$ that appear in the rules whose results build a Match $(M_1 \uplus M_2)$, such as EVAL-BBE-AND-3, EVAL-PAT-WHEN-3, EVAL-PAT-AND-3, EVAL-PAT-VIEW-2 and EVAL-PAT-VIEW-3. These premises ensure that a naive interpreter never reaches a situation where it needs to rely on typing information to determine how to merge two maps of bindings. As our type soundness proof establishes (Lemma 4), the typing premises of the form $\text{pv}(p_1) \cap \text{pv}(p_2) = \varnothing$ guarantee that during the evaluation the unions of maps of the form Match $(M_1 \uplus M_2)$ indeed are *disjoint* unions.

Another interesting aspect is the way in which bindings propagate throughout a pattern. When evaluating a pattern conjunction $v \triangleright (p_1 \,\&\, p_2)$, the bindings M_1 produced by $v \triangleright p_1$,

$$\textsc{eval-pat-var}\ \frac{}{v \triangleright x^? \Downarrow_{\text{pat}} \text{Match } \{x \mapsto v\}} \qquad \textsc{eval-pat-wild}\ \frac{}{v \triangleright \_ \Downarrow_{\text{pat}} \text{Match } \varnothing} \qquad \textsc{eval-pat-int-1}\ \frac{}{n \triangleright n \Downarrow_{\text{pat}} \text{Match } \varnothing}$$

$$\textsc{eval-pat-int-2}\ \frac{n_1 \neq n_2}{n_1 \triangleright n_2 \Downarrow_{\text{pat}} \text{Mismatch}} \qquad \textsc{eval-pat-when-1}\ \frac{v \triangleright p \Downarrow_{\text{pat}} \text{Mismatch}}{v \triangleright p \textbf{ when } b \Downarrow_{\text{pat}} \text{Mismatch}}$$

$$\textsc{eval-pat-when-2}\ \frac{v \triangleright p \Downarrow_{\text{pat}} \text{Match } M_1 \qquad \text{Subst}(M_1, b) \Downarrow_{\text{bbe}} \text{Mismatch}}{v \triangleright p \textbf{ when } b \Downarrow_{\text{pat}} \text{Mismatch}} \qquad \textsc{eval-pat-when-3}\ \frac{v \triangleright p \Downarrow_{\text{pat}} \text{Match } M_1 \qquad \text{Subst}(M_1, b) \Downarrow_{\text{bbe}} \text{Match } M_2 \qquad M_1 \# M_2}{v \triangleright p \textbf{ when } b \Downarrow_{\text{pat}} \text{Match } (M_1 \uplus M_2)}$$

$$\textsc{eval-pat-and-1}\ \frac{v \triangleright p_1 \Downarrow_{\text{pat}} \text{Mismatch}}{v \triangleright (p_1 \,\&\, p_2) \Downarrow_{\text{pat}} \text{Mismatch}} \qquad \textsc{eval-pat-and-2}\ \frac{v \triangleright p_1 \Downarrow_{\text{pat}} \text{Match } M_1 \qquad v \triangleright \text{Subst}(M_1, p_2) \Downarrow_{\text{pat}} \text{Mismatch}}{v \triangleright (p_1 \,\&\, p_2) \Downarrow_{\text{pat}} \text{Mismatch}}$$

$$\textsc{eval-pat-and-3}\ \frac{v \triangleright p_1 \Downarrow_{\text{pat}} \text{Match } M_1 \qquad v \triangleright \text{Subst}(M_1, p_2) \Downarrow_{\text{pat}} \text{Match } M_2 \qquad M_1 \# M_2}{v \triangleright (p_1 \,\&\, p_2) \Downarrow_{\text{pat}} \text{Match } (M_1 \uplus M_2)} \qquad \textsc{eval-pat-or-1}\ \frac{v \triangleright p_1 \Downarrow_{\text{pat}} \text{Mismatch} \qquad v \triangleright p_2 \Downarrow_{\text{pat}} \text{Mismatch}}{v \triangleright (p_1 \mid p_2) \Downarrow_{\text{pat}} \text{Mismatch}}$$

$$\textsc{eval-pat-or-2}\ \frac{v \triangleright p_1 \Downarrow_{\text{pat}} \text{Mismatch} \qquad v \triangleright p_2 \Downarrow_{\text{pat}} \text{Match } M_2}{v \triangleright (p_1 \mid p_2) \Downarrow_{\text{pat}} \text{Match } M_2} \qquad \textsc{eval-pat-or-3}\ \frac{v \triangleright p_1 \Downarrow_{\text{pat}} \text{Match } M_1}{v \triangleright (p_1 \mid p_2) \Downarrow_{\text{pat}} \text{Match } M_1} \qquad \textsc{eval-pat-not-1}\ \frac{b \Downarrow_{\text{bbe}} \text{Mismatch}}{\textbf{not } b \Downarrow_{\text{bbe}} \text{Match } \varnothing}$$

$$\textsc{eval-pat-not-2}\ \frac{b \Downarrow_{\text{bbe}} \text{Match } M}{\textbf{not } b \Downarrow_{\text{bbe}} \text{Mismatch}} \qquad \textsc{eval-pat-as-1}\ \frac{v \triangleright p \Downarrow_{\text{pat}} \text{Mismatch}}{v \triangleright p \textbf{ as } x^? \Downarrow_{\text{pat}} \text{Mismatch}}$$

$$\textsc{eval-pat-as-2}\ \frac{v \triangleright p \Downarrow_{\text{pat}} \text{Match } M \qquad x \notin \text{dom}(M)}{v \triangleright p \textbf{ as } x^? \Downarrow_{\text{pat}} \text{Match } (M \uplus \{x \mapsto v\})} \qquad \textsc{eval-pat-pred-1}\ \frac{g\,(v) \Downarrow_{\text{trm}} \text{false}}{v \triangleright g \Downarrow_{\text{pat}} \text{Mismatch}} \qquad \textsc{eval-pat-pred-2}\ \frac{g\,(v) \Downarrow_{\text{trm}} \text{true}}{v \triangleright g \Downarrow_{\text{pat}} \text{Match } \varnothing}$$

$$\textsc{eval-pat-view-1}\ \frac{t_0\,(v) \Downarrow_{\text{trm}} \text{None}}{v \triangleright t_0\ (p_1, ..., p_n) \Downarrow_{\text{pat}} \text{Mismatch}}$$

$$\textsc{eval-pat-view-2}\ \frac{t_0\,(v) \Downarrow_{\text{trm}} \text{Some}\,(v_1, ..., v_n) \qquad \left(\forall i \in [1,k).\ v_i \triangleright \text{Subst}((\biguplus_{1 \le j < i} M_j), p_i) \Downarrow_{\text{pat}} \text{Match } M_i\right) \qquad v_k \triangleright \text{Subst}((\biguplus_{1 \le j < k} M_j), p_k) \Downarrow_{\text{pat}} \text{Mismatch}\right) \qquad \forall i, j \in [1,n].\ i \neq j \Rightarrow M_i \# M_j}{v \triangleright t_0\ (p_1, ..., p_n) \Downarrow_{\text{pat}} \text{Mismatch}}$$

$$\textsc{eval-pat-view-3}\ \frac{t_0\,(v) \Downarrow_{\text{trm}} \text{Some}\,(v_1, ..., v_n) \qquad \forall i \in [1,n].\ v_i \triangleright \text{Subst}((\biguplus_{1 \le j < i} M_j), p_i) \Downarrow_{\text{pat}} \text{Match } M_i \qquad \forall i, j \in [1,n].\ i \neq j \Rightarrow M_i \# M_j}{v \triangleright t_0\ (p_1, ..., p_n) \Downarrow_{\text{pat}} \text{Match } (\biguplus_{1 \le i \le n} M_i)}$$

Fig. 9. Semantics of patterns

in case of success, are substituted into p_2, leading to the evaluation of $v \triangleright \text{Subst}(M_1, p_2)$. Likewise, in the evaluation of pattern views, the bindings obtained from the subpattern p_i are substituted into the subpatterns p_j for $j > i$.

Threading of the state. As stated earlier, we did omit details about mutable memory state. Let us illustrate on one rule how the state could be added. Let us write σ for the mutable state. The evaluation judgment for terms in the presence of mutable state takes the form $t/\sigma_1 \Downarrow_{\text{trm}} v/\sigma_2$, asserting that the evaluation of t in σ_1 produces a result v in a state σ_2. BBEs and patterns can include effectful terms, hence they also need to include a state. For example, the evaluation judgment for BBEs would take the form: $b/\sigma_1 \Downarrow_{\text{bbe}} r/\sigma_2$. To illustrate of threading a state throughout an evaluation rule, consider the rule for while loops, which would be augmented as follows.

EVAL-TRM-WHILE-1-WITH-STATE

$$\frac{b/\sigma_0 \Downarrow_{\text{bbe}} (\text{Match } M)/\sigma_1 \qquad (\text{Subst}(M, t);\ \textbf{while}\ b\ \textbf{do}\ t\ \textbf{done})/\sigma_1 \Downarrow_{\text{trm}} v/\sigma_2}{(\textbf{while}\ b\ \textbf{do}\ t\ \textbf{done})/\sigma_0\ \Downarrow_{\text{trm}}\ v/\sigma_2}$$

In the case where several subterms are evaluated, it is necessary to thread the evaluation order. Consider the semantic rule of the while construct.

EVAL-TRM-MATCH-VAL

$$\frac{\begin{array}{c}\forall i \in [1, k).\ (v_0 \triangleright p_i)/\sigma_{i-1} \Downarrow_{\text{pat}} \text{Mismatch}/\sigma_i \\ v_0 \triangleright p_k/\sigma_{k-1} \Downarrow_{\text{pat}} \text{Match } M/\sigma_k \qquad \text{Subst}(M, t_k)/\sigma_k \Downarrow_{\text{trm}} v/\sigma_{k+1}\end{array}}{(\textbf{match}\ v_0\ \textbf{with} \mid p_1 \to t_1 \mid \ldots \mid p_n \to t_n)/\sigma_0\ \Downarrow_{\text{trm}}\ v/\sigma_{k+1}}$$

3 BACKTRACKING IN PATTERN MATCHING

3.1 Labeled Constructs

This section introduces the **next** construct, which empowers the programmer with fine-grained control for backtracking within a pattern-matching construct. A motivating example for this construct is discussed in the appendix (Section A).

We refine the source language with several *labeled* constructs: a labeled-function is written $\lambda_L(x_1, ..., x_n).\, t$, a labeled-match is written $\textbf{match}_L\ t_0\ \textbf{with} \mid p_1 \to t_1 \mid \ldots \mid p_n \to t_n$, a labeled-swith is written $\textbf{switch}_L \mid (\textbf{case}\ b_1\ \textbf{then}\ t_1) \mid \ldots \mid (\textbf{case}\ b_n\ \textbf{then}\ t_n)$, and a labeled-if is written $\textbf{if}_L\ b_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2$. A labeled-match corresponds to a particular case of a labeled-switch, and a labeled-switch corresponds to a cascade of labeled-ifs.

The *next-branch* expression, written **next** L, may appear in any of the continuations from a match or switch construct (in any t_i), and it may appear in the first branch of a labeled-if statement (in t_1).

As we show throughout the rest of this section, the *next-branch* can be formalized in a very similar way to the *exit-block* construct. In short, inside a *labeled block*, written $L : \{ t \}$, anywhere in-depth in t, the programmer can place an *exit-block* construct, written **exit** $L\ t'$, to abort the execution of the block L, and provide t' as result value for this block.

The exit-block construct, introduced by Common Lisp (1970s), is available in numerous languages, e.g., Javascript or Rust. It can be used to directly encode control-flow constructs such as **break**, **continue** and **return**. While many programs can be naturally written

Terms, refined with labels

t +=	$\lambda_L(x_1, ..., x_n).\, t$	labeled function
	$\textbf{if}_L\ b_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2$	labeled conditional
	$\textbf{while}_L\ b\ \textbf{do}\ t\ \textbf{done}$	labeled loop
	$\textbf{switch}_L\ (\textbf{case}\ b_1\ \textbf{then}\ t_1) \mid ...$	labeled switch
	$\textbf{match}_L\ t_0\ \textbf{with} \mid p_1 \to t_1 \mid ... \mid$	labeled match

Terms, extended

t +=	$L : \{\, t \,\}$	labeled block
	$\textbf{exit}\ L\ t$	exit from a block
	$\textbf{return}\ L\ t$	exit from a function body
	$\textbf{break}\ L$	break out of a loop
	$\textbf{continue}\ L$	jump to next iteration of a loop
	$\textbf{next}\ L$	jump to next branch of a labeled if/switch/match

Syntactic sugar for implicit labels

t +=	$\textbf{exit}\ t$	exit nearest block with argument term
	$\textbf{return}\ t$	return argument term for nearest function
	$\textbf{break}$	break out of nearest enclosing loop
	$\textbf{continue}$	next iteration on nearest enclosing loop
	$\textbf{next}$	next branch of nearest enclosing if/switch/match

Fig. 10. Adding block-exit and adding labels on branching term constructs

without such constructs (recall that the OCaml does not include them), there are large classes of algorithms whose description is arguably more concise and more elegant when leveraging these constructs.

Exit-blocks can be simulated using exceptions, yet with runtime overhead (and syntactic overhead). The point of the exit-block construct is to provide statically scoped exceptions, that can be compiled efficiently using simple jumps at the assemby level. Similarly, the point of the **next** L construct is to a provide statically scoped construct for backtracking inside conditionals or pattern matching. Whereas our naive prototype implementation compiles **next** into ML exceptions, a realistic compiler would compile **next** similarly to exit-blocks, using simple jumps.

3.2 Syntax and Typing for Abort Constructs

To stress the similarities between **next** and the constructs **exit**, **break**, **continue** and **return**, we include all of them in our presentation. Figure 10 gives the syntax of labeled terms. The expression **next** without argument is syntactic sugar for **next** L with L being the label of nearest enclosing branching construct (if, switch, or match), and likewise for other constructs.

The typing judgment for terms is generalized to the form $E; F \vdash_{\text{trm}} t : T$, where F denotes a *label environment*, which lists the labels in scope, and binds certain labels to a type. Figure 11 formalizes the grammar of labeled environments. An entry LblLoop L or

Typing environment for labels

F	:=	$\varnothing$	empty environment
	|	F, (LblBlock L : T)	binding the label of a block
	|	F, (LblFun L : T)	binding the label of a function
	|	F, (LblLoop L)	binding the label of a loop
	|	F, (LblBranch L)	binding the label of a conditional or switch

Fig. 11. Grammar of label environments.

LblBranch L indicates that L is a label associated with a while-loop or with an if/switch/-match construct, respectively. An entry "LblBlock L : T" or "LblFun L : T" indicates that L is a label associated with a labeled-block of type T or with a function whose body has type T, respectively.

The typing judgments for BBEs and patterns does *not* depend on labeled environments. Even though BBEs and patterns may contain general terms in depth, we purposely disallow them to trigger abrupt termination behaviors. We believe that allowing them would result in obfuscated control flows.

Importantly, we need to disallow function closures to capture abort operations that would escape the scope of the closure. This restriction is standard. For example, inside a while-loop, one cannot define a function that invokes **break** to break out of this surrounding loop. To enforce this restriction, we typecheck the body t of a function $\lambda_L(x_1, ..., x_n).\, t$ in a singleton label environment that binds only the label L.

The typing rules for terms are shown in Figure 12. The typing rules for BBEs and patterns are unchanged, except where a subexpression from the category of terms is involved. In such cases, we typecheck the subterm in an empty label environment, as illustrated in the last three rules from Figure 12.

3.3 Semantics for Abort Constructs

To give a semantics to language constructs that trigger an *abrupt termination*, a.k.a. *abort behavior*, we introduce a grammar of *outcomes*. As shown in Figure 13, outcomes include both regular results, written Res v, labeled-abort outcomes, written Abort a, as well as conventional exceptions, written Exn e. The grammar of abrupt terminations, written a, includes in particular Exit L v and Next L. The exception NoBranch is produced by a match or a switch with no matching branch (it corresponds to the exception `Match_failure` of OCaml). Throughout the rest of the paper, we do not discuss exceptions further, as their semantics is standard and orthogonal to the labeled-abort behaviors on which we focus.

In the Appendix B, we present the most important evaluation rules in big-step style. In the next section, we present an exhaustive set of evaluation rules in small-step style.

4 A MINIMAL LANGUAGE FOR BBE AND PATTERNS

This section presents a minimal language, with a smaller number of constructs, in which all the features presented so far can be encoded. We use this minimal language to simplify the proof of type soundness, and to describe the compilation scheme towards a standard,

$$\textsc{typ-trm-fun-label}\quad \frac{E, x_1 : T_1, ..., x_n : T_n;\ (\text{LblFun } L\ :\ T) \vdash_{\text{trm}} t : T}{E; F \vdash_{\text{trm}} \lambda_L(x_1, ..., x_n).\, t : (T_1, ..., T_n)\ \to\ T}$$

$$\textsc{typ-trm-if-label}\quad \frac{E \vdash_{\text{bbe}} b_0 \rightsquigarrow B \qquad E, B_0; F, (\text{LblBranch } L) \vdash_{\text{trm}} t_1 : T \qquad E; F \vdash_{\text{trm}} t_2 : T}{E; F \vdash_{\text{trm}} \textbf{if}_L\ b_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 : T}$$

$$\textsc{typ-trm-while-label}\quad \frac{E \vdash_{\text{bbe}} b \rightsquigarrow B \qquad E, B; F, (\text{LblLoop } L) \vdash_{\text{trm}} t : \text{unit}}{E; F \vdash_{\text{trm}} (\textbf{while}_L\ b\ \textbf{do}\ t\ \textbf{done}) : \text{unit}}$$

$$\textsc{typ-trm-switch-label}\quad \frac{\forall i \in [1, n].\ E \vdash_{\text{bbe}} b_i \rightsquigarrow B_i\ \wedge\ E, B_i; F, (\text{LblBranch } L) \vdash_{\text{trm}} t_i : T}{E; F \vdash_{\text{trm}} \textbf{switch}_L \mid (\textbf{case}\ b_1\ \textbf{then}\ t_1) \mid ... \mid (\textbf{case}\ b_n\ \textbf{then}\ t_n) : T}$$

$$\textsc{typ-trm-match-label}\quad \frac{E \vdash_{\text{trm}} t_0 : T_0 \qquad \big(\forall i \in [1, n].\ E \vdash_{\text{pat}} p_i : T_0 \rightsquigarrow B_i\ \wedge\ E, B_i; F, (\text{LblBranch } L) \vdash_{\text{trm}} t_i : T\big)}{E; F \vdash_{\text{trm}} \textbf{match}_L\ t_0\ \textbf{with} \mid p_1 \to t_1 \mid ... \mid p_n \to t_n : T}$$

$$\textsc{typ-trm-block}\quad \frac{E; F, (\text{LblBlock } L\ :\ T) \vdash_{\text{trm}} t : T}{E; F \vdash_{\text{trm}} (L : \{\, t \,\}) : T} \qquad \textsc{typ-trm-exit}\quad \frac{(\text{LblBlock } L\ :\ T) \in F \qquad E; F \vdash_{\text{trm}} t : T}{E; F \vdash_{\text{trm}} \textbf{exit}\ L\ t : T'}$$

$$\textsc{typ-trm-return}\quad \frac{(\text{LblFun } L\ :\ T) \in F \qquad E; F \vdash_{\text{trm}} t : T}{E; F \vdash_{\text{trm}} \textbf{return}\ L\ t : T'} \qquad \textsc{typ-trm-break}\quad \frac{(\text{LblLoop } L) \in F}{E; F \vdash_{\text{trm}} \textbf{break}\ L : \text{unit}}$$

$$\textsc{typ-trm-continue}\quad \frac{(\text{LblLoop } L) \in F}{E; F \vdash_{\text{trm}} \textbf{continue}\ L : \text{unit}} \qquad \textsc{typ-trm-next}\quad \frac{(\text{LblBranch } L) \in F}{E; F \vdash_{\text{trm}} \textbf{next}\ L : T'} \qquad \textsc{typ-bbe-trm}\quad \frac{E; \varnothing \vdash_{\text{trm}} t : \text{bool}}{E \vdash_{\text{bbe}} t\ \rightsquigarrow\ \varnothing}$$

$$\textsc{typ-bbe-is}\quad \frac{E; \varnothing \vdash_{\text{trm}} t : T \qquad E \vdash_{\text{pat}} p : T \rightsquigarrow B}{E \vdash_{\text{bbe}} (t\ \textbf{is}\ p)\ \rightsquigarrow\ B}$$

$$\textsc{typ-pat-view}\quad \frac{\begin{array}{c} E; \varnothing \vdash_{\text{trm}} t_0 : T \to (T_1, ..., T_n)\ \text{option} \\ \forall i \in [1, n].\quad E, (\biguplus_{1 \le j < i} B_j) \vdash_{\text{pat}} p_i : T_i \rightsquigarrow B_i \\ \forall i, j \in [1, n].\ i \neq j \Rightarrow \text{pv}(p_i) \cap \text{pv}(p_j) = \varnothing \end{array}}{E \vdash_{\text{pat}} t_0\ (p_1, ..., p_n)\ :\ T\ \rightsquigarrow\ \biguplus_{1 \le i \le n} B_i}$$

Fig. 12. Typing rules for constructs involving labels.

Core-ML language. For the minimal language, we present both a big-step and a small-step semantics. We will use the big-step semantics to prove the correctness of the compilation scheme, and use the small-step semantics to establish type soundness.

4.1 Syntax of the Core Language and Encodings

Figure 14 presents the grammar of the language, which consists essentially of a subset of the language presented so far. We remove the let-pattern construct, and keep only

Outcome of term evaluation

o	:=	Res v	result that consists of an output value v
	|	Abort a	result that consists of a labeled-abort a
	|	Exn v	result that consists of an exception v

Abrupt termination reaching a label L

a	:=	Exit L v	exit from a block with value v
	|	Return L v	exit from a function with value v
	|	Break L	exit from a loop
	|	Continue L	jump to next iteration of a loop
	|	Next L	jump to next branch in a if or a switch

Fig. 13. Entities involved in the semantics of labeled constructs

t, g	:=	x	variable
	|	n	integer
	|	**let** $x = t_1$ **in** t_2	let-binding
	|	$\lambda(x_1, ..., x_n).\, t$	function definition
	|	$t_0\ (t_1, ..., t_n)$	application, including constructor
	|	**if**$_L$ b_0 **then** t_1 **else** t_2	backtracking branching instruction
	|	**while** b **do** t **done**	conditional loop
	|	$L : \{\, t \,\}$	labeled block
	|	**exit** L t	block-exiting
	|	**next** L	branch-exiting
b	:=	t **is** p	match against a pattern
	|	**if**$^{\text{bbe}}$ b_0 **then** b_1 **else** b_2	branching, as part of a BBE
p	:=	$x^?$	pattern variable
	|	p **when** b	guarded pattern
	|	Some$(p_1, ..., p_n)$	tuple option pattern

Fig. 14. Grammar of the minimal language

the simpler form **let** $x = t_1$ **in** t_2. Such a term is semantically equivalent to "$(\lambda x.\, t_2)\, t_1$", however we do not want to suggest that a closure allocation is required. We keep function definitions and while loops, but only in their unlabeled form. We restrict the view-pattern to only support matching against an option on a tuple, in the form "t **is** Some$(p_1, ..., p_n)$".

Importantly, we introduce one new construct, written **if**$^{\text{bbe}}$ b_0 **then** b_1 **else** b_2. This construct can be used to encode BBE conjunction, disjunction, and negation. The construct **if**$^{\text{bbe}}$ b_0 **then** b_1 **else** b_2 first evaluates b_0. If the result is Mismatch, then it returns the result of b_2. If the result is Match M_0, then b_1 is evaluated. If the result is Mismatch, the final result is Mismatch. Otherwise, if the result is Match M_1, then the final result is Match $(M_0 \uplus M_1)$.

Figure 15 presents encodings for the language constructs that are not part of the minimal language. A let-pattern is encoded using a conditional with a **is** statement. The **switch**

Encoded terms

$$\begin{array}{ll}
\textbf{let } p = t_1 \textbf{ in } t_2 & \equiv \textbf{if } (t_1 \textbf{ is } p) \textbf{ then } t_2 \textbf{ else } (\text{raise Match_Failure}) \\
\textbf{switch}_L \mid (\textbf{case } b_1 \textbf{ then } t_1) \mid \ldots \mid c_n & \equiv \textbf{if}_L\ b_1 \textbf{ then } t_1 \textbf{ else } (\textbf{switch}_L \mid c_2 \mid \ldots \mid c_n) \\
\textbf{match}_L\ t_0 \textbf{ with } \mid p_1 \to t_1 \mid \ldots \mid p_n \to t_n & \equiv \textbf{let } x = t_0 \textbf{ in switch}_L \mid (\textbf{case } x \textbf{ is } p_1 \textbf{ then } t_1) \mid \ldots
\end{array}$$

$$\left[\begin{array}{l}
\lambda_L(x_1, \ldots, x_n). \\
\quad \ldots \\
\quad \ldots \textbf{return } L\ t'; \\
\quad \ldots
\end{array}\right] \equiv \left[\begin{array}{l}
\lambda(x_1, \ldots, x_n).\, L : \{ \\
\qquad \ldots \\
\qquad \ldots \textbf{exit } L\ t'; \\
\qquad \ldots \}
\end{array}\right]$$

$$\left[\begin{array}{l}
\\
\textbf{while}_L\ b \textbf{ do} \\
\quad \ldots \\
\quad \ldots \textbf{break } L; \\
\quad \ldots \\
\quad \ldots \textbf{continue } L; \\
\quad \ldots \\
\textbf{done} \\
\\
\end{array}\right] \equiv \left[\begin{array}{l}
L : \{\textbf{while } b \text{ do} \\
\quad L' : \{ \\
\qquad \ldots \\
\qquad \ldots \textbf{exit } L\ tt; \\
\qquad \ldots \\
\qquad \ldots \textbf{exit } L'\ tt; \\
\qquad \ldots \\
\quad \} \\
\}
\end{array}\right]$$

Encoded BBEs

$$\begin{array}{ll}
t & \equiv t \textbf{ is } \text{true} \\
b_1 \textbf{ and } b_2 & \equiv \textbf{if}^{\text{bbe}}\ b_1 \textbf{ then } b_2 \textbf{ else } \text{false} \\
b_1 \textbf{ or } b_2 & \equiv \textbf{if}^{\text{bbe}}\ b_1 \textbf{ then } \text{true} \textbf{ else } b_2 \\
\textbf{not } b & \equiv \textbf{if}^{\text{bbe}}\ b \textbf{ then } \text{false} \textbf{ else } \text{true}
\end{array}$$

Encoded patterns—for a globally fresh variable x

$$\begin{array}{ll}
_ & \equiv x^? \\
n & \equiv x^? \textbf{ when } (x = n) \\
p_1\ \&\ p_2 & \equiv x^? \textbf{ when } (x \textbf{ is } p_1) \textbf{ and } (x \textbf{ is } p_2) \\
p_1 \mid p_2 & \equiv x^? \textbf{ when } (x \textbf{ is } p_1) \textbf{ or } (x \textbf{ is } p_2) \\
\neg\ p & \equiv x^? \textbf{ when not } (x \textbf{ is } p) \\
p \textbf{ as } x^? & \equiv x^? \textbf{ when } (x \textbf{ is } p) \\
g & \equiv x^? \textbf{ when } g\,(x) \\
t_0\ (p_1, \ldots, p_n) & \equiv x^? \textbf{ when } f\,(x) \textbf{ is } \text{Some } (p_1, \ldots, p_n)
\end{array}$$

Fig. 15. Encoding into the minimal language of other language constructs

construct is encoded using a cascade of conditionals. The **break** and **continue** expressions are encoded using **exit** on different labels. A term t in BBE position is encoded as "t **is** true". Other BBE constructs are encoded using $\textbf{if}^{\text{bbe}}$. Derived pattern constructs are encoded using **when** clauses.

$$\textsc{typ-bbe-if}\quad \frac{E \vdash_{\text{bbe}} b_0 \rightsquigarrow B_0 \qquad E, B_0 \vdash_{\text{bbe}} b_1 \rightsquigarrow B_1 \qquad E \vdash_{\text{bbe}} b_2 \rightsquigarrow B_2 \qquad \text{pv}(b_0) \cap \text{pv}(b_1) = \varnothing}{E \vdash_{\text{bbe}} \textbf{if}^{\text{bbe}}\ b_0\ \textbf{then}\ b_1\ \textbf{else}\ b_2 \ \rightsquigarrow\ (B_0 \uplus B_1) \cap B_2}$$

$$\textsc{typ-bbe-false}\quad \frac{}{E \vdash_{\text{bbe}} \text{false} \ \rightsquigarrow\ B}$$

Fig. 16. Typing rules for $\textbf{if}^{\text{bbe}}$, which encodes the BBE operations **and**, **or**, **not**, and the additional typing rule for false, needed to make TYP-BBE-AND derivable.

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

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

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

Fig. 17. Evaluation rules for $\textbf{if}^{\text{bbe}}$.

4.2 Typing and Semantics for Low-Level Constructs

Figure 16 shows the typing rule for the new construct $\textbf{if}^{\text{bbe}}\ b_0\ \textbf{then}\ b_1\ \textbf{else}\ b_2$. In order to derive the typing rule for BBE conjunction (TYP-BBE-AND), we need a special typing rule for false appearing as part of a BBE. This special rule TYP-BBE-FALSE overlaps with TYP-BBE-TRM. Recall that the judgment "b has type B" asserts that if b evaluates to Match M, then the bindings in M have the type described by the map B. Because false evaluates to Mismatch, it is safe to claim that false admits any type B. The following derivation justifies that TYP-BBE-AND is derivable.

$$\textsc{typ-bbe-and-derived-from-if}\quad \frac{\dfrac{\overline{E \vdash_{\text{bbe}} b_0 \ \rightsquigarrow\ B_0} \quad \overline{E, B_0 \vdash_{\text{bbe}} b_1 \ \rightsquigarrow\ B_1} \quad \overline{E \vdash_{\text{bbe}} \text{false} \ \rightsquigarrow\ (B_0 \uplus B_1)} \quad \overline{\text{pv}(b_0) \cap \text{pv}(b_1) = \varnothing}}{E \vdash_{\text{bbe}} \textbf{if}^{\text{bbe}}\ b_0\ \textbf{then}\ b_1\ \textbf{else}\ \text{false} \ \rightsquigarrow\ (B_0 \uplus B_1) \cap (B_0 \uplus B_1)}}{E \vdash_{\text{bbe}} b_0\ \textbf{and}\ b_1 \ \rightsquigarrow\ B_0 \uplus B_1}$$

The evaluation rules for $\textbf{if}^{\text{bbe}}$ appear in Figure 17. The tuple-option pattern construct that we have included is just a special case of the view pattern, hence we omit its typing and evaluation rules.

Figures 29, 30 and 31 in Appendix G, present the expanded typing and semantics rules of the minimal language.

4.3 Small-Step Semantics

In addition to the big-step semantics presented above for the minimal language, we present an equivalent small-step semantics. The reduction judgment takes the form $e \mapsto e'$, where

G :=	$\mathbf{let}\ x = \square\ \mathbf{in}\ t_2$	argument of a let-binding
|	$\square\ (t_1, ..., t_n)$	function of an application
|	$v_0\ (v_1, ..., v_k, \square, t_{k+2}, ..., t_n)$	one argument of an application
|	$\mathbf{if}_L\ \square\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$	argument of a conditional
|	$\mathbf{ifthen}_L\ \square\ \mathbf{else}\ t_2$	first branch of a labeled conditional
|	$L : \{\square\}$	argument of a labeled block
|	$\mathbf{exit}\ L\ \square$	argument of an exit instruction
|	$\square\ \mathbf{is}\ p$	argument of a BBE matching
|	$\mathbf{if}^{\mathrm{bbe}}\ \square\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2$	argument of a BBE conditional
|	$\mathbf{if}^{\mathrm{bbe}}\ M\ \mathbf{then}\ \square$	first branch of a BBE conditional

Fig. 18. Evaluation contexts for the small-step semantics of the minimal language

e is an entity. The grammar of entities corresponds to the flattening of the grammar of terms (written t), BBEs (written b), outcomes (written o), results (written r), maps (written M), as well as the pairs of values and patterns (written $v \triangleright p$).

Entities also included *intermediate forms* for terms and BBEs. The form $\mathbf{ifthen}_L\ \square\ \mathbf{else}\ t_2$ denotes a conditional with ongoing execution of the first branch, but with the possibility of backtracking into t_2 if a **next** L is triggered. Similarly, the form $\mathbf{if}^{\mathrm{bbe}}\ M\ \mathbf{then}\ \square$ denotes a BBE conditional with ongoing execution of its first branch, such that if this branch results in a Match M', then the conditional produces Match $(M \uplus M')$.

The grammar of evaluation contexts is shown in Figure 18. The reduction rules are given in Figure 19. Note that there is no reduction rules for patterns per se, but only rules for BBEs of the form $v \triangleright p$.

Let us state the equivalence between the small-step and the big-step judgments. The proof follows standard techniques, hence we omit the details.

Theorem 1 (small-step matches big-step semantics, for terminating programs).

$$(t \mapsto^* o \iff t \Downarrow_{\mathrm{trm}} o) \wedge (b \mapsto^* r \iff b \Downarrow_{\mathrm{bbe}} r) \wedge ((v \triangleright p) \mapsto^* r \iff v \triangleright p \Downarrow_{\mathrm{pat}} r)$$

4.4 Type soundness

Details of the type soundness proof may be found in Appendix C. Here, we only reproduce the statement of the theorems. Recall that an outcome consists either of a value or an abort outcome.

Theorem 2 (Preservation). *The following statements hold:*

(1) $E; F \vdash_{\mathrm{trm}} e : T \quad \wedge \quad e \mapsto e' \quad \Rightarrow \quad E; F \vdash_{\mathrm{trm}} e' : T$
(2) $E \vdash_{\mathrm{bbe}} e \rightsquigarrow B \quad \wedge \quad e \mapsto e' \quad \Rightarrow \quad E \vdash_{\mathrm{bbe}} e' : B$

Theorem 3 (Progress). *The following statements hold:*

(1) $\varnothing; F \vdash_{\mathrm{trm}} t : T \quad \Rightarrow \quad t$ is an outcome $o \quad \vee \quad \exists t'.\ t \mapsto t'$
(2) $\varnothing \vdash_{\mathrm{bbe}} b \rightsquigarrow B \quad \Rightarrow \quad b$ is a result $r \quad \vee \quad \exists b'.\ b \mapsto b'$

$$\textsc{red-ctx}\ \frac{e \mapsto e'}{G[e] \mapsto G[e']} \qquad \textsc{red-abt}\ \frac{\begin{array}{c}\forall L, t_1, t_2.\ G \neq (\mathbf{ifthen}_L\ \square\ \mathbf{else}\ t_2) \\ \forall L.\ G \neq (L : \{\square\})\end{array}}{G[\text{Abort } a] \mapsto \text{Abort } a}$$

RED-INT	n	$\mapsto \text{Res } \bar{n}$
RED-FUN	$\lambda(x_1, ..., x_n).\, t$	$\mapsto \text{Res } (\bar{\lambda}(x_1, ..., x_n).t)$
RED-CSTR	$(\text{Res } C)\ (\text{Res } v_1, ..., \text{Res } v_n)$	$\mapsto \text{Res } (\bar{C}\ (v_1, ..., v_n))$
RED-LET	$\mathbf{let}\ x = \text{Res } v_1\ \mathbf{in}\ t_2$	$\mapsto \text{Subst}(\{x \mapsto v_1\}, t_2)$
RED-BETA	$(\text{Res } (\bar{\lambda}(x_1, ..., x_n).t))$ $(\text{Res } v_1, ..., \text{Res } v_n)$	$\mapsto \text{Subst}(\{x_1 \mapsto v_1, ..., x_n \mapsto v_n\}, t)$
RED-IF-1	$\mathbf{if}_L\ \text{Mismatch}\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$	$\mapsto t_2$
RED-IF-2	$\mathbf{if}_L\ \text{Match}\ M\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$	$\mapsto \mathbf{ifthen}_L\ \text{Subst}(M, t_1)\ \mathbf{else}\ t_2$
RED-IFTHEN-1	$\mathbf{ifthen}_L\ \text{Abort}\ (\text{Next } L)\ \mathbf{else}\ t_2$	$\mapsto t_2$
RED-IFTHEN-2	$\mathbf{ifthen}_L\ o\ \mathbf{else}\ t_2$	$\mapsto o$ *when* $o \neq \text{Abort}\ (\text{Next } L)$
RED-WHILE	$\mathbf{while}\ b\ \mathbf{do}\ t\ \mathbf{done}$	$\mapsto \mathbf{if}\ b\ \mathbf{then}\ (\mathbf{let}\ x = t\ \mathbf{in}\ \mathbf{while}\ b\ \mathbf{do}\ t\ \mathbf{done})$ $\mathbf{else}\ tt$ *where* x *is a free variable*
RED-BLOCK-1	$L : \{\ \text{Abort}\ (\text{Exit } L\ v)\ \}$	$\mapsto \text{Res } v$
RED-BLOCK-2	$L : \{\ o\ \}$	$\mapsto o$ *when* $o \neq \text{Abort}\ (\text{Exit } L\ v)$
RED-EXIT	$\mathbf{exit}\ L\ (\text{Res } v)$	$\mapsto \text{Abort}\ (\text{Exit } L\ v)$
RED-NEXT	$\mathbf{next}\ L$	$\mapsto \text{Abort}\ (\text{Next } L)$
RED-IS	$(\text{Res } v)\ \mathbf{is}\ p$	$\mapsto v \triangleright p$
RED-IF-BBE-1	$\mathbf{if}^{\text{bbe}}\ \text{Mismatch}\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2$	$\mapsto b_2$
RED-IF-BBE-2	$\mathbf{if}^{\text{bbe}}\ \text{Match}\ M_0\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2$	$\mapsto \mathbf{if}^{\text{bbe}}\ M_0\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2$
RED-IF-BBE-3	$\mathbf{if}^{\text{bbe}}\ M_0\ \mathbf{then}\ \text{Mismatch}\ \mathbf{else}\ b_2$	$\mapsto \text{Mismatch}$
RED-IF-BBE-4	$\mathbf{if}^{\text{bbe}}\ M_0\ \mathbf{then}\ \text{Match}\ M_1\ \mathbf{else}\ b_2$	$\mapsto \text{Match}\ (M_0 \uplus M_1)$ *when* $M_0 \# M_1$
RED-PAT-VAR	$v \triangleright x^?$	$\mapsto \text{Match}\ \{x \mapsto v\}$
RED-WHEN	$v \triangleright (p\ \mathbf{when}\ b)$	$\mapsto (v \triangleright p)\ \mathbf{and}\ b$ *where* $(b_1\ \mathbf{and}\ b_2)$ $\equiv (\mathbf{if}^{\text{bbe}}\ b_1\ \mathbf{then}\ b_2\ \mathbf{else}\ \text{false})$
RED-VIEW-1	$\text{None} \triangleright \text{Some}(p_1, ..., p_n)$	$\mapsto \text{Mismatch}$
RED-VIEW-2	$\text{Some}(v_1, ..., v_2) \triangleright \text{Some}(p_1, ..., p_n)$	$\mapsto (v_1 \triangleright p_1)\ \mathbf{and}\ ...\ \mathbf{and}\ (v_n \triangleright p_n)$

Fig. 19. Small-step semantics for the minimal language

5 COMPILATION INTO CORE-ML

In this section, we show how to compile our minimal language into a core subset of a standard ML language. This compilation scheme corresponds to the one implemented in our prototype.

For the target language, we use OCaml syntax. We assume the target language to support exceptions and the simplest possible form of pattern matching: testing a constructor and,

$$
\begin{array}{lll}
[\![x]\!] & \equiv & \mathtt{x} \\
[\![n]\!] & \equiv & \mathtt{n} \\
[\![\textbf{let } x = t_1 \textbf{ in } t_2]\!] & \equiv & \mathtt{let\ x\ =\ } [\![t_1]\!] \mathtt{\ in\ } [\![t_2]\!] \\
[\![\lambda(x_1, ..., x_n).\, t]\!] & \equiv & \mathtt{(fun\ (x_1, ..., x_n)\ \to\ } [\![t]\!]\mathtt{)} \\
[\![t_0\ (t_1, ..., t_n)]\!] & \equiv & [\![t_0]\!]\ ([\![t_1]\!], ..., [\![t_n]\!]) \\
[\![\textbf{if}_L\ b_0 \textbf{ then } t_1 \textbf{ else } t_2]\!] & \equiv & \mathtt{let\ k() = } [\![t_2]\!] \mathtt{\ in\ } [\![b_0]\!]^{\mathtt{try}\ [\![t_1]\!]\ \mathtt{with\ |\ Next\ L\ \to\ k()}}_{\mathtt{k()}} \\
[\![\textbf{while } b \textbf{ do } t \textbf{ done}]\!] & \equiv & \left(\mathtt{fix\ f\ ()\ =\ } [\![b]\!]^{([\![t]\!]\mathtt{;f())}}_{\mathtt{()}}\right)\ \mathtt{()} \\
[\![L : \{\, t \,\}]\!] & \equiv & \mathtt{try\ } [\![t]\!] \mathtt{\ with\ |\ Exit\ L\ x\ \to\ x} \\
[\![\textbf{exit } L\ t]\!] & \equiv & \mathtt{raise\ (Exit\ (L,\ } [\![t]\!]\mathtt{))} \\
[\![\textbf{next } L]\!] & \equiv & \mathtt{raise\ (Next\ L)} \\
[\![b]\!]^{\mathtt{u}}_{\mathtt{u}'} & \equiv & \mathtt{u}' \qquad \textit{when } \mathrm{bv}(b) = \mathbb{V} \\
[\![t \textbf{ is } p]\!]^{\mathtt{u}}_{\mathtt{u}'} & \equiv & \mathtt{let\ y\ =\ } [\![t]\!] \mathtt{\ in\ } [\![\mathtt{y} \blacktriangleright p]\!]^{\mathtt{u}}_{\mathtt{u}'} \qquad \textit{for a globally fresh variable } \mathtt{y} \\
[\![\textbf{if}^{\mathrm{bbe}}\ b_0 \textbf{ then } b_1 \textbf{ else } b_2]\!]^{\mathtt{u}}_{\mathtt{u}'} & \equiv & \mathtt{let\ k_1(x_1, ..., x_n)\ =\ u\ in\ let\ k_2()\ =\ u'\ in\ } [\![b_0]\!]^{[\![b_1]\!]^{\mathtt{k_1(x_1, ..., x_n)}}_{\mathtt{k2()}}}_{[\![b_2]\!]^{\mathtt{k_1(x_1, ..., x_n)}}_{\mathtt{k2()}}} \\
 & & \textit{where } \{\mathtt{x_1}; ...; \mathtt{x_n}\} = (\mathrm{bv}(b_0) \cup \mathrm{bv}(b_1)) \cap \mathrm{bv}(b_2) \cap \mathrm{fv}(\mathtt{u}) \\
[\![\textbf{if}^{\mathrm{bbe}}\ b_0 \textbf{ then } b_1 \textbf{ else } b_2]\!]^{\mathtt{u}}_{\mathtt{u}'} & \equiv & \mathtt{let\ k_2()\ =\ u'\ in\ } [\![b_0]\!]^{[\![b_1]\!]^{\mathtt{assert\ false}}_{\mathtt{k2()}}}_{[\![b_2]\!]^{\mathtt{assert\ false}}_{\mathtt{k2()}}} \\
 & & \textit{when } ((\mathrm{bv}(b_0) \cup \mathrm{bv}(b_1)) \cap \mathrm{bv}(b_2)) \text{ is an infinite set} \\
[\![\mathtt{y} \blacktriangleright p]\!]^{\mathtt{u}}_{\mathtt{u}'} & \equiv & \mathtt{u}' \qquad \textit{when } \mathrm{bv}(p) = \mathbb{V} \\
[\![\mathtt{y} \blacktriangleright x^?]\!]^{\mathtt{u}}_{\mathtt{u}'} & \equiv & \mathtt{let\ x\ =\ y\ in\ u} \\
[\![\mathtt{y} \blacktriangleright p \textbf{ when } b]\!]^{\mathtt{u}}_{\mathtt{u}'} & \equiv & \mathtt{let\ k()\ =\ u'\ in\ } [\![\mathtt{y} \blacktriangleright p]\!]^{[\![b]\!]^{\mathtt{u}}_{\mathtt{k()}}}_{\mathtt{k()}} \\
[\![\mathtt{y} \blacktriangleright \mathrm{Some}(p_1, ..., p_n)]\!]^{\mathtt{u}}_{\mathtt{u}'} & \equiv & \mathtt{let\ k()\ =\ u'\ in\ match\ y\ with} \\
 & & \mathtt{|\ Some(x_1, ..., x_n)\ \to\ } [\![(x_1 \textbf{ is } p_1) \textbf{ and } ... \textbf{ and } (x_n \textbf{ is } p_n)]\!]^{\mathtt{u}}_{\mathtt{k()}} \\
 & & \mathtt{|\ \_\ \to\ k()} \\
 & & \textit{for globally fresh variables } \{\mathtt{x_1}; ...; \mathtt{x_n}\}
\end{array}
$$

Fig. 20. Compilation of the minimal language into Core-ML, when applicable, the duplication-avoiding closures (e.g., `k()`) are bound with globally fresh variables.

in case of success, binding the constructor arguments. To simplify the translation, we use uncurried functions in the target language, but curried functions would work as well. To encode while loop with a BBE argument, we use recursive functions, written `fix f` $(\mathtt{x_1}, ..., \mathtt{x_n}) \to \mathtt{u}$, which is equivalent to `let rec f` $(\mathtt{x_1}, ..., \mathtt{x_n}) \to \mathtt{u}$ `in f`. The appendix (Section D) details the syntax (Figure 23) and the semantics (Figure 24) of the target language. We write $\mathrm{fv}(\mathtt{u})$ the set of free variables appearing in a Core-ML term u.

The compilation scheme is presented in Figure 20. We encode the abort outcomes of our minimal language, namely **exit** and **next**, into two corresponding exceptions that we define in the target language.

We write $[\![t]\!]$ the compilation of a term t. We write $[\![b]\!]^{\mathtt{u}}_{\mathtt{u}'}$ the compilation of a BBE b, where u and $\mathtt{u}'$ are Core-ML terms that denote the success and the failure continuations, respectively. The free variables of u should be included in the set of bindings exported by b according to the typing rules. The term $\mathtt{u}'$ should be closed, i.e., have no free variable.

Besides, we write $[\![\mathtt{y} \blacktriangleright p]\!]^{\mathtt{u}}_{\mathtt{u}'}$ for the 4-argument meta-function that describes the compilation of the matching of y against p, with success continuation u and failure continuation u′. We comment on the two most interesting cases.

The first interesting case is the compilation of a conditional $\mathbf{if}_L\ b_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$. It essentially compiles to: $[\![b_0]\!]^{\texttt{try } [\![t_1]\!] \texttt{ with | Next L} \rightarrow [\![t_2]\!]}_{[\![t_2]\!]}$. This term evaluates b_0. In case of failure, it continues with $[\![t_2]\!]$. Otherwise, it evaluates, in a context where the bindings of b_0 are available, the try-with expression. This expression evaluates $[\![t_1]\!]$, but if a `Next L` exception is raised, then it aborts and evaluates $[\![t_2]\!]$. To avoid code duplication, the continuation $[\![t_2]\!]$ is named `k()`.

The other interesting case is the compilation of the BBE-conditional construct. Essentially, $[\![\mathbf{if}^{\mathrm{bbe}}\ b_0\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2]\!]^{\mathtt{u}}_{\mathtt{u}'}$ compiles to $[\![b_0]\!]^{[\![b_1]\!]^{\mathtt{u}}_{\mathtt{u}'}}_{[\![b_2]\!]^{\mathtt{u}}_{\mathtt{u}'}}$. To avoid code duplication, the continuation u is named $\mathtt{k}(\mathtt{x}_1, ..., \mathtt{x}_n)$, where the $\mathtt{x}_i$ correspond to the free variables of u.[4]

Finally, as stated in figure Figure 15, the construct $\mathbf{if}^{\mathrm{bbe}}\ b_0\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2$ encodes the boolean operations on BBEs **and**, **or** and **not**. In particular, we have

$$[\![b_1\ \mathbf{and}\ b_2]\!]^{\mathtt{u}}_{\mathtt{u}'} \equiv \texttt{let } \mathtt{k}_1(\mathtt{x}_1, ..., \mathtt{x}_n) \texttt{ = u in let } \mathtt{k}_2() \texttt{ = u' in } [\![b_1]\!]^{[\![b_2]\!]^{\mathtt{k}_1(\mathtt{x}_1,...,\mathtt{x}_n)}_{\mathtt{k}_2()}}_{\mathtt{k}_2()}.$$

6 CONCLUSION

We hope that our work will facilitate the adoption of binding-boolean-expressions and extended pattern matching, both in existing and yet-to-appear programming languages. We look forward to formalizing our definitions and proofs using Rocq.

[4]The introduction of this list of arguments to avoid code duplication appears as the only reason preventing the size of the compiled term to be linear in the size of the input term. We speculate that a compilation scheme targeting a assembly-level language could be proved to be size-preserving.

REFERENCES

Arthur Charguéraud. 2011. Characteristic Formulae for the Verification of Imperative Programs. In *International Conference on Functional Programming*. 418–430. http://www.chargueraud.org/research/2011/cfml/main.pdf

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

David Delahaye. 2000. A Tactic Language for the System Coq. In *Logic for Programming and Automated Reasoning*, Michel Parigot and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 85–95.

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

Martin Erwig. 1996. Active Patterns. In *Implementation of Functional Languages*. https://api.semanticscholar.org/CorpusID:29734599

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

Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. 1992. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.* 27, 5 (May 1992), 1–164. https://doi.org/10.1145/130697.130699

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

Xavier Leroy. 2023. Proving the correctness of a compiler. https://xavierleroy.org/courses/EUTypes-2019/slides.pdf

Xavier Leroy and Michel Mauny. 1993. Dynamics in ML. *Journal of Functional Programming* 3, 4 (1993), 431–463. https://doi.org/10.1017/S0956796800000848

Daniel Licata and Simon Peyton Jones. 2007. View patterns: lightweight views for Haskell. https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/view_patterns.html

Luc Maranget. 2007. Warnings for pattern matching. *Journal of Functional Programming* 17, 3 (2007), 387–421. https://doi.org/10.1017/S0956796807006223

Nicholas D. Matsakis and Felix S. Klock. 2014. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (Portland, Oregon, USA) *(HILT '14)*. Association for Computing Machinery, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Robin Milner. 1984. A proposal for standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA) *(LFP '84)*. Association for Computing Machinery, New York, NY, USA, 184–197. https://doi.org/10.1145/800055.802035

Cyril Flurin Moser, Thodoris Sotiropoulos, Chengyu Zhang, and Zhendong Su. 2025. Validating Soundness and Completeness in Pattern-Match Coverage Analyzers. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 393 (Oct. 2025), 27 pages. https://doi.org/10.1145/3763171

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. An overview of the Scala programming language. (2004).

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

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

Giacomo Servadei. 2017. *Toward a more expressive pattern matching in Haskell.* Master thesis. POLITECNICO DI MILANO.

Don Syme, Gregory Neverov, and James Margetson. 2007a. Extensible pattern matching via a lightweight language extension. *SIGPLAN Not.* 42, 9 (Oct. 2007), 29–40. https://doi.org/10.1145/1291220.1291159

Don Syme, Gregory Neverov, and James Margetson. 2007b. 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.

D Turner. 1986. An overview of Miranda. *SIGPLAN Not.* 21, 12 (Dec. 1986), 158–166. https://doi.org/10.1145/15042.15053

D. A. Turner. 1981. The semantic elegance of applicative languages. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture* (Portsmouth, New Hampshire, USA) *(FPCA '81)*. Association for Computing Machinery, New York, NY, USA, 85–92. https://doi.org/10.1145/800223.806766

D. A. Turner. 1985. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

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

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

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

Andrew K. Wright and Robert Cartwright. 1994. A practical soft type system for Scheme. *SIGPLAN Lisp Pointers* VII, 3 (July 1994), 250–262. https://doi.org/10.1145/182590.182485

A MOTIVATION FOR THE "NEXT" CONSTRUCT

As mentioned in the introduction, certain algorithms involve a mixture of pattern matching and backtracking. To motivate the introduction of the **next** construct in Section 3.1, consider the `match` construct from Rocq's Ltac language.

```
match t0 with (* Ltac backtracking pattern matching *)
| p1 -> t1
| p2 -> t2
| p3 -> t3
```

Its semantics is as follows: let `x` denote the result of `t0`. If `x` matches `p1`, then evaluate the tactic `t1`. If the execution of `t1` fails for any reason, move on to the next branch, and test whether `x` matches `p2`, and so on. If all branches are exhausted, the pattern matching construct ultimately raises a failure.

How could such a behavior be described using a combination of exceptions and standard pattern matching, e.g., in standard OCaml? Suppose the `ti` are large expressions that should not be duplicated. We show below a first, direct attempt, involving a cascade of continuations defined in the reverse order compared with the original branches.

```
let k4 x = raise Fail in
let k3 x = match x with p3 -> (try t3 with Fail -> k4 x) | _ -> k4 x in
let k2 x = match x with p2 -> (try t2 with Fail -> k3 x) | _ -> k3 x in
let k1 x = match x with p1 -> (try t1 with Fail -> k2 x) | _ -> k2 x in
k1 t0
```

Could we introduce a higher-order combinator to express the branches in the original order? Yes, it could be realized using a combinator named `backtracking_match`, as follows.

```
backtracking_match t0
  [ (fun x -> match x with p1 -> Some t1 | _ -> None);
    (fun x -> match x with p2 -> Some t2 | _ -> None);
    (fun x -> match x with p3 -> Some t3 | _ -> None) ]

let backtracking_match (e:'a) (bs:'a->'b) : 'b =
  let x = e in
  let rec aux bs =
    match bs with
    | [] -> raise Fail
    | b::bs' -> match b x with Some r -> r | None -> aux bs'
    in
  aux bs
```

While this approach is semantically correct, the allocation of closures, lists and options make the code very inefficient. It might be possible for a hypothetical compiler to eliminate all these intermediate artefact (e.g., by means of partial evaluation). Even if a compiler were able to handle this particular example, it is not certain that it would handle more complex scenarios involving backtracking patterns.

In this paper, we introduce a construct, written `next L`, for aborting the execution of a branch of a pattern matching construct decorated with label `L`, and move on the next branch. The previous example could be written as follows—in a hypothetical syntax.

```
match^L t0 with
| p1 -> (try t1 with Fail -> next L)
| p2 -> (try t2 with Fail -> next L)
| p3 -> (try t3 with Fail -> next L)
```

As the example suggests, the `next` constructs empowers the programmer with fine-grained control, going beyond what a simple syntactic sugar for backtracking pattern matching could offer. Indeed, the programmer may decide, on a per-branch basis, which exception(s) should lead the program execution to test the remaining branches.

The `next` construct goes beyond what a simple syntactic sugar for backtracking pattern matching could offer. Indeed, the programmer may decide, on a per-branch basis, which exception(s) should lead the program execution to test the remaining branches.

B SELECTED BIG-STEP EVALUATION RULES FOR LABELED CONSTRUCTS

Figure 21 presents selected big-step rules for labeled constructs. A number of previously presented rules remain essentially unchanged, only updating the result from a value v to an outcome o. We do not include the rules for propagating exceptions and labeled-abort behaviors through all language constructs. We refer to the small-step semantics for a complete set of rules.

C TYPE SOUNDNESS

In this section, we prove preservation and progress by induction for the minimal language, w.r.t. the small-step semantics. For the language constructs not included in the minimal language, we prove, via the two lemmas shown below, that their evaluation and typing rules are derivable w.r.t. the encodings presented in Figure 15.

Theorem 4 (Soundness of derived evaluation rules). *For every encoded language construct (Figure 15), the big-step rules (Figure 21) are derivable w.r.t. the big-step rules for the minimal language (Figure 21 plus Figure 17).*

Theorem 5 (Soundness of derived typing rules). *For every encoded language construct (Figure 15), the typing rules (Figure 3, Figure 4, Figure 5 and Figure 12) are derivable w.r.t. the typing for the minimal language (Figure 29).*

To establish type soundness, we first need to introduce additional typing rules for values ($\vdash_{\text{val}} v : T$), outcomes ($F \vdash_{\text{out}} o : T$), abort outcomes ($F \vdash_{\text{abt}} a : T$), maps ($E \vdash_{\text{map}} M : B$), and results ($\vdash_{\text{res}} r : T$), as well as typing rules for the intermediate expressions that were introduced for stating the small-step semantics.

We reinterpret the existing rules, allowing outcomes and results to appear in evaluation position. For example, TYP-TRM-APP could typecheck the term $(\text{Res } v_0)(\text{Res } v_1, ..., \text{Res } v_k, o_{k+1}, t_{k+2}, ..., t_n)$. In the particular case of the bbe-conditional construct, we introduce an explicit rule TYP-BBE-IF-RES to typecheck expressions of the form $\textbf{if}^{\text{bbe}}\ M_0\ \textbf{then}\ b_1\ \textbf{else}\ b_2$. An alternative route would be to define $\text{pv}(M)$ as $\text{dom}(M)$.

The new typing rules appear in Figure 22. Interestingly, for the proof of soundness, we need to add a rule TYP-BBE-WEAKEN, which asserts that if b admits type B, and B' is a subset of B, then b also admits type B'. Recall that, intuitively, a BBE b has type B is a promise that if b evaluates to Match M, then the M binds at least all the names from the domain of B.

As expected, the proof involves the usual weakening and substitution lemmas.

Lemma 1 (Weakening lemmas for typing).
The following implications hold:

(1) $E; F \vdash_{\text{trm}} t : T \quad \Rightarrow \quad E, E'; F, F' \vdash_{\text{trm}} t : T$
(2) $E \vdash_{\text{bbe}} b \rightsquigarrow B \quad \Rightarrow \quad E, E' \vdash_{\text{bbe}} b \rightsquigarrow B$
(3) $E \vdash_{\text{pat}} p : T \rightsquigarrow B \quad \Rightarrow \quad E, E' \vdash_{\text{pat}} p : T \rightsquigarrow B$

Lemma 2 (Substitution lemmas for typing).
Assume $\vdash_{\text{map}} M : B$. *Then, the following implications hold:*

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

The next three lemmas are key to establishing type soundness: they explain how the bindings in result maps compare with the bindings computed in types.

Lemma 3 (only pattern variables are bound in result maps).

$$b \Downarrow_{\text{bbe}} \text{Match } M \quad \Rightarrow \quad \text{dom}(M) \subseteq \text{pv}(b)$$

Proof. Straightforward by induction on the evaluation judgment: only the pattern variable construct ($x^?$) introduces bindings in result maps, and the $\text{pv}(b)$ operation gathers all the pattern variables appearing in depth inside b. Hence, the result map contains □

Lemma 4 (disjoint pattern variables implies disjoint result maps).

$$b_1 \Downarrow_{\text{bbe}} \text{Match } M_1 \quad \wedge \quad b_2 \Downarrow_{\text{bbe}} \text{Match } M_2 \quad \wedge \quad \text{pv}(b_1) \cap \text{pv}(b_2) = \varnothing \quad \Rightarrow \quad M_1 \# M_2$$

Proof. Immediate corollary of the previous lemma. □

Lemma 5 (result maps contain all bindings expected from typing).

$$E \vdash_{\text{bbe}} b \rightsquigarrow B \quad \wedge \quad b \Downarrow_{\text{bbe}} \text{Match } M \quad \Rightarrow \quad \text{dom}(B) \subseteq \text{dom}(M)$$

$$E \vdash_{\text{val}} v : T \quad \wedge \quad \vdash_{\text{pat}} p : T \rightsquigarrow B \quad \wedge \quad v \triangleright p \Downarrow_{\text{pat}} \text{Match } M \quad \Rightarrow \quad \text{dom}(B) \subseteq \text{dom}(M)$$

Proof. By induction on the evaluation derivation, and case analysis on the typing derivation. For both judgments, bindings are introduced by pattern variables. In the typing judgment, bindings can be removed when computing intersections or when applying the weakening rule. In the evaluation judgment, bindings are only accumulated, never removed. Hence, the evaluation judgment returns no less bindings than the typing judgment. □

We are now ready to state and prove the preservation and progress theorems.

Theorem 6 (Preservation). *The following statements hold:*

(1) $E; F \vdash_{\text{trm}} e : T \quad \wedge \quad e \mapsto e' \quad \Rightarrow \quad E; F \vdash_{\text{trm}} e' : T$
(2) $E \vdash_{\text{bbe}} e \rightsquigarrow B \quad \wedge \quad e \mapsto e' \quad \Rightarrow \quad E \vdash_{\text{bbe}} e' \rightsquigarrow B$

Proof. The proof is done by induction on the small-step reduction rules. Let us justify how applications of the weakening rule (TYP-BBE-WEAKEN) for BBE-typechecking can be eliminated from the input derivation: if $e \rightsquigarrow B'$ is proved from $e \rightsquigarrow B$ for a B larger than B', then we can apply the preservation by induction hypothesis to derive $e' \rightsquigarrow B$, and conclude by applying the weakening rule to derive $e' \rightsquigarrow B'$.

- RED-CTX. Say in this context that $e = G[e_1]$ and $e' = G[e_1']$. By inversion of RED-CTX, e_1 reduces to e_1'. By inversion on the typing assumption of $G[e_1]$, e_1 has some type T_1. By IH, this means that e_1' has type T_1. Typing of $G[e_1']$ only depends on the type of e_1', not of the expression itself, meaning that from the same assumptions, $G[e_1]$ is well-typed.
- RED-ABT. By inversion of RED-ABT, Abort a is well-typed to some type T_1. By inversion of TYP-OUT-ABT, a is also well-typed. We conclude by applying the same rule with the type T.

From now on, we can assume that, if e is neither an if-then nor a label block, then the subterm is evaluation position is either of the form Res v, or a result r. (as any other form would be captured by either RED-CTX or RED-ABT). In these cases, we will systematically apply TYP-OUT-VAL to get the type of v.

- RED-INT. Both n and $\bar{n}$ have type int.
- RED-FUN. With Lemma 1 and TYP-VAL-FUN.
- RED-CSTR. By inversion of TYP-TRM-APP and TYP-TRM-PRIM, C has type $(T_1, ..., T_n) \to T$, and each v_i has type T_i. We deduce exactly the premises to apply TYP-VAL-CONSTR and conclude.
- RED-LET. By inversion of TYP-TRM-LET-VAR, v_1 has type T_1. We conclude by applying Lemma 2 to get that $\text{Subst}(\{x \mapsto v_1\}, t_2)$ has type T.
- RED-BETA. By inversion of TYP-TRM-APP, and TYP-TRM-FUN, we get that each v_i has type T_i, and that $\bar{\lambda}(x_1, ..., x_n).t$ has type $(T_1, ..., T_n) \to T$. We conclude with Lemma 2 on t.
- RED-IF-1. By inversion of TYP-TRM-IF-RES-LABEL, we obtain that t_2 has type T.
- RED-IF-2. By inversion of TYP-TRM-IF-RES-LABEL then of TYP-RES-MATCH, we deduce that the map M has a type B, that t_1 has type T in environment $E, B; F$, LblBranch L, and that t_2 has type T in E. Applying these premises with Lemma 2 to TYP-TRM-IFTHEN-LABEL is enough to conclude.
- RED-IFTHEN-1. By inversion of TYP-TRM-IFTHEN-LABEL, we obtain that t_2 has type T.
- RED-IFTHEN-2. By inversion of TYP-TRM-IFTHEN-LABEL, we obtain that o has type T.
- RED-WHILE. By assumption we get that **while** b **do** t **done** is well-typed, and by inversion of TYP-TRM-WHILE, b binds B, t has type unit with bindings from B. With these, we conclude by applying TYP-TRM-LET-VAR and TYP-TRM-IF.
- RED-BLOCK-1. By assuption $L : \{ \text{Abort } (\text{Exit } L\ v) \}$ is well-typed with type T. By inversion of TYP-TRM-BLOCK, Abort (Exit L v) is well-typed in a context where the label L has type T. By inversion of TYP-ABT-EXIT, v has type T. We conclude by applying TYP-OUT-VAL on v.

- RED-BLOCK-2. By inversion of TYP-TRM-BLOCK, we obtain that o has type T.
- RED-EXIT. By inversion of TYP-TRM-EXIT, the value v have type T and the the label L is bound in F to the same type T. We conclude by applying TYP-OUT-ABT, then TYP-ABT-EXIT and TYP-OUT-VAL on v.
- RED-NEXT. By inversion of TYP-TRM-NEXT, the label L is bound in the environment. We conclude by applying TYP-OUT-ABT, then TYP-ABT-NEXT.
- RED-IS. By inversion of TYP-BBE-IS, we get the exact premises to apply TYP-BBE-PAT.
- RED-IF-BBE-1. By assumption, we have that $\mathbf{if}^{\text{bbe}}$ Mismatch **then** b_1 **else** b_2 has some type B. By inversion of TYP-BBE-IF, we get that $B = (B_0 \uplus B_1) \cap B_2$, where Mismatch has type B_0, b_1 has type B_1, and b_2 has type B_2. The goal is to show that b_2 has type $(B_0 \uplus B_1) \cap B_2$. Since $(B_0 \uplus B_1) \cap B_2 \subseteq B_2$, we conclude by applying TYP-BBE-WEAKEN.
- RED-IF-BBE-2. By inversion of TYP-BBE-IF, we get that $B = (B_0 \uplus B_1) \cap B_2$, where Match M_0 has type B_0 (which means that M_0 has type B_0), b_1 has type B_1 with bindings from B_0, and b_2 has type B_2. We conclude by applying TYP-BBE-IF-RES.
- RED-IF-BBE-3. Applying TYP-RES-MISMATCH is enough.
- RED-IF-BBE-4. The input expression e is $\mathbf{if}^{\text{bbe}}$ M_0 **then** r_1 **else** b_2 with r_1 = Match M_1. Moreover, we have $M_0 \# M_1$. By inversion of TYP-BBE-IF-RES-RES, we obtain the typing assumptions $\vdash_{\text{map}} M_0 : B_0$ and $\vdash_{\text{res}}$ Match $M_1 : B_1$ and $\varnothing \vdash_{\text{bbe}} b_2 \rightsquigarrow B_2$. By inversion on TYPE-RES-MATCH, we deduce $\vdash_{\text{map}} M_1 : B_1$ We conclude with TYP-RES-MATCH, arguing that $M_0 \uplus M_1$ has type $(B_0 \uplus B_1) \cap B_2$. By TYP-BBE-WEAKEN, is suffices to show that $M_0 \uplus M_1$ has type $B_0 \uplus B_1$, which follows from $\vdash_{\text{map}} M_0 : B_0$ and $\vdash_{\text{map}} M_1 : B_1$.
- RED-PAT-VAR. By inversion of TYP-BBE-PAT and TYP-PAT-VAR, the variable x is bound to the type of v. We can condlude with TYP-RES-MATCH.
- RED-WHEN. By inversion of TYP-BBE-PAT, we get that v has type T and p **when** b has type $T \rightsquigarrow B$. Inversion of TYP-PAT-WHEN gives us that $B = B_1 \uplus B_2$, where p has type $T \rightsquigarrow B_1$, b has type B_2 with bindings from B_1, and $\text{pv}(p) \cap \text{pv}(b) = \varnothing$. By applying TYP-BBE-PAT, $v \triangleright p$ has type B_1. Since $\text{pv}(v \triangleright p) = \text{pv}(p)$, we get that $\text{pv}(v \triangleright p) \cap \text{pv}(b) = \varnothing$. We conclude by applying TYP-BBE-IF.
- RED-VIEW-1. Mismatch has any type.
- RED-VIEW-2. By inversion of TYP-BBE-PAT and TYP-VAL-PRIM on the constructor Some, each v_i has type T_i, and that for all i and j, the pattern variables appearing in p_i and p_j are disjoint. Since $\text{pv}(v_i \triangleright p_i) = \text{pv}(p_i)$, for all i and j, the pattern variables appearing in $\text{pv}(v_i \triangleright p_i)$ and $\text{pv}(v_j \triangleright p_j)$ are disjoint. We conclude by applying TYP-BBE-IF $(n-1)$ times.

□

THEOREM 7 (PROGRESS). *The following statements hold:*

(1) $\varnothing; F \vdash_{\text{trm}} t : T \quad \Rightarrow \quad t$ is an outcome $o \quad \vee \quad \exists t'.\, t \mapsto t'$
(2) $\varnothing \vdash_{\text{bbe}} b \rightsquigarrow B \quad \Rightarrow \quad b$ is a result $r \quad \vee \quad \exists b'.\, b \mapsto b'$

PROOF. The proof is done by induction on the typing rules. For the typing rules, we factorize the similar cases as follows. Consider an entity e.

- If e is already an outcome o or a result r, the result is immediate.

- If e is of the form $G[e_1]$, and the e_1 in evaluation position is either a term t or a BBE b, it suffices to invoke the induction hypothesis (IH) and the reduction RED-CTX applies.
- If e is of the form $G[\text{Abort } a]$, and G is not a labeled-block nor a ifthen-construct, the Abort a propagate outwards: the reduction RED-ABT applies.

Therefore there remains to consider well-typed terms in which the subterm in evaluation position, if any, is either a Res v or a r; or an Abort a in the case of a labeled-block or ifthen-construct.

- TYP-TRM-VAR. A variable x cannot be well-typed in an empty environment, so the assumption is contradictory.
- TYP-TRM-INT. The rule RED-INT applies.
- TYP-TRM-LET-VAR. The only case to discuss is that where t_1 is of the form Res v. We can conclude by applying RED-LET.
- TYP-TRM-FUN. The rule RED-FUN applies.
- TYP-TRM-APP. The case to discuss is of the form $(\text{Res } v_0)\ (\text{Res } v_1, ..., \text{Res } v_n)$ The function v_0 has an arrow type. Therefore it is either a primitive function or a lambda-abstraction.
  - If v_0 is a constructor C, we apply RED-CSTR.
  - If v_0 is an abstraction $(\bar{\lambda}(x_1, ..., x_n).t)$, we apply RED-BETA.
- TYP-TRM-IF-LABEL. The argument b is not yet a result, hence term t at hand is of the form $G[b]$, and the rule RED-CTX applies.
- TYP-TRM-WHILE. The while loop unfolds, by application of RED-WHILE.
- TYP-TRM-BLOCK. Consider a block $L : \{ o \}$.
  - If o is of the form Abort $(\text{Exit } L\ v)$, we apply RED-BLOCK-1.
  - If o is not of this form, we apply RED-BLOCK-2.
- TYP-TRM-EXIT. The rule RED-EXIT applies.
- TYP-TRM-NEXT. The rule RED-NEXT applies.
- TYP-TRM-IF-RES-LABEL. The case to discuss is when the argument of the condition is a result r. If r is Mismatch, then RED-IF-1 applies. If r is of the form Match M, then RED-IF-2 applies.
- TYP-TRM-IFTHEN-LABEL. Consider $\textbf{ifthen}_L\ o\ \textbf{else}\ t_2$.
  - If o is of the form Abort $(\text{Next } L)$, we apply RED-IFTHEN-1.
  - If o is not of this form, we apply RED-IFTHEN-2.
- TYP-BBE-IS. The rule RED-IS applies.
- TYP-BBE-IF. The argument of the conditional is either a Mismatch or a Match M. We apply the rule RED-IF-BBE-1 or RED-IF-BBE-2 accordingly.
- TYP-BBE-IF-RES. The BBE at hand is $\textbf{if}^{\text{bbe}}\ M_0\ \textbf{then}\ b_1\ \textbf{else}\ b_2$. The b_1 is in evaluation position, hence RED-CTX applies.
- TYP-BBE-IF-RES-RES. The case to consider is $\textbf{if}^{\text{bbe}}\ M_0\ \textbf{then}\ r_1\ \textbf{else}\ b_2$. The typing premise available is $\forall M_1.\ r_1 = \text{Match } M_1 \Rightarrow M_0 \# M_1$. By case analysis on r_1. If r_1 is Mismatch then RED-IF-BBE-3 applies. If, however, r_1 is of the form Match M_1, then RED-IF-BBE-4 applies, under the condition $M_0 \# M_1$, which comes from specializing the typing premise.

- TYP-BBE-PAT. By case analysis on the grammar of patterns: p is either pattern variable, or when-pattern, or if-some pattern; Each of these three cases are covered by the following rules.
- TYP-PAT-VAR. The rule RED-PAT-VAR applies.
- TYP-PAT-WHEN. The rule RED-WHEN applies.
- TYP-PAT-SOME. Consider $v \triangleright \mathrm{Some}(p_1, ..., p_n)$. By inversion on the typing rule TYP-BBE-PAT, v has some type T and the pattern also has type T. By inversion on the typing rule TYP-PAT-SOME for the pattern, we deduce that T is of the form $(T_1, ..., T_n)$ option. By inversion on the typing of v with type $(T_1, ..., T_n)$ option, v must be an option constructor. If v is None then the rule RED-VIEW-1 applies. Otherwise, v is $\mathrm{Some}(v_1, ..., v_n)$ and the rule RED-VIEW-2 applies.
- TYP-BBE-WEAKEN. The induction hypothesis applies immediately.

□

D SYNTAX AND SEMANTICS OF CORE-ML, THE COMPILATION TARGET

Figure 23 and Figure 24 give the semantics of the target language.

E AUXILIARY OPERATIONS ON VARIABLES

Figure 25, Figure 26 and Figure 27 formally define the operations for computing pattern variables, free variables, and bound variables, respectively.

The BBE false is a special case. Even though no variables are syntactically bound by false, any possible evaluation would result to a Mismatch, in the same vein as the typing rule TYP-BBE-FALSE. For this reason, we define $\mathrm{bv}(\mathrm{false})$ as $\mathbb{V}$, where $\mathbb{V}$ is the infinite set containing every possible variable.

F CORRECTNESS OF THE COMPILATION SCHEME

To state the semantic-preservation theorem, we formalize the encoding of values and outcomes in Figure 28.

The semantic-preservation theorem asserts that if a closed term t evaluates to an outcome o in the source language, then the translation of t evaluates to the translation of o. This forward-simulation result suffices to establish a compiler correctness result because the language that we considered is deterministic [Leroy 2023].

To set up the proof by induction, we need two auxiliary statements, for the semantic preservation of BBEs and patterns. They involve the auxiliary predicates $\mathrm{TrCont}(r, \mathsf{u}, \mathsf{u}', \mathsf{q})$ and $\mathrm{FvCont}(B, \mathsf{u}, \mathsf{u}')$. Intuitively, $\mathrm{TrCont}(r, \mathsf{u}, \mathsf{u}', \mathsf{q})$ constrains the evaluation result of the continuations u and u' involved in the translation $[\![b]\!]^{\mathsf{u}}_{\mathsf{u}'}$ of a BBE b (or of a pattern). $\mathrm{FvCont}(B, \mathsf{u}, \mathsf{u}')$ constrains the set of free variables that may appear in the continuations u and u'. The proof is carried out by mutual induction on the following three statements.

LEMMA 6 (BOUND ON FREE VARIABLES). *The following statements hold:*

(1) $\mathrm{fv}([\![t]\!]) \subseteq \mathrm{fv}(t)$
(2) $\mathrm{fv}([\![b]\!]^{\mathsf{u}}_{\mathsf{u}'}) \subseteq \mathrm{fv}(b) \cup (\mathrm{fv}(\mathsf{u}) \setminus \mathrm{bv}(b)) \cup \mathrm{fv}(\mathsf{u}')$
(3) $\mathrm{fv}([\![\mathsf{y} \blacktriangleright p]\!]^{\mathsf{u}}_{\mathsf{u}'}) \subseteq \mathrm{fv}(p) \cup (\mathrm{fv}(\mathsf{u}) \setminus \mathrm{bv}(p)) \cup \mathrm{fv}(\mathsf{u}')$

Proof. Straightforward proof by induction on the size of the AST. Inclusion comes from the possibility of $\mathrm{bv}(b) = \mathbb{V}$. □

Remark. In particular, by definition,

- $(\mathrm{bv}(b) = \mathbb{V}) \;\Rightarrow\; (\mathrm{fv}([\![b]\!]^{\mathsf{u}}_{\mathsf{u}'}) \subseteq \mathrm{fv}(\mathsf{u}'))$
- $(\mathrm{bv}(p) = \mathbb{V}) \;\Rightarrow\; (\mathrm{fv}([\![\mathsf{y} \blacktriangleright p]\!]^{\mathsf{u}}_{\mathsf{u}'}) \subseteq \mathrm{fv}(\mathsf{u}'))$

Lemma 7 (Variables bound by well-typed BBEs and patterns). *The following statements hold:*

(1) $E; F \vdash_{\mathrm{bbe}} b \rightsquigarrow B \;\Rightarrow\; \mathrm{dom}(B) \subseteq \mathrm{bv}(b)$
(2) $E; F \vdash_{\mathrm{pat}} p : T \rightsquigarrow B \;\Rightarrow\; \mathrm{dom}(B) \subseteq \mathrm{bv}(p)$

Proof. Straightforward by induction on the typing judgment □

Lemma 8 (Free variables of well-typed entities). *The following statements hold:*

(1) $E; F \vdash_{\mathrm{trm}} t : T \;\Rightarrow\; \mathrm{fv}(t) \subseteq \mathrm{dom}(E)$
(2) $E \vdash_{\mathrm{bbe}} b \rightsquigarrow B \;\Rightarrow\; \mathrm{fv}(b) \subseteq \mathrm{dom}(E)$
(3) $E \vdash_{\mathrm{pat}} p : T \rightsquigarrow B \;\Rightarrow\; \mathrm{fv}(p) \subseteq \mathrm{dom}(E)$

Proof. Straightforward by induction on the typing judgment. □

Lemma 9 (Type soudness big-step style). *The following statement holds:* $(t \Downarrow_{\mathrm{trm}} \mathrm{Res}\ v) \;\wedge\; (\vdash_{\mathrm{trm}} t : T) \;\Rightarrow\; (\vdash_{\mathrm{val}} v : T)$

Proof. The result comes by type soundness of the small-step semantics, and equivalence between the small-step and big-step evaluation rules. □

Theorem 8 (Compilation correctness). *The following statements hold:*

(1) $(t \Downarrow_{\mathrm{trm}} o) \;\wedge\; \varnothing; F \vdash_{\mathrm{trm}} t : T \;\Rightarrow\; ([\![t]\!] \Downarrow_{\mathrm{ml}} [\![o]\!])$
(2) $(b \Downarrow_{\mathrm{bbe}} r) \;\wedge\; \vdash_{\mathrm{bbe}} b \rightsquigarrow B \;\wedge\; \mathrm{TrCont}(r, \mathsf{u}, \mathsf{u}', \mathsf{q}) \;\wedge\; \mathrm{FvCont}(B, \mathsf{u}, \mathsf{u}') \;\Rightarrow$
$([\![b]\!]^{\mathsf{u}}_{\mathsf{u}'} \Downarrow_{\mathrm{ml}} \mathsf{q})$
(3) $(v \triangleright p \Downarrow_{\mathrm{pat}} r) \;\wedge\; (\vdash_{\mathrm{val}} v : T) \;\wedge\; (\vdash_{\mathrm{pat}} p : T \rightsquigarrow B) \;\wedge$
$\mathrm{TrCont}(r, \mathsf{u}, \mathsf{u}', \mathsf{q}) \;\wedge\; \mathrm{FvCont}(B, \mathsf{u}, \mathsf{u}') \;\Rightarrow\; ((([\![v]\!] \blacktriangleright p]\!]^{\mathsf{u}}_{\mathsf{u}'}) \Downarrow_{\mathrm{ml}} \mathsf{q}) \cdot$

where

$$\begin{array}{lcl} \mathrm{TrCont}(r, \mathsf{u}, \mathsf{u}', \mathsf{q}) & := & (r = \mathrm{Mismatch} \;\wedge\; \mathsf{u}' \Downarrow_{\mathrm{ml}} \mathsf{q}) \\ & & \vee\; (\exists M.\; r = \mathrm{Match}\ M \;\wedge\; \mathrm{Subst}([\![M]\!], \mathsf{u}) \Downarrow_{\mathrm{ml}} \mathsf{q}) \\ \mathrm{FvCont}(B, \mathsf{u}, \mathsf{u}') & := & (\mathrm{fv}(\mathsf{u}') = \varnothing) \;\wedge\; (\mathrm{fv}(\mathsf{u}) \subseteq \mathrm{dom}(B)) \end{array}$$

Proof. The proof is done by mutual induction on the evaluation rules.

The arguments for justifying typing conditions in each case are very similar to the ones in the proof of Theorem 2, and are left out for conciseness.

1. Evaluation rules for terms.

- eval-trm-int.
  The conclusion of the evaluation rule is $n \Downarrow_{\mathrm{trm}} \mathrm{Res}\ \bar{n}$. We have to prove that $[\![n]\!] \Downarrow_{\mathrm{ml}} [\![\mathrm{Res}\ \bar{n}]\!]$, which simplifies to $\mathsf{n} \Downarrow_{\mathrm{ml}} \texttt{Result n}$.
  This result is an application of ml-int.

- EVAL-TRM-LET.
  The conclusion of the evaluation rule is **let** $x = t_1$ **in** $t_2 \Downarrow_{\text{trm}} o$. Its premises are $t_1 \Downarrow_{\text{trm}} \text{Res}\ v_1$ and $\text{Subst}(\{x \mapsto v_1\}, t_2) \Downarrow_{\text{trm}} o$. By application of IH (1) on both premises, we obtain "$[\![t_1]\!] \Downarrow_{\text{ml}} \texttt{Result}\ [\![v_1]\!]$" and "$[\![\text{Subst}(\{x \mapsto v_1\}, t_2)]\!] \Downarrow_{\text{ml}} [\![o]\!]$", from which we get "$\text{Subst}(\{\texttt{x} \mapsto [\![v_1]\!]\}, [\![t_2]\!]) \Downarrow_{\text{ml}} [\![o]\!]$" by Lemma 10.
  We have to prove that $\texttt{let x =}\ [\![t_1]\!]\ \texttt{in}\ [\![t_2]\!] \Downarrow_{\text{ml}} [\![o]\!]$.
  We prove this by applying ML-LET-1, with $[\![t_1]\!] \Downarrow_{\text{ml}} \texttt{Result}\ [\![v_1]\!]$ and $\text{Subst}(\{\texttt{x} \mapsto [\![v_1]\!]\}, [\![t_2]\!]) \Downarrow_{\text{ml}} [\![o]\!]$.
- EVAL-TRM-FUN.
  The conclusion of the evaluation rule is $\lambda(x_1, ..., x_n).\, t \Downarrow_{\text{trm}} \text{Res}\ \bar{\lambda}(x_1, ..., x_n).t$.
  We have to prove that $[\![\lambda(x_1, ..., x_n).\, t]\!] \Downarrow_{\text{ml}} [\![\text{Res}\ \bar{\lambda}(x_1, ..., x_n).t]\!]$, which simplifies to $\texttt{(fun (x}_1, ..., \texttt{x}_n\texttt{) } \rightarrow\ [\![t]\!]) \Downarrow_{\text{ml}} \texttt{Result (fun (x}_1, ..., \texttt{x}_n\texttt{) } \rightarrow\ [\![t]\!])$.
  This result is an application of ML-FIX.
- EVAL-TRM-BETA.
  The conclusion of the evaluation rule is $t_0\ (t_1, ..., t_n) \Downarrow_{\text{trm}} o$. Its premises are "$t_0 \Downarrow_{\text{trm}} \text{Res}\ (\bar{\lambda}(x_1, ..., x_n).t)$" and "$\forall i \in [1, n].\ t_i \Downarrow_{\text{trm}} \text{Res}\ v_i$" and "$\text{Subst}(M, t) \Downarrow_{\text{trm}} o$", where $M = \{x_1 \mapsto v_1, ..., x_n \mapsto v_n\}$. By application of IH (1) on each premise, we have "$[\![t_0]\!] \Downarrow_{\text{ml}} \texttt{Result (fun (x}_1, ..., \texttt{x}_n\texttt{) } \rightarrow\ [\![t]\!])$" and "$\forall i \in [1, n].\ [\![t_i]\!] \Downarrow_{\text{ml}} \texttt{Result}\ [\![v_i]\!]$" and "$[\![\text{Subst}(M,\ t)]\!] \Downarrow_{\text{ml}} [\![o]\!]$", from which we get "$\text{Subst}([\![M]\!], [\![t]\!]) \Downarrow_{\text{ml}} [\![o]\!]$" by Lemma 10.
  We have to prove that $[\![t_0]\!]\ ([\![t_1]\!], ..., [\![t_n]\!]) \Downarrow_{\text{ml}} [\![o]\!]$.
  We prove this by applying ML-BETA with "$[\![t_0]\!] \Downarrow_{\text{ml}} \texttt{Result (fun (x}_1, ..., \texttt{x}_n\texttt{) } \rightarrow\ [\![t]\!])$" and "$\forall i \in [1, n].\ [\![t_i]\!] \Downarrow_{\text{ml}} \texttt{Result}\ [\![v_i]\!]$" and "$\text{Subst}([\![M]\!], [\![t]\!]) \Downarrow_{\text{ml}} [\![o]\!]$".
- EVAL-TRM-IF-LABEL-1.
  The conclusion of the evaluation rule is "$\textbf{if}_L\ b_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \Downarrow_{\text{trm}} o$". Its premises are "$b_0 \Downarrow_{\text{bbe}} \text{Mismatch}$" and "$t_2 \Downarrow_{\text{trm}} o$".
  By application of IH (1) on "$t_2 \Downarrow_{\text{trm}} o$", we have $[\![t_2]\!] \Downarrow_{\text{ml}} [\![o]\!]$.
  The typing assumption is "$\varnothing;\ F \vdash_{\text{trm}} \textbf{if}_L\ b_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 : T$", from which we get by inversion that $\vdash_{\text{bbe}} b_0 \rightsquigarrow B_0$ and $B_0;\ F,\ (\text{LblBranch}\ L\ :\ T) \vdash_{\text{trm}} t_1 : T$ and $\varnothing;\ F \vdash_{\text{trm}} t_2 : T$.
  By using Lemma 8 on the typing judgments, we have "$\text{fv}(b_0) = \varnothing$" and "$\text{fv}(t_1) \subseteq \text{dom}(B_0)$" and "$\text{fv}(t_2) = \varnothing$".
  We have to prove that $\texttt{let k() =}\ [\![t_2]\!]\ \texttt{in}\ [\![b_0]\!]^{\texttt{try}\ [\![t_1]\!]\ \texttt{with Next L} \rightarrow \texttt{k()}}_{\texttt{k()}} \Downarrow_{\text{ml}} [\![o]\!]$.
  By application of ML-LET-1, it suffices to prove that $[\![b_0]\!]^{\texttt{u}}_{\texttt{u}'} \Downarrow_{\text{ml}} [\![o]\!]$, where $\texttt{w}_{[\![t_2]\!]} = \texttt{(fun () } \rightarrow\ [\![t_2]\!])$ and $\texttt{u} = \texttt{try}\ [\![t_1]\!]\ \texttt{with Next L} \rightarrow\ \texttt{w}_{[\![t_2]\!]}\texttt{()}$ and $\texttt{u}' = \texttt{w}_{[\![t_2]\!]}\texttt{()}$.
  We prove this by applying IH (2) on "$b \Downarrow_{\text{bbe}} \text{Mismatch}$".
  There remains to prove the premises of IH (2), that is:
  $\text{TrCont}(\text{Mismatch}, \texttt{u}, \texttt{u}', [\![o]\!])$ and $\text{FvCont}(B_0, \texttt{u}, \texttt{u}')$.

  The property $\text{TrCont}(\text{Mismatch}, \texttt{u}, \texttt{u}', [\![o]\!])$ simplifies to $\texttt{w}_{[\![t_2]\!]}\texttt{()} \Downarrow_{\text{ml}} [\![o]\!]$, which we prove by applying ML-BETA with $[\![t_2]\!] \Downarrow_{\text{ml}} [\![o]\!]$.

  The property $\text{FvCont}(B, \texttt{u}, \texttt{u}')$ simplifies to "$\text{fv}(\texttt{u}') = \varnothing \wedge \text{fv}(\texttt{u}) \subseteq \text{dom}(B)$".

We prove $\mathrm{fv}(\mathsf{u}') = \mathrm{fv}(\mathsf{w}_{[\![t_2]\!]}\,()) = \mathrm{fv}([\![t_2]\!]) = \varnothing$ by applying Lemma 6 to get $\mathrm{fv}([\![t_2]\!]) \subseteq \mathrm{fv}(t_2) = \varnothing$.
We have that $\mathrm{fv}(\mathsf{u}) = \mathrm{fv}(\texttt{try}\ [\![t_1]\!]\ \texttt{with Next L} \to \mathsf{w}_{[\![t_2]\!]}\,()) = \mathrm{fv}([\![t_1]\!]) \cup \mathrm{fv}(\mathsf{w}_{[\![t_2]\!]}\,()) = \mathrm{fv}([\![t_1]\!]) \cup \mathrm{fv}([\![t_2]\!]) = \mathrm{fv}([\![t_1]\!])$ since $\mathrm{fv}([\![t_2]\!]) = \varnothing$. We conclude by applying Lemma 6 to get $\mathrm{fv}([\![t_1]\!]) \subseteq \mathrm{fv}(t_1) \subseteq \mathrm{dom}(B)$.

- EVAL-TRM-IF-LABEL-2.
The conclusion of the evaluation rule is $\mathbf{if}_L\ b_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \Downarrow_{\mathrm{trm}} o$. Its premises are "$b_0 \Downarrow_{\mathrm{bbe}}$ Match M_0" and "Subst$(M_0, t_1) \Downarrow_{\mathrm{trm}}$ Abort (Next L)" and "$t_2 \Downarrow_{\mathrm{trm}} o$".
By application of IH (1), we have $[\![\mathrm{Subst}(M_0, t_1)]\!] \Downarrow_{\mathrm{ml}} \texttt{Exn (Next L)}$, from which we get "Subst$([\![M_0]\!], [\![t_1]\!]) \Downarrow_{\mathrm{ml}} \texttt{Exn (Next L)}$" by Lemma 10, as well as $[\![t_2]\!] \Downarrow_{\mathrm{ml}} [\![o]\!]$.
The typing assumption is "$\varnothing; F \vdash_{\mathrm{trm}} \mathbf{if}_L\ b_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : T$", from which we get by inversion "$\vdash_{\mathrm{bbe}} b_0 \rightsquigarrow B_0$" and "$B_0; F, (\mathrm{LblBranch}\ L\ :\ T) \vdash_{\mathrm{trm}} t_1 : T$" and "$\varnothing; F \vdash_{\mathrm{trm}} t_2 : T$".
By using Lemma 8 on the typing judgments, we have $\mathrm{fv}(b_0) = \varnothing$ and $\mathrm{fv}(t_1) \subseteq \mathrm{dom}(B_0)$ and $\mathrm{fv}(t_2) = \varnothing$.
We have to prove that $\texttt{let k() =}\ [\![t_2]\!]\ \texttt{in}\ [\![b_0]\!]^{\texttt{try}\ [\![t_1]\!]\ \texttt{with Next L} \to \texttt{k()}}_{\texttt{k()}} \Downarrow_{\mathrm{ml}} [\![o]\!]$.
By application of ML-LET-1, it suffices to prove that $[\![b_0]\!]^{\mathsf{u}}_{\mathsf{u}'} \Downarrow_{\mathrm{ml}} [\![o]\!]$, where $\mathsf{w}_{[\![t_2]\!]} = (\texttt{fun () →}\ [\![t_2]\!])$ and $\mathsf{u} = \texttt{try}\ [\![t_1]\!]\ \texttt{with Next L} \to \mathsf{w}_{[\![t_2]\!]}\,()$ and $\mathsf{u}' = \mathsf{w}_{[\![t_2]\!]}\,()$.
We prove this by applying IH (2) on "$b_0 \Downarrow_{\mathrm{bbe}}$ Match M_0".
There remains to prove the premises of IH (2), that is:
TrCont(Match M_0, u, u′, $[\![o]\!]$) and FvCont(B_0, u, u′).

The property TrCont(Match M_0, u, u′, $[\![o]\!]$) simplifies to
Subst$([\![M_0]\!], \texttt{try}\ [\![t_1]\!]\ \texttt{with Next L} \to \mathsf{w}_{[\![t_2]\!]}\,()) \Downarrow_{\mathrm{ml}} [\![o]\!]$, which itself simplifies to
$\texttt{try}\ \mathrm{Subst}([\![M_0]\!], [\![t_1]\!])\ \texttt{with Next L} \to \mathrm{Subst}([\![M_0]\!], \mathsf{w}_{[\![t_2]\!]}\,()) \Downarrow_{\mathrm{ml}} [\![o]\!]$.
This is proved by applying ML-TRY-WITH-1, with Subst$([\![M_0]\!], [\![t_1]\!]) \Downarrow_{\mathrm{ml}} \texttt{Exn (Next L)}$ and $[\![t_2]\!] \Downarrow_{\mathrm{ml}} [\![o]\!]$.

The property FvCont(B_0, u, u′) is proved like in the case EVAL-TRM-IF-LABEL-1.

- EVAL-TRM-IF-LABEL-3.
The conclusion of the evaluation rule is $\mathbf{if}_L\ b_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \Downarrow_{\mathrm{trm}} o$. Its premises are "$b_0 \Downarrow_{\mathrm{bbe}}$ Match M_0" and "Subst$(M_0, t_1) \Downarrow_{\mathrm{trm}} o$" and "$o \neq$ Abort (Next L)".
By application of IH (1) on the premises, we have "$[\![\mathrm{Subst}(M_0, t_1)]\!] \Downarrow_{\mathrm{ml}} [\![o]\!]$", from which we get "Subst$([\![M_0]\!], [\![t_1]\!]) \Downarrow_{\mathrm{ml}} [\![o]\!]$" by Lemma 10.
The typing assumption is $\varnothing; F \vdash_{\mathrm{trm}} \mathbf{if}_L\ b_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : T$, from which we get by inversion that $\vdash_{\mathrm{bbe}} b_0 \rightsquigarrow B_0$ and $B_0; F, (\mathrm{LblBranch}\ L\ :\ T) \vdash_{\mathrm{trm}} t_1 : T$ and $\varnothing; F \vdash_{\mathrm{trm}} t_2 : T$.
By using Lemma 8 on the typing judgments, we have $\mathrm{fv}(b_0) = \varnothing$ and $\mathrm{fv}(t_1) \subseteq \mathrm{dom}(B_0)$ and $\mathrm{fv}(t_2) = \varnothing$.
By inspecting Figure 28, only results of the form "Next L" can translate to Exn (Next L).
We deduce from $o \neq$ Abort (Next L) that $[\![o]\!] \neq \texttt{Exn (Next L)}$.
We have to prove that $\texttt{let k() =}\ [\![t_2]\!]\ \texttt{in}\ [\![b_0]\!]^{\texttt{try}\ [\![t_1]\!]\ \texttt{with Next L} \to \texttt{k()}}_{\texttt{k()}} \Downarrow_{\mathrm{ml}} [\![o]\!]$.
By application of ML-LET-1, it suffices to prove that $[\![b_0]\!]^{\mathsf{u}}_{\mathsf{u}'} \Downarrow_{\mathrm{ml}} [\![o]\!]$, where $\mathsf{w}_{[\![t_2]\!]} = (\texttt{fun () →}\ [\![t_2]\!])$ and $\mathsf{u} = \texttt{try}\ [\![t_1]\!]\ \texttt{with Next L} \to \mathsf{w}_{[\![t_2]\!]}\,()$ and $\mathsf{u}' = \mathsf{w}_{[\![t_2]\!]}\,()$.

We prove this by applying IH (2) on "$b_0 \Downarrow_{\text{bbe}} \text{Match } M_0$".
There remains to prove the premises of IH (2), that is:
TrCont(Match M_0, u, u′, $[\![o]\!]$) and FvCont(B, u, u′).

The property TrCont(Match M_0, u, u′, $[\![o]\!]$) simplifies to
`try` $[\![t_1]\!]$ `with Next L →` $\mathtt{w}_{[\![t_2]\!]}\,() \Downarrow_{\text{ml}} [\![o]\!]$, which itself simplifies to
`try` Subst($[\![M_0]\!]$, $[\![t_1]\!]$) `with Next L →` Subst($[\![M_0]\!]$, $\mathtt{w}_{[\![t_2]\!]}\,()$) $\Downarrow_{\text{ml}} [\![o]\!]$.
This is proved by applying ML-TRY-WITH-2, with Subst($[\![M_0]\!]$, $[\![t_1]\!]$) $\Downarrow_{\text{ml}} [\![o]\!]$ and $[\![o]\!] \neq$ `Exn (Next L)`.

The property FvCont(B_0, u, u′) is proved like in the case EVAL-TRM-IF-LABEL-1.

- EVAL-TRM-WHILE-1.
The conclusion of the evaluation rule is "**while** b **do** t **done** $\Downarrow_{\text{trm}} o$". Its premises are "$b \Downarrow_{\text{bbe}} \text{Match } M$" and "Subst($M$, t); **while** b **do** t **done** $\Downarrow_{\text{trm}} o$".
By inversion on "Subst(M, t); **while** b **do** t **done** $\Downarrow_{\text{trm}} o$", we have Subst($M$, t) $\Downarrow_{\text{trm}}$ Res v_1 and **while** b **do** t **done** $\Downarrow_{\text{trm}} o$. Recall that we have omitted the mutable state, but the premise corresponds to a state σ' reached after executing the body once.
By application of IH (1) on each subpremise, we have $[\![$**while** b **do** t **done**$]\!] \Downarrow_{\text{ml}} [\![o]\!]$ (in state σ') and $[\![\text{Subst}(M, t)]\!] \Downarrow_{\text{trm}}$ `Result` $[\![v_1]\!]$, from which we can deduce Subst($[\![M]\!]$, $[\![t]\!]$) $\Downarrow_{\text{trm}}$ `Result` $[\![v_1]\!]$.
The typing assumption is "$\varnothing$; $F \vdash_{\text{trm}}$ **while** b **do** t **done** : T", from which we get by inversion that "$\vdash_{\text{bbe}} b \rightsquigarrow B$" and "$B$; $F \vdash_{\text{trm}} t$: unit".
By using Lemma 8 on the typing judgments, we have $\text{fv}(b) = \varnothing$ and $\text{fv}(t) \subseteq \text{dom}(B)$.
By definition of the translation, $[\![$**while** b **do** t **done**$]\!] = \mathtt{w}_F\,()$,
where $\mathtt{w}_F$ = `(fix f () →` $[\![b]\!]^{([\![t]\!];\mathtt{f}())}_{()}$`)`.
We have to show that "$\mathtt{w}_F\,() \Downarrow_{\text{ml}} [\![o]\!]$".
By applying ML-BETA, it suffices to show that $[\![b]\!]^{\mathtt{u}}_{()} \Downarrow_{\text{ml}} [\![o]\!]$, where u = $[\![t]\!]$; $\mathtt{w}_F\,()$.
We prove this by applying IH (2) on "$b \Downarrow_{\text{bbe}} \text{Match } M$".
There remains to prove the premises of IH (2), that is: TrCont(Match M, u, `()`, $[\![o]\!]$) and FvCont(B, u, `()`).

The property TrCont(Match M, u, `()`, $[\![o]\!]$) simplifies to "Subst($[\![M]\!]$, $[\![t]\!]$; $\mathtt{w}_F\,()$) $\Downarrow_{\text{ml}}$ $[\![o]\!]$", which itself simplifies to "Subst($[\![M]\!]$, $[\![t]\!]$); Subst($[\![M]\!]$, $\mathtt{w}_F\,()$) $\Downarrow_{\text{ml}} [\![o]\!]$".
Remember that $\mathtt{w}_F\,() = [\![$**while** b **do** t **done**$]\!]$. We get from Lemma 6 that $\text{fv}(\mathtt{w}_F\,()) \subseteq \text{fv}($**while** b **do** t **done**$) = \varnothing$. This means that the substitution operation in Subst($[\![M]\!]$, $\mathtt{w}_F\,()$) is the identity. We get from this that Subst($[\![M]\!]$, $\mathtt{w}_F\,()$) = $\mathtt{w}_F\,()$, so that $\mathtt{w}_F\,() \Downarrow_{\text{ml}} [\![o]\!]$.
We conclude by applying ML-LET-1 with Subst($[\![M]\!]$, $[\![t]\!]$) $\Downarrow_{\text{ml}}$ `Result` $[\![v_1]\!]$ and $\mathtt{w}_F\,() \Downarrow_{\text{ml}} [\![o]\!]$.

The property FvCont(B, u, `()`) simplifies to ($\text{fv}([\![t]\!]; \mathtt{w}_F\,()) \subseteq \text{dom}(B)$). We have $\text{fv}([\![t]\!]; \mathtt{w}_F\,()) = \text{fv}([\![t]\!]) \cup \text{fv}(\mathtt{w}_F\,())$. We proved earlier that $\text{fv}(\mathtt{w}_F\,()) = \varnothing$, and we have by Lemma 7 that $\text{fv}([\![t]\!]) \subseteq \text{fv}(t) \subseteq \text{dom}(B)$.
We conclude that $\text{fv}([\![t]\!];\ \mathtt{w}_F\,()) \subseteq \text{dom}(B)$.

- EVAL-TRM-WHILE-2.
  The conclusion of the evaluation rule is **while** b **do** t **done** $\Downarrow_{\text{trm}}$ tt. Its premise is "$b \Downarrow_{\text{bbe}}$ Mismatch".
  The typing assumption is "$\varnothing$; $F \vdash_{\text{trm}}$ **while** b **do** t **done** : T", from which we get by inversion that "$\vdash_{\text{bbe}} b \rightsquigarrow B$" and "$B$; $F \vdash_{\text{trm}} t$: unit".
  By using Lemma 8 on the typing judgments, we have $\text{fv}(b) = \varnothing$ and $\text{fv}(t) \subseteq \text{dom}(B)$.
  By definition of the translation, $[\![$**while** b **do** t **done**$]\!]$ = $\mathtt{w_F}$ `()`,
  where $\mathtt{w_F}$ = `(fix f () →` $[\![b]\!]^{([\![t]\!];\mathtt{f()})}_{()}$`)`.
  We have to show that "$\mathtt{w_F}$ `()` $\Downarrow_{\text{ml}}$ $[\![tt]\!]$".
  By applying ML-BETA, it suffices to show that $[\![b]\!]^{\mathtt{u}}_{()}$ $\Downarrow_{\text{ml}}$ $[\![tt]\!]$, where u = $[\![t]\!]$; $\mathtt{w_F}$ `()`.
  We prove this by applying IH (2) on "$b \Downarrow_{\text{bbe}}$ Mismatch".
  There remains to prove the premises of IH (2), that is: TrCont(Mismatch, u, `()`, $[\![tt]\!]$) and FvCont(B, u, `()`).

  The property TrCont(Mismatch, u, `()`, $[\![tt]\!]$) simplifies to "`()` $\Downarrow_{\text{ml}}$ $[\![tt]\!]$".
  By definition, $[\![tt]\!]$ = `Result ()`, and we can conclude.

  The property FvCont(B, u, `()`) is proved like in the case EVAL-TRM-WHILE-1.
- EVAL-TRM-BLOCK-1.
  The conclusion of the evaluation rule is "$L : \{\, t \,\} \Downarrow_{\text{trm}}$ Res v". Its premise is "$t \Downarrow_{\text{trm}}$ Abort (Exit L v)".
  By application of IH (1), we have $[\![t]\!]$ $\Downarrow_{\text{ml}}$ `Exn (Exit (L,` $[\![v]\!]$`))`.
  We have to show that "`try` $[\![t]\!]$ `with Exit (L, x) → x` $\Downarrow_{\text{ml}}$ `Result` $[\![v]\!]$".
  By applying ML-TRY-WITH-1 with $[\![t]\!]$ $\Downarrow_{\text{ml}}$ `Exn (Exit (L,` $[\![v]\!]$`))`, it suffices to show that Subst($\{\mathtt{x} \mapsto [\![v]\!]\}$, x) $\Downarrow_{\text{ml}}$ `Result` $[\![v]\!]$, that is, $[\![v]\!]$ $\Downarrow_{\text{ml}}$ `Result` $[\![v]\!]$.
  As $[\![v]\!]$ is a value, it is either of the form "n" or "`(fun (`$\mathtt{x}_1, ..., \mathtt{x}_n$`) →` u`)`" or "`C (`$\mathtt{w}_1$`, ...,` $\mathtt{w}_n$`)`".
  We conclude by applying the corresponding evaluation rule, that is: ML-INT or ML-FIX or ML-CONSTR.
- EVAL-TRM-BLOCK-2.
  The conclusion of the evaluation rule is "$L : \{\, t \,\} \Downarrow_{\text{trm}} o$". Its premises are "$t \Downarrow_{\text{trm}} o$" and $\forall v.\ o \neq$ Abort (Exit L v).
  By application of IH (1) on "$t \Downarrow_{\text{trm}} o$", we have $[\![t]\!]$ $\Downarrow_{\text{ml}}$ $[\![o]\!]$.
  We have to show that "`try` $[\![t]\!]$ `with Exit (L, x) → x` $\Downarrow_{\text{ml}}$ $[\![o]\!]$".
  By applying ML-TRY-WITH-2 with $[\![t]\!]$ $\Downarrow_{\text{ml}}$ $[\![o]\!]$, it suffices to show that $\forall$w. $[\![o]\!] \neq$ `Exn (Exit (L, w))`.
  To prove this, assume that there exists a w such that $[\![o]\!]$ = `Exn (Exit (L, w))`. By inspecting Figure 28, we see that only results of the form Abort (Exit L v) can translate into result of the form `Exn Exit (L, w)`. We deduce that there exists a v such that o = Abort (Exit L v), which contradicts the assumption.
- EVAL-TRM-EXIT-1.
  The conclusion of the evaluation rule is "**exit** L t $\Downarrow_{\text{trm}}$ Abort (Exit L v)". Its premise is $t \Downarrow_{\text{trm}}$ Res v, from which we have by application of IH (1) that $[\![t]\!]$ $\Downarrow_{\text{ml}}$ `Result` $[\![v]\!]$.
  We have to prove that `raise (Exit (L,` $[\![t]\!]$`))` $\Downarrow_{\text{ml}}$ `Exn (Exit (L,` $[\![v]\!]$`))`.

By applying ML-RAISE, it suffices to prove that
`Exit (L, ⟦t⟧)` $\Downarrow_{\text{ml}}$ `Result (Exit (L, ⟦v⟧))`.
We prove this by applying ML-CONSTR with $[\![t]\!] \Downarrow_{\text{ml}}$ `Result ⟦v⟧`.

- EVAL-TRM-EXIT-2.
  The conclusion of the evaluation rule is "**exit** $L\ t \Downarrow_{\text{trm}}$ Abort a". Its premise is $t \Downarrow_{\text{trm}}$ Abort a, from which we have by application of IH (1) that $[\![t]\!] \Downarrow_{\text{ml}} [\![\text{Abort } a]\!]$.
  We have to prove that `raise Exit (L, ⟦t⟧)` $\Downarrow_{\text{ml}} [\![\text{Abort } a]\!]$.
  We have by definition of the translation that $[\![\text{Abort } a]\!]$ is of the form `Exn w` for some value `w`.
  We can conclude with the ML-RAISE exception propagation rule.
- EVAL-TRM-NEXT.
  We have to prove that `raise (Next L)` $\Downarrow_{\text{ml}}$ `Exn Next L`.
  We prove this with ML-RAISE and ML-CONSTR, in the same way as the case EVAL-TRM-EXIT-1.

2. Evaluation rules for BBEs.

- EVAL-BBE-IS.
  The conclusion of the evaluation rule is t **is** $p \Downarrow_{\text{bbe}} r$. Its premises are "$t \Downarrow_{\text{trm}}$ Res v" and "$v \triangleright p \Downarrow_{\text{pat}} r$".
  By application of IH (1) on "$t \Downarrow_{\text{trm}}$ Res v", we have $[\![t]\!] \Downarrow_{\text{ml}}$ `Result ⟦v⟧`.
  The typing assumption is "$\vdash_{\text{bbe}} t$ **is** $p \rightsquigarrow B$" from which we get by inversion that "$\vdash_{\text{trm}} t : T$" and "$\vdash_{\text{pat}} p : T \rightsquigarrow B$".
  The additional hypotheses are $\text{TrCont}(r, \mathsf{u}, \mathsf{u}', \mathsf{q})$ and $\text{FvCont}(B, \mathsf{u}, \mathsf{u}')$.
  We have to prove that `let y = ⟦t⟧ in` $[\![\mathsf{y} \blacktriangleright p]\!]^{\mathsf{u}}_{\mathsf{u}'} \Downarrow_{\text{ml}} \mathsf{q}$.
  By applying ML-LET-1, it suffices to prove that $\text{Subst}(\{\mathsf{y} \mapsto [\![v]\!]\}, [\![\mathsf{y} \blacktriangleright p]\!]^{\mathsf{u}}_{\mathsf{u}'}) \Downarrow_{\text{ml}} \mathsf{q}$.
  Recall that y is globally fresh, in particular $\mathsf{y} \notin (\text{fv}(p) \cup \text{fv}(\mathsf{u}) \cup \text{fv}(\mathsf{u}'))$. A straightforward induction shows: $\text{Subst}(\{\mathsf{y} \mapsto \mathsf{w}\}, [\![\mathsf{y} \blacktriangleright p]\!]^{\mathsf{u}}_{\mathsf{u}'}) = [\![\mathsf{w} \blacktriangleright p]\!]^{\mathsf{u}}_{\mathsf{u}'}$.
  Thus we have to prove that $[\![[\![v]\!] \blacktriangleright p]\!]^{\mathsf{u}}_{\mathsf{u}'} \Downarrow_{\text{ml}} \mathsf{q}$.
  By applying IH (3) on $v \triangleright p \Downarrow_{\text{pat}} r$, with $\text{TrCont}(r, \mathsf{u}, \mathsf{u}', \mathsf{q})$ and $\text{FvCont}(B, \mathsf{u}, \mathsf{u}')$, there is left to prove the typing requirements, that is: $\vdash_{\text{val}} v : T$ and $\vdash_{\text{pat}} p : T \rightsquigarrow B$.
  We have previously stated $\vdash_{\text{pat}} p : T \rightsquigarrow B$ by inversion of the typing assumption.
  We get $\vdash_{\text{val}} v : T$ by applying Lemma 9 with "$t \Downarrow_{\text{trm}}$ Res v" and "$\vdash_{\text{trm}} t : T$".
- EVAL-BBE-IF-1.
  The conclusion of the evaluation rule is $\textbf{if}^{\text{bbe}}\ b_0\ \textbf{then}\ b_1\ \textbf{else}\ b_2 \Downarrow_{\text{bbe}} r$. Its premises are "$b_0 \Downarrow_{\text{bbe}}$ Mismatch" and "$b_2 \Downarrow_{\text{bbe}} r$".
  The typing assumption is $\vdash_{\text{bbe}} \textbf{if}^{\text{bbe}}\ b_0\ \textbf{then}\ b_1\ \textbf{else}\ b_2 \rightsquigarrow B$, from which we get by inversion that: $\vdash_{\text{bbe}} b_0 \rightsquigarrow B_0$ and $B_0 \vdash_{\text{bbe}} b_1 \rightsquigarrow B_1$ and $\vdash_{\text{bbe}} b_2 \rightsquigarrow B_2$, where $B = (B_0 \uplus B_1) \cap B_2$.
  By using Lemma 8 on the typing judgments, we have "$\text{fv}(b_0) = \varnothing$" and "$\text{fv}(b_1) \subseteq \text{dom}(B_0)$" and "$\text{fv}(b_2) = \varnothing$".
  The additional hypotheses are $\text{TrCont}(r, \mathsf{u}, \mathsf{u}', \mathsf{q})$, and $\text{FvCont}(B, \mathsf{u}, \mathsf{u}')$.

We have to show that
$\texttt{let k}_1\ \bar{\texttt{x}} = \texttt{u in let k}_2() = \texttt{u}' \texttt{ in } [\![b_0]\!]^{[\![b_1]\!]^{\texttt{k}_1\ \bar{\texttt{x}}}_{\texttt{k}_2()}}_{[\![b_2]\!]^{\texttt{k}_1\ \bar{\texttt{x}}}_{\texttt{k}_2()}} \Downarrow_{\text{ml}} \texttt{q}$, where $\bar{\texttt{x}} = (\texttt{x}_1, ..., \texttt{x}_n)$, and $\{\texttt{x}_1; ...; \texttt{x}_n\} = (\text{bv}(b_0) \cup \text{bv}(b_1)) \cap \text{bv}(b_2) \cap \text{fv}(\texttt{u})$.

By applying ML-LET-1 twice, there remains to show that $[\![b_0]\!]^{[\![b_1]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}()}}_{[\![b_2]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}()}} \Downarrow_{\text{ml}} \texttt{q}$, where $\texttt{w}_\texttt{u} = (\texttt{fun}\ \bar{\texttt{x}}\ \rightarrow\ \texttt{u})$ and $\texttt{w}_{\texttt{u}'} = (\texttt{fun}\ ()\ \rightarrow\ \texttt{u}')$

We prove this by applying IH (2) on $b_0 \Downarrow_{\text{bbe}}$ Mismatch, with continuations $[\![b_1]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}()}$ and $[\![b_2]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}()}$.

There remains to prove the premises of IH (2), that is: TrCont(Mismatch, $[\![b_1]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}()}$, $[\![b_2]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}()}$, q) and FvCont($B$, $[\![b_1]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}()}$, $[\![b_2]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}()}$)

The property TrCont(Mismatch, $[\![b_1]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}()}$, $[\![b_2]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}()}$, q) simplifies to $[\![b_2]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}()} \Downarrow_{\text{ml}} \texttt{q}$.

This is proved by applying IH (2) on "$b_2 \Downarrow_{\text{bbe}} r$" with continuations $\texttt{w}_\texttt{u}\ \bar{\texttt{x}}$ and $\texttt{w}_{\texttt{u}'}()$.

There remains to prove the premises of IH (2), that is: TrCont(r, $\texttt{w}_\texttt{u}\ \bar{\texttt{x}}$, $\texttt{w}_{\texttt{u}'}()$, q) and FvCont($B_2$, $\texttt{w}_\texttt{u}\ \bar{\texttt{x}}$, $\texttt{w}_{\texttt{u}'}()$).

- We prove the property TrCont(r, $\texttt{w}_\texttt{u}\ \bar{\texttt{x}}$, $\texttt{w}_{\texttt{u}'}()$, q) by case analysis.

  If r = Mismatch, then the property simplifies to $\texttt{w}_{\texttt{u}'}() \Downarrow_{\text{ml}} \texttt{q}$.

  By applying ML-BETA, there is left to prove that $\texttt{u}' \Downarrow_{\text{ml}} \texttt{q}$, which we check with FvCont(B, u, $\texttt{u}'$).

  If r = Match M_2, then the property simplifies to Subst($[\![M_2]\!]$, $\texttt{w}_\texttt{u}\ \bar{\texttt{x}}$) $\Downarrow_{\text{ml}}$ q, which itself simplifies to "Subst($[\![M_2]\!]$, $\texttt{w}_\texttt{u}$) Subst($[\![M_2]\!]$, $\bar{\texttt{x}}$) $\Downarrow_{\text{ml}}$ q" .

  By definition, we have $\text{fv}(\bar{\texttt{x}}) = (\text{bv}(b_0) \cup \text{bv}(b_1)) \cap \text{bv}(b_2) \cap \text{fv}(\texttt{u})$.

  We have by FvCont(B, u, $\texttt{u}'$) that $\text{fv}(\texttt{u}) \subseteq \text{dom}(B) = \text{dom}((B_0 \uplus B_1) \cap B_2)$. This means in particular that $\text{fv}(\texttt{u}) \subseteq \text{dom}(B_2)$. By Lemma 5 with "$\vdash_{\text{bbe}} b_2 \rightsquigarrow B_2$" and "$b_2 \Downarrow_{\text{bbe}}$ Match M_2", we have $\text{dom}(B_2) \subseteq \text{dom}(M_2)$. We conclude that $\text{fv}(\texttt{u}) \subseteq \text{dom}(M_2)$, from which we have $\text{fv}(\bar{\texttt{x}}) \subseteq \text{dom}(M_2)$.

  We also have by Lemma 7 that $\text{dom}(B_0) \subseteq \text{bv}(b_0)$, $\text{dom}(B_1) \subseteq \text{bv}(b_1)$ and $\text{dom}(B_2) \subseteq \text{bv}(b_2)$. Hence $\text{dom}(B) = \text{dom}((B_0 \uplus B_1) \cap B_2) \subseteq (\text{bv}(b_0) \cup \text{bv}(b_1)) \cap \text{bv}(b_2)$. Together with $\text{fv}(\texttt{u}) \subseteq \text{dom}(B)$, we conclude that $\text{fv}(\texttt{u}) \subseteq (\text{bv}(b_0) \cup \text{bv}(b_1)) \cap \text{bv}(b_2) \cap \text{fv}(\texttt{u}) = \text{fv}(\bar{\texttt{x}})$. Hence $\text{fv}(\texttt{w}_\texttt{u}) = \varnothing$.

  The property to prove thus simplifies to $\texttt{w}_\texttt{u}$ Subst($[\![M_2]\!]$, $\bar{\texttt{x}}$) $\Downarrow_{\text{ml}}$ q.

  From $\text{fv}(\bar{\texttt{x}}) \subseteq \text{dom}(M_2)$, we have that, for all $\texttt{x}_\texttt{i}$ in $\text{fv}(\bar{\texttt{x}})$, there exists a value $\texttt{w}_\texttt{i}$ such that $[\![M_2]\!][\texttt{x}_\texttt{i}] = \texttt{w}_\texttt{i}$.

  By applying ML-BETA, there remains to prove that Subst($[\![M_2]\!]$, u) $\Downarrow_{\text{ml}}$ q, which is exactly the property TrCont(Match M_2, u, $\texttt{u}'$, q).
- The property FvCont(B_2, $\texttt{w}_\texttt{u}\ \bar{\texttt{x}}$, $\texttt{w}_{\texttt{u}'}()$) simplifies to $\text{fv}(\texttt{w}_{\texttt{u}'}()) = \varnothing \wedge \text{fv}(\texttt{w}_\texttt{u}\ \bar{\texttt{x}}) \subseteq \text{dom}(B_2)$.

  We have by definition that $\text{fv}(\texttt{w}_{\texttt{u}'}()) = \text{fv}(\texttt{fun}\ ()\ \rightarrow\ \texttt{u}') = \text{fv}(\texttt{u}')$. We conclude with FvCont(B, u, $\texttt{u}'$), from which we have $\text{fv}(\texttt{u}') = \varnothing$.

We have by definition that $\mathrm{fv}(\mathsf{w_u}\ \bar{\mathsf{x}}) = \mathrm{fv}(\mathsf{fun}\ \bar{\mathsf{x}}\ \to\ \mathsf{u}) \cup \mathrm{fv}(\bar{\mathsf{x}}) = \mathrm{fv}(\mathsf{u}) \cup \mathrm{fv}(\bar{\mathsf{x}})$, where $\mathrm{fv}(\bar{\mathsf{x}}) = \{\mathsf{x}_1; ...; \mathsf{x}_n\}$. We also have by definition that $\mathrm{fv}(\bar{\mathsf{x}}) \subseteq \mathrm{fv}(\mathsf{u})$, which means that $\mathrm{fv}(\mathsf{w_u}\ \bar{\mathsf{x}}) \subseteq \mathrm{fv}(\mathsf{u})$.
Finally, we get from $\mathrm{FvCont}(B, \mathsf{u}, \mathsf{u}')$ that $\mathrm{fv}(\mathsf{u}) \subseteq \mathrm{dom}(B) = \mathrm{dom}((B_0 \uplus B_1) \cap B_2) \subseteq \mathrm{dom}(B_2)$, from which we conclude.

The property $\mathrm{FvCont}(B, [\![b_1]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}, [\![b_2]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()})$ simplifies to
$\mathrm{fv}([\![b_2]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}) = \varnothing \wedge \mathrm{fv}([\![b_1]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}) \subseteq \mathrm{dom}(B)$.

- By Lemma 6, $\mathrm{fv}([\![b_2]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}) \subseteq \mathrm{fv}(b_2) \cup (\mathrm{fv}(\mathsf{w_u}\ \bar{\mathsf{x}}) \setminus \mathrm{bv}(b_2)) \cup \mathrm{fv}(\mathsf{w_{u'}}\,())$.
  We stated earlier that $\mathrm{fv}(b_2) = \varnothing$, and proved that $\mathrm{fv}(\mathsf{w_{u'}}\,()) = \varnothing$.
  All there is left to prove is that $\mathrm{fv}(\mathsf{w_u}\ \bar{\mathsf{x}}) \setminus \mathrm{bv}(b_2) = \varnothing$, which simplifies to $\mathrm{fv}(\mathsf{w_u}\ \bar{\mathsf{x}}) \subseteq \mathrm{bv}(b_2)$.
  By definition, we have to prove that $\mathrm{fv}(\mathsf{u}) \cup \mathrm{fv}(\bar{\mathsf{x}}) \subseteq \mathrm{bv}(b_2)$, which simplifies to proving both $\mathrm{fv}(\mathsf{u}) \subseteq \mathrm{bv}(b_2)$ and $\mathrm{fv}(\bar{\mathsf{x}}) \subseteq \mathrm{bv}(b_2)$.
  We have from $\mathrm{FvCont}(B, \mathsf{u}, \mathsf{u}')$ that $\mathrm{fv}(\mathsf{u}) \subseteq \mathrm{dom}(B)$, and from Lemma 7 that $\mathrm{dom}(B) \subseteq \mathrm{bv}(b_2)$, which is enough to conclude the former.
  By definition, $\mathrm{fv}(\bar{\mathsf{x}}) = (\mathrm{bv}(b_0) \cup \mathrm{bv}(b_1)) \cap \mathrm{bv}(b_2) \cap \mathrm{fv}(\mathsf{u}) \subseteq \mathrm{bv}(b_2)$, which is exactly the latter.
- By Lemma 6, $\mathrm{fv}([\![b_1]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}) \subseteq \mathrm{fv}(b_1) \cup (\mathrm{fv}(\mathsf{w_u}\ \bar{\mathsf{x}}) \setminus \mathrm{bv}(b_1)) \cup \mathrm{fv}(\mathsf{w_{u'}}\,())$.
  We stated earlier that $\mathrm{fv}(b_1) \subseteq \mathrm{dom}(B_0) \subseteq \mathrm{dom}(B)$, and proved that $\mathrm{fv}(\mathsf{w_{u'}}\,()) = \varnothing$.
  All there is left to prove is that $\mathrm{fv}(\mathsf{w_u}\ \bar{\mathsf{x}}) \setminus \mathrm{bv}(b_1) \subseteq \mathrm{dom}(B)$, which simplifies to $\mathrm{fv}(\mathsf{w_u}\ \bar{\mathsf{x}}) \subseteq \mathrm{bv}(b_1) \cup \mathrm{dom}(B)$.
  By definition, we have to prove that $\mathrm{fv}(\mathsf{u}) \cup \mathrm{fv}(\bar{\mathsf{x}}) \subseteq \mathrm{bv}(b_1) \cup \mathrm{dom}(B)$, which simplifies to proving both $\mathrm{fv}(\mathsf{u}) \subseteq \mathrm{bv}(b_1) \cup \mathrm{dom}(B)$ and $\mathrm{fv}(\bar{\mathsf{x}}) \subseteq \mathrm{bv}(b_1) \cup \mathrm{dom}(B)$.
  We have from $\mathrm{FvCont}(B, \mathsf{u}, \mathsf{u}')$ that $\mathrm{fv}(\mathsf{u}) \subseteq \mathrm{dom}(B)$, which is exactly the former.
  By definition, $\mathrm{fv}(\bar{\mathsf{x}}) = (\mathrm{bv}(b_0) \cup \mathrm{bv}(b_1)) \cap \mathrm{bv}(b_2) \cap \mathrm{fv}(\mathsf{u}) \subseteq \mathrm{fv}(\mathsf{u}) \subseteq \mathrm{dom}(B)$, which is exactly the latter.

- EVAL-BBE-IF-2.
  The conclusion of the evaluation rule is $\mathbf{if}^{\mathrm{bbe}}\ b_0\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 \Downarrow_{\mathrm{bbe}} \mathrm{Mismatch}$. Its premises are "$b_0 \Downarrow_{\mathrm{bbe}} \mathrm{Match}\ M_0$" and "$\mathrm{Subst}(M_0, b_1) \Downarrow_{\mathrm{bbe}} \mathrm{Mismatch}$".
  The typing assumption is $\vdash_{\mathrm{bbe}} \mathbf{if}^{\mathrm{bbe}}\ b_0\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 \rightsquigarrow B$, from which we get by inversion that: $\vdash_{\mathrm{bbe}} b_0 \rightsquigarrow B_0$ and $B_0 \vdash_{\mathrm{bbe}} b_1 \rightsquigarrow B_1$ and $\vdash_{\mathrm{bbe}} b_2 \rightsquigarrow B_2$, where $B = (B_0 \uplus B_1) \cap B_2$.
  The additional hypotheses are $\mathrm{TrCont}(\mathrm{Mismatch}, \mathsf{u}, \mathsf{u}', \mathsf{q})$, and $\mathrm{FvCont}(B, \mathsf{u}, \mathsf{u}')$.
  We have to show that
  $\mathtt{let\ k_1\ \bar{x}\ =\ u\ in\ let\ k_2()\ =\ u'\ in}\ [\![b_0]\!]^{[\![b_1]\!]^{\mathsf{k_1}\ \bar{\mathsf{x}}}_{\mathsf{k_2}\,()}}_{[\![b_2]\!]^{\mathsf{k_1}\ \bar{\mathsf{x}}}_{\mathsf{k_2}\,()}} \Downarrow_{\mathrm{ml}} \mathsf{q}$, where $\bar{\mathsf{x}} = (\mathsf{x}_1, ..., \mathsf{x}_n)$, and $\{\mathsf{x}_1; ...; \mathsf{x}_n\} = (\mathrm{bv}(b_0) \cup \mathrm{bv}(b_1)) \cap \mathrm{bv}(b_2) \cap \mathrm{fv}(\mathsf{u})$.

By applying ML-LET-1 twice, there remains to show that $[\![b_0]\!]^{[\![b_1]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}}_{[\![b_2]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}} \Downarrow_{\mathrm{ml}} \mathsf{q}$, where $\mathsf{w_u} = (\mathtt{fun}\ \bar{\mathsf{x}} \to \mathsf{u})$ and $\mathsf{w_{u'}} = (\mathtt{fun}\ () \to \mathsf{u'})$

We prove this by applying IH (2) on $b_0 \Downarrow_{\mathrm{bbe}}$ Match M_0, with continuations $[\![b_1]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}$ and $[\![b_2]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}$.

There remains to prove the premises of IH (2), that is: TrCont(Match M_0, $[\![b_1]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}$, $[\![b_2]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}$, q) and FvCont($B$, $[\![b_1]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}$, $[\![b_2]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}$)

The property TrCont(Match M_0, $[\![b_1]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}$, $[\![b_2]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}$, q) simplifies to Subst($[\![M_0]\!]$, $[\![b_1]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}$) $\Downarrow_{\mathrm{ml}}$ q.

By applying Lemma 10, it suffices to prove $[\![\mathrm{Subst}(M_0, b_1)]\!]^{\mathrm{Subst}([\![M_0]\!], \mathsf{w_u}\ \bar{\mathsf{x}})}_{\mathsf{w_{u'}}\,()} \Downarrow_{\mathrm{ml}} \mathsf{q}$ and $\mathrm{fv}(\mathsf{w_{u'}}\,()) = \varnothing$ and $\mathrm{fv}(\mathsf{w_u}\ \bar{\mathsf{x}}) \subseteq (\mathrm{dom}(M_0) \cup \mathrm{bv}(b_1))$.

- We prove $[\![\mathrm{Subst}(M_0, b_1)]\!]^{\mathrm{Subst}([\![M_0]\!], \mathsf{w_u}\ \bar{\mathsf{x}})}_{\mathsf{w_{u'}}\,()} \Downarrow_{\mathrm{ml}} \mathsf{q}$ by applying IH (2) on Subst(M_0, b_1) $\Downarrow_{\mathrm{bbe}}$ Mismatch.
  There remains to prove the premises of IH (2), that is: TrCont(Mismatch, Subst($[\![M_0]\!]$, $\mathsf{w_u}\ \bar{\mathsf{x}}$), $\mathsf{w_{u'}}\,()$, q) and FvCont(B_1, Subst($[\![M_0]\!]$, $\mathsf{w_u}\ \bar{\mathsf{x}}$), $\mathsf{w_{u'}}\,()$).
  The property TrCont(Mismatch, Subst($[\![M_0]\!]$, $\mathsf{w_u}\ \bar{\mathsf{x}}$), $\mathsf{w_{u'}}\,()$, q) simplifies to $\mathsf{w_{u'}}\,() \Downarrow_{\mathrm{ml}} \mathsf{q}$.
  We prove this by applying ML-BETA with $\mathsf{u'} \Downarrow_{\mathrm{ml}} \mathsf{q}$.
  The property FvCont(B_1, Subst($[\![M_0]\!]$, $\mathsf{w_u}\ \bar{\mathsf{x}}$), $\mathsf{w_{u'}}\,()$) simplifies to $\mathrm{fv}(\mathsf{w_{u'}}\,()) = \varnothing \wedge \mathrm{fv}(\mathrm{Subst}([\![M_0]\!], \mathsf{w_u}\ \bar{\mathsf{x}})) \subseteq \mathrm{dom}(B_1)$.
  We proved the former in the case EVAL-BBE-IF-1.
  To prove the latter, it suffices to prove that $\mathrm{fv}(\mathsf{w_u}\ \bar{\mathsf{x}}) \subseteq (\mathrm{dom}(B_1) \cup \mathrm{dom}(M_0))$.
  By Lemma 5, we have $\mathrm{dom}(B_0) \subseteq \mathrm{dom}(M_0)$, which means that it suffices to prove that $\mathrm{fv}(\mathsf{w_u}\ \bar{\mathsf{x}}) \subseteq (\mathrm{dom}(B_1) \cup \mathrm{dom}(B_0)) = \mathrm{dom}(B_0 \uplus B_1)$.
  As stated in the case EVAL-BBE-IF-1, $\mathrm{fv}(\mathsf{w_u}\ \bar{\mathsf{x}}) \subseteq \mathrm{fv}(\mathsf{u}) \subseteq \mathrm{dom}(B)$, where $B = (B_0 \uplus B_1) \cap B_2$.
  We conclude the latter with $(B_0 \uplus B_1) \cap B_2 \subseteq B_0 \uplus B_1$.
- The property $\mathrm{fv}(\mathsf{w_{u'}}\,()) = \varnothing$ is proved in the case EVAL-BBE-IF-1.
- To prove the property $\mathrm{fv}(\mathsf{w_u}\ \bar{\mathsf{x}}) \subseteq (\mathrm{dom}(M_0) \cup \mathrm{bv}(b_1))$, it suffices to prove $\mathrm{fv}(\mathsf{w_u}\ \bar{\mathsf{x}}) \subseteq (\mathrm{dom}(M_0) \uplus \mathrm{dom}(B_1))$ since $\mathrm{dom}(B_1) \subseteq \mathrm{bv}(b_1)$ by Lemma 7. This exact property was proved in the previous case.

The property FvCont(B, $[\![b_1]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}$, $[\![b_2]\!]^{\mathsf{w_u}\ \bar{\mathsf{x}}}_{\mathsf{w_{u'}}\,()}$) is proved like in the case EVAL-BBE-IF-1.

- EVAL-BBE-IF-3.

The conclusion of the evaluation rule is "$\mathbf{if}^{\mathrm{bbe}}\ b_0\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 \Downarrow_{\mathrm{bbe}}$ Match $(M_0 \uplus M_1)$". Its premises are "$b_0 \Downarrow_{\mathrm{bbe}}$ Match M_0" and "Subst(M_0, b_1) $\Downarrow_{\mathrm{bbe}}$ Match M_1" and "$M_0 \# M_1$".

The typing assumption is $\vdash_{\mathrm{bbe}} \mathbf{if}^{\mathrm{bbe}}\ b_0\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2 \rightsquigarrow B$, from which we get by inversion that: $\vdash_{\mathrm{bbe}} b_0 \rightsquigarrow B_0$ and $B_0 \vdash_{\mathrm{bbe}} b_1 \rightsquigarrow B_1$ and $\vdash_{\mathrm{bbe}} b_2 \rightsquigarrow B_2$, where $B = (B_0 \uplus B_1) \cap B_2$.

The additional hypotheses are TrCont(Match $(M_0 \uplus M_1)$, u, u′, q), and FvCont(B, u, u′).
We have to show that
$\texttt{let k}_1\ \bar{\texttt{x}} = \texttt{u in let k}_2\texttt{() = u}'\ \texttt{in}\ [\![b_0]\!]^{[\![b_1]\!]^{\texttt{k}_1\ \bar{\texttt{x}}}_{\texttt{k}_2\texttt{()}}}_{[\![b_2]\!]^{\texttt{k}_1\ \bar{\texttt{x}}}_{\texttt{k}_2\texttt{()}}} \Downarrow_{\text{ml}} \texttt{q}$, where $\bar{\texttt{x}} = (\texttt{x}_1, ..., \texttt{x}_n)$, and $\{\texttt{x}_1; ...; \texttt{x}_n\} = (\text{bv}(b_0) \cup \text{bv}(b_1)) \cap \text{bv}(b_2) \cap \text{fv}(\texttt{u})$.
By applying ML-LET-1 twice, there remains to show that $[\![b_0]\!]^{[\![b_1]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}\texttt{()}}}_{[\![b_2]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}\texttt{()}}} \Downarrow_{\text{ml}} \texttt{q}$, where $\texttt{w}_\texttt{u} = \texttt{(fun}\ \bar{\texttt{x}}\ \texttt{->}\ \texttt{u)}$ and $\texttt{w}_{\texttt{u}'} = \texttt{(fun ()} \rightarrow \texttt{u}')$.
We prove this by applying IH (2) on "$b_0 \Downarrow_{\text{bbe}} \text{Match } M_0$", with continuations $[\![b_1]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}\texttt{()}}$ and $[\![b_2]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}\texttt{()}}$.
There remains to prove the premises of IH (2), that is: TrCont(Match M_0, $[\![b_1]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}\texttt{()}}$, $[\![b_2]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}\texttt{()}}$, q) and FvCont($B$, $[\![b_1]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}\texttt{()}}$, $[\![b_2]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}\texttt{()}}$)

The property TrCont(Match M_0, $[\![b_1]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}\texttt{()}}$, $[\![b_2]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}\texttt{()}}$, q) simplifies to Subst($[\![M_0]\!]$, $[\![b_1]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}\texttt{()}}$) $\Downarrow_{\text{ml}}$ q.
By applying Lemma 10, it suffices to prove $[\![\text{Subst}(M_0, b_1)]\!]^{\text{Subst}([\![M_0]\!], \texttt{w}_\texttt{u}\ \bar{\texttt{x}})}_{\texttt{w}_{\texttt{u}'}\texttt{()}} \Downarrow_{\text{ml}} \texttt{q}$ and $\text{fv}(\texttt{w}_{\texttt{u}'}\texttt{()}) = \varnothing$ and $\text{fv}(\texttt{w}_\texttt{u}\ \bar{\texttt{x}}) \subseteq (\text{dom}(M_0) \cup \text{bv}(b_1))$.

- We prove $[\![\text{Subst}(M_0, b_1)]\!]^{\text{Subst}([\![M_0]\!], \texttt{w}_\texttt{u}\ \bar{\texttt{x}})}_{\texttt{w}_{\texttt{u}'}\texttt{()}} \Downarrow_{\text{ml}} \texttt{q}$ by applying IH (2) on Subst(M_0, b_1) $\Downarrow_{\text{bbe}}$ Match M_1.
  There remains to prove the premises of IH (2), that is: TrCont(Match M_1, Subst($[\![M_0]\!]$, $\texttt{w}_\texttt{u}\ \bar{\texttt{x}}$), $\texttt{w}_{\texttt{u}'}\texttt{()}$, q) and FvCont($B_1$, Subst($[\![M_0]\!]$, $\texttt{w}_\texttt{u}\ \bar{\texttt{x}}$), $\texttt{w}_{\texttt{u}'}\texttt{()}$).
  The property TrCont(Match M_1, Subst($[\![M_0]\!]$, $\texttt{w}_\texttt{u}\ \bar{\texttt{x}}$), $\texttt{w}_{\texttt{u}'}\texttt{()}$, q) simplifies to Subst($[\![M_1]\!]$, Subst($[\![M_0]\!]$, $\texttt{w}_\texttt{u}\ \bar{\texttt{x}}$)) $\Downarrow_{\text{ml}}$ q, which itself simplifies to Subst($[\![M_0 \uplus M_1]\!]$, $\texttt{w}_\texttt{u}\ \bar{\texttt{x}}$) $\Downarrow_{\text{ml}}$ q, as $M_0 \# M_1$. Finally, by spreading the substitution, the property to prove simplifies to Subst($[\![M_0 \uplus M_1]\!]$, $\texttt{w}_\texttt{u}$) Subst($[\![M_0 \uplus M_1]\!]$, $\bar{\texttt{x}}$) $\Downarrow_{\text{ml}}$ q.
  By definition, $\text{fv}(\bar{\texttt{x}}) = ((\text{bv}(b_0) \cup \text{bv}(b_1)) \cap \text{bv}(b_2)) \cap \text{fv}(\texttt{u})$, in particular we have $\text{fv}(\bar{\texttt{x}}) \subseteq \text{fv}(\texttt{u})$.
  By FvCont(B, u, u′), $\text{fv}(\texttt{u}) \subseteq \text{dom}(B) = \text{dom}((B_0 \uplus B_1) \cap B_2) \subseteq \text{dom}(B_0 \uplus B_1) \subseteq \text{dom}(M_0 \uplus M_1)$, by Lemma 5.
  We have from this that $\text{fv}(\texttt{w}_\texttt{u}) = \varnothing$, and $\text{fv}(\bar{\texttt{x}}) \subseteq \text{dom}(M_0 \uplus M_1)$. With a similar reasoning as in the case EVAL-BBE-IF-1, there is left to prove that Subst($[\![M_0 \uplus M_1]\!]$, u) $\Downarrow_{\text{ml}}$ q, which is exactly the property TrCont(Match $M_0 \uplus M_1$, u, u′, q).
  The property FvCont(B_1, Subst($[\![M_0]\!]$, $\texttt{w}_\texttt{u}\ \bar{\texttt{x}}$), $\texttt{w}_{\texttt{u}'}\texttt{()}$) is proved like in the case EVAL-BBE-IF-2.
- The property $\text{fv}(\texttt{w}_{\texttt{u}'}\texttt{()}) = \varnothing$ is proved in the case EVAL-BBE-IF-1.
- To prove the property $\text{fv}(\texttt{w}_\texttt{u}\ \bar{\texttt{x}}) \subseteq (\text{dom}(M_0) \cup \text{bv}(b_1))$, it suffices to prove $\text{fv}(\texttt{w}_\texttt{u}\ \bar{\texttt{x}}) \subseteq (\text{dom}(M_0) \uplus \text{dom}(B_1))$ since $\text{dom}(B_1) \subseteq \text{bv}(b_1)$ by Lemma 7. This exact property was proved in the previous case.

The property FvCont(B, $[\![b_1]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}\texttt{()}}$, $[\![b_2]\!]^{\texttt{w}_\texttt{u}\ \bar{\texttt{x}}}_{\texttt{w}_{\texttt{u}'}\texttt{()}}$) is proved like in the case EVAL-BBE-IF-1.

3. Evaluation rules for patterns.

- EVAL-PAT-WHEN-1.
  The conclusion of the evalation rule is "$v \triangleright p$ **when** $b \Downarrow_{\text{pat}}$ Mismatch". Its premise is "$v \triangleright p \Downarrow_{\text{pat}}$ Mismatch".
  The typing assumptions are "$\vdash_{\text{val}} v : T$" and "$\vdash_{\text{pat}} p$ **when** $b : T \rightsquigarrow B$", from which we get by inversion: "$\vdash_{\text{pat}} p : T \rightsquigarrow B_1$" and "$B_1 \vdash_{\text{bbe}} b \rightsquigarrow B_2$", where $B = B_1 \uplus B_2$.
  By applying Lemma 8 on each typing assumption, we have: "$\text{fv}(p) = \varnothing$" and "$\text{fv}(b) \subseteq \text{dom}(B_1)$".
  The additional hypotheses are "TrCont(Mismatch, u, u', q)", which simplifies to $\mathsf{u}' \Downarrow_{\text{ml}} \mathsf{q}$, and "FvCont($B$, u, u')", which is equivalent to "$\text{fv}(\mathsf{u}') = \varnothing \wedge \text{fv}(\mathsf{u}) \subseteq \text{dom}(B)$".
  We have to show that $\texttt{let k() = u' in } [\![[\![v]\!] \blacktriangleright p]\!]^{[\![b]\!]^{\mathsf{u}}_{\mathsf{k}()}}_{\mathsf{k}()} \Downarrow_{\text{ml}} \mathsf{q}$.
  By applying ML-LET-1, there remains to prove that $[\![[\![v]\!] \blacktriangleright p]\!]^{[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}}_{\mathsf{w_{u'}}()} \Downarrow_{\text{ml}} \mathsf{q}$, where $\mathsf{w_{u'}} = (\texttt{fun () } \rightarrow \texttt{ u'})$.
  We prove this by applying IH (3) on "$v \triangleright p \Downarrow_{\text{pat}}$ Mismatch", with $\mathsf{u} = [\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$ and $\mathsf{u}' = \mathsf{w_{u'}}()$.
  There remains to prove the premises of IH (3), that is: TrCont(Mismatch, $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$, $\mathsf{w_{u'}}()$, q) and FvCont(B, $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$, $\mathsf{w_{u'}}()$).
  The property TrCont(Mismatch, $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$, $\mathsf{w_{u'}}()$, q) simplifies to $\mathsf{w_{u'}}() \Downarrow_{\text{ml}} \mathsf{q}$.
  We conclude by applying ML-BETA with $\mathsf{u}' \Downarrow_{\text{ml}} \mathsf{q}$.
  The property FvCont(B, $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$, $\mathsf{w_{u'}}()$) simplifies to "$(\text{fv}(\mathsf{w_{u'}}()) = \varnothing) \wedge (\text{fv}([\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}) \subseteq \text{dom}(B))$".
  The former comes from $\text{fv}(\mathsf{u}') = \varnothing$.
  To prove the latter, first see that we have by Lemma 6 that $\text{fv}([\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}) \subseteq \text{fv}(b) \cup (\text{fv}(\mathsf{u}) \setminus \text{bv}(b)) \cup \text{fv}(\mathsf{w_{u'}}())$.
  We stated earlier that $\text{fv}(b) \subseteq \text{dom}(B_1) \subseteq \text{dom}(B)$ by typing, that $\text{fv}(\mathsf{w_{u'}}()) = \varnothing$, and that $\text{fv}(\mathsf{u}) \subseteq \text{dom}(B)$ by FvCont(B, u, u'), which is enough to conclude.
- EVAL-PAT-WHEN-2.
  The conclusion of the evaluation rule is "$v \triangleright p$ **when** $b \Downarrow_{\text{pat}}$ Mismatch". Its premises are "$v \triangleright p \Downarrow_{\text{pat}}$ Match M_1" and "Subst(M_1, b) $\Downarrow_{\text{bbe}}$ Mismatch".
  The typing assumptions are "$\vdash_{\text{val}} v : T$" and "$\vdash_{\text{pat}} p$ **when** $b : T \rightsquigarrow B$", from which we get by inversion that $\vdash_{\text{pat}} p : T \rightsquigarrow B_1$ and $B_1 \vdash_{\text{bbe}} b \rightsquigarrow B_2$ where $B = B_1 \uplus B_2$, $\mathsf{u}' \Downarrow_{\text{ml}} \mathsf{q}$, $\text{fv}(\mathsf{u}') = \varnothing$ and $\text{fv}(\mathsf{u}) \subseteq \text{dom}(B)$.
  The additional hypotheses are "TrCont(Mismatch, u, u', q)", which simplifies to $\mathsf{u}' \Downarrow_{\text{ml}} \mathsf{q}$ and "FvCont(B, u, u')".
  We have to show that $\texttt{let k() = u' in } [\![[\![v]\!] \blacktriangleright p]\!]^{[\![b]\!]^{\mathsf{u}}_{\mathsf{k}()}}_{\mathsf{k}()} \Downarrow_{\text{ml}} \mathsf{q}$.
  By applying ML-LET-1, there remains to prove that $[\![[\![v]\!] \blacktriangleright p]\!]^{[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}}_{\mathsf{w_{u'}}()} \Downarrow_{\text{ml}} \mathsf{q}$, where $\mathsf{w_{u'}} = \texttt{fun () } \rightarrow \texttt{ u'}$.

We prove this by applying IH (3) on "$v \triangleright p \Downarrow_{\text{pat}} \text{Match } M_1$" with continuations $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$ and $\mathsf{w_{u'}}()$.
There remains to prove the premises of IH (3), that is: TrCont(Match M_1, $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$, $\mathsf{w_{u'}}()$, q) and FvCont(B, $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$, $\mathsf{w_{u'}}()$).

The property TrCont(Match M_1, $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$, $\mathsf{w_{u'}}()$, q) simplifies to Subst($[\![M_1]\!]$, $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$) $\Downarrow_{\text{ml}}$ q.
This is proved by applying Lemma 10. There remains to prove its premises, that is: $[\![\text{Subst}(M_1, b)]\!]^{\text{Subst}([\![M_1]\!],\mathsf{u})}_{\mathsf{w_{u'}}()} \Downarrow_{\text{ml}} \mathsf{q}$ and $\text{fv}(\mathsf{w_{u'}}()) = \varnothing$ and $\text{fv}(\mathsf{u}) \subseteq (\text{dom}(M_1) \cup \text{bv}(b))$.

The property $[\![\text{Subst}(M_1,\ b)]\!]^{\text{Subst}([\![M_1]\!],\mathsf{u})}_{\mathsf{w_{u'}}()} \Downarrow_{\text{ml}} \mathsf{q}$ is proved by applying IH (2) on "Subst(M_1, b) $\Downarrow_{\text{bbe}}$ Mismatch" with continuations Subst($[\![M_1]\!]$, u) and $\mathsf{w_{u'}}()$.
There remains to prove the premises of IH (2), that is: TrCont(Mismatch, Subst($[\![M_1]\!]$, u), $\mathsf{w_{u'}}()$, q) and FvCont(B_2, Subst($[\![M_1]\!]$, u), $\mathsf{w_{u'}}()$).

- The property TrCont(Mismatch, Subst($[\![M_1]\!]$, u), $\mathsf{w_{u'}}()$, q) simplifies to $\mathsf{w_{u'}}() \Downarrow_{\text{ml}} \mathsf{q}$, which we prove by application of ML-BETA on $\mathsf{u'} \Downarrow_{\text{ml}} \mathsf{q}$.
- The property FvCont(B_2, Subst($[\![M_1]\!]$, u), $\mathsf{w_{u'}}()$) simplifies to : $(\text{fv}(\mathsf{w_{u'}}()) = \varnothing) \wedge (\text{fv}(\text{Subst}([\![M_1]\!], \mathsf{u})) \subseteq \text{dom}(B_2))$.
  By definition $\text{fv}(\mathsf{u'}()) = \text{fv}(\mathsf{u'}) = \varnothing$ by FvCont(B, u, $\mathsf{u'}$).
  The property $\text{fv}(\text{Subst}([\![M_1]\!], \mathsf{u})) \subseteq \text{dom}(B_2)$ simplifies to $\text{fv}(\mathsf{u}) \subseteq \text{dom}([\![M_1]\!]) \cup \text{dom}(B_2)$. By application of Lemma 5 with $v \triangleright p \Downarrow_{\text{pat}} \text{Match } M_1$ and $\vdash_{\text{pat}} p : T \rightsquigarrow B_1$ and $\vdash_{\text{val}} v : T$, we have $\text{dom}(B_1) \subseteq \text{dom}(M_1)$. With that, there remains to prove that $\text{fv}(\mathsf{u}) \subseteq \text{dom}(B_1 \uplus B_2)$, which is checked by FvCont(B, u, $\mathsf{u'}$).

The property $\text{fv}(\mathsf{w_{u'}}()) = \varnothing$ comes from $\text{fv}(\mathsf{u'}) = \varnothing$.

To prove the property $\text{fv}(\mathsf{u}) \subseteq (\text{dom}(M_1) \cup \text{bv}(b))$, it suffices to prove $\text{fv}(\mathsf{u}) \subseteq (\text{dom}(B_1 \uplus B_2))$, as we have both $\text{dom}(B_1) \subseteq \text{dom}(M_1)$ by Lemma 5 and $\text{dom}(B_2) \subseteq \text{bv}(b)$ by Lemma 7.

The property FvCont(B, $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$, $\mathsf{w_{u'}}()$) is proved like in the case EVAL-PAT-WHEN-1.

- EVAL-PAT-WHEN-3.
  The conclusion of the evaluation rule is "$v \triangleright p$ **when** $b \Downarrow_{\text{pat}} \text{Match } M_1 \uplus M_2$". Its premises are "$v \triangleright p \Downarrow_{\text{pat}} \text{Match } M_1$" and "Subst($M_1$, b) $\Downarrow_{\text{bbe}}$ Match M_2".
  The typing assumptions are "$\vdash_{\text{val}} v : T$" and "$\vdash_{\text{pat}} p$ **when** $b : T \rightsquigarrow B$", from which we get by inversion that $\vdash_{\text{pat}} p : T \rightsquigarrow B_1$ and $B_1 \vdash_{\text{bbe}} b \rightsquigarrow B_2$ where $B = B_1 \uplus B_2$, $\mathsf{u'} \Downarrow_{\text{ml}} \mathsf{q}$, $\text{fv}(\mathsf{u'}) = \varnothing$ and $\text{fv}(\mathsf{u}) \subseteq \text{dom}(B)$.
  The additional hypotheses are "TrCont(Match $M_1 \uplus M_2$, u, $\mathsf{u'}$, q)", which simplifies to Subst($[\![M_1 \uplus M_2]\!]$, u) $\Downarrow_{\text{ml}}$ q and "FvCont(B, u, $\mathsf{u'}$)".
  We have to show that $\texttt{let k() = u' in } [\![[\![v]\!] \blacktriangleright p]\!]^{[\![b]\!]^{\mathsf{u}}_{\mathsf{k}()}}_{\mathsf{k}()} \Downarrow_{\text{ml}} \mathsf{q}$.
  By applying ML-LET-1, there remains to prove that $[\![[\![v]\!] \blacktriangleright p]\!]^{[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}}_{\mathsf{w_{u'}}()} \Downarrow_{\text{ml}} \mathsf{q}$, where $\mathsf{w_{u'}} = \texttt{fun () -> u'}$.
  We prove this by applying IH (3) on "$v \triangleright p \Downarrow_{\text{pat}} \text{Match } M_1$" with continuations $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$ and $\mathsf{w_{u'}}()$.

There remains to prove the premises of IH (3), that is: TrCont(Match M_1, $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$, $\mathsf{w_{u'}}()$, q) and FvCont(B, $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$, $\mathsf{w_{u'}}()$).

The property TrCont(Match M_1, $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$, $\mathsf{w_{u'}}()$, q) simplifies to Subst($[\![M_1]\!]$, $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$) $\Downarrow_{\mathrm{ml}}$ q.
This is proved by applying Lemma 10. There remains to prove its premises, that is: $[\![\mathrm{Subst}(M_1, b)]\!]^{\mathrm{Subst}([\![M_1]\!],\mathsf{u})}_{\mathsf{w_{u'}}()} \Downarrow_{\mathrm{ml}} \mathsf{q}$ and $\mathrm{fv}(\mathsf{w_{u'}}()) = \varnothing$ and $\mathrm{fv}(\mathsf{u}) \subseteq (\mathrm{dom}(M_1) \cup \mathrm{bv}(b))$.

The property $[\![\mathrm{Subst}(M_1,\ b)]\!]^{\mathrm{Subst}([\![M_1]\!],\mathsf{u})}_{\mathsf{w_{u'}}()} \Downarrow_{\mathrm{ml}} \mathsf{q}$ is proved by applying IH (2) on "Subst(M_1, b) $\Downarrow_{\mathrm{bbe}}$ Match M_2" with continuations Subst($[\![M_1]\!]$, u) and $\mathsf{w_{u'}}()$.
There remains to prove the premises of IH (2), that is: TrCont(Match M_2, Subst($[\![M_1]\!]$, u), $\mathsf{w_{u'}}()$, q) and FvCont(B_2, Subst($[\![M_1]\!]$, u), $\mathsf{w_{u'}}()$).

- The property TrCont(Match M_2, Subst($[\![M_1]\!]$, u), $\mathsf{w_{u'}}()$, q) simplifies to Subst($[\![M_2]\!]$, Subst($[\![M_1]\!]$, u)) $\Downarrow_{\mathrm{ml}}$ q, which itself simplifies to Subst($[\![M_1 \uplus M_2]\!]$, u) $\Downarrow_{\mathrm{ml}}$ q by disjointness of the maps M_1 and M_2, which we have by TrCont(Match $M_1 \uplus M_2$, u, $\mathsf{u'}$, q).
- The property FvCont(B_2, Subst($[\![M_1]\!]$, u), $\mathsf{w_{u'}}()$) is proved like in the case EVAL-PAT-WHEN-2.

The property $\mathrm{fv}(\mathsf{w_{u'}}()) = \varnothing$ comes from $\mathrm{fv}(\mathsf{u'}) = \varnothing$.

To prove the property $\mathrm{fv}(\mathsf{u}) \subseteq (\mathrm{dom}(M_1) \cup \mathrm{bv}(b))$, it suffices to prove $\mathrm{fv}(\mathsf{u}) \subseteq (\mathrm{dom}(B_1 \uplus B_2))$, as we have both $\mathrm{dom}(B_1) \subseteq \mathrm{dom}(M_1)$ by Lemma 5 and $\mathrm{dom}(B_2) \subseteq \mathrm{bv}(b)$ by Lemma 7.

The property FvCont(B, $[\![b]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$, $\mathsf{w_{u'}}()$) is proved like in the case EVAL-PAT-WHEN-1.

- EVAL-PAT-SOME-1.
The conclusion of the evaluation rule is $v \triangleright \mathrm{Some}(p_1, ..., p_n) \Downarrow_{\mathrm{pat}} \mathrm{Mismatch}$. Its premise is $\forall v_1, ..., v_n.\ v \neq \mathrm{Some}(v_1, ..., v_n)$.
The typing assumptions are $\vdash_{\mathrm{val}} v : T$ and $\vdash_{\mathrm{pat}} \mathrm{Some}(p_1, ..., p_n) : T \rightsquigarrow B$, from which we get by inversion that: $\forall i \in [1, n].\ (\biguplus_{1 \le j < i} B_j) \vdash_{\mathrm{pat}} p_i : T_i \rightsquigarrow B_i$, and $\forall i, j \in [1, n].\ i \neq j \Rightarrow \mathrm{pv}(p_i) \cap \mathrm{pv}(p_j) = \varnothing$ where $B = \biguplus_{j \in 1..n} B_j$.
By using Lemma 8 on the typing judgments, we have $\forall i \in [1, n].\ \mathrm{fv}(p_i) \subseteq \mathrm{dom}(\biguplus_{1 \le j < i} B_j)$.
The additional hypotheses are TrCont(Mismatch, u, $\mathsf{u'}$, q), and FvCont(B, u, $\mathsf{u'}$).
We have to show that
`let k() = u' in match [[v]] with` $\Downarrow_{\mathrm{ml}} \mathsf{q}$.
| Some(x_1, ..., x_n) → $[\![(x_1$ **is** $p_1)$ **and** ... **and** $(x_n$ **is** $p_n)]\!]^{\mathsf{u}}_{\mathsf{k}()}$
| _ → k()
By applying ML-LET-1, it suffices to prove
`match [[v]] with` $\Downarrow_{\mathrm{ml}} \mathsf{q}$,
| Some(x_1, ..., x_n) → $[\![(x_1$ **is** $p_1)$ **and** ... **and** $(x_n$ **is** $p_n)]\!]^{\mathsf{u}}_{\mathsf{w_{u'}}()}$
| _ → $\mathsf{w_{u'}}()$
where $\mathsf{w_{u'}}$ = (fun () → u').
By applying ML-MATCH-2, it suffices to prove that $\forall \mathsf{w}_1, ..., \mathsf{w}_n.\ [\![v]\!] \neq \mathtt{Some}\ (\mathsf{w}_1, ..., \mathsf{w}_n)$ and $\mathsf{w_{u'}}() \Downarrow_{\mathrm{ml}} \mathsf{q}$.

To prove the former, assume that there exists values $\mathsf{w}_1, ..., \mathsf{w}_n$ such that $[\![v]\!] = \mathtt{Some}\ (\mathsf{w}_1, ..., \mathsf{w}_n)$.
By inspecting Figure 28, only results of the form Some(v_1, ..., v_n) can translate into result of the form $\mathtt{Some}\ (\mathsf{w}_1, ..., \mathsf{w}_n)$. We deduce that there exists values $v_1, ..., v_n$ such that v = Some(v_1, ..., v_n), which contradicts the assumption.
We conclude that $[\![v]\!] \neq \mathtt{Some}\ (\mathsf{w}_1, ..., \mathsf{w}_n)$.
We prove the latter by applying ML-BETA with $\mathsf{u}' \Downarrow_{\text{ml}} \mathsf{q}$, that we get from TrCont(r, u, u′, q).

- EVAL-PAT-SOME-2 and EVAL-PAT-SOME-3
  We define the following notations for clarity.
  $N_k = \biguplus_{i \in 1..k} M_i$.
  $p_i^k = \text{Subst}(N_{k-1}, p_i)$.
  $\bigwedge_{i \in 1..n}(v_i \textbf{ is } p_i) = (v_1 \textbf{ is } p_1) \textbf{ and } ... \textbf{ and } (v_n \textbf{ is } p_n)$.
  $b_k = \bigwedge_{i \in k..n}(v_i \textbf{ is } p_i^k)$.
  $b'_{k+1} = \bigwedge_{i \in (k+1)..n}(v_i \textbf{ is } p_i^k)$.
  $\mathsf{w}_{\mathsf{u}'} = (\mathtt{fun}\ ()\ \rightarrow\ \mathsf{u}')$.
  $\mathsf{u}_k = \text{Subst}([\![N_{k-1}]\!], \mathsf{u})$.
  $\text{fv}(\bar{\mathsf{x}}_k) = (\bigcup_{i \in k..n} \text{bv}(p_i)) \cap \text{fv}(\mathsf{u}_k)$.
  $\mathsf{w}_{\mathsf{u}_k} = \mathtt{fun}\ \bar{\mathsf{x}}_k\ \rightarrow\ \mathsf{u}_k$.
  Since all values are closed, we have by definition, $b_{k+1} = \text{Subst}(M_k, b'_{k+1})$, and that by disjunction of the result maps, $\mathsf{u}_{k+1} = \text{Subst}([\![M_k]\!], \mathsf{u}_k)$.
  Notice that the premises for the two rules EVAL-PAT-SOME-2 and EVAL-PAT-SOME-3 are very similar, as we can factorize the premises in such a way:
  We denote m the first index such that $v_i \triangleright p_i^i \Downarrow_{\text{pat}}$ Mismatch. If there is no such pattern, we set by convention $m = n + 1$.
  We have as premises "$\forall i \in [1, m).\ v_i \triangleright p_i^i \Downarrow_{\text{pat}}$ Match M_i" and "$m \leq n \Rightarrow v_m \triangleright p_m^m \Downarrow_{\text{pat}}$ Mismatch", and assume the induction hypothesis on these premises.
  The result Some(v_1, ..., v_n) $\triangleright$ Some(p_1, ..., p_n) $\Downarrow_{\text{pat}} r$ depends only on the value of m. If $m \leq n$, the premises prove that r = Mismatch. Whereas if $m = n + 1$, the premises prove that r = Match $\biguplus_{i \in 1..n} M_i$.
  The typing assumptions are $\vdash_{\text{val}} v : T$ and $\vdash_{\text{pat}}$ Some(p_1, ..., p_n) $: T \rightsquigarrow B$, from which we get by inversion that: $\forall i \in [1, n].\ (\biguplus_{1 \leq j < i} B_j) \vdash_{\text{pat}} p_i : T_i \rightsquigarrow B_i$ and $\forall i, j \in [1, n].\ i \neq j \Rightarrow \text{pv}(p_i) \cap \text{pv}(p_j) = \varnothing$ where $B = \biguplus_{j \in 1..n} B_j$.
  By using Lemma 8 on the typing judgments, we have that
  $\forall i \in [1, n].\ \text{fv}(p_i) \subseteq \text{dom}(\biguplus_{1 \leq j < i} B_j)$.
  The additional hypotheses are TrCont(r, u, u′, q), and FvCont(B, u, u′).
  We have to show that
  $\mathtt{let\ k()\ =\ u'\ in\ match\ Some\ (}[\![v_1]\!], ..., [\![v_n]\!]\mathtt{)\ with}\ \Downarrow_{\text{ml}} \mathsf{q}$.
  $\mathtt{|\ Some(}\mathsf{x}_1, ..., \mathsf{x}_n\mathtt{)}\ \rightarrow\ [\![\bigwedge_{i \in 1..n}(x_i \textbf{ is } p_i)]\!]^{\mathsf{u}}_{\mathsf{k}()}$
  $\mathtt{|\ \_\ \rightarrow\ k()}$

By applying ML-LET-1, it suffices to prove that
`match Some (`$[\![v_1]\!], ..., [\![v_n]\!]$`) with` $\Downarrow_{\text{ml}}$ q.
`| Some(`$x_1, ..., x_n$`)` $\rightarrow$ $[\![\bigwedge_{i\in 1..n}(x_i \textbf{ is } p_i)]\!]^{u}_{w_{u'}()}$
`| _` $\rightarrow$ $w_{u'}()$
By applying ML-MATCH-1 with `Some(`$[\![v_1]\!], ..., [\![v_n]\!]$`)` $\Downarrow_{\text{ml}}$ `Result (Some(`$[\![v_1]\!], ..., [\![v_n]\!]$`))`, there is left to prove that
$\text{Subst}([\![M]\!], [\![\bigwedge_{i\in 1..n}(x_i \textbf{ is } p_i)]\!]^{u}_{w_{u'}()}) \Downarrow_{\text{ml}} q$, where $M = \{x_1 \mapsto v_1, ..., x_n \mapsto v_n\}$.
By definition, the variables x_i are globally fresh in the program. This means in particular that $\text{dom}(M) \cap \text{fv}(u) = \varnothing$.
By applying Lemma 13 on $\text{Subst}([\![M]\!], [\![\bigwedge_{i\in 1..n}(x_i \textbf{ is } p_i)]\!]^{u}_{w_{u'}()}) \Downarrow_{\text{ml}} q$, and by disjointness of the map M, it suffices to prove that
$\text{fv}(u) \subseteq \text{bv}(\bigwedge_{i\in 1..n}(x_i \textbf{ is } p_i))$ and $\text{fv}(w_{u'}()) = \varnothing$ and
$[\![\text{Subst}([\![M]\!], \bigwedge_{i\in 1..n}(x_i \textbf{ is } p_i))]\!]^{u}_{w_{u'}()} \Downarrow_{\text{ml}} q$.

- We have by definition that $\text{bv}(\bigwedge_{i\in 1..n}(x_i \textbf{ is } p_i)) = \bigcup_{1\le i\le n} \text{bv}(p_i)$. We also have by $\text{FvCont}(B, u, u')$ that $\text{fv}(u) \subseteq \text{dom}(B) = \text{dom}(\biguplus_{i\in 1..n} B_i)$. By successive application of Lemma 7, we have the inclusions: $\text{dom}(B_i) \subseteq \text{bv}(p_i)$ for all i, from which we conclude that $\text{fv}(u) \subseteq \bigcup_{1\le i\le n} \text{bv}(p_i)$.
- We have by $\text{FvCont}(B, u, u')$ that $\text{fv}(u') = \varnothing$, which is enough to conclude that $\text{fv}(w_{u'}()) = \varnothing$.
- The variables x_i are globally fresh in the program. This means in particular that $\forall p_i.\ \text{dom}(M) \cap \text{fv}(p_i) = \varnothing$.
  The property $[\![\text{Subst}(M, \bigwedge_{i\in 1..n}(x_i \textbf{ is } p_i))]\!]^{u}_{w_{u'}()} \Downarrow_{\text{ml}} q$ simplifies to
  $[\![\bigwedge_{i\in 1..n}(v_i \textbf{ is } p_i)]\!]^{u}_{w_{u'}()} \Downarrow_{\text{ml}} q$.
  We conclude this with the key property that we prove below, with k = 1:
  $\forall k \in 1..n.\ (k \le m) \Rightarrow [\![b_k]\!]^{u_k}_{u'} \Downarrow_{\text{ml}} q$,
  We prove the property by recurrence on $n - k$.
  - Inductive case: $1 \le k < n$.
    If $k > m$, then there is nothing to prove. We assume for the rest of this case that $k \le m$.
    In this case, we have by IH that $[\![b_{k+1}]\!]^{u_{k+1}}_{u'} \Downarrow_{\text{ml}} q$.
    We have to show that $[\![b_k]\!]^{u_k}_{u'} \Downarrow_{\text{ml}} q$.
    By inlining the definition of the **and** construct, there is left to prove that:
    $[\![(v_k \textbf{ is } p^k_k) \textbf{ and } b'_{k+1}]\!]^{u_k}_{u'} \Downarrow_{\text{ml}} q$.
    By translating the **and** construct, there is left to prove that:
    `let` $k_1 \bar{x}_k$ `=` u_k `in let` $k_2()$ `=` u' `in` $[\![v_k \textbf{ is } p^k_k]\!]^{[\![b'_{k+1}]\!]^{k_1 \bar{x}_k}_{k_2()}}_{k_2()} \Downarrow_{\text{ml}} q$.
    By applying ML-LET-1 twice, there remains to show that $[\![v_k \textbf{ is } p^k_k]\!]^{[\![b'_{k+1}]\!]^{w_{u_k} \bar{x}_k}_{w_{u'}()}}_{w_{u'}()} \Downarrow_{\text{ml}} q$,
    By translation of the **is** construct, this simplifies to
    `let y =` $[\![v_k]\!]$ `in` $[\![y \blacktriangleright p^k_k]\!]^{[\![b'_{k+1}]\!]^{w_{u_k} \bar{x}_k}_{w_{u'}()}}_{w_{u'}()} \Downarrow_{\text{ml}} q$, for a globally fresh variable y.

By a similar reasoning as in the case EVAL-BBE-IS, the property simplifies to $[\![[\![v_k]\!] \blacktriangleright p_k^k]\!]_{\mathsf{w_{u'}}()}^{[\![b'_{k+1}]\!]_{\mathsf{w_{u'}}()}^{\mathsf{w_{u_k}}\bar{\mathsf{x}}_k}} \Downarrow_{\mathrm{ml}} \mathsf{q}$.

We prove this by applying IH (3) on $v_k \triangleright p_k^k \Downarrow_{\mathrm{pat}} r_k$, with continuations $[\![b'_{k+1}]\!]_{\mathsf{w_{u'}}()}^{\mathsf{w_{u_k}}\bar{\mathsf{x}}_k}$ and $\mathsf{w_{u'}}()$, where r_k is the expected result of the evaluation of p_k^k against v_k, which depends solely on the value of k relative to m.

There is left to prove the premises of IH (3), that is: $\mathrm{TrCont}(r_k, [\![b'_{k+1}]\!]_{\mathsf{w_{u'}}()}^{\mathsf{w_{u_k}}\bar{\mathsf{x}}_k}, \mathsf{w_{u'}}(), \mathsf{q})$ and $\mathrm{FvCont}(B_k, [\![b'_{k+1}]\!]_{\mathsf{w_{u'}}()}^{\mathsf{w_{u_k}}\bar{\mathsf{x}}_k}, \mathsf{w_{u'}}())$.

- We prove $\mathrm{TrCont}(r_k, [\![b'_{k+1}]\!]_{\mathsf{w_{u'}}()}^{\mathsf{w_{u_k}}\bar{\mathsf{x}}_k}, \mathsf{w_{u'}}(), \mathsf{q})$ by case disjunction on k.

  1. If $k = m$, then $v_k \triangleright p_k^k \Downarrow_{\mathrm{pat}} \mathrm{Mismatch}$.

  The property $\mathrm{TrCont}(\mathrm{Mismatch}, [\![b'_{k+1}]\!]_{\mathsf{w_{u'}}()}^{\mathsf{w_{u_k}}\bar{\mathsf{x}}_k}, \mathsf{w_{u'}}(), \mathsf{q})$ simplifies to $\mathsf{w_{u'}}() \Downarrow_{\mathrm{ml}} \mathsf{q}$.

  In this case the premises are the ones of EVAL-PAT-SOME-2, this means that the end result is $r = \mathrm{Mismatch}$. We deduce that $\mathrm{TrCont}(r, \mathsf{u}, \mathsf{u'}, \mathsf{q})$ simplifies to $\mathsf{u'} \Downarrow_{\mathrm{ml}} \mathsf{q}$.

  We conclude by application of ML-BETA with $\mathsf{u'} \Downarrow_{\mathrm{ml}} \mathsf{q}$.

  2. If $k < m$, then $v_k \triangleright p_k^k \Downarrow_{\mathrm{pat}} \mathrm{Match}\ M_k$.

  The property $\mathrm{TrCont}(\mathrm{Match}\ M_k, [\![b'_{k+1}]\!]_{\mathsf{w_{u'}}()}^{\mathsf{w_{u_k}}\bar{\mathsf{x}}_k}, \mathsf{w_{u'}}(), \mathsf{q})$ simplifies to $\mathrm{Subst}([\![M_k]\!], [\![b'_{k+1}]\!]_{\mathsf{w_{u'}}()}^{\mathsf{w_{u_k}}\bar{\mathsf{x}}_k}) \Downarrow_{\mathrm{ml}} \mathsf{q}$.

  This is proved by applying Lemma 10. There remains to prove its premises, that is: $[\![\mathrm{Subst}(M_k,\ b'_{k+1})]\!]_{\mathsf{w_{u'}}()}^{\mathrm{Subst}([\![M_k \setminus \mathrm{bv}(b'_{k+1})]\!], \mathsf{w_{u_k}}\bar{\mathsf{x}}_k)} \Downarrow_{\mathrm{ml}} \mathsf{q}$ and $\mathrm{fv}(\mathsf{w_{u'}}()) = \varnothing$ and $\mathrm{fv}(\mathsf{w_{u_k}}\bar{\mathsf{x}}_k) \subseteq (\mathrm{dom}(M_k) \cup \mathrm{bv}(b'_{k+1}))$.

  2.1 $[\![\mathrm{Subst}(M_k, b'_{k+1})]\!]_{\mathsf{w_{u'}}()}^{\mathrm{Subst}([\![M_k \setminus \mathrm{bv}(b'_{k+1})]\!], \mathsf{w_{u_k}}\bar{\mathsf{x}}_k)} \Downarrow_{\mathrm{ml}} \mathsf{q}$

  Notice that by definition $\mathrm{bv}(b'_{k+1}) = \bigcup_{i \in (k+1)..n} \mathrm{bv}(p_i)$. By assumption, the $\mathrm{pv}(p_i)$ are disjoint, and we have $\mathrm{bv}(p_i) \subseteq \mathrm{pv}(p_i)$. We deduce that the $\mathrm{bv}(p_i)$ are disjoint.

  In particular, since $\mathrm{dom}(M_k) \subseteq \mathrm{pv}(p_k)$ by Lemma 3, it is disjoint with $\bigcup_{i \in (k+1)..n} \mathrm{bv}(p_i) = \mathrm{bv}(b'_{k+1})$, so $\mathrm{Subst}([\![M_k \setminus \mathrm{bv}(b'_{k+1})]\!], \mathsf{w_{u_k}}\bar{\mathsf{x}}_k)$ simplifies to $\mathrm{Subst}([\![M_k]\!], \mathsf{w_{u_k}}\bar{\mathsf{x}}_k)$.

  There is left to prove that $[\![b_{k+1}]\!]_{\mathsf{w_{u'}}()}^{\mathrm{Subst}([\![M_k]\!], \mathsf{w_{u_k}}\bar{\mathsf{x}}_k)} \Downarrow_{\mathrm{ml}} \mathsf{q}$.

  We prove this by congruence with the IH $[\![b_{k+1}]\!]_{\mathsf{u'}}^{\mathsf{u}_{k+1}} \Downarrow_{\mathrm{ml}} \mathsf{q}$. It is clear that if $\mathsf{w_{u'}}() \equiv \mathsf{u'}$ and $\mathrm{Subst}([\![M_k]\!], \mathsf{w_{u_k}}\bar{\mathsf{x}}_k) \equiv \mathsf{u}_{k+1}$, then $[\![b_{k+1}]\!]_{\mathsf{u'}}^{\mathsf{u}_{k+1}} \Downarrow_{\mathrm{ml}} \mathsf{q}$ is enough to prove that $[\![b_{k+1}]\!]_{\mathsf{w_{u'}}()}^{\mathrm{Subst}([\![M_k]\!], \mathsf{w_{u_k}}\bar{\mathsf{x}}_k)} \Downarrow_{\mathrm{ml}} \mathsf{q}$.

  For the former, we have by $\mathrm{FvCont}(B, \mathsf{u}, \mathsf{u'})$ that $\mathrm{fv}(\mathsf{u'}) = \varnothing$. This means that any substitution on either $\mathsf{w_{u'}}()$ or $\mathsf{u'}$ is the identity. We can then conclude by application (or inversion) of ML-BETA.

  We prove the latter by η-equivalence. Notice that if $x \in \mathrm{fv}(t)$, $(\lambda x.t)x \equiv t$.

This means that proving that $\mathrm{fv}(\bar{\mathsf{x}}_k) \subseteq \mathrm{fv}(u_k)$ is enough to prove that $\mathsf{w}_{\mathsf{u}_k}\bar{\mathsf{x}}_k \equiv \mathsf{u}_k$, which is enough to prove that $\mathrm{Subst}(\llbracket M_k \rrbracket, \mathsf{w}_{\mathsf{u}_k}\bar{\mathsf{x}}_k) \equiv \mathrm{Subst}(\llbracket M_k \rrbracket, \mathsf{u}_k) = \mathsf{u}_{k+1}$.
By definition, $\mathrm{fv}(\bar{\mathsf{x}}_k) = (\bigcup_{i\in k..n} \mathrm{bv}(p_i)) \cap \mathrm{fv}(\mathsf{u}_k) \subseteq \mathrm{fv}(\mathsf{u}_k)$, which is enough to conclude.
2.2 $\mathrm{fv}(\mathsf{w}_{\mathsf{u}'}()) = \varnothing$ We have $\mathrm{fv}(\mathsf{w}_{\mathsf{u}'}()) = \mathrm{fv}(\mathsf{u}') = \varnothing$ by $\mathrm{FvCont}(B, \mathsf{u}, \mathsf{u}')$.
2.3 $\mathrm{fv}(\mathsf{w}_{\mathsf{u}_k}\bar{\mathsf{x}}_k) \subseteq (\mathrm{dom}(M_k) \cup \mathrm{bv}(b'_{k+1}))$.
This case is detailed in the case "2." of the proof of $\mathrm{FvCont}(B_k, \llbracket b'_{k+1} \rrbracket^{\mathsf{w}_{\mathsf{u}_k}\bar{\mathsf{x}}_k}_{\mathsf{w}_{\mathsf{u}'}()}, \mathsf{w}_{\mathsf{u}'}())$.

- The property $\mathrm{FvCont}(B_k, \llbracket b'_{k+1} \rrbracket^{\mathsf{w}_{\mathsf{u}_k}\bar{\mathsf{x}}_k}_{\mathsf{w}_{\mathsf{u}'}()}, \mathsf{w}_{\mathsf{u}'}())$ simplifies to $\mathrm{fv}(\mathsf{w}_{\mathsf{u}'}()) = \varnothing \wedge \mathrm{fv}(\llbracket b'_{k+1} \rrbracket^{\mathsf{w}_{\mathsf{u}_k}\bar{\mathsf{x}}_k}_{\mathsf{w}_{\mathsf{u}'}()}) \subseteq \mathrm{dom}(B_k)$
  We have by $\mathrm{FvCont}(B, \mathsf{u}, \mathsf{u}')$ that $\mathrm{fv}(\mathsf{u}') = \varnothing$, which is enough to conclude the former.
  We have by Lemma 6 that $\mathrm{fv}(\llbracket b'_{k+1} \rrbracket^{\mathsf{w}_{\mathsf{u}_k}\bar{\mathsf{x}}_k}_{\mathsf{w}_{\mathsf{u}'}()}) \subseteq \mathrm{fv}(b'_{k+1}) \cup (\mathrm{fv}(\mathsf{w}_{\mathsf{u}_k}\bar{\mathsf{x}}_k) \setminus \mathrm{bv}(b'_{k+1})) \cup \mathrm{fv}(\mathsf{w}_{\mathsf{u}'}())$.
  We have stated earlier that $\mathrm{fv}(\mathsf{w}_{\mathsf{u}'}()) = \varnothing$. We have by inspection of the operation $\mathrm{bv}()$ that $\mathrm{bv}(p_i^k) = \mathrm{bv}(p_i)$. Remember that $b'_{k+1} = \bigwedge_{i\in(k+1)..n}(v_i \textbf{ is } p_i^k)$.
  This means that $\mathrm{fv}(\mathsf{w}_{\mathsf{u}_k}\bar{\mathsf{x}}_k) \setminus \mathrm{bv}(b'_{k+1}) = \mathrm{fv}(\mathsf{w}_{\mathsf{u}_k}\bar{\mathsf{x}}_k) \setminus \bigcup_{i\in(k+1)..n} \mathrm{bv}(p_i)$.
  There is left to prove both inclusions: $\mathrm{fv}(b'_{k+1}) \subseteq \mathrm{dom}(B_k)$ and $\mathrm{fv}(\mathsf{w}_{\mathsf{u}_k}\bar{\mathsf{x}}_k) \setminus \bigcup_{i\in(k+1)..n} \mathrm{bv}(p_i) \subseteq \mathrm{dom}(B_k)$
  1. By using Lemma 2 on the typing hypothesis "$\forall i \in [1, n].\ (\biguplus_{1\le j<i} B_j) \vdash_{\mathrm{pat}} p_i : T_i \rightsquigarrow B_i$", we have $\forall i \in [k+1,\ n].\ \biguplus_{j\in[k,i)} B_j \vdash_{\mathrm{pat}} p_i^k : T_i \rightsquigarrow B_i$.
  We gather by successive application of Lemma 8 that $\mathrm{fv}(p_i^k) \subseteq \biguplus_{j\in[k,i)} \mathrm{dom}(B_j)$.
  By definition, $\mathrm{fv}(b'_{k+1}) = \bigcup_{i\in(k+1)..n}(\mathrm{fv}(p_i^k) \setminus \bigcup_{k\in[k+1,i)} \mathrm{bv}(p_j))$.
  This means by typing that it is enough to prove that $\bigcup_{i\in(k+1)..n}(\biguplus_{j\in[k,i)} \mathrm{dom}(B_j) \setminus \bigcup_{j\in[k+1,i)} \mathrm{bv}(p_j)) \subseteq \mathrm{dom}(B_k)$.
  By Lemma 7, $\forall i \in 1..n.\ \mathrm{dom}(B_i) \subseteq \mathrm{bv}(p_i)$, this means in particular that $\bigcup_{j\in[k+1,i)} \mathrm{dom}(B_j) \subseteq \bigcup_{j\in[k+1,i)} \mathrm{bv}(p_j)$.
  It thus suffices to prove that $\bigcup_{i\in(k+1)..n}(\biguplus_{j\in[k,i)} \mathrm{dom}(B_j) \setminus \bigcup_{j\in[k+1,i)} \mathrm{dom}(B_j)) \subseteq \mathrm{dom}(B_k)$, which simplifies to $(\bigcup_{i\in(k+1)..n} \mathrm{dom}(B_k)) \subseteq \mathrm{dom}(B_k)$, which is true by definition.
  2. $\mathrm{fv}(\mathsf{w}_{\mathsf{u}_k}\bar{\mathsf{x}}_k) \setminus \bigcup_{i\in(k+1)..n} \mathrm{bv}(p_i) \subseteq \mathrm{dom}(B_k)$
  By successive application of Lemma 7, we have $\bigcup_{i\in(k+1)..n} \mathrm{dom}(B_i) \subseteq \bigcup_{i\in(k+1)..n} \mathrm{bv}(p_i)$.
  By definition, $\mathrm{fv}(\mathsf{w}_{\mathsf{u}_k}\bar{\mathsf{x}}_k) = \mathrm{fv}(\mathsf{w}_{\mathsf{u}_k}) \cup \mathrm{fv}(\bar{\mathsf{x}}_k)$, it suffices to prove both inclusions $\mathrm{fv}(\mathsf{w}_{\mathsf{u}_k}) \setminus \bigcup_{i\in(k+1)..n} \mathrm{dom}(B_i) \subseteq \mathrm{dom}(B_k)$ and $\mathrm{fv}(\bar{\mathsf{x}}_k) \setminus \bigcup_{i\in(k+1)..n} \mathrm{dom}(B_i) \subseteq \mathrm{dom}(B_k)$.

2.1. $\mathrm{fv}(\mathsf{w}_{\mathsf{u}_k}) \setminus \bigcup_{i\in(k+1)..n} \mathrm{dom}(B_i) \subseteq \mathrm{dom}(B_k)$
We have by definition $\mathrm{fv}(\mathsf{w}_{\mathsf{u}_k}) \subseteq \mathrm{fv}(\mathsf{u}_k) =$
$\mathrm{fv}(\mathrm{Subst}(\llbracket \biguplus_{j\in1..(k-1)} M_j \rrbracket, \mathsf{u})) = \mathrm{fv}(\mathsf{u}) \setminus (\mathrm{dom}(\biguplus_{j\in1..(k-1)} M_j))$.
By successive application of Lemma 5, we have $\mathrm{dom}(\biguplus_{j\in1..(k-1)} B_j) \subseteq$
$\mathrm{dom}(\biguplus_{j\in1..(k-1)} M_j)$.
Also, by $\mathrm{FvCont}(B, \mathsf{u}, \mathsf{u}')$, we have $\mathrm{fv}(\mathsf{u}) \subseteq \mathrm{dom}(B) = \mathrm{dom}(\biguplus_{i\in1..n} B_i)$.
We deduce that it suffices to prove that
$\mathrm{dom}(\biguplus_{i\in1..n} B_i) \setminus (\mathrm{dom}(\biguplus_{j\in1..(k-1)} B_j) \cup \bigcup_{i\in(k+1)..n} \mathrm{dom}(B_i)) \subseteq \mathrm{dom}(B_k)$,
which simplifies to $\mathrm{dom}(B_k) \subseteq \mathrm{dom}(B_k)$, which is true.
2.2. $\mathrm{fv}(\bar{\mathsf{x}}_k) \setminus \bigcup_{i\in(k+1)..n} \mathrm{dom}(B_i) \subseteq \mathrm{dom}(B_k)$
We have stated earlier that $\mathrm{fv}(\bar{\mathsf{x}}_k) = (\bigcup_{i\in k..n} \mathrm{bv}(p_i)) \cap \mathrm{fv}(\mathsf{u}_k)$.
In particular, $\mathrm{fv}(\bar{\mathsf{x}}_k) \subseteq \mathrm{fv}(\mathsf{u}_k)$.
We proved in the previous point that $\mathrm{fv}(\mathsf{u}_k) \setminus \bigcup_{i\in(k+1)..n} \mathrm{dom}(B_i) \subseteq$
$\mathrm{dom}(B_k)$, which is enough to conclude.

- Base case: $k = n$.
If $n = m$, then the proof is exactly the same as Case 1 above. We thus assume that $n < m$.
By definition of m, this implies that $m = n + 1$. Hence $v_n \triangleright p_n^n \Downarrow_{\mathrm{pat}}$ Match M_n. Moreover, by the discussion above on the result of the whole evaluation, we have r = Match N_n = Match $\biguplus_{i\in1..n} M_i$. In particular, $\mathrm{TrCont}(r, \mathsf{u}, \mathsf{u}', \mathsf{q})$ simplifies to $\mathrm{Subst}(\llbracket N_n \rrbracket, \mathsf{u}) \Downarrow_{\mathrm{ml}} \mathsf{q}$, that is, $\mathsf{u}_{n+1} \Downarrow_{\mathrm{ml}} \mathsf{q}$.
We have to show that $\llbracket b_n \rrbracket^{\mathsf{u}_n}_{\mathsf{u}'} \Downarrow_{\mathrm{ml}} \mathsf{q}$.
By definition of b_n, there is left to prove that $\llbracket v_n \textbf{ is } p_n^n \rrbracket^{\mathsf{u}_n}_{\mathsf{u}'} \Downarrow_{\mathrm{ml}} \mathsf{q}$.
By translation of the **is** construct, there is left to prove that:
`let y =` $\llbracket v_n \rrbracket$ `in` $\llbracket \mathsf{y} \blacktriangleright p_n^n \rrbracket^{\mathsf{u}_n}_{\mathsf{u}'} \Downarrow_{\mathrm{ml}} \mathsf{q}$, for a globally fresh variable y.
By a similar reasoning as in the case EVAL-BBE-IS, the property simplifies to $\llbracket \llbracket v_n \rrbracket \blacktriangleright p_n^n \rrbracket^{\mathsf{u}_n}_{\mathsf{u}'} \Downarrow_{\mathrm{ml}} \mathsf{q}$.
We prove this by applying IH (3) on $v_n \triangleright p_n^n \Downarrow_{\mathrm{pat}}$ Match M_n, with continuations u_n and u'.
There is left to prove the premises of IH (3), that is: $\mathrm{TrCont}(\text{Match } M_n, \mathsf{u}_n, \mathsf{u}', \mathsf{q})$ and $\mathrm{FvCont}(B_n, \mathsf{u}_n, \mathsf{u}')$.
1. The property $\mathrm{TrCont}(\text{Match } M_n, \mathsf{u}_n, \mathsf{u}', \mathsf{q})$ simplifies to $\mathrm{Subst}(\llbracket M_n \rrbracket, \mathsf{u}_n) \Downarrow_{\mathrm{ml}} \mathsf{q}$.
By definition, we have $\mathrm{Subst}(\llbracket M_n \rrbracket, \mathsf{u}_n) = \mathsf{u}_{n+1}$. We conclude with $\mathrm{TrCont}(r, \mathsf{u}, \mathsf{u}', \mathsf{q})$, since r = Match N_n.
2. The property $\mathrm{FvCont}(B_n, \mathsf{u}_n, \mathsf{u}')$ simplifies to $\mathrm{fv}(\mathsf{u}') = \varnothing \wedge \mathrm{fv}(\mathsf{u}_n) \subseteq \mathrm{dom}(B_n)$.
The former follows from $\mathrm{FvCont}(B, \mathsf{u}, \mathsf{u}')$.
For the latter, by definition, $\mathrm{fv}(\mathsf{u}_n) = \mathrm{fv}(\mathrm{Subst}(\llbracket N_{n-1} \rrbracket, \mathsf{u})) = \mathrm{fv}(\mathsf{u}) \setminus \mathrm{dom}(N_{n-1})$.
Also, by $\mathrm{FvCont}(B, \mathsf{u}, \mathsf{u}')$, we have $\mathrm{fv}(\mathsf{u}) \subseteq \mathrm{dom}(B) = \mathrm{dom}(\biguplus_{i\in1..n} B_i)$.

By successive application of the pattern domain-inclusion lemma on "$\forall i \in [1, n).\ v_i \triangleright p_i^i \Downarrow_{\text{pat}} \text{Match } M_i$", we have $\text{dom}(\biguplus_{i \in 1..(n-1)} B_i) \subseteq \text{dom}(N_{n-1})$.
We deduce that $\text{fv}(\mathsf{u}_n) \subseteq \text{dom}(B_n)$.

□

The proof involves, as expected, substitution lemmas, which we state next.

Lemma 10 (Substitution lemmas for translation). *The following statements hold:*

(1) $[\![\text{Subst}(M, t)]\!] \Downarrow_{\text{ml}} \mathsf{q} \Rightarrow (\text{Subst}([\![M]\!], [\![t]\!]) \Downarrow_{\text{ml}} \mathsf{q})$
(2) $[\![\text{Subst}(M, b)]\!]_{\mathsf{u}'}^{\text{Subst}([\![M \setminus \text{bv}(b)]\!], \mathsf{u})} \Downarrow_{\text{ml}} \mathsf{q} \wedge \text{fv}(\mathsf{u}') = \varnothing \wedge \text{fv}(\mathsf{u}) \subseteq (\text{dom}(M) \cup \text{bv}(b)) \Rightarrow (\text{Subst}([\![M]\!], [\![b]\!]_{\mathsf{u}'}^{\mathsf{u}}) \Downarrow_{\text{ml}} \mathsf{q})$
(3) $[\![\mathsf{y} \blacktriangleright \text{Subst}(M, p)]\!]_{\mathsf{u}'}^{\text{Subst}([\![M \setminus \text{bv}(p)]\!], \mathsf{u})} \Downarrow_{\text{ml}} \mathsf{q} \wedge \text{fv}(\mathsf{u}') = \varnothing \wedge \text{fv}(\mathsf{u}) \subseteq (\text{dom}(M) \cup \text{bv}(p)) \Rightarrow (\text{Subst}([\![M]\!], [\![\mathsf{y} \blacktriangleright p]\!]_{\mathsf{u}'}^{\mathsf{u}}) \Downarrow_{\text{ml}} \mathsf{q})$

Definition 1. We say that two terms u and u' are congruent, written $\mathsf{u} \equiv \mathsf{u}'$ if, for any map M,

$$\text{Subst}(M, \mathsf{u}) \Downarrow_{\text{ml}} \mathsf{q} \iff \text{Subst}(M, \mathsf{u}') \Downarrow_{\text{ml}} \mathsf{q}$$

Lemma 11 (Substitution lemmas for translation (congruent version)). *The following statements hold:*

(1) $[\![\text{Subst}(M, t)]\!] \equiv \text{Subst}([\![M]\!], [\![t]\!])$
(2) $\text{fv}(\mathsf{u}') = \varnothing \wedge \text{fv}(\mathsf{u}) \subseteq (\text{dom}(M) \cup \text{bv}(b)) \Rightarrow ([\![\text{Subst}(M, b)]\!]_{\mathsf{u}'}^{\text{Subst}([\![M \setminus \text{bv}(b)]\!], \mathsf{u})} \equiv \text{Subst}([\![M]\!], [\![b]\!]_{\mathsf{u}'}^{\mathsf{u}}))$
(3) $\text{fv}(\mathsf{u}') = \varnothing \wedge \text{fv}(\mathsf{u}) \subseteq (\text{dom}(M) \cup \text{bv}(p)) \Rightarrow ([\![\mathsf{y} \blacktriangleright \text{Subst}(M, p)]\!]_{\mathsf{u}'}^{\text{Subst}([\![M \setminus \text{bv}(p)]\!], \mathsf{u})} \equiv \text{Subst}([\![M]\!], [\![\mathsf{y} \blacktriangleright p]\!]_{\mathsf{u}'}^{\mathsf{u}}))$

Proof. We prove (1), (2), and (3) by strong mutual induction on the size of the AST. The induction hypothesis (IH) is: all three statements hold for every strict subterm (hence every strictly smaller AST).

- **Clause (1): terms.**
  - Case $t = x$.
    Hypotheses: none.
    Goal: $[\![\text{Subst}(M, x)]\!] \equiv \text{Subst}([\![M]\!], [\![x]\!])$.
    Immediate by unfolding substitution.
  - Case $t = n$.
    Hypotheses: none.
    Goal: $[\![\text{Subst}(M, n)]\!] \equiv \text{Subst}([\![M]\!], [\![n]\!])$.
    Immediate by unfolding.
  - Case $t = \textbf{let } x = t_1 \textbf{ in } t_2$ (representative detailed case).
    Hypotheses: IH for t_1 and t_2 (both strictly smaller).
    Goal: $[\![\text{Subst}(M, \textbf{let } x = t_1 \textbf{ in } t_2)]\!] \equiv \text{Subst}([\![M]\!], [\![\textbf{let } x = t_1 \textbf{ in } t_2]\!])$.
    By unfolding substitution and translation on both sides, this goal reduces to congruence of the two translated subterms corresponding to t_1 and t_2 (with

the standard restriction of M under binders). These are exactly IH instances. We conclude by congruence compatibility of `let`-contexts.

- Case $t = \lambda(x_1, ..., x_n).\, t_0$.
  Hypotheses: IH for t_0.
  Goal: $[\![\mathrm{Subst}(M, \lambda(x_1, ..., x_n).\, t_0)]\!] \equiv \mathrm{Subst}([\![M]\!], [\![\lambda(x_1, ..., x_n).\, t_0]\!])$. By unfolding (with restriction under binders) and IH.
- Case $t = t_0\ (t_1, ..., t_n)$.
  Hypotheses: IH for each t_i.
  Goal: $[\![\mathrm{Subst}(M,\ t_0\ (t_1, ..., t_n))]\!] \equiv \mathrm{Subst}([\![M]\!], [\![t_0\ (t_1, ..., t_n)]\!])$. By unfolding and IH on all arguments.
- Case $t = \mathbf{if}_L\ b_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$.
  Hypotheses: IH for b_0, t_1, and t_2.
  Goal: $[\![\mathrm{Subst}(M, \mathbf{if}_L\ b_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2)]\!] \equiv \mathrm{Subst}([\![M]\!], [\![\mathbf{if}_L\ b_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2]\!])$. By unfolding and IH-(2) on b_0, IH-(1) on t_1, t_2.
- Case $t = \mathbf{while}\ b\ \mathbf{do}\ t_0\ \mathbf{done}$.
  Hypotheses: IH for b and t_0.
  Goal: $[\![\mathrm{Subst}(M, \mathbf{while}\ b\ \mathbf{do}\ t_0\ \mathbf{done})]\!] \equiv \mathrm{Subst}([\![M]\!], [\![\mathbf{while}\ b\ \mathbf{do}\ t_0\ \mathbf{done}]\!])$. By unfolding and IH.
- Case $t = L : \{\ t_0\ \}$.
  Hypotheses: IH for t_0.
  Goal: $[\![\mathrm{Subst}(M, L : \{\ t_0\ \})]\!] \equiv \mathrm{Subst}([\![M]\!], [\![L : \{\ t_0\ \}]\!])$. By unfolding and IH.
- Case $t = \mathbf{exit}\ L\ t_0$.
  Hypotheses: IH for t_0.
  Goal: $[\![\mathrm{Subst}(M, \mathbf{exit}\ L\ t_0)]\!] \equiv \mathrm{Subst}([\![M]\!], [\![\mathbf{exit}\ L\ t_0]\!])$. By unfolding and IH.
- Case $t = \mathbf{next}\ L$.
  Hypotheses: none.
  Goal: $[\![\mathrm{Subst}(M, \mathbf{next}\ L)]\!] \equiv \mathrm{Subst}([\![M]\!], [\![\mathbf{next}\ L]\!])$.
  Immediate by unfolding.

- **Clause (2): BBEs.**
  - Case $b = (t\ \mathbf{is}\ p)$ (representative detailed case).
    Hypotheses: $\mathrm{fv}(\mathsf{u}') = \varnothing$ and $\mathrm{fv}(\mathsf{u}) \subseteq (\mathrm{dom}(M) \cup \mathrm{bv}((t\ \mathbf{is}\ p)))$.
    Goal: $[\![\mathrm{Subst}(M, (t\ \mathbf{is}\ p))]\!]_{\mathsf{u}'}^{\mathrm{Subst}([\![M \setminus \mathrm{bv}((t\ \mathbf{is}\ p))]\!], \mathsf{u})} \equiv \mathrm{Subst}([\![M]\!], [\![(t\ \mathbf{is}\ p)]\!]_{\mathsf{u}'}^{\mathsf{u}})$.
    Unfolding translation of **is**reduces the goal to the corresponding congruent properties for the strict subterms t and p. Apply IH-(1) to t and IH-(3) to p; side conditions are inherited from the hypothesis and $\mathrm{bv}((t\ \mathbf{is}\ p)) = \mathrm{bv}(p)$. Conclude by congruence compatibility of `let` and pattern-translation contexts.
  - Case $b = \mathbf{if}^{\mathrm{bbe}}\ b_0\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2$.
    Hypotheses: $\mathrm{fv}(\mathsf{u}') = \varnothing$ and $\mathrm{fv}(\mathsf{u}) \subseteq (\mathrm{dom}(M) \cup \mathrm{bv}(\mathbf{if}^{\mathrm{bbe}}\ b_0\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2))$, plus IH on b_0, b_1, b_2.
    Goal: $[\![\mathrm{Subst}(M, \mathbf{if}^{\mathrm{bbe}}\ b_0\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2)]\!]_{\mathsf{u}'}^{\mathrm{Subst}([\![M \setminus \mathrm{bv}(\mathbf{if}^{\mathrm{bbe}}\ b_0\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2)]\!], \mathsf{u})} \equiv \mathrm{Subst}([\![M]\!], [\![\mathbf{if}^{\mathrm{bbe}}\ b_0\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2]\!]_{\mathsf{u}'}^{\mathsf{u}})$. By unfolding and IH on each branch.
- **Clause (3): patterns.**
  - Case $p = x^?$.

Hypotheses: $\mathrm{fv}(\mathsf{u}') = \varnothing$ and $\mathrm{fv}(\mathsf{u}) \subseteq (\mathrm{dom}(M) \cup \mathrm{bv}(x^?))$.
Goal: $[\![\mathsf{y} \blacktriangleright \mathrm{Subst}(M, x^?)]\!]_{\mathsf{u}'}^{\mathrm{Subst}([\![M \setminus \mathrm{bv}(x^?)]\!], \mathsf{u})} \equiv \mathrm{Subst}([\![M]\!], [\![\mathsf{y} \blacktriangleright x^?]\!]_{\mathsf{u}'}^{\mathsf{u}})$. Immediate by unfolding.

– Case $p = (p_0 \ \mathbf{when}\ b)$ (representative detailed case).
Hypotheses: $\mathrm{fv}(\mathsf{u}') = \varnothing$ and $\mathrm{fv}(\mathsf{u}) \subseteq (\mathrm{dom}(M) \cup \mathrm{bv}((p_0 \ \mathbf{when}\ b)))$.
Goal: $[\![\mathsf{y} \blacktriangleright \mathrm{Subst}(M, (p_0 \ \mathbf{when}\ b))]\!]_{\mathsf{u}'}^{\mathrm{Subst}([\![M \setminus \mathrm{bv}((p_0 \ \mathbf{when}\ b))]\!], \mathsf{u})} \equiv \mathrm{Subst}([\![M]\!], [\![\mathsf{y} \blacktriangleright (p_0 \ \mathbf{when}\ b)]\!]_{\mathsf{u}'}^{\mathsf{u}})$.
Unfolding translation of PAT-WHEN reduces to strict subterms p_0 and b. Apply IH-(3) on p_0 and IH-(2) on b; side conditions follow from $\mathrm{bv}((p_0 \ \mathbf{when}\ b)) = \mathrm{bv}(p_0) \cup \mathrm{bv}(b)$. Conclude by context congruence.

– Case $p = \mathrm{Some}(p_1, ..., p_n)$.
Hypotheses: $\mathrm{fv}(\mathsf{u}') = \varnothing$ and $\mathrm{fv}(\mathsf{u}) \subseteq (\mathrm{dom}(M) \cup \mathrm{bv}(\mathrm{Some}(p_1, ..., p_n)))$, plus IH on each p_i.
Goal: $[\![\mathsf{y} \blacktriangleright \mathrm{Subst}(M, \mathrm{Some}(p_1, ..., p_n))]\!]_{\mathsf{u}'}^{\mathrm{Subst}([\![M \setminus \mathrm{bv}(\mathrm{Some}(p_1, ..., p_n))]\!], \mathsf{u})} \equiv \mathrm{Subst}([\![M]\!], [\![\mathsf{y} \blacktriangleright \mathrm{Some}(p_1, ..., p_n)]\!]_{\mathsf{u}'}^{\mathsf{u}})$. By unfolding and IH on every p_i.

In every case, recursive calls are only on strict subterms, hence on strictly smaller ASTs, exactly as required. □

Remark. Lemma 10 is a direct corollary of Lemma 11 when choosing an identity substitution.

To exploit the substitution lemmas, we need to justify inclusion results w.r.t. set of free variables. The following lemmas provide the necessary results.

LEMMA 12. *The following statements hold:*

$$\begin{array}{lcl} E \vdash_{\mathrm{trm}} t : T & \Rightarrow & \mathrm{fv}(t) \subseteq \mathrm{dom}(E) \\ E \vdash_{\mathrm{bbe}} b \rightsquigarrow B & \Rightarrow & \mathrm{fv}(b) \subseteq \mathrm{dom}(E) \\ E \vdash_{\mathrm{pat}} p : T \rightsquigarrow B & \Rightarrow & \mathrm{fv}(p) \subseteq \mathrm{dom}(E) \end{array}$$

LEMMA 13. *The following statements holds:*

(1) $(\mathrm{dom}(M) \cap \mathrm{fv}(\mathsf{u}) = \varnothing) \wedge ([\![\mathrm{Subst}(M, b)]\!]_{\mathsf{u}'}^{\mathsf{u}}, \Downarrow_{\mathrm{ml}} \mathsf{q}) \wedge (\mathrm{fv}(\mathsf{u}') = \varnothing) \wedge (\mathrm{fv}(\mathsf{u}) \subseteq \mathrm{bv}(b)) \Rightarrow (\mathrm{Subst}([\![M]\!], [\![b]\!]_{\mathsf{u}'}^{\mathsf{u}}) \Downarrow_{\mathrm{ml}} \mathsf{q})$

(2) $(\mathrm{dom}(M) \cap \mathrm{fv}(\mathsf{u}) = \varnothing) \wedge ([\![\mathsf{y} \blacktriangleright \mathit{SubstMp}]\!]_{\mathsf{u}'}^{\mathsf{u}}, \Downarrow_{\mathrm{ml}} \mathsf{q}) \wedge (\mathrm{fv}(\mathsf{u}') = \varnothing) \wedge (\mathrm{fv}(\mathsf{u}) \subseteq \mathrm{bv}(p)) \Rightarrow (\mathrm{Subst}([\![M]\!], [\![\mathsf{y}]\!]_{\mathsf{u}}^{p}\mathsf{u}') \Downarrow_{\mathrm{ml}} \mathsf{q})$

PROOF. Direct corollary of Lemma 10. □

G TYPING AND SEMANTICS OF THE MINIMAL LANGUAGE

$$\textsc{eval-trm-exit-1}\ \frac{t \Downarrow_{\text{trm}} \text{Res}\ v}{\textbf{exit}\ L\ t\ \Downarrow_{\text{trm}}\ \text{Abort}\ (\text{Exit}\ L\ v)} \qquad \textsc{eval-trm-exit-2}\ \frac{t \Downarrow_{\text{trm}} \text{Abort}\ a}{\textbf{exit}\ L\ t\ \Downarrow_{\text{trm}}\ \text{Abort}\ a}$$

$$\textsc{eval-trm-return-1}\ \frac{t \Downarrow_{\text{trm}} \text{Res}\ v}{\textbf{return}\ L\ t\ \Downarrow_{\text{trm}}\ \text{Abort}\ (\text{Return}\ L\ v)} \qquad \textsc{eval-trm-return-2}\ \frac{t \Downarrow_{\text{trm}} \text{Abort}\ a}{\textbf{return}\ L\ t\ \Downarrow_{\text{trm}}\ \text{Abort}\ a}$$

$$\textsc{eval-trm-break}\ \frac{}{\textbf{break}\ L\ \Downarrow_{\text{trm}}\ \text{Abort}\ (\text{Break}\ L)} \qquad \textsc{eval-trm-continue}\ \frac{}{\textbf{continue}\ L\ \Downarrow_{\text{trm}}\ \text{Abort}\ (\text{Continue}\ L)}$$

$$\textsc{eval-trm-next}\ \frac{}{\textbf{next}\ L\ \Downarrow_{\text{trm}}\ \text{Abort}\ (\text{Next}\ L)} \qquad \textsc{eval-trm-block-1}\ \frac{t \Downarrow_{\text{trm}} \text{Res}\ v}{L : \{\, t \,\}\ \Downarrow_{\text{trm}}\ \text{Res}\ v} \qquad \textsc{eval-trm-block-2}\ \frac{t \Downarrow_{\text{trm}} \text{Abort}\ (\text{Exit}\ L\ v)}{L : \{\, t \,\}\ \Downarrow_{\text{trm}}\ \text{Res}\ v}$$

$$\textsc{eval-trm-block-3}\ \frac{t \Downarrow_{\text{trm}} \text{Abort}\ a \qquad \forall v.\ a \neq \text{Exit}\ L\ v}{L : \{\, t \,\}\ \Downarrow_{\text{trm}}\ \text{Abort}\ a} \qquad \textsc{eval-trm-if-label-1}\ \frac{b_0 \Downarrow_{\text{bbe}} \text{Mismatch} \qquad t_2 \Downarrow_{\text{trm}} o}{\textbf{if}_L\ b_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \Downarrow_{\text{trm}}\ o}$$

$$\textsc{eval-trm-if-label-2}\ \frac{b_0 \Downarrow_{\text{bbe}} \text{Match}\ M_0 \qquad \text{Subst}(M_0, t_1) \Downarrow_{\text{trm}} \text{Abort}\ (\text{Next}\ L) \qquad t_2 \Downarrow_{\text{trm}} o}{\textbf{if}_L\ b_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \Downarrow_{\text{trm}}\ o}$$

$$\textsc{eval-trm-if-label-3}\ \frac{b_0 \Downarrow_{\text{bbe}} \text{Match}\ M_0 \qquad \text{Subst}(M_0, t_1) \Downarrow_{\text{trm}} o \qquad o \neq \text{Abort}\ (\text{Next}\ L)}{\textbf{if}_L\ b_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \Downarrow_{\text{trm}}\ o}$$

$$\textsc{eval-trm-switch-fail}\ \frac{\forall i \in [1, n].\ b_i \Downarrow_{\text{bbe}} \text{Mismatch}}{\textbf{switch}_L \mid (\textbf{case}\ b_1\ \textbf{then}\ t_1) \mid \ldots \mid (\textbf{case}\ b_n\ \textbf{then}\ t_n)\ \Downarrow_{\text{trm}}\ \text{Exn NoBranch}}$$

$$\textsc{eval-trm-match-fail}\ \frac{\forall i \in [1, n].\ v_0 \triangleright p_i \Downarrow_{\text{pat}} \text{Mismatch}}{\textbf{match}_L\ v_0\ \textbf{with} \mid p_1 \to t_1 \mid \ldots \mid p_n \to t_n\ \Downarrow_{\text{trm}}\ \text{Exn NoBranch}}$$

Fig. 21. Selected evaluation rules for constructs involving labels, using the generalized judgment $t \Downarrow_{\text{trm}} o$, where o denotes an evaluation outcome. Numerous rules, not shown, are directly adapted from the evaluation rules from Figure 7 which defined the judgment $t \Downarrow_{\text{trm}} v$.

$$\textsc{typ-val-int}\ \frac{}{\vdash_{\text{val}} \bar{n} : \text{int}} \qquad \textsc{typ-val-constr}\ \frac{(C : (T_1, ..., T_n) \ \to\ T) \in \Sigma \qquad \forall i \in [1, n].\ \vdash_{\text{val}} v_i : T_i}{\vdash_{\text{val}} \bar{C}\ (v_1, ..., v_n) : T} \qquad \textsc{typ-val-prim}\ \frac{(\pi : T) \in \Sigma}{E \vdash_{\text{val}} \bar{\pi} : T}$$

$$\textsc{typ-val-fun}\ \frac{x_1 : T_1, ..., x_n : T_n;\ \varnothing \vdash_{\text{trm}} t : T}{\vdash_{\text{val}} \bar{\lambda}(x_1, ..., x_n).t : (T_1, ..., T_n) \ \to\ T}$$

$$\textsc{typ-trm-if-res-label}\ \frac{\vdash_{\text{res}} r \rightsquigarrow B \qquad E, B;\ F, (\text{LblBranch}\ L) \vdash_{\text{trm}} t_1 : T \qquad E;\ F \vdash_{\text{trm}} t_2 : T}{E;\ F \vdash_{\text{trm}} \textbf{if}_L\ r\ \textbf{then}\ t_1\ \textbf{else}\ t_2 : T}$$

$$\textsc{typ-trm-ifthen-label}\ \frac{E;\ F, (\text{LblBranch}\ L) \vdash_{\text{trm}} t_1 : T \qquad E;\ F \vdash_{\text{trm}} t_2 : T}{E;\ F \vdash_{\text{trm}} \textbf{ifthen}_L\ t_1\ \textbf{else}\ t_2 : T}$$

$$\textsc{typ-bbe-if-res}\ \frac{\vdash_{\text{map}} M_0 : B_0 \qquad E, B_0 \vdash_{\text{bbe}} b_1 \rightsquigarrow B_1 \qquad E \vdash_{\text{bbe}} b_2 \rightsquigarrow B_2 \qquad \text{dom}(M_0) \cap \text{pv}(b_1) = \varnothing}{E \vdash_{\text{bbe}} \textbf{if}^{\text{bbe}}\ M_0\ \textbf{then}\ b_1\ \textbf{else}\ b_2 \rightsquigarrow ((B_0 \uplus B_1) \cap B_2)}$$

$$\textsc{typ-bbe-if-res-res}\ \frac{\vdash_{\text{map}} M_0 : B_0 \qquad \vdash_{\text{res}} r_1 : B_1 \qquad E \vdash_{\text{bbe}} b_2 \rightsquigarrow B_2 \qquad \left(\forall M_1.\ r_1 = \text{Match}\ M_1\ \Rightarrow\ M_0 \# M_1\right)}{E \vdash_{\text{bbe}} \textbf{if}^{\text{bbe}}\ M_0\ \textbf{then}\ r_1\ \textbf{else}\ b_2 \rightsquigarrow ((B_0 \uplus B_1) \cap B_2)}$$

$$\textsc{typ-bbe-pat}\ \frac{\vdash_{\text{val}} v : T \qquad E \vdash_{\text{pat}} p : T \rightsquigarrow B}{E \vdash_{\text{bbe}} v \triangleright p \ \rightsquigarrow\ B} \qquad \textsc{typ-bbe-weaken}\ \frac{E \vdash_{\text{bbe}} b \rightsquigarrow B \qquad B' \subseteq B}{E \vdash_{\text{bbe}} b \rightsquigarrow B'} \qquad \textsc{typ-out-val}\ \frac{\vdash_{\text{val}} v : T}{F \vdash_{\text{out}} \text{Res}\ v : T}$$

$$\textsc{typ-out-abt}\ \frac{F \vdash_{\text{abt}} a}{F \vdash_{\text{out}} \text{Abort}\ a : T} \qquad \textsc{typ-abt-exit}\ \frac{(\text{LblBlock}\ L\ :\ T) \in F \qquad \vdash_{\text{val}} v : T}{F \vdash_{\text{abt}} \text{Exit}\ L\ v}$$

$$\textsc{typ-abt-return}\ \frac{(\text{LblFun}\ L\ :\ T) \in F \qquad \vdash_{\text{val}} v : T}{F \vdash_{\text{abt}} \text{Return}\ L\ v} \qquad \textsc{typ-abt-break}\ \frac{(\text{LblLoop}\ L) \in F}{F \vdash_{\text{abt}} \text{Break}\ L} \qquad \textsc{typ-abt-continue}\ \frac{(\text{LblLoop}\ L) \in F}{F \vdash_{\text{abt}} \text{Continue}\ L}$$

$$\textsc{typ-abt-next}\ \frac{(\text{LblBranch}\ L) \in F}{F \vdash_{\text{abt}} \text{Next}\ L} \qquad \textsc{typ-map}\ \frac{\text{dom}(B) \subseteq \text{dom}(M) \qquad \forall\ (x, T) \in B.\ \vdash_{\text{val}} M[x] : T}{\vdash_{\text{map}} M : B} \qquad \textsc{typ-res-match}\ \frac{\vdash_{\text{map}} M : B}{\vdash_{\text{res}} \text{Match}\ M : B}$$

$$\textsc{typ-res-mismatch}\ \frac{}{\vdash_{\text{res}} \text{Mismatch} : B}$$

Fig. 22. Additional typing rules for establishing type preservation on the minimal language. The new judgments are: $\vdash_{\text{val}} v : T$ and $F \vdash_{\text{out}} o : T$ and $F \vdash_{\text{abt}} a : T$ and $E \vdash_{\text{map}} M : B$ and $\vdash_{\text{res}} r : T$.

Core-ML values

w :=	n	integer
	L	labels (e.g., strings or polymorphic variants)
	C $(w_1, ..., w_n)$	constructor value, including exceptions
	fix f $(x_1, ..., x_n) \to u$	n-ary recursive function closure

Core-ML terms

u :=	x	variable
	C $(u_1, ..., u_n)$	constructor application
	fix f $(x_1, ..., x_n) \to u$	n-ary recursive function definition
	u_0 $(u_1, ..., u_n)$	n-ary function application
	let x = u_1 in u_2	simple let-binding
	match u_0 with | C $(x_1, ..., x_n) \to u_1$ | _ $\to u_2$	simple pattern matching
	try u_0 with | C $(x_1, ..., x_n) \to u_1$	exception handling
	raise u	exception raising

Core-ML exceptions defined for use in the translation

e :=	Exit (L, w)	exception for encoding exit blocks
	Next L	exception for encoding labeled conditionals

Fig. 23. Grammar of Core-ML programs, target of the compilation scheme

ML term outcome

$$\begin{array}{lll} \mathtt{q} := & \mid \ \mathtt{Result\ w} & \text{regular outcome} \\ & \mid \ \mathtt{Exn\ w} & \text{exception outcome} \end{array}$$

ML-INT
$$\frac{}{\mathtt{n} \Downarrow_{\mathrm{ml}} \mathtt{Result\ n}}$$

ML-CONSTR
$$\frac{\forall i.\quad \mathtt{u_i} \Downarrow_{\mathrm{ml}} \mathtt{Result\ w_i}}{\mathtt{C\ (u_1, ..., u_n)} \Downarrow_{\mathrm{ml}} \mathtt{Result\ (C\ (w_1, ..., w_n))}}$$

ML-FIX
$$\frac{}{\mathtt{fix\ f\ (x_1, ..., x_n) \to u} \Downarrow_{\mathrm{ml}} \mathtt{Result\ (fix\ f\ (x_1, ..., x_n) \to u)}}$$

ML-BETA
$$\frac{\mathtt{u_0} \Downarrow_{\mathrm{ml}} \mathtt{Result\ w_0} \qquad \mathtt{w_0 = (fix\ f\ (x_1, ..., x_n) \to u)} \qquad \left(\forall i.\quad \mathtt{u_i} \Downarrow_{\mathrm{ml}} \mathtt{Result\ w_i}\right) \qquad \mathrm{Subst}(\{\mathtt{f} \mapsto \mathtt{w_0}, \mathtt{x_1} \mapsto \mathtt{w_1}, ...\mathtt{x_n} \mapsto \mathtt{w_n}\}, \mathtt{u}) \Downarrow_{\mathrm{ml}} \mathtt{q}}{\mathtt{u_0\ (u_1, ..., u_n)} \Downarrow_{\mathrm{ml}} \mathtt{q}}$$

ML-LET-1
$$\frac{\mathtt{u_1} \Downarrow_{\mathrm{ml}} \mathtt{Result\ w_1} \qquad \mathrm{Subst}(\{\mathtt{x} \mapsto \mathtt{w_1}\}, \mathtt{u_2}) \Downarrow_{\mathrm{ml}} \mathtt{q}}{\mathtt{let\ x = u_1\ in\ u_2} \Downarrow_{\mathrm{ml}} \mathtt{q}}$$

ML-LET-2
$$\frac{\mathtt{u_1} \Downarrow_{\mathrm{ml}} \mathtt{Exn\ w}}{\mathtt{let\ x = u_1\ in\ u_2} \Downarrow_{\mathrm{ml}} \mathtt{Exn\ w}}$$

ML-MATCH-1
$$\frac{\mathtt{u_0} \Downarrow_{\mathrm{ml}} \mathtt{Result\ w_0} \qquad \mathtt{w_0 = C\ (w_1, ..., w_n)} \qquad \mathrm{Subst}(\{\mathtt{x_1} \mapsto \mathtt{w_1}, ..., \mathtt{x_n} \mapsto \mathtt{w_n}\}, \mathtt{u_1}) \Downarrow_{\mathrm{ml}} \mathtt{q}}{\mathtt{match\ u_0\ with\ \mid\ C\ (x_1, ..., x_n) \to u_1\ \mid\ \_ \to u_2} \Downarrow_{\mathrm{ml}} \mathtt{q}}$$

ML-MATCH-2
$$\frac{\mathtt{u_0} \Downarrow_{\mathrm{ml}} \mathtt{Result\ w_0} \qquad \left(\forall \mathtt{w_1..w_n}.\ \mathtt{w_0} \neq \mathtt{C\ (w_1, ..., w_n)}\right) \qquad \mathtt{u_2} \Downarrow_{\mathrm{ml}} \mathtt{q}}{\mathtt{match\ u_0\ with\ \mid\ C\ (x_1, ..., x_n) \to u_1\ \mid\ \_ \to u_2} \Downarrow_{\mathrm{ml}} \mathtt{q}}$$

ML-MATCH-3
$$\frac{\mathtt{u_0} \Downarrow_{\mathrm{ml}} \mathtt{Exn\ w}}{\mathtt{match\ u_0\ with\ \mid\ C\ (x_1, ..., x_n) \to u_1\ \mid\ \_ \to u_2} \Downarrow_{\mathrm{ml}} \mathtt{Exn\ w}}$$

ML-TRY-WITH-1
$$\frac{\mathtt{u_0} \Downarrow_{\mathrm{ml}} \mathtt{Exn\ C\ (w_1, ..., w_n)} \qquad \mathrm{Subst}(\{\mathtt{x_1} \mapsto \mathtt{w_1}, ..., \mathtt{x_n} \mapsto \mathtt{w_n}\}, \mathtt{u_1}) \Downarrow_{\mathrm{ml}} \mathtt{q}}{\mathtt{try\ u_0\ with\ \mid\ C\ (x_1, ..., x_n) \to u_1} \Downarrow_{\mathrm{ml}} \mathtt{q}}$$

ML-TRY-WITH-2
$$\frac{\mathtt{u_0} \Downarrow_{\mathrm{ml}} \mathtt{q} \qquad \forall \mathtt{w_1..w_n}.\ \mathtt{q} \neq \mathtt{Exn\ C\ (w_1, ..., w_n)}}{\mathtt{try\ u_0\ with\ \mid\ C\ (x_1, ..., x_n) \to u_1} \Downarrow_{\mathrm{ml}} \mathtt{q}}$$

ML-RAISE
$$\frac{\mathtt{u} \Downarrow_{\mathrm{ml}} \mathtt{Result\ w}}{\mathtt{raise\,u} \Downarrow_{\mathrm{ml}} \mathtt{Exn\ w}}$$

Fig. 24. Semantics of Core-ML programs, target of the compilation scheme. The usual rules for propagating exceptions are not shown, except for the examples of ML-LET-2 and ML-MATCH-3.

$$
\begin{array}{lcl}
\mathrm{pv}(t) & \equiv & \varnothing \\
\mathrm{pv}(t\ \textbf{is}\ p) & \equiv & \mathrm{pv}(p) \\
\mathrm{pv}(\textbf{if}^{\mathrm{bbe}}\ b_0\ \textbf{then}\ b_1\ \textbf{else}\ b_2) & \equiv & \mathrm{pv}(b_0) \cup \mathrm{pv}(b_1) \cup \mathrm{pv}(b_2) \\
\mathrm{pv}(b_1\ \textbf{and}\ b_2) & \equiv & \mathrm{pv}(b_1) \cup \mathrm{pv}(b_2) \\
\mathrm{pv}(b_1\ \textbf{or}\ b_2) & \equiv & \mathrm{pv}(b_1) \cup \mathrm{pv}(b_2) \\
\mathrm{pv}(\textbf{not}\ b) & \equiv & \mathrm{pv}(b) \\
\mathrm{pv}(x^?) & \equiv & \{x\} \\
\mathrm{pv}(\_) & \equiv & \varnothing \\
\mathrm{pv}(n) & \equiv & \varnothing \\
\mathrm{pv}(p_1 \mid p_2) & \equiv & \mathrm{pv}(p_1) \cup \mathrm{pv}(p_2) \\
\mathrm{pv}(p_1\ \&\ p_2) & \equiv & \mathrm{pv}(p_1) \cup \mathrm{pv}(p_2) \\
\mathrm{pv}(\neg\ p) & \equiv & \mathrm{pv}(p) \\
\mathrm{pv}(p\ \textbf{as}\ x^?) & \equiv & \mathrm{pv}(p) \cup \{x\} \\
\mathrm{pv}(p\ \textbf{when}\ b) & \equiv & \mathrm{pv}(p) \cup \mathrm{pv}(b) \\
\mathrm{pv}(g) & \equiv & \varnothing \\
\mathrm{pv}(t_0\ (p_1, ..., p_n)) & \equiv & \bigcup_i \mathrm{pv}(p_i)
\end{array}
$$

Fig. 25. Definition of the pv() operations for computing pattern variables

$$
\begin{array}{lcl}
\mathrm{fv}(x) & \equiv & \{x\} \\
\mathrm{fv}(n) & \equiv & \varnothing \\
\mathrm{fv}(\textbf{let}\ x = t_1\ \textbf{in}\ t_2) & \equiv & \mathrm{fv}(t_1) \cup (\mathrm{fv}(t_2) \setminus \{x\}) \\
\mathrm{fv}(\lambda(x_1, ..., x_n).\, t) & \equiv & \mathrm{fv}(t) \setminus \{x_1, ..., x_n\} \\
\mathrm{fv}(t_0\ (t_1, ..., t_n)) & \equiv & \bigcup_{i \in [0,n]} \mathrm{fv}(t_i) \\
\mathrm{fv}(\textbf{if}_L\ b_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2) & \equiv & \mathrm{fv}(b_0) \cup (\mathrm{fv}(t_1) \setminus \mathrm{bv}(b_0)) \cup \mathrm{fv}(t_2) \\
\mathrm{fv}(\textbf{while}\ b\ \textbf{do}\ t\ \textbf{done}) & \equiv & \mathrm{fv}(b) \cup (\mathrm{fv}(t) \setminus \mathrm{bv}(b)) \\
\mathrm{fv}(L : \{\ t\ \}) & \equiv & \mathrm{fv}(t) \\
\mathrm{fv}(\textbf{exit}\ L\ t) & \equiv & \mathrm{fv}(t) \\
\mathrm{fv}(\textbf{next}\ L) & \equiv & \varnothing \\
\mathrm{fv}(t\ \textbf{is}\ p) & \equiv & \mathrm{fv}(t) \cup \mathrm{fv}(p) \\
\mathrm{fv}(\textbf{if}^{\mathrm{bbe}}\ b_0\ \textbf{then}\ b_1\ \textbf{else}\ b_2) & \equiv & \mathrm{fv}(b_0) \cup (\mathrm{fv}(b_1) \setminus \mathrm{bv}(b_0)) \cup \mathrm{fv}(b_2) \\
\mathrm{fv}(x^?) & \equiv & \varnothing \\
\mathrm{fv}(\_) & \equiv & \varnothing \\
\mathrm{fv}(p\ \textbf{when}\ b) & \equiv & \mathrm{fv}(p) \cup (\mathrm{fv}(b) \setminus \mathrm{bv}(p)) \\
\mathrm{fv}(t_0\ (p_1, ..., p_n)) & \equiv & \mathrm{fv}(f) \cup \bigcup_i (\mathrm{fv}(p_i) \setminus \bigcup_{1 \le j < i} (\mathrm{bv}(p_i)))
\end{array}
$$

Fig. 26. Definition of fv() for computing free variables, on the minimal language

$$
\begin{array}{lcl}
\mathrm{bv}(\mathrm{false}) & \equiv & \mathbb{V} \\
\quad \textit{where false is a BBE} & & \quad \textit{where and } \mathbb{V} \textit{ is the set of all variables} \\
\quad \text{Recall that false is desugared as false } \textbf{is} \text{ true} & & \\
\mathrm{bv}(t \textbf{ is } p) & \equiv & \mathrm{bv}(p) \\
\mathrm{bv}(\textbf{if}^{\mathrm{bbe}}\ b_0 \textbf{ then } b_1 \textbf{ else } b_2) & \equiv & (\mathrm{bv}(b_0) \cup \mathrm{bv}(b_1)) \cap \mathrm{bv}(b_2) \\
\mathrm{bv}(x^?) & \equiv & \{x\} \\
\mathrm{bv}(\_) & \equiv & \varnothing \\
\mathrm{bv}(p \textbf{ when } b) & \equiv & \mathrm{bv}(p) \cup \mathrm{bv}(b) \\
\mathrm{bv}(\mathrm{Some}(p_1, ..., p_n)) & \equiv & \bigcup_i \mathrm{bv}(p_i)
\end{array}
$$

Fig. 27. Definition of bv() for computing bound variables, on the minimal language,

$$
\begin{array}{lcl}
[\![\bar{n}]\!] & \equiv & \texttt{n} \\
[\![\bar{\lambda}(x_1, ..., x_n).t]\!] & \equiv & \texttt{fun (x}_1\texttt{, ..., x}_\texttt{n}\texttt{) } \rightarrow \texttt{ } [\![t]\!] \\
[\![\bar{C}\ (v_1, ..., x_n)]\!] & \equiv & \texttt{C ([[}v_1\texttt{]], ..., [[}v_n\texttt{]])} \\
 & & \\
[\![\mathrm{Res}\ v]\!] & \equiv & \texttt{Result } [\![v]\!] \\
[\![\mathrm{Abort}\ (\mathrm{Exit}\ L\ v)]\!] & \equiv & \texttt{Exn (Exit (L, } [\![v]\!]\texttt{))} \\
[\![\mathrm{Abort}\ (\mathrm{Next}\ L)]\!] & \equiv & \texttt{Exn (Next L)}
\end{array}
$$

Fig. 28. Compilation scheme for values and outcomes

TYP-TRM-VAR
$$\frac{(x : T) \in E}{E;\, F \vdash_{\text{trm}} x : T}$$

TYP-TRM-INT
$$\frac{}{E;\, F \vdash_{\text{trm}} n : \mathsf{int}}$$

TYP-TRM-LET-VAR
$$\frac{E;\, F \vdash_{\text{trm}} t_1 : T_1 \qquad E,\, (x : T_1);\, F \vdash_{\text{trm}} t_2 : T_2}{E;\, F \vdash_{\text{trm}} \textbf{let}\ x = t_1\ \textbf{in}\ t_2 : T_2}$$

TYP-TRM-FUN
$$\frac{E,\, x_1 : T_1, ..., x_n : T_n;\, \varnothing \vdash_{\text{trm}} t : T}{E;\, F \vdash_{\text{trm}} \lambda(x_1, ..., x_n).\, t : (T_1, ..., T_n) \ \to\ T}$$

TYP-TRM-APP
$$\frac{E;\, F \vdash_{\text{trm}} t_0 : (T_1, ..., T_n) \ \to\ T \qquad \forall i.\quad E;\, F \vdash_{\text{trm}} t_i : T_i}{E;\, F \vdash_{\text{trm}} t_0\ (t_1, ..., t_n) : T}$$

TYP-TRM-IF-LABEL
$$\frac{\begin{array}{c} E \vdash_{\text{bbe}} b \rightsquigarrow B \\ E,\, B;\, F,\, (\text{LblBranch}\ L) \vdash_{\text{trm}} t_1 : T \\ E;\, F \vdash_{\text{trm}} t_2 : T \end{array}}{E;\, F \vdash_{\text{trm}} \textbf{if}_L\ b\ \textbf{then}\ t_1\ \textbf{else}\ t_2 : T}$$

TYP-TRM-WHILE
$$\frac{E \vdash_{\text{bbe}} b \rightsquigarrow B \qquad E,\, B;\, \varnothing \vdash_{\text{trm}} t : \mathsf{unit}}{E;\, F \vdash_{\text{trm}} \textbf{while}\ b\ \textbf{do}\ t\ \textbf{done} : \mathsf{unit}}$$

TYP-TRM-BLOCK
$$\frac{E;\, F,\, (\text{LblBlock}\ L\ :\ T) \vdash_{\text{trm}} t : T}{E;\, F \vdash_{\text{trm}} (L : \{\, t \,\}) : T}$$

TYP-TRM-EXIT
$$\frac{(\text{LblBlock}\ L\ :\ T) \in F \qquad E;\, F \vdash_{\text{trm}} t : T}{E;\, F \vdash_{\text{trm}} \textbf{exit}\ L\ t : T'}$$

TYP-TRM-NEXT
$$\frac{(\text{LblBranch}\ L) \in F}{E;\, F \vdash_{\text{trm}} \textbf{next}\ L : T'}$$

TYP-BBE-IS
$$\frac{E;\, \varnothing \vdash_{\text{trm}} t : T \qquad E \vdash_{\text{pat}} p : T \rightsquigarrow B}{E \vdash_{\text{bbe}} (t\ \textbf{is}\ p) \ \rightsquigarrow\ B}$$

TYP-BBE-IF
$$\frac{\begin{array}{cc} E \vdash_{\text{bbe}} b_0 \rightsquigarrow B_0 & E,\, B_0 \vdash_{\text{bbe}} b_1 \rightsquigarrow B_1 \\ E \vdash_{\text{bbe}} b_2 \rightsquigarrow B_2 & \text{pv}(b_0) \cap \text{pv}(b_1) = \varnothing \end{array}}{E \vdash_{\text{bbe}} \textbf{if}^{\text{bbe}}\ b_0\ \textbf{then}\ b_1\ \textbf{else}\ b_2 \ \rightsquigarrow\ (B_0 \uplus B_1) \cap B_2}$$

TYP-PAT-VAR
$$\frac{}{E \vdash_{\text{pat}} x^{?} \ :\ T \ \rightsquigarrow\ \{x : T\}}$$

TYP-PAT-WHEN
$$\frac{\begin{array}{cc} & E \vdash_{\text{pat}} p : T \rightsquigarrow B_1 \\ E,\, B_1 \vdash_{\text{bbe}} b \rightsquigarrow B_2 & \text{pv}(p) \cap \text{pv}(b) = \varnothing \end{array}}{E \vdash_{\text{pat}} p\ \textbf{when}\ b\ :\ T\ \rightsquigarrow\ B_1 \uplus B_2}$$

TYP-PAT-SOME
$$\frac{\begin{array}{c} \forall i \in [1, n].\quad E,\, (\biguplus_{1 \le j < i} B_j) \vdash_{\text{pat}} p_i : T_i \rightsquigarrow B_i \\ \forall i, j \in [1, n].\ i \neq j \Rightarrow \text{pv}(p_i) \cap \text{pv}(p_j) = \varnothing \end{array}}{E \vdash_{\text{pat}} \text{Some}(p_1, ..., p_n) \ :\ (T_1, ..., T_n)\ \text{option} \ \rightsquigarrow\ \biguplus_{1 \le j < i} B_j}$$

Fig. 29. Typing rules for the minimal language

$$\textsc{eval-trm-int}\quad \frac{}{n \Downarrow_{\text{trm}} \text{Res } \bar{n}}$$

$$\textsc{eval-trm-let}\quad \frac{t_1 \Downarrow_{\text{trm}} \text{Res } v_1 \qquad \text{Subst}(\{x \mapsto v_1\}, t_2) \Downarrow_{\text{trm}} o}{\textbf{let } x = t_1 \textbf{ in } t_2 \ \Downarrow_{\text{trm}} \ o}$$

$$\textsc{eval-trm-fun}\quad \frac{}{\lambda(x_1, ..., x_n).\, t \Downarrow_{\text{trm}} \text{Res } (\bar{\lambda}(x_1, ..., x_n).t)}$$

$$\textsc{eval-trm-beta}\quad \frac{t_0 \Downarrow_{\text{trm}} \text{Res } (\bar{\lambda}(x_1, ..., x_n).t) \quad \left(\forall i \in [1, n].\quad t_i \Downarrow_{\text{trm}} \text{Res } v_i\right) \quad \text{Subst}(\{x_1 \mapsto v_1, ..., x_n \mapsto v_n\}, t) \Downarrow_{\text{trm}} o}{t_0\ (t_1, ..., t_n) \ \Downarrow_{\text{trm}} \ o}$$

$$\textsc{eval-trm-if-label-1}\quad \frac{b_0 \Downarrow_{\text{bbe}} \text{Mismatch} \qquad t_2 \Downarrow_{\text{trm}} o}{\textbf{if}_L\ b_0 \textbf{ then } t_1 \textbf{ else } t_2 \ \Downarrow_{\text{trm}} \ o}$$

$$\textsc{eval-trm-if-label-2}\quad \frac{b_0 \Downarrow_{\text{bbe}} \text{Match } M_0 \qquad \text{Subst}(M_0, t_1) \Downarrow_{\text{trm}} \text{Abort } (\text{Next } L) \qquad t_2 \Downarrow_{\text{trm}} o}{\textbf{if}_L\ b_0 \textbf{ then } t_1 \textbf{ else } t_2 \ \Downarrow_{\text{trm}} \ o}$$

$$\textsc{eval-trm-if-label-3}\quad \frac{b_0 \Downarrow_{\text{bbe}} \text{Match } M_0 \qquad \text{Subst}(M_0, t_1) \Downarrow_{\text{trm}} o \qquad o \neq \text{Abort } (\text{Next } L)}{\textbf{if}_L\ b_0 \textbf{ then } t_1 \textbf{ else } t_2 \ \Downarrow_{\text{trm}} \ o}$$

$$\textsc{eval-trm-while-1}\quad \frac{b \Downarrow_{\text{bbe}} \text{Match } M \qquad (\text{Subst}(M, t);\ \textbf{while } b \textbf{ do } t \textbf{ done}) \Downarrow_{\text{trm}} o}{\textbf{while } b \textbf{ do } t \textbf{ done} \ \Downarrow_{\text{trm}} \ o}$$

$$\textsc{eval-trm-while-2}\quad \frac{b \Downarrow_{\text{bbe}} \text{Mismatch}}{\textbf{while } b \textbf{ do } t \textbf{ done} \ \Downarrow_{\text{trm}} \ \text{Res } tt}$$

$$\textsc{eval-trm-block-1}\quad \frac{t \Downarrow_{\text{trm}} \text{Abort } (\text{Exit } L\ v)}{L : \{\, t \,\} \ \Downarrow_{\text{trm}} \ \text{Res } v}$$

$$\textsc{eval-trm-block-2}\quad \frac{t \Downarrow_{\text{trm}} o \qquad \forall v.\ o \neq \text{Abort } (\text{Exit } L\ v)}{L : \{\, t \,\} \ \Downarrow_{\text{trm}} \ o}$$

$$\textsc{eval-trm-exit-1}\quad \frac{t \Downarrow_{\text{trm}} \text{Res } v}{\textbf{exit } L\ t \ \Downarrow_{\text{trm}} \ \text{Abort } (\text{Exit } L\ v)}$$

$$\textsc{eval-trm-exit-2}\quad \frac{t \Downarrow_{\text{trm}} \text{Abort } a}{\textbf{exit } L\ t \ \Downarrow_{\text{trm}} \ \text{Abort } a}$$

$$\textsc{eval-trm-next}\quad \frac{}{\textbf{next } L \ \Downarrow_{\text{trm}} \ \text{Abort } (\text{Next } L)}$$

Fig. 30. Semantics of terms in the minimal language, omitting obvious abort propagation.

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

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

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

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

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

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

$$\textsc{eval-pat-when-2}\quad \frac{v \triangleright p \Downarrow_{\text{pat}} \text{Match } M_1 \qquad \text{Subst}(M_1, b) \Downarrow_{\text{bbe}} \text{Mismatch}}{v \ \triangleright \ p \textbf{ when } b \ \Downarrow_{\text{pat}} \ \text{Mismatch}}$$

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

$$\textsc{eval-pat-some-1}\quad \frac{\forall v_1, ..., v_n.\ v \neq \text{Some}(v_1, ..., v_n)}{v \ \triangleright \ \text{Some}(p_1, ..., p_n) \ \Downarrow_{\text{pat}} \ \text{Mismatch}}$$

$$\textsc{eval-pat-some-2}\quad \frac{\begin{array}{c}\left(\forall i \in [1, k).\ v_i \triangleright \text{Subst}(\biguplus_{1 \le j < i} M_j, p_i) \Downarrow_{\text{pat}} \text{Match } M_i\right) \\ v_k \triangleright \text{Subst}(\biguplus_{1 \le j < k} M_j, p_k) \Downarrow_{\text{pat}} \text{Mismatch} \\ \forall i, j \in [1, k).\ i \neq j \Rightarrow M_i \# M_j\end{array}}{\text{Some}(v_1, ..., v_n) \ \triangleright \ \text{Some}(p_1, ..., p_n) \ \Downarrow_{\text{pat}} \ \text{Mismatch}}$$

$$\textsc{eval-pat-some-3}\quad \frac{\begin{array}{c}\forall i \in [1, n].\ v_i \triangleright \text{Subst}(\biguplus_{1 \le j < i} M_j, p_i) \Downarrow_{\text{pat}} \text{Match } M_i \\ \forall i, j \in [1, n].\ i \neq j \Rightarrow M_i \# M_j\end{array}}{\text{Some}(v_1, ..., v_n) \ \triangleright \ \text{Some}(p_1, ..., p_n) \ \Downarrow_{\text{pat}} \ \text{Match } (\biguplus_{1 \le j < i} M_j)}$$

Fig. 31. Semantics of BBEs and patterns in the minimal language