The Ultimate Conditional Syntax

LUYU CHENG, HKUST, Hong Kong, China
LIONEL PARREAUX, HKUST, Hong Kong, China

Functional programming languages typically support expressive pattern-matching syntax allowing programmers to write concise and type-safe code, especially appropriate for manipulating algebraic data types. Many features have been proposed to enhance the expressiveness of stock pattern-matching syntax, such as pattern bindings, pattern alternatives (a.k.a. disjunction), pattern conjunction, view patterns, pattern guards, pattern synonyms, active patterns, 'if-let' patterns, multi-way if-expressions, etc. In this paper, we propose a new pattern-matching syntax that is both more expressive and (we argue) simpler and more readable than previous alternatives. Our syntax supports parallel and nested matches interleaved with computations and intermediate bindings. This is achieved through a form of nested multi-way if-expressions with a *condition-splitting* mechanism to factor common conditional prefixes as well as a binding technique we call *conditional pattern flowing*. We motivate this new syntax with many examples in the setting of MLscript, a new ML-family programming language. We describe a straightforward desugaring pass from our rich *source* syntax into a minimal *core* syntax that only supports flat patterns and has an intuitive small-step semantics. We then provide a translation from the core syntax into a *normalized* syntax without backtracking, which is more amenable to coverage checking and compilation, and formally prove that our translation is semantics-preserving. We view this work as a step towards rethinking pattern matching to make it more powerful and natural to use. Our syntax can easily be integrated, in part or in whole, into existing as well as future programming language designs.

CCS Concepts: • **Software and its engineering** → **Control structures**; *Semantics*; *Syntax*; Functional languages; • **Theory of computation** → **Pattern matching**.

Additional Key Words and Phrases: pattern matching, syntax, programming language design

1 Introduction

Pattern matching on algebraic data types is the bread and butter of programming languages derived from or inspired by ML, such as OCaml, Haskell, SML, Scala, Rust, etc. This programming paradigm helps make illegal states unrepresentable and ensure that all possible cases are always handled, removing large classes of programming errors common in lower-level imperative languages.

However, at least in its most basic incarnation, ML-style pattern matching can be frustrating to use due to its lack of flexibility. For instance, it only allows matching a single item at a time and it does not allow parts of a pattern to depend on other parts of the same pattern; it does not allow interleaving patterns with computations; and it sometimes requires nesting many levels of pattern matches, which can decrease readability. Many extensions to ML-style pattern matching have been proposed over the years to alleviate some of these limitations, but we argue that they have all fallen short in one way or another. Moreover, we argue that sticking with the **`match-with`** (or **`case-of`**) construct is unnecessary and leads to less readable code.

In this paper, we propose *turning pattern matching on its head*, treating it as a generalization of **`if-then-else`** by adding destructuring and binding capabilities to boolean expressions, rather than treating **`if-then-else`** as a special case of pattern matching, as is usually done in ML languages. In so doing, we attain a new conditional syntax that is both more flexible than ML-style pattern matching with extensions, and, we argue, more readable and easier to learn.

We facetiously dub our syntax the *Ultimate Conditional Syntax* (UCS). Using the UCS, one can:

- Successively pattern-match several items in the same condition:

```
if x is Some(a) and y is Some(b) then a + b else 0
```

Authors' Contact Information: Luyu Cheng, HKUST, Hong Kong, China, luyu.cheng@connect.ust.hk; Lionel Parreaux, HKUST, Hong Kong, China, parreaux@cse.ust.hk.

This could be achieved in ML by matching on the pair `(x, y)`, but doing so would be less convenient, as all the branches in the corresponding pattern matching expression would need to destructure the pair, even when only one of its components is needed. Such pairing is also problematic in languages like Scala that have opt-in lazy and by-name evaluation but where pairs are not lazy, as it forces the evaluation of all components (including `y` here).

- Write patterns that depend on other patterns:

  ```
  ... x is Some(y) and y is Some(z) ...
  ```

  In ML, one could use a nested pattern `Some(Some(z) as y)`, but nested patterns are less expressive when combined with the other capabilities of the UCS, such as the capability to...
- Interleave patterns and computations:

  ```
  ... x is Some(y) and f(y) is Some(z) ...
  ```

  More concretely, consider needing to find whether an extracted name is part of a context:

  ```
  ... expr is Var(name) and ctx.get(name) is Some(value) and ...
  ```

- Avoid repetition by *splitting* conditional prefixes in arbitrary places. Below, the code on the left is equivalent to the more verbose ML expression on the right:

```
if foo(args)                                let tmp1 = foo(args) in
     == 0    then "null"                    if tmp1 == 0      then "null"
     |> abs                                 else let tmp2 = tmp1 |> abs in
       > 100 then "large"                   if tmp2 > 100     then "large"
       < 10  then "small"                   else if tmp2 < 10 then "small"
       else       "medium"                  else                   "medium"
```

Conditional splits allow us to recover concise pattern-matching style branching structures. For instance, below we match some `x` value against `Right`/`Left` patterns and match on a subsequent computation. We do this while avoiding repeating scrutinee expressions like '`x is ...`':

```
if x is
  Right(None)         then defaultValue
  Right(Some(cached)) then f(cached)
  Left(input)
    and compute(input) is
      None            then defaultValue
      Some(result)    then f(result)
```

Since matches on successive patterns are easily chained together, we can often avoid the excessive nesting of pattern-matching and if-then-else expressions. Below, we perform a number of successive checks and transformations on some input string, some of which return optional results that need to be pattern-matched as we go, laying out a large condition on one flat conditional level:

```
// Match identifiers of the form _1234
if name.startsWith("_")
  and name.tailOption is Some(namePostfix)
  and namePostfix.toIntOption is Some(index)
  and 0 <= index and index < arity
  then Right([index, name])
else Left("invalid identifier: " + name)
```

We implemented the UCS in MLscript, a new ML-inspired programming language currently being developed at HKUST.[1] MLscript uses a *structural* take on ML typing, based on algebraic subtyping [Dolan 2017; Dolan and Mycroft 2017; Parreaux 2020] generalized to a *Boolean algebra of types* that includes union, intersection, and negation types [Parreaux and Chau 2022]. This type

[1] Public repository at https://github.com/hkust-taco/mlscript/.

system allows one to pattern match on arbitrary data constructors even when these constructors do not appear in a common data type definition, similarly to extensible variants.[2] But the ideas of the UCS are general and can be applied to any programming language with pattern matching, not just MLscript. In this paper, we focus on the runtime semantics of pattern matching and leave the specifics of static typing to the language's underlying type system.

Much previous work has been dedicated to exploring various approaches to compile pattern matching efficiently. Unfortunately, most of these approaches are not directly applicable to the UCS, as they focus on the restricted structure of ML-style pattern matching. Compiling the UCS is more challenging because it is strictly more general than previous approaches, allowing the interleaving of computations, binding, and pattern destructuring. In this paper, we provide the first step towards a practical UCS compilation strategy by focusing on simplicity, semantics, and correctness. We leave for future work the exploration of more efficient compilation strategies taking inspiration from and reusing the results of previous work, where applicable.

The specific contributions we make are the following:

- We motivate the need for a concise pattern-matching syntax that is more flexible than traditional ML-style pattern-matching and present the UCS as a solution (Section 2).
- We describe the rich *source* syntax of the UCS and its *core* syntax, along with a desugaring from the former to the latter; we formalize the small-step semantics of the core syntax and its translation into a *normalized* representation amenable for type checking, exhaustiveness checking, and compilation; and we prove the correctness of this translation in terms of semantics preservation (Section 3).
- We describe our UCS implementation in MLscript and discuss technical aspects that are important for a practical realization of the UCS (Section 4).
- We compare the UCS to pattern-matching approaches in other languages, showing how the UCS is more readable and more flexible than other conditional syntaxes (Section 5).

2 Motivation and Presentation

Consider the following excerpt of an OCaml pattern matching expression that contains many cases:

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

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

This code suffers from "Boolean blindness" [Harper 2016], which is when a program branches based on a Boolean value and when the validity of the corresponding branches implicitly depends on the value of that Boolean. Our code is problematic because `Map.find` is a partial function that is only safe to call when accessing a key that is known to be in the map. Since the check that the key is indeed in the map is not explicitly connected to the call to `Map.find`, changes like a seemingly-innocuous refactoring could silently break the implicit assumption, resulting in runtime failure, for example if we added `... || value == 0` to the **`when`** clause.

It would be better to use the `Map.find_opt` function instead, which returns an optional value that is `None` when the key is not found. This would also be more efficient, as it would only require a

[2] One can also see a similarity with refinement types [Freeman and Pfenning 1991], whereby subtypes are used to precisely state which constructors are handled by each pattern matching expression.

single map query. However, due to the limitations of pattern matching in OCaml, there is no *local*, *non-disruptive* way of changing our program to using `find_opt` instead of `find`. If the program above had other `Lit` cases listed below, for example as in:

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

then we would need to completely change the structure of our pattern matching expression, merging all `Lit` branches into a single one and avoiding repetition by using an extra `helper` function:

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

With the Ultimate Conditional Syntax, one would write the following instead:

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

which shows that `Map.find_opt` no longer disrupts the existing flow of pattern matching.

The example above relies on several crucial features of the UCS:

Pattern flowing. A key idea of the UCS is that variables bound in patterns on the right-hand side of the '**`is`**' operator become available based on the truth of the current expression, i.e., in the "*logical continuation*" of the expression. A logical continuation can be a **`then`**, for example in '**`if`** `x` **`is`** `Some(x)` **`then`** `x + 1`', but it can also be an **`and`**, as in '**`if`** `x` **`is`** `Some(x)` **`and`** `x > 0` **`then`** `...`'. In this way, information flows through Boolean connectives and control structures in a way that is reminiscent of *flow-typing*, also called *occurrence-typing* [Castagna et al. 2022; Pearce 2013; Tobin-Hochstadt and Felleisen 2008].

A nested multi-way 'if' with interleaved 'let'. Second, the structure of if-then-else expressions is akin to a *nested* version of what Haskell calls "*multi-way if-expressions*"[3] but in which bindings can be interleaved. We generalize this further with *splits*, to allow expressing nested structures, as in:

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

```
if
  A and
      let P1 = ...
      B and
          let P2 = ...
          C then ...
          else ...
  let P3 = ...
  D then ...
  else ...
```

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

Conditional splits. Third and finally, a UCS expression may be *split* in arbitrary ways, leading to one or more conditional branches. For example, all of the following splits are legal:

```
if x ==                  if x                          if predicate of
  0 then "zero"            == 0 then "null"              0, 1 then "A"
  1 then "unit"            > 0  then "positive"          2, 3 then "B"
  _ then "?"               else      "negative"        else "C"
```

In MLscript, '`f` **`of`** `a, b, c`' means '`f(a, b, c)`' where '**`of`**' is right-associative (like Haskell's '$'). Splits on '**`is`**' are merely special cases of the general operator-splitting syntax:

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

As another example, consider the '`zip_with`' function below, which zips two lists together and returns `None` when the lists have mismatched lengths:

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

Note that the MLscript is indentation-sensitive and uses the "offside rule" [Landin 1966], so we do not need to use the **`in`** keyword after each **`let`** and we do not need a '`|`' operator to separate pattern matching cases (similar to F# and Haskell). However, the UCS does not fundamentally rely on indentation. We could allow the use braces and semicolons instead of indentation to split operators, as in '**`if`** `x { > 0` **`then`** `"positive"; == 0` **`then`** `"null"; else "?" }`'.

Coverage Checking. The UCS is meant to be a fully type-safe alternative to traditional conditional structures. Therefore, it is crucial to statically check for the *exhaustiveness* of conditional expressions, preventing match errors from occurring at runtime. We also perform some *redundancy* checking to warn about unreachable code due to impossible or already-handled patterns.

To perform these checks, we consider the entire conditional expression, made of uses of the **`and`**, **`is`**, and **`then`** operators, as a single entity. We also merge concomitant conditional expression, such as '**`if`** *A* **`else if`** *B*', into a single conditional expression '**`if`** { *A*; *B* }'. Top-level Boolean expressions

not part of a conditional expression, like the argument to `foo` in '`foo(x is None)`', are treated as a conditional expressions returning Boolean literals, as in '`foo(if x is None then True else False)`'. We treat plain Boolean expressions like '`x > 0`' as shorthands for, in this case, '`x > 0 is True`'; these are never treated as exhaustive and must be associated with a default branch. In the future, a form of normalization could be used to identify pure expressions like `x > 2`, `2 < x`, and `not (x <= 2)`, allowing matches such as '`if { x > 2 then` A`; x <= 2 then` B`}`' to be recognized as exhaustive[4].

As part of the UCS compilation process, we transform the conditional expression by unnesting patterns and handling each matched operand individually, in its order of appearance, making sure that all cases are covered in the process.

MLscript supports a form of subtyping known as *Boolean-algebraic subtyping* [Parreaux and Chau 2022] that admits principal ML-style type inference at the cost of some approximations in the meaning of types. In particular, in MLscript the union of tuple types $[\tau_1, \tau_2] \mid [\tau_2, \tau_3]$ and the tuple type with union components $[\tau_1 \mid \tau_2, \tau_2 \mid \tau_3]$ are equivalent. Therefore, it is important that the conditional expression below be considered inexhaustive and that it be rejected by the compiler:

```
if x is
  [0, 0] then true  // Match specifically the tuple value [0, 0]
  [1, 1] then false // Match specifically the tuple value [1, 1]
```

Our UCS implementation complains that patterns `[0, 1]` and `[1, 0]` are not handled. In a language like TypeScript or CDuce [Castagna et al. 2015], such a conditional expression could theoretically be typed as exhaustive by assuming $[0, 0] \mid [1, 1]$ for the type of `x`, which in these languages, unlike in MLscript, is *not* equivalent to $[0 \mid 1, 0 \mid 1]$ and precisely denotes a set of two tuple values.

Note that the motivating example above could also be handled by Haskell-style pattern guards. However, the UCS is more general and can also encode patterns like the following, which do not have a direct equivalent in Haskell:

```
if e is
  ...
  Var(name) and Map.find_opt(env, name) is
    Some(Right(value)) then Some(value)
    Some(Left(thunk))  then Some(thunk())
  App(lhs, rhs)        then ...
  ...
```

Here, we are splitting the pattern guard expression into several independent patterns leading to distinct `then` branches, which cannot be done using plain Haskell-style pattern guards.

3 Formalization

In this section, we formalize the UCS and its relevant algorithms, with special focus on the translation of UCS terms to a lower-level representation for compilation as well as the correctness proof of that translation.

3.1 The UCS Pipeline

Compiling the UCS is more difficult than compiling traditional pattern matching because the UCS is more flexible, notably allowing the interleaving of intermediate bindings and computations within the matching process. The UCS compilation pipeline consists of four stages, as illustrated in Fig. 1.

1. We start from the *source* syntax (Fig. 2) and desugar it into a *core* syntax (Fig. 3).

[4]We could naturally also imagine the use of an SMT solver to decide exhaustiveness in the presence of decidable theories like linear integer arithmetic [Xi and Pfenning 1998, 1999].

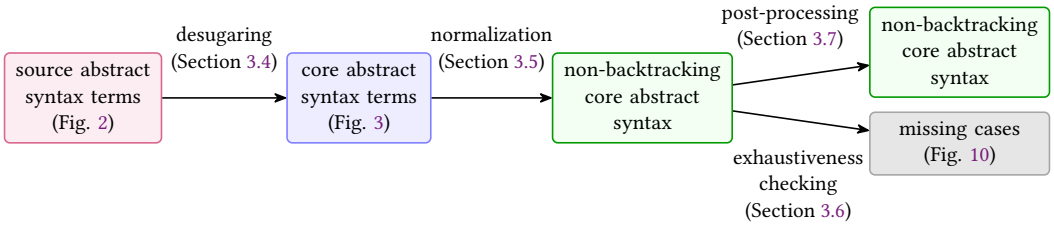

Fig. 1. High-level view of the UCS translation.

2. Core UCS terms make use of backtracking, which we eliminate through a *normalization* process in order to facilitate efficient compilation (Section 3.5).
3. Normalized terms are then processed in two parallel stages:
   a. The post-processing stage (Section 3.7) rewrites normalized terms to consolidate pattern matches on the same scrutinee on a best-effort basis.
   b. The exhaustiveness checking stage (Section 3.6) detects uncovered cases in post-processed terms by a simple form of abstract interpretation.

We formalize the semantics of core and normalized terms and prove that the normalization process is semantic-preserving.

3.2 Source Abstract Syntax

The source abstract syntax of the UCS is given in Fig. 2. Throughout our formalization, we use meta-variable ε to represent empty syntax forms, which are often omitted from examples. We use the notation $\overline{e}$ to denote the repetition of syntax forms e that are delimited by comma. For instance, $\overline{x}$ represents syntax forms such as 'ε', 'x', 'x, x', 'x, x, x', and so forth. Moreover, we use the notation $\overline{e_i}^{\,i}$ to denote the repetition of the syntax form e referring to index i.

$$
\begin{array}{rrcl}
\textit{Variable} & x, y, z & & \\
\textit{Constructor} & C, D & & \\
\textit{Term} & e & ::= & x \mid e\ e \mid C(\overline{e}) \mid \lambda x.\ e \mid e \oplus e \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \textbf{if}\ \{\, \mathcal{S}[\mathcal{T}] \,\} \\
\textit{Binary operator} & \oplus & ::= & + \mid - \mid \times \mid = \mid \neq \mid < \mid \leq \mid > \mid \geq \mid \ldots \\
\textit{Pattern} & p & ::= & x \mid C(\overline{p}) \\
\textit{Term branch} & \mathcal{T} & ::= & e\ \textbf{then}\ e \mid e\ \textbf{and}\ \{\, \mathcal{S}[\mathcal{T}] \,\} \mid e\ \textbf{is}\ \{\, \mathcal{S}[\mathcal{P}] \,\} \mid e\ \{\, \mathcal{S}[\mathcal{O}] \,\} \mid e \oplus \{\, \mathcal{S}[\mathcal{T}] \,\} \\
\textit{Pattern branch} & \mathcal{P} & ::= & p\ \textbf{then}\ e \mid p\ \textbf{and}\ \{\, \mathcal{S}[\mathcal{T}] \,\} \\
\textit{Operator branch} & \mathcal{O} & ::= & \textbf{is}\ \{\, \mathcal{S}[\mathcal{P}] \,\} \mid \oplus \{\, \mathcal{S}[\mathcal{T}] \,\} \\
\textit{Split} & \mathcal{S}[\mathcal{X}] & ::= & \mathcal{X}\ ;\ \mathcal{S}[\mathcal{X}] \mid \textbf{let}\ p = e\ ;\ \mathcal{S}[\mathcal{X}] \mid \textbf{else}\ e \mid \varepsilon \\
\textit{Branch} & \mathcal{X} & ::= & \mathcal{O} \mid \mathcal{P} \mid \mathcal{T}
\end{array}
$$

Fig. 2. Source abstract syntax.

3.2.1 Terms. We use meta-variable e to represent terms in source abstract syntax. Objects are denoted by $C(\overline{e})$, where C is an object *constructor* and terms $\overline{e}$ are arguments. We do not include data type declarations and inheritance in the source abstract syntax for simplicity, although our implementation supports them. We also omit literals from our syntax, given that every literal value can be represented using a nullary object constructor. Meta-variable $\oplus$ ranges over binary operators and can be used as infix expressions. We call expressions of the form '$\textbf{if}\ \{\, \mathcal{S}[\mathcal{T}] \,\}$' *UCS terms*.

3.2.2 Splits. A split represents a series of branches of the same kind which are interspersed with 'let' bindings and optionally ended with a default term. There are three kinds of branches, thus three kinds of splits. For simplicity, we represent splits using $\mathcal{S}[\mathcal{X}]$ where $\mathcal{X}$ is a meta-variable ranging over $\mathcal{O}$, $\mathcal{P}$, and $\mathcal{T}$, representing operator, pattern, and term branches, respectively.

As for the semantics, splits are evaluated sequentially. Each branch of a split is evaluated in the order it appears: if the previous branch does not hold, the following branch will be evaluated. Each interspersed 'let' binding is evaluated when none of the previous branches hold. As a result, variables declared by 'let' bindings are only bound in the branches and other 'let' bindings following it. If no branch in this split holds, the default term is evaluated, and if there is no default term, it will cause a match error at runtime.

3.2.3 Term Branches. Term branch $\mathcal{T}$ has five productions. All productions start with a non-terminal symbol e, which we call *initial terms*. The simplest term branch is 'e_1 **then** e_2'. It evaluates e_1 and checks if the result is the object True[5], and if this is the case, the entire enclosing UCS term evaluates to e_2. The similar term branch 'e **and** $\{\,\mathcal{S}[\mathcal{T}]\,\}$' makes use of special connective operator '**and**'. It evaluates e, and continues to evaluate the nested split of term branches if the condition holds. Pattern matching is expressed through production 'e **is** $\{\,\mathcal{S}[\mathcal{P}]\,\}$'. We first evaluate the scrutinee e, and then match it against each pattern branch in '$\mathcal{S}[\mathcal{P}]$', which is discussed later.

Conditional splits are represented by the last two productions. The fourth production '$e\ \{\,\mathcal{S}[\mathcal{O}]\,\}$' first evaluates the term e, which can be viewed as a left-hand side term missing the operator and the right-hand side term. Then, it evaluates the nested split of operator branches, where the result of e serves as the left-hand side and is paired with the operator and further evaluated. The last production '$e \oplus \{\,\mathcal{S}[\mathcal{T}]\,\}$' is akin to the fourth one. The difference is that it already has the left-hand side term e and binary operator $\oplus$, thus missing the right-hand side term. When evaluating the inner $\mathcal{S}[\mathcal{T}]$, partially applied function '$e \oplus$' is applied to the initial terms of branches in $\mathcal{S}[\mathcal{T}]$.

3.2.4 Pattern Branches. Before entering the pattern branch, we already have a scrutinee as the subject of pattern matching. A pattern p is either a variable x or a constructor pattern $C(\overline{p})$ in which each parameter can nest more patterns. Pattern branch $\mathcal{P}$ has two productions. The simplest branch 'p **then** e' checks whether the scrutinee matches pattern p. If it does, the entire UCS term evaluates to e. The other production 'p **and** $\{\,\mathcal{S}[\mathcal{T}]\,\}$' is akin to the first production but continue to evaluate a nested split of term branches, i.e., $\mathcal{S}[\mathcal{T}]$, if the scrutinee matches p.

For example, the following term checks whether variable x is an instance of Some. If that is the case, the sole argument is bound to variable xv and the term evaluates to the variable xv. Otherwise, the term checks that x is an instance of None and then evaluates to zero[6].

$$\textbf{if}\ \{\ x\ \textbf{is}\ \{\ \mathsf{Some}(xv)\ \textbf{then}\ xv\,;\ \mathsf{None}\ \textbf{then}\ 0\ \}\ \}$$

3.2.5 Operator Branches. Operator branch $\mathcal{O}$ has two productions. Before entering an operator branch, we already have a left-hand side term, namely e_L, for the operator. The first production '**is** $\{\,\mathcal{S}[\mathcal{P}]\,\}$' accepts '**is**' as the operator and leads to a split of pattern branches with e_L being the scrutinee. The second production '$\oplus\,\{\,\mathcal{S}[\mathcal{T}]\,\}$' takes a binary operator '$\oplus$' and leads to a nested split of term branches. Similar to the treatment of the last production of term branches, we apply the partially applied function '$e_L \oplus$' to the initial terms of branches in $\mathcal{S}[\mathcal{T}]$.

For example, the term below determines the sign of variable x. It contains two comparisons: $x > 0$ and $x < 0$. Because these two terms share the same left-hand side expression, we use operator

[5]Object True is a shorthand for 'True(ε)'. We omit empty argument lists for the sake of simplicity.

[6]For clarity, we assume that each integer corresponds to a unique nullary constructor, and the use of integer literals is a shorthand for using corresponding nullary object constructors.

$$
\begin{array}{ll}
\textit{Term} & t ::= x \mid t\ t \mid C(\overline{t_i}^{\,i}) \mid \lambda x.\ t \mid \textbf{let}\ x = t\ \textbf{in}\ t \mid \textbf{if}\ \{\ \sigma\ \} \\
\textit{Value} & v ::= C(\overline{v_i}^{\,i}) \mid \lambda x.\ t \\
\textit{Split} & \sigma ::= t\ \textbf{is}\ C(\overline{x_i}^{\,i}) \rightarrow \{\ \sigma\ \}\ ;\ \sigma \mid \textbf{let}\ x = t\ ;\ \sigma \mid \textbf{else}\ t \mid \varepsilon
\end{array}
$$

Fig. 3. Core abstract syntax.

branches to make it more concise.

$$\textbf{if}\ \{\ x\ \{\ \{\ <\ \{\ 0\ \textbf{then}\ -1\ \}\ \}\ ;\ \{\ >\ \{\ 0\ \textbf{then}\ 1\ \}\ \}\ ;\ \textbf{else}\ 0\ \}\ \}$$

3.2.6 Conclusion. Intuitively speaking, a split contains multiple branches, each of which is not limited to one condition and can introduce nested splits through connective '**and**', thus extending more branches. Meanwhile, split allows the introduction of variables whose values are only evaluated in certain branches through interleaved 'let' bindings. Although the source abstract syntax is comprehensible and straightforward, many of its rules remain trivial to some extent. In formalization and implementation, we transform it into an equivalent but more condensed syntax.

3.3 Core Abstract Syntax

Our discussion on core abstract syntax (Fig. 3) focuses on its differences from source abstract syntax. First of all, values are either lambda abstractions or objects of which arguments are values. Moreover, variable patterns are desugared away and patterns cannot nest. Most importantly, all sorts of branches are condensed into a simple form of predicate 'x **is** $C(\overline{x})$' and connectives such as '**and** $\{\ \mathcal{S}[\mathcal{T}]\ \}$' and '**then** e' are uniformly represented by '$\rightarrow\ \{\ \sigma\ \}$'. For all split like '$t$ **is** $C(\overline{x_i}^{\,i}) \rightarrow \{\ \sigma_1\ \}\ ;\ \sigma_2$', we refer to '$t$ **is** $C(\overline{x_i}^{\,i})$' as the *leading match*. We also refer to σ_1 and σ_2 as the *consequent split* and the *alternative split* regarding the leading match respectively.

Definition 3.1 (UCS Terms). Terms of the form '**if** $\{\ \sigma\ \}$' are referred to as the *UCS terms*. Its contained split σ is called the *topmost split* of that UCS term.

The core abstract syntax is considerably more straightforward than the source syntax, while it still retains the core expressiveness—a UCS term is a multi-level pattern matching expression whose branches can have different scrutinees and branches can be interspersed with 'let' bindings.

Definition 3.2 (Fullness of Splits). For all σ, if σ ends with '**else** t', then we say σ is *full*.

Split concatenation, represented by the binary operator ++, joins together two splits. If the left-hand side split is full, the right-hand side split is discarded. Otherwise, the right-hand side split is appended to the end of the left-hand side split. We adhere to Barendregt [1984] variable convention, that if σ_1 and σ_2 are from different branches, then all their bound variables are chosen to be different from the free variables. Therefore, split concatenation does not cause accidental variable capture. Split concatenation is necessary in the evaluation because the latter split acts as an alternative when the entire former split does not hold.

The small-step evaluation semantics of core abstract syntax is given in Fig. 4. The relation $t \rightsquigarrow t'$ reads "term t evaluates to term t' in one step". We reduce UCS terms by reducing the topmost split. Rule E-IfLet reduces 'let' bindings in splits in a way similar to E-Let. Rule E-Else reduces UCS terms whose topmost splits only contain a default branch '**else** t' to t. E-Match and E-NotMatch discuss whether the leading match of the topmost split holds or not. If the scrutinee is an object in the form as specified by pattern '$C(\overline{x_i}^{\,i})$', we substitute occurrences of each x_i with v_i in σ_1 and concatenate the resulting split with σ_2. Otherwise, we discard the consequent split σ_1 and let σ_2 be the topmost split, as indicated by E-NotMatch.

$$E[\square] ::= \square\ t \mid v\ \square \mid C(\overline{v}, \square, \overline{t}) \mid \textbf{let}\ x = \square\ \textbf{in}\ t \mid \textbf{if}\ \{\square\ \textbf{is}\ C(\overline{x}) \to \{\sigma\};\ \sigma\} \mid \textbf{if}\ \{\textbf{let}\ x = \square;\ \sigma\}$$

$$\begin{array}{rcl}
\textsc{E-Ctx} & E[t] \rightsquigarrow E[t'] & \text{if } t \rightsquigarrow t' \\
\textsc{E-AppAbs} & (\lambda x.\ t)\ v \rightsquigarrow [x \mapsto v]\ t & \\
\textsc{E-Let} & \textbf{let}\ x = v\ \textbf{in}\ t \rightsquigarrow [x \mapsto v]\ t & \\
\textsc{E-IfLet} & \textbf{if}\ \{\textbf{let}\ x = v;\ \sigma\} \rightsquigarrow \textbf{if}\ \{[x \mapsto v]\ \sigma\} & \\
\textsc{E-Else} & \textbf{if}\ \{\textbf{else}\ t\} \rightsquigarrow t & \\
\textsc{E-Match} & \textbf{if}\ \{C(\overline{v_i}^{\,i})\ \textbf{is}\ C(\overline{x_i}^{\,i}) \to \{\sigma_1\};\ \sigma_2\} \rightsquigarrow \textbf{if}\ \{[\overline{x_i \mapsto v_i}^{\,i}]\ \sigma_1 +\!\!+ \sigma_2\} & \\
\textsc{E-NotMatch} & \textbf{if}\ \{v\ \textbf{is}\ C(\overline{x_i}^{\,i}) \to \{\sigma_1\};\ \sigma_2\} \rightsquigarrow \textbf{if}\ \{\sigma_2\} & \text{if } v \text{ is not } C(\overline{v_i}^{\,i})
\end{array}$$

$$\boxed{\sigma +\!\!+ \sigma\ : \sigma}$$

$$\begin{aligned}
(t\ \textbf{is}\ C(\overline{x}) \to \{\sigma_1\};\ \sigma_2) +\!\!+ \sigma_3 &= t\ \textbf{is}\ C(\overline{x}) \to \{\sigma_1\};\ (\sigma_2 +\!\!+ \sigma_3) \\
(\textbf{let}\ x = t;\ \sigma_1) +\!\!+ \sigma_2 &= \textbf{let}\ x = t;\ (\sigma_1 +\!\!+ \sigma_2) \\
(\textbf{else}\ t) +\!\!+ \sigma\ &= \textbf{else}\ t \\
\varepsilon +\!\!+ \sigma\ &= \sigma
\end{aligned}$$

Fig. 4. The small-step evaluation rules of core abstract syntax.

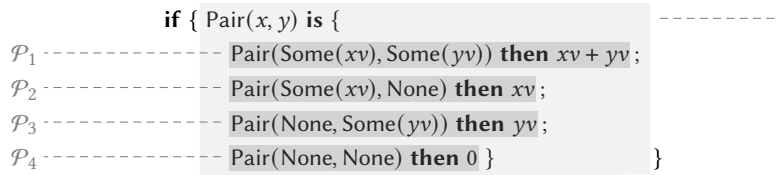

Fig. 5. An example of source abstract syntax.

Finally, we introduce a notational shorthand, which is extensively used in subsequent examples and proofs, for the sake of simplicity.

Definition 3.3 (Shorthands for Splits Containing Single Else). We denote splits like 't_1 **is** $C(\overline{x_i}^{\,i})$ $\to$ { **else** t_2 }; σ' by 't_1 **is** $C(\overline{x_i}^{\,i}) \to t_2$; σ'.

3.4 Desugaring from Source to Core Abstract Syntax

The initial step in translating the UCS involves desugaring terms from source abstract syntax to core abstract syntax. The desugaring algorithm is a straightforward recursive function $\mathcal{D}[\![e]\!]$ that operates on source abstract syntax terms and produces core abstract syntax terms. While the algorithm does introduce fresh variables, it is completely local and syntax-directed: it does not require any reasoning, and the expectation is that programmers should intuitively understand what the source abstract syntax terms they write desugar to. In our implementation, source abstract syntax terms are desugared on the fly during parsing.

Fig. 5 shows a term in source abstract syntax. Some branches within the term are labeled for our ease of reference. The example includes three constructors: Pair takes two arguments and

represents 2-tuples (a.k.a. pairs); Some, which takes one argument, and None, which takes no arguments, collectively represent an algebraic data type of possibly optional values[7].

The term determines the sum of two optional values x and y, returning the sum of corresponding constructor arguments if both are Some, the only argument if only one is Some, or zero if both are None. Programmers often write code akin to it in common programming tasks.

The top-level term split has only one term branch $\mathcal{T}_0$. We first bind its scrutinee to a fresh variable x_0, which makes sure that it is only evaluated once. Next, the nested pattern split contains four pattern branches: $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$, and $\mathcal{P}_4$. Because core abstract syntax does not allow nested patterns, we must expand sub-patterns into nested splits. For each nested sub-pattern, we create a fresh variable as a sub-scrutinee and expand the sub-pattern with the sub-scrutinee.

For instance, $\mathcal{P}_1$'s pattern '$\mathsf{Pair}(\mathsf{Some}(xv), \mathsf{Some}(yv))$' has two sub-patterns: '$\mathsf{Some}(xv)$' and '$\mathsf{Some}(yv)$'. We create two fresh variables, namely x_1 and x_2, as their sub-scrutinees. We can obtain the following split for $\mathcal{P}_1$.

$$x_0 \textbf{ is } \mathsf{Pair}(x_1, x_2) \rightarrow \{\, x_1 \textbf{ is } \mathsf{Some}(yv) \rightarrow \{\, x_2 \textbf{ is } \mathsf{Some}(xv) \rightarrow \{\, \textbf{else } xv + yv \,\} \,\} \,\}$$

By repeating the above operation for branches $\mathcal{P}_2$, $\mathcal{P}_3$, and $\mathcal{P}_4$, we can obtain the term in Fig. 6. Note that the fresh variables for the sub-scrutinees of the same parameter of Pair are different in

$$\begin{aligned}
&\textbf{if } \{\, \textbf{let } x_0 = \mathsf{Pair}(x, y)\,; \\
&\quad x_0 \textbf{ is } \mathsf{Pair}(x_1, x_2) \rightarrow \{\, x_1 \textbf{ is } \mathsf{Some}(xv) \rightarrow \{\, x_2 \textbf{ is } \mathsf{Some}(yv) \rightarrow \{\, \textbf{else } + xv\ yv \,\} \,\} \,\}\,; \\
&\quad x_0 \textbf{ is } \mathsf{Pair}(x_3, x_4) \rightarrow \{\, x_3 \textbf{ is } \mathsf{Some}(xv) \rightarrow \{\, x_4 \textbf{ is } \mathsf{None} \rightarrow \{\, \textbf{else } xv \,\} \,\} \,\}\,; \\
&\quad x_0 \textbf{ is } \mathsf{Pair}(x_5, x_6) \rightarrow \{\, x_5 \textbf{ is } \mathsf{None} \rightarrow \{\, x_6 \textbf{ is } \mathsf{Some}(yv) \rightarrow \{\, \textbf{else } yv \,\} \,\} \,\}\,; \\
&\quad x_0 \textbf{ is } \mathsf{Pair}(x_7, x_8) \rightarrow \{\, x_7 \textbf{ is } \mathsf{None} \rightarrow \{\, x_8 \textbf{ is } \mathsf{None} \rightarrow \{\, \textbf{else } 0 \,\} \,\} \,\} \,\}
\end{aligned}$$

Fig. 6. Desugaring of the term of Fig. 5.

each pattern branch, which aligns with the assumption in our formalization that all variables are unique. In implementation, this can minimize naming conflicts when merging branches. Readers may notice that the innermost splits are all in the form of '**else** t', which can appear somewhat verbose. Therefore, we use the shorthands in Definition 3.3 in subsequent discussions.

Now, we can illustrate the backtracking nature of the semantics of core abstract syntax using the desugared term above as an example. Suppose x is Some and y is None. After we test 'x_0 **is** $\mathsf{Pair}(x_1, x_2)$' and 'x_1 **is** $\mathsf{Some}(xv)$', we find that 'x_2 **is** $\mathsf{Some}(yv)$' does not hold. Therefore, we should return to the topmost split and proceed to the second branch. However, the first two predicates of this branch have already been tested, leading to redundant evaluations. A logical approach is to *factorize* two branches if they both begin with a match of the same scrutinee variable and the same constructor. In Fig. 6, the topmost split has four idential matches which check whether x_0 is Pair at the beginning. We can move their consequent splits to the same split (this can be achieved through split concatenation), and substitute the variables in Pair pattern. As a result, even if the previous branches fail during the evaluation, the entire UCS term only needs to check once whether x_0 is Pair. This idea forms the basis of the normalization described in the next section.

We illustrate the desugaring algorithm through several representative examples in Fig. 7. Due to space constraints, we cannot display the full algorithm here. The desugaring algorithm is provided in the appendix in Fig. 13 , along with the auxiliary desugaring functions referenced in Fig. 14 .

[7]This emulates the use of types `Option[A]` in Scala, `'a option` in OCaml, and `Maybe a` in Haskell.

[8]In these examples, x_i are fresh variables generated by desugaring algorithm; whereas y_i are user-defined variables.

Source Abstract Syntax Terms	$\mathcal{D}$esugared Terms
if { e_1 **is** { Some(Left(lv)) **then** e_2 ; Some(Right(rv)) **then** e_3 ; None **then** e_4 } }	**if** { **let** $x_1 = \mathcal{D}[\![e_1]\!]$; x_1 **is** Some(x_2) $\rightarrow$ { x_2 **is** Left(lv) $\rightarrow \mathcal{D}[\![e_2]\!]$ } ; x_1 **is** Some(x_2) $\rightarrow$ { x_2 **is** Right(rv) $\rightarrow \mathcal{D}[\![e_3]\!]$ } ; x_1 **is** None $\rightarrow \mathcal{D}[\![e_4]\!]$ }
if { y_1 **is** { Some(v) **and** { $p(v)$ **then** e_1 } ; Some(v) **then** e_2 ; None **then** e_3 }}	**if** { y_1 **is** Some(v) $\rightarrow$ { $p(v)$ **is** True $\rightarrow \mathcal{D}[\![e_1]\!]$ } ; y_1 **is** Some(v) $\rightarrow \mathcal{D}[\![e_2]\!]$; y_1 **is** None $\rightarrow \mathcal{D}[\![e_3]\!]$ }
if { x { { < { 0 **then** -1 } } ; { > { 0 **then** 1 } } ; **else** 0 } }	**if** { < x 0 **is** True $\rightarrow -1$; > x 0 **is** True $\rightarrow$ 1 ; **else** 0 }

Fig. 7. Source abstract syntax terms and corresponding core abstract syntax terms.[8]

Normalization of Splits $\mathcal{M}[\![\sigma]\!] : \sigma$

$$\mathcal{M}[\![x \textbf{ is } C(\overline{x}) \rightarrow \{\sigma_1\};\ \sigma_2]\!] = x \textbf{ is } C(\overline{x}) \rightarrow \textbf{if } \{\mathcal{M}[\![\mathcal{S}^+_{x \textbf{ is } C(\overline{x})}[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!]\};\ \mathcal{M}[\![\mathcal{S}^-_{x \textbf{ is } C(\overline{x})}[\![\sigma_2]\!]]\!]$$
$$\mathcal{M}[\![t \textbf{ is } C(\overline{x}) \rightarrow \{\sigma_1\};\ \sigma_2]\!] = t \textbf{ is } C(\overline{x}) \rightarrow \textbf{if } \{\mathcal{M}[\![\sigma_1 +\!\!+ \sigma_2]\!]\};\ \mathcal{M}[\![\sigma_2]\!] \qquad \text{if } t \text{ is not } x$$
$$\mathcal{M}[\![\textbf{let } x = t;\ \sigma]\!] = \textbf{let } x = t;\ \mathcal{M}[\![\sigma]\!] \quad \mathcal{M}[\![\textbf{else } t]\!] = \textbf{else } t \quad \mathcal{M}[\![\varepsilon]\!] = \varepsilon$$

Specialization of Splits $\mathcal{S}^{\pm}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![\sigma]\!] : \sigma$

$$\mathcal{S}^{\pm}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![t \textbf{ is } D(\overline{y_i}^{\,i}) \rightarrow \{\sigma_1\};\ \sigma_2]\!] = t \textbf{ is } D(\overline{y_i}^{\,i}) \rightarrow \{\mathcal{S}^{\pm}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![\sigma_1]\!]\};\ \mathcal{S}^{\pm}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![\sigma_2]\!]$$
$$\mathcal{S}^{\pm}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![y \textbf{ is } D(\overline{y_i}^{\,i}) \rightarrow \{\sigma_1\};\ \sigma_2]\!] = y \textbf{ is } D(\overline{y_i}^{\,i}) \rightarrow \{\mathcal{S}^{\pm}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![\sigma_1]\!]\};\ \mathcal{S}^{\pm}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![\sigma_2]\!] \qquad \text{if } x \neq y$$
$$\mathcal{S}^{+}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![x \textbf{ is } D(\overline{y_i}^{\,i}) \rightarrow \{\sigma_1\};\ \sigma_2]\!] = \mathcal{S}^{+}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![\sigma_2]\!] \qquad \text{if } C \neq D$$
$$\mathcal{S}^{+}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![x \textbf{ is } C(\overline{y_i}^{\,i}) \rightarrow \{\sigma_1\};\ \sigma_2]\!] = \mathcal{S}^{+}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![[\overline{y_i \mapsto x_i}^{\,i}]\ \sigma_1 +\!\!+ \sigma_2]\!]$$
$$\mathcal{S}^{-}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![x \textbf{ is } C(\overline{y_i}^{\,i}) \rightarrow \{\sigma_1\};\ \sigma_2]\!] = \mathcal{S}^{-}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![\sigma_2]\!]$$
$$\mathcal{S}^{\pm}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![\textbf{let } y = t;\ \sigma]\!] = \textbf{let } y = t;\ \mathcal{S}^{\pm}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![\sigma]\!]$$
$$\mathcal{S}^{\pm}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![\textbf{else } t]\!] = \textbf{else } t$$
$$\mathcal{S}^{\pm}_{x \textbf{ is } C(\overline{x_i}^{\,i})}[\![\varepsilon]\!] = \varepsilon$$

Fig. 8. Normalization of splits.

3.5 Addressing Backtracking with Normalization

In the previous section, we demonstrated the nature of backtracking in core abstract syntax through Fig. 6. Next, we introduce normalization, a translation from the core abstract syntax to a strict subset of the core abstract syntax whose semantics do not require backtracking.

3.5.1 Normalization. Normalization can be applied to terms and splits, which are denoted by $\mathcal{N}[\![t]\!]$ and $\mathcal{M}[\![\sigma]\!]$ respectively. Term normalization $\mathcal{N}[\![t]\!]$ simply traverses the given term and applies $\mathcal{M}[\![\sigma]\!]$ to σ in '**if** { σ }'. Therefore, the definition of $\mathcal{N}[\![t]\!]$ is congruence rules. The main focus of normalization is primarily on splits, which is given in Fig. 8.

3.5.2 Specialization. The essence of split normalization is to factorize each split according to the leading match. This operation follows the idea that each branch is tested only once. To be specific,

for splits like 'x **is** $C(\overline{x}) \to \{\, \sigma_1 \,\};\ \sigma_2$', we remove all branches (and their nested splits) that match x against constructors other than C, and skip all branches (but preserve their nested splits) that match x against constructor C in $\sigma_1 +\!\!+ \sigma_2$. The resulting split serves as the new consequent split. We also remove all branches (and their nested splits) that match x against constructor C in σ_2, with the produced split as the new alternative split. Subsequently, we recursively perform the same operation on the newly obtained consequent split and alternative split. We call this recursive transformation *specialization* (Fig. 8). There are two modes of specialization, one for dealing with consequent splits and the other for dealing with alternative splits. We use $\mathcal{S}^+$ and $\mathcal{S}^-$ to denote them respectively. Both of them accept three parameters: a variable scrutinee x, a constructor pattern $C(\overline{y})$, and a split σ. For the brevity of definitions, we denote functions by $\mathcal{S}^\pm$ if both $\mathcal{S}^+$ and $\mathcal{S}^-$ have the same definition in some cases.

Note that in the first case of the definition of $\mathcal{M}[\![\, t \,]\!]$, we first concatenate the consequent split and the alternative split, then perform specialization, and finally normalize the resulting split. This captures the semantics of splits: if there are no branches that hold in the consequent split, we evaluate the alternative split instead.

3.5.3 Example. We use the example in Fig. 6 to demonstrate the normalization step by step. Observe the top-level split in Fig. 6, the first branch we consider is 'x_0 **is** $\mathsf{Pair}(x_1, x_2)$'. By applying the specialization, we obtain the consequent split and the alternative split below.

$$
\begin{aligned}
\mathcal{S}^+_{x_0 \text{ is } \mathsf{Pair}(x_1,x_2)}[\![\sigma]\!] \;=\;& x_1 \text{ is } \mathsf{Some}(xv) \to \{\, x_2 \text{ is } \mathsf{Some}(yv) \to +\, xv\ yv \,\};\\
& x_1 \text{ is } \mathsf{Some}(xv) \to \{\, x_2 \text{ is } \mathsf{None} \to xv \,\};\\
& x_1 \text{ is } \mathsf{None} \quad\ \to \{\, x_2 \text{ is } \mathsf{Some}(yv) \to yv \,\};\\
& x_1 \text{ is } \mathsf{None} \quad\ \to \{\, x_2 \text{ is } \mathsf{None} \to 0 \,\}\\
\mathcal{S}^-_{x_0 \text{ is } \mathsf{Pair}(x_1,x_2)}[\![\sigma]\!] \;=\;& \varepsilon
\end{aligned}
$$

Subsequently, we continue to normalize the splits above by specializing regarding x_1 **is** $\mathsf{Some}(xv)$.

$$
\begin{aligned}
\mathcal{S}^+_{x_1 \text{ is } \mathsf{Some}(xv)}[\![\sigma]\!] \;=\;& x_2 \text{ is } \mathsf{Some}(yv) \to +\, xv\ yv;\ x_2 \text{ is } \mathsf{None} \to xv\\
\mathcal{S}^-_{x_1 \text{ is } \mathsf{Some}(xv)}[\![\sigma]\!] \;=\;& x_1 \text{ is } \mathsf{None} \to \{\, x_2 \text{ is } \mathsf{Some}(yv) \to yv \,\};\\
& x_1 \text{ is } \mathsf{None} \to \{\, x_2 \text{ is } \mathsf{None} \to 0 \,\}
\end{aligned}
$$

By repeating the operations above to the remaining split, the normalization finally produces the term in Fig. 9. We can observe that all matches are factorized. Therefore, regardless of whether x and y are Some or None, each scrutinee is matched against each constructor only once.

$$
\begin{aligned}
&\textbf{if } \{\ \textbf{let } x_0 = \mathsf{Pair}(x, y)\,;\\
&\quad x_0 \text{ is } \mathsf{Pair}(x_1, x_2) \to \{\\
&\qquad x_1 \text{ is } \mathsf{Some}(xv) \to \{\, x_2 \text{ is } \mathsf{None} \to xv\,;\\
&\qquad\qquad\qquad\quad x_2 \text{ is } \mathsf{Some}(yv) \to +\, xv\ yv \,\}\,;\\
&\qquad x_1 \text{ is } \mathsf{None} \to \{\, x_2 \text{ is } \mathsf{None} \to 0\,;\\
&\qquad\qquad\qquad\quad x_2 \text{ is } \mathsf{Some}(yv) \to yv \,\}\ \}\ \}
\end{aligned}
$$

Fig. 9. Normalized form of the term of Fig. 6.

3.6 Coverage Checking

Coverage checking is two-fold: redundancy checking and exhaustiveness checking. Redundancy checking is not a separate stage and is jointly accomplished by the prior stages. We first exemplify redundancy and then describe the coverage checking algorithm.

3.6.1 Redundancy Checking. Redundancy refers to those unreachable branches that have already been handled by previous branches, so they can be safely removed from the UCS terms. Redundancy can generally be divided into the following three categories. We illustrate each category with desugared core abstract syntax terms.

Contradictory cases. Contradictory cases contain conflicting patterns, connecting two contradictory patterns against the same scrutinee with '**and**' operator. For example, the first branch requires x to be A and B simultaneously in the following term.[9]

$$\textbf{if } \{\, x \textbf{ is } \mathsf{A} \to \{\, x \textbf{ is } \mathsf{B} \to t_1 \,\};\ x \textbf{ is } \mathsf{B} \to t_2 \,;\ x \textbf{ is } \mathsf{C} \to t_3 \,\}$$

The conflicting split 'x **is** $B \to t_1$' is detected and removed by $\mathcal{S}^{+}_{x \textbf{ is } \mathsf{A}}[\![\cdot]\!]$ during normalization.

Duplicated cases. If the identical pattern appears twice in parallel, then split led by the latter pattern is never evaluated, as t_2 is redundant due to t_1 in the following term.

$$\textbf{if } \{\, x \textbf{ is } \mathsf{A} \to t_1 \,;\ x \textbf{ is } \mathsf{A} \to t_2 \,;\ x \textbf{ is } \mathsf{B} \to t_3 \,\}$$

Duplicated 'x **is** $\mathsf{A} \to t_2$' is detected and removed by $\mathcal{S}^{-}_{x \textbf{ is } \mathsf{A}}[\![\cdot]\!]$ in the specialization of the alternative split during normalization.

Unreachable branches. Apart from the two categories mentioned above, the most common scenario is that the condition of the previous split is always met and it is already full, so the subsequent split will not be evaluated under any circumstances. For example, in

$$\textbf{if } \{\, f(x) \textbf{ is } y \textbf{ and } \{\, p(y) \textbf{ then } t_1 \,;\ \textbf{else } t_2 \,\};\ g(x) \textbf{ then } t_3 \,\}$$

term split '$g(x)$ **then** t_3' is unreachable because '$f(x)$ **is** y' is unconditional and the inner split is full. This kind of branches will be detected and removed in desugaring.

3.6.2 Scrutinee Identification. Before discussing exhaustiveness checking, we first address an important issue: how to determine if two scrutinees are identical. Observe that sub-scrutinees in nested patterns are replaced with fresh variables during normalization. Thus, a scrutinee may have aliases if the pattern occurred in many places. We introduce a mechanism that assigns a unique identifier to each scrutinee.

For example, term

$$\textbf{if } \{\ e_1 \textbf{ is } \{\ \mathsf{Some}(\mathsf{Left}(y_1)) \textbf{ then } e_2 \,;\ \mathsf{Some}(\mathsf{Right}(y_2)) \textbf{ then } e_3 \,;\ \mathsf{None} \textbf{ then } e_3 \ \}\ \}$$

is desugared to

$$\begin{aligned}
&\textbf{if } \{\ \textbf{let } x_0 = \mathcal{D}[\![e_1]\!]\,; \\
&\quad x_0 \textbf{ is } \mathsf{Some}(x_1) \to \{\, x_1 \textbf{ is } \mathsf{Left}(y_1) \to \mathcal{D}[\![e_1]\!] \,\}\,; \\
&\quad x_0 \textbf{ is } \mathsf{Some}(x_2) \to \{\, x_2 \textbf{ is } \mathsf{Right}(y_2) \to \mathcal{D}[\![e_2]\!] \,\}\,; \\
&\quad x_0 \textbf{ is } \mathsf{None} \quad\ \to \{\, \textbf{else } t_2 \,\}\ \}.
\end{aligned}$$

Note that scrutinees x_1 and x_2 refer to the same value. To make sure that identifiers of x_1 and x_2 are identically identified, We need to traverse the entire term and associate each variable with a unique identifier. The syntax of identifiers is defined as follows.

$$\textit{Identifier} \quad n ::= [x] \mid n/C.i \quad i = 1, 2, 3, \ldots$$

An identifier is either a root identifier '$[x]$' representing an independent scrutinee that is not bound by any pattern, or a sub-identifier '$n/C.i$', representing the i-th parameter of a parent identifier n when n is determined to be constructed by C. The mapping from variables to identifiers is denoted by $N : x \mapsto n$. For example, the mapping aggregated from the term above is

$$\{\ (x_0 \mapsto [x_0]),\ (x_1 \mapsto [x_0]/\mathsf{Some}.1),\ (x_2 \mapsto [x_0]/\mathsf{Some}.1)\ \}$$

[9] Note that this is possible in a system with subtyping or intersection types.

It can be seen that both x_1 and x_2 are associated with the same identifier. Given a variable x, we can obtain its identifier by $N(x)$. The function used for aggregating the mapping N, which is denoted by $collect(\sigma)$, is a straightforward recursive function and is given in Fig. 10.

3.6.3 Exhaustiveness Checking. Non-exhaustive cases represent scenarios that the program has not accounted for but may encounter at runtime and lead to unexpected failures. An example of non-exhaustive cases can be obtained by removing any branch of the desugared term in Fig. 6. To check for exhaustiveness, we collect the constructors that each scrutinee is matched with and check that these constructors are matched in all branches.

Definitions of Sets and Maps

$$\begin{aligned} &\textit{Identifier map} && N ::= \{\ \overline{x \mapsto n}\ \} \\ &\textit{Pattern set} && U, W, M ::= \{\ \overline{(n, C)}\ \} \\ &\textit{Missing case set} && K ::= \{\ \overline{M \Rightarrow W}\ \} \end{aligned}$$

Operations on Pattern Sets

$$\textit{Extracting Subsets by Identifiers} \qquad U(n) = \{\ (n', C) \in U \mid n' = n\ \}$$

$$\boxed{collect(\sigma)\ :\ (U, N)}$$

$$\begin{aligned} &collect(\varepsilon) && = (\varnothing, \varnothing) \\ &collect(\textbf{else}\ t) && = (\varnothing, \varnothing) \\ &collect(x\ \textbf{is}\ C(\overline{x_i}^{i \in [1,n]}) \rightarrow \{\ \sigma_1\ \};\ \sigma_2) && = (\{\ N_1(x) \mapsto C\ \} \cup collect(\sigma_1) \cup collect(\sigma_2),\ N_1) \end{aligned}$$

$$\text{where } N_0 = \begin{cases} N & \text{if } x \mapsto n \in N \\ N \cup \{x \mapsto [x]\} & \text{otherwise} \end{cases},\ N_1 = N_0 \cup \{x_i \mapsto N(x)/C.i \mid i \in [1, n]\}$$

$$\boxed{check_{U,N}(\sigma, W, M)\ :\ K}$$

$$\begin{aligned} &check_{U,N}(\varepsilon, W, M) && = \{\ M \Rightarrow \{\ (n, C) \in W \mid W(n) \neq U(n)\ \}\ \} \\ &check_{U,N}(\textbf{else}\ t, W, M) && = \varnothing \\ &check_{U,N}(x\ \textbf{is}\ C(\overline{x_i}^{\,i}) \rightarrow \{\ \textbf{else if}\ \{\ \sigma_1\ \};\ \sigma_2\ \}, W, M) && = check_{U,N}(\sigma_1, W_1, M_1) \cup check_{U,N}(\sigma_2, W_2, M) \end{aligned}$$

$$\text{where } N(x) = n,\ W_1 = \{\ (n', C) \in W \mid n' \neq n\ \},\ W_2 = W - \{\ (n, C)\ \},\ \text{and } M_1 = M \cup \{\ (n, C)\ \}$$

Fig. 10. The exhaustiveness checking algorithm.

Our next step is to summarize a *match set* M describing all patterns that each scrutinee is matched against. The function is denoted by $collect(\sigma)$ and is given in Fig. 10. Each element of set M is a pair of identifiers and constructors. To demonstrate how the function works, we apply $collect(\sigma)$ to the example above, then we can obtain the following pattern set.

$$M = \{\ ([x_0], \text{Some}),\ ([x_0], \text{None}),\ ([x_0]/\text{Some.1}, \text{Left}),\ ([x_0]/\text{Some.1}, \text{Right})\ \}$$

The match set can be used to check the exhaustiveness of the UCS term from which it is aggregated. The coverage checking function, denoted by $check_{U,N}(\sigma, U, \varnothing)$, is given in Fig. 10. Finally, we can define exhaustiveness by examining the result of the checking function.

Definition 3.4 (Exhaustiveness of UCS Terms). For any UCS term '**if** $\{\ \sigma\ \}$', we say it is exhaustive if $check_{U,N}(\sigma, U, \varnothing) = \varnothing$ where $collect(\sigma) = (U, N)$.

The result of checking function represents cases missing from the UCS term and is therefore useful in generating diagnostic messages.

3.7 Post-Processing

The normalization stage removes backtracking from the UCS terms, but the resulting core abstract syntax terms may repeat the same matches for multiple times. For example, the following is adapted from the desugared term in Fig. 6 by adjusting the order of matches.

$$
\begin{array}{l}
\textbf{if}\ \{\ \textbf{let}\ x_0 = \mathsf{Pair}(x, y)\,; \\
\quad x_0\ \textbf{is}\ \mathsf{Pair}(x_3, x_4) \rightarrow \{\ x_3\ \textbf{is}\ \mathsf{Some}(xv) \rightarrow \{\ x_4\ \textbf{is}\ \mathsf{None} \rightarrow xv\ \}\ \}\,; \\
\quad x_0\ \textbf{is}\ \mathsf{Pair}(x_7, x_8) \rightarrow \{\ x_8\ \textbf{is}\ \mathsf{None} \rightarrow \{\ x_7\ \textbf{is}\ \mathsf{None} \rightarrow 0\ \}\ \}\,; \\
\quad x_0\ \textbf{is}\ \mathsf{Pair}(x_1, x_2) \rightarrow \{\ x_2\ \textbf{is}\ \mathsf{Some}(yv) \rightarrow \{\ x_1\ \textbf{is}\ \mathsf{Some}(xv) \rightarrow +\ xv\ yv\ \}\ \}\,; \\
\quad x_0\ \textbf{is}\ \mathsf{Pair}(x_5, x_6) \rightarrow \{\ x_5\ \textbf{is}\ \mathsf{None} \rightarrow \{\ x_6\ \textbf{is}\ \mathsf{Some}(yv) \rightarrow yv\ \}\ \}\ \}
\end{array}
$$

The core abstract syntax term after normalization is as follows.

$$
\begin{array}{l}
\textbf{if}\ \{\ \textbf{let}\ x_0 = \mathsf{Pair}(x, y)\,; \\
\quad x_0\ \textbf{is}\ \mathsf{Pair}(x_1, x_2) \rightarrow \{ \\
\quad\quad x_1\ \textbf{is}\ \mathsf{Some}(xv) \rightarrow \{\ x_2\ \textbf{is}\ \mathsf{None} \rightarrow xv\,; \\
\quad\quad\quad\quad\quad\quad x_2\ \textbf{is}\ \mathsf{Some}(yv) \rightarrow +\ xv\ yv\ \}\,; \\
\quad\quad x_2\ \textbf{is}\ \mathsf{None} \rightarrow \{\ x_1\ \textbf{is}\ \mathsf{None} \rightarrow 0\ \}\,; \\
\quad\quad x_2\ \textbf{is}\ \mathsf{Some}(yv) \rightarrow \{\ x_1\ \textbf{is}\ \mathsf{None} \rightarrow yv\ \}\ \}\ \}
\end{array}
$$

We can see that x_1 is first matched against Some, and if not, x_2 is matched against None and Some before matching x_1 against None. The match 'x_1 **is** None' appears twice in different splits.

This way of organizing matches is rarely seen with traditional pattern-matching structures (e.g., Haskell's `case` expression, Scala and Rust's `match` expressions/statements, etc), which have one scrutinee followed by multiple patterns and corresponding branches. Because many compilation and optimization techniques are based on trees formed by traditional pattern-matching structures, as we will see in in Section 5.3, the normalized core abstract syntax terms may become less amenable to those techniques. A manageable solution is to lift up 'x_1 **is** None', turning it into the successive match of 'x_1 **is** Some', and push down two matches on x_2.

The idea is fulfilled in the post-processing stage, the final step of the UCS translation. The stage gathers together matches on the same scrutinee by adjusting the order of certain pattern matching heuristically. It can make the program better suited to traditional pattern matching structure and preserve the semantics of the program.

In this case, the post-processing stage transforms the normalized term into the term below.

$$
\begin{array}{l}
\textbf{if}\ \{\ \textbf{let}\ x_0 = \mathsf{Pair}(x, y)\,; \\
\quad x_0\ \textbf{is}\ \mathsf{Pair}(x_1, x_2) \rightarrow \{ \\
\quad\quad x_1\ \textbf{is}\ \mathsf{Some}(xv) \rightarrow \{\ x_2\ \textbf{is}\ \mathsf{None} \rightarrow xv\,; \\
\quad\quad\quad\quad\quad\quad x_2\ \textbf{is}\ \mathsf{Some}(yv) \rightarrow +\ xv\ yv\ \}\,; \\
\quad\quad x_1\ \textbf{is}\ \mathsf{None} \rightarrow \{\ x_2\ \textbf{is}\ \mathsf{None} \rightarrow 0\,; \\
\quad\quad\quad\quad\quad\quad x_2\ \textbf{is}\ \mathsf{Some}(yv) \rightarrow yv\ \}\ \}\ \}
\end{array}
$$

The term first determines x_0 is Pair, then determines whether x_1 is Some or None, and finally matches x_2 against Some and None. It is clear that the term is identical to Fig. 9. Since each scrutinee is matched against corresponding patterns consecutively, the term can be translated to traditional pattern-matching expressions easily and handled by existing optimization strategies.

In conclusion, the post-processing stage eliminates duplicated matches and congregate matches on the same scrutinees. The post-processing stage is implemented heuristically. Our implementation

lifts matches whose scrutinees are closer to the topmost split of the UCS term. It could be devised differently and may produce more optimal results with different reordering strategies.

3.8 Correctness of Translation

We now introduce the semantic preservation property of the translation, which says that the normalized UCS term should be evaluated to the same value as the original UCS term does.

Theorem 3.5 (Semantic Preservation of Translation). *For all term t, if $t \overset{*}{\rightsquigarrow} v$, then $\mathcal{N}[\![t]\!] \overset{*}{\rightsquigarrow} \mathcal{N}[\![v]\!]$.*

Because the definition of $\mathcal{N}[\![t]\!]$, which is given in the appendix, is mostly congruence rules, the core of this proof is to demonstrate the correctness of normalization of splits, which is formulated by the following lemma. Its complete proof can be found in the appendix.

Lemma 3.6 (Correctness of normalization of splits). *For all split σ and substitution $\phi = \overline{x_i \mapsto v_i}^{i}$, if $[\phi]$ **if** $\{ \sigma \} \overset{*}{\rightsquigarrow}$ **if** $\{$ **else** $t \}$, then $[\mathcal{N}[\![\phi]\!]]$ **if** $\{ \mathcal{M}[\![\sigma]\!] \} \overset{*}{\rightsquigarrow}$ **if** $\{$ **else** $\mathcal{N}[\![t]\!] \}$.*

4 Implementation

We now briefly describe important practical aspects of the UCS and how it was implemented as part of the MLscript programming language. Note that our implementation will be submitted to the artifact evaluation process and will be made available online.

4.1 Parsing

While conditional splitting might appear unconventional, it is neither ambiguous nor difficult to parse. Our parser uses a straightforward recursive descent implementation for better recovery and error messages. When parsing sub-expressions, we return an `Either[Term, ThenElseBlock]` value, where the latter stands for a block of **`let`**, **`then`**, and an optional **`else`** clause *or* an expression that ends with such nested blocks. When we find a `ThenElseBlock` instead of a `Term`, depending on where the sub-expression was parsed, we either yield an error (if this position does not accept then-else blocks) or we propagate the then-else block out to the outer expression, until we finally reach an enclosing **`if`**. In the end, the parser generates an abstract syntax tree quite close to the source language of Fig. 2.

4.2 Syntax Extensions

We have designed and implemented various syntax extensions based on the UCS. These syntax extensions only modify the source abstract syntax and do not affect the expressiveness of the core abstract syntax.

4.2.1 Boolean Test Shorthands. The "**if**" keyword and splits can be omitted if one wants to know the Boolean result of the test. For example, the term in source abstract syntax 'x **is** Some(xv) **and** $f(xv)$' is equivalent to '**if** $\{$ x **is** Some(xv) **and** $f(xv)$ **then** True ; **else** False $\}$'. Compared to traditional "and" expressions, its flexibility lies in that the preceding predicate can match expressions and bind variables that are accessible in the subsequent predicate.

4.2.2 Alias Pattern. The introduction mentions the keyword **as**, which binds the matched value of arbitrary sub-patterns to a variable flexibly. For example, pattern 'Some(Pair(x, y) **as** p)' binds the value matched by the sub-pattern 'Pair(x, y)' to p. We did not mention it in the formalization because it is relatively trivial to implement. Recall that we assign the scrutinees of sub-patterns to temporary variables during desugaring. Thus, we can just replace these temporary variables with user-defined alias variables.

4.2.3 Tuple Pattern. In the paper, we represent 2-tuples as constructor Pair, but in MLscript, the values of tuples of different length are constructed directly with tuple literals, e.g., `[1, 2, 3]`, rather than using a separate constructor. Therefore, we introduced an isomorphic pattern syntax for matching tuples. The example in Fig. 5 can be rewritten in a more concise manner as follows.

$$\begin{array}{lll}\textbf{if}\ \{\ [\,x\,,\ y\,]\ \textbf{is}\ \{\ [\,\mathsf{Some}(xv)\,,\ \mathsf{Some}(yv)\,] & \textbf{then}\ xv + yv\,; \\ \qquad\qquad\quad [\,\mathsf{Some}(xv)\,,\ \mathsf{None}\,] & \textbf{then}\ xv\,; \\ \qquad\qquad\quad [\,\mathsf{None}\,,\ \mathsf{Some}(yv)\,] & \textbf{then}\ yv\,; \\ \qquad\qquad\quad [\,\mathsf{None}\,,\ \mathsf{None}\,] & \textbf{then}\ 0\ \}\ \}\end{array}$$

4.3 Type Checking

We perform type checking *after* translation into the *core* MLscript language (which has only a flat pattern matching expression form). Specifically, the UCS translation is completed in the pre-typing stage, which occurs before typing. Only name resolution is completed during this stage, meaning that the typing information accessible to the UCS is very limited. This greatly simplifies the MLscript implementation, as it means that the type checker does not have to be aware of the UCS, which can be considered a purely syntactic abstraction. We found that this works well in practice although type error messages could be improved by performing type checking *before* UCS normalization, which we plan to do in the future. Because most UCS errors are reported before type checking, it is not yet clear how the MLscript compiler will need to be modified to properly handle GADTs, where type information usually can influence exhaustiveness checking. We leave this as an open question, to be addressed in future work.

4.4 Inheritance and Overlapping Constructors

The language presented in this paper does not support inheritance, which simplifies the normalization algorithm, as we can assume any two constructors are disjoint. However, our implementation does support inheritance and has to consider the case where constructors may overlap. For instance, assuming constructor Int is the base of integer constants, consider the following source term:

$$\textbf{if}\ \{\ x\ \textbf{is}\ \{\ \mathsf{Int}\ \textbf{and}\ \{f(x)\ \textbf{then}\ e_1\}\,;\ 0\ \textbf{then}\ e_2\ \}\ \}$$

Since the 0 constructor overlaps with (and is actually included in) constructor Int, the second branch e_2 should be merged into the first branch. Thus, the normalization of the desugared term above should produce the following term:

$$\textbf{if}\ \{\ x\ \textbf{is}\ \mathsf{Int} \to \{f(x)\ \textbf{is}\ \mathsf{True} \to \mathcal{D}[\![e_1]\!]\,;\ x\ \textbf{is}\ 0 \to \mathcal{D}[\![e_2]\!]\ \}\ \}$$

In our implementation, because of singe-inheritance, when two constructors overlap, one always include the other. So specialization only needs to reason about inclusion between constructors. For cases of the form $\mathcal{S}^{+}_{x_1\ \textbf{is}\ C_1(\overline{x})}[\![x_1\ \textbf{is}\ C_2(\overline{x}) \to \{\,\sigma_1\,\}\,;\ \sigma_2]\!]$, we should keep σ_1 if $C_2 <: C_1$, and for cases of the form $\mathcal{S}^{-}_{x_1\ \textbf{is}\ C_1(\overline{x})}[\![x_1\ \textbf{is}\ C_2(\overline{x}) \to \{\,\sigma_1\,\}\,;\ \sigma_2]\!]$, we should discard σ_1 if $C_1 <: C_2$.

4.5 Subtyping and Extensible Variants

Note that the core language of MLscript is based on MLstruct, which supports something akin to extensible variant pattern matching. One can define constructors independently of each other, without assuming that they are parts of some specific parent algebraic data type definition. Consider the following program, which implements a `map` function over the `Some` constructor.

```
class Some[out A](value: A)
module None

fun mapSome(f, opt) = if opt is Some then f(opt) else opt
```

Where a **module** declaration defines a parameterless class and its singleton instance, here the usual `None` constructor for option types. The inferred type of function `mapSome` is

$$\text{mapSome} : \forall \alpha, \beta.\ (\alpha \to \beta,\ \alpha\ \&\ \text{Some} \mid \beta\ \&\ \sim\text{Some}) \to \beta$$

Here, type 'Some' is the nominal tag of class `Some`, representing the identity of that class. The negation type '~Some' is the supertype of all types disjoint from `Some`, meaning all types known not to include instances of the `Some` class among their values. Type variables α and β represent whatever is coming along with 'Some' or '~Some', respectively, whichever the second parameter `opt` ends up being. In other words, `mapSome` accepts as `opt` either an instance of `Some` that also has type α, or an instance of any other class that also has type β. Moreover, `mapSome` only puts the former through its first argument `f`, which has type $\alpha \to \beta$, finally returning values of type β.

This makes defining and extending ad-hoc algebraic data types easy. For example, we can extend `Some` to add a field to represent some extra payload:

```
class SomeAnd[A, P](value: A, payload: P) extends Some[A](value)
```

Strictly speaking, class `None` is not a true singleton: it supports class extensions, as shown below:

```
class Err(msg: Str) extends None
```

Suppose the user constructs a term that can be an instance of either `SomeAnd` or `Err`. Such a term can still be passed to our original `mapSome` function *and* it retains precise type information:

```
let arg = if ⟨arbitrary condition⟩ then SomeAnd("Hello", "World")
                                  else Err("error: ...")
mapSome(arg, x => { msg: x.value ++ x.payload }).msg  // : Str
```

Another example is `Either[A, B]`, which represents a value of one of two possible types (a disjoint union). There are cases where it is natural to extend this disjoint union with a `Both` constructor that contains both possible types at the same time. In MLstruct, we can use the union of types `Either` and `Both` as defined below:

```
class Both[A, B](left: A, right: B)

fun foo(params): Either[Int, Str] | Both[Int, Str] = ...
```

which can be used in a pattern matching expression by first teasing out the `Both` constructor and then using the leftover value in the default branch like a normal `Either` value:

```
if foo(args) is
  Both(l, r) then l.n + r.m
  else Either.fold(b, x => x.n, x => x.m)
```

4.6 Destructing Constructor Parameters

Classes in MLscript are defined with named constructor parameters. For example, a class for 2-tuples can be declared as **class** `Pair[A, B](first: A, second: B)` and an instance can be constructed by `Pair(0, 1)`, which is consistent with the constructor in the source and core syntax.

The core abstract syntax supports destructuring classes and extracting these constructor arguments at the same time. However, the pattern-matching syntax of core MLscript corresponds to *class-instance matching* without class parameter bindings (e.g., **case** x **of** $\{ C_1 \to t_1\ C_2 \to t_2 \}$).

To solve the discrepancy, our current implementation generates an `unapply` function for each class. Given a class instance, its `unapply` function returns the arguments used to construct this instance. For example, if a scrutinee is determined to be an instance of `Pair`, it will be applied to `Pair.unapply`, then the extracted tuple elements will be bound to corresponding variables. In this way, the UCS translation does not need to know the specific definition within each class.

4.7 Error Handling

The errors in the UCS translation mainly come from two stages: parsing and exhaustiveness checking. Other stages only generate warnings, for example, redundancy mentioned in Section 3.6.1. Moreover, failures in the UCS translation are non-fatal. Expressions that cause parsing errors are discarded without further translation. If errors occur during exhaustiveness checking, the desugared and normalized terms are still be passed to compiler passes after the UCS translation. In our implementation, the results of the exhaustiveness checking are compiled into warning and error messages that inform users of the missing cases.

5 Comparison With Existing Approaches

In this section, we compare the UCS with other conditional syntaxes and pattern-matching extensions in terms of three aspects: expressiveness, type safety, and efficient compilation. At the end of this section, we briefly showcase the practical and unique aspects of the UCS that other pattern matching syntaxes cannot achieve.

5.1 Expressiveness

In early functional programming languages, for example, ISWIM [Landin 1966], PAL [Evans 1968], and SASL [Turner 1979], pattern matching refers to the destruction of expressions into their components using 'let' bindings, as in '`let (a, y) = decons(x)`'. Burstall, in an early work on structural induction, first proposed `cases` expressions, as a syntactic sugar for chained if-then-else expressions, to conduct case analysis on data types [Burstall 1969]. Burstall stated that the `cases` expression were partly suggested by the **`case`** switch on type introduced into BCPL by Richards [1967]. During the development of NPL, Darlington was inspired by Kleene's recursion equations and replaced case expressions with equational function definitions [Burstall and Darlington 1977], where functions are defined by a sequence of one or more equations, and each equation specifies the function over some subsets of the possible argument values that are described by patterns on the left-hand side of the equation. Following that, NPL evolved into HOPE, the first polymorphic functional programming language with algebraic data types [Burstall et al. 1980].

The earliest version of ML did not have pattern matching, and structures were analyzed using conditions, tests, and projection functions [Milner 1978]. When designing Standard ML, Milner added HOPE-style algebraic data types, and pattern matching over data type constructors, as well as equational function definitions based on LCF/ML [Milner 1984]. Since then, pattern matching has become an indispensable part of the ML family of languages [MacQueen et al. 2020].

Guards first appeared in KRC, a language based on SASL, which allows adding extra conditions to the function equations in a way that patterns cannot describe [Turner 1981]. Miranda, another language influenced by SASL, also inherited pattern matching using equational function definitions and extended the syntax of guards [Turner 1986]. Wadler proposed *views* as a language feature for abstracting patterns based on Miranda. The definition of views is bidirectional as they serve not only as patterns in pattern matching but also as constructors in expressions [Wadler 1987b].

Haskell, which integrates the pattern matching constructs of all the aforementioned languages, not only inherits equational function definitions, but also provides conditional expressions, guards, and **`case`** expressions. Many novel pattern-matching extensions are implemented based on Haskell.

Haskell's *view patterns*[10] partially implements Wadler's views, which allows passing values to be pattern-matched through functions and matching on the result [Burton et al. 1996]. The main limitation of view patterns is that they cannot be shared easily between patterns — while they have a heuristic for avoiding repeated work, it is quite limited (it does not always apply) and in any

[10] https://gitlab.haskell.org/ghc/ghc/-/wikis/view-patterns

GHC Haskell:

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

Scala 3 (proposed extension):

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

MLscript:

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

MLscript (after desugaring):

```
case e of
  Var(x) →
    let tmp0 = context.get(x)
    case tmp0 of
      Some(IntVal(v))  → Left(v)
      Some(BoolVal(v)) → Right(v)
  Lit(IntVal(v))  → Left(v)
  Lit(BoolVal(v)) → Right(v)
  ...
```

Fig. 11. More complex example in Haskell, Scala 3 (with a proposed extension), and MLscript (with desugaring).

case, still incurs textual repetition. We argue that views arise from the fundamental inability of traditional pattern matching syntaxes to interleave computations with matches, which is already relaxed in the UCS. In the following example (adapted to MLscript syntax), we use a GHC-style view called '`np k`' to match integers that are not less than the given `k`, which is a standard use case.

```
fun np(k)(n) = if k <= n then Some(n - k) else None
fun fib(n) = case n of
  0 → 1
  1 → 1
  (np(2) → Some(n')) → fib(n' + 1) + fib(n')
```

This can be straightforwardly translated into the UCS as follows:

```
fun fib(n) = if n is
  0 then 1
  1 then 1
  m and np(2)(m) is Some(n') then fib(n' + 1) + fib(n')
```

McBride and McKinna [2004] studied pattern matching view in dependent type systems and extended it with a construct that allows users to specify the order of case splitting explicitly.

Haskell's pattern guards are strictly more powerful than the alternatives above, in that, they can execute arbitrary logic during pattern matching and match on the resulting intermediate values. The following is an example from Simon Peyton Jones's notes on pattern guards.

```
lookup :: FiniteMap → Int → Maybe Int
addLookup env var1 var2
  | Just val1 ← lookup env var1
  , Just val2 ← lookup env var2
  = val1 + val2
{-...other equations...-}
```

This can be rewritten using the UCS, resulting in a boost of readability.

```
fun lookup: FiniteMap → Int → Option[Int]
fun addLookup(env, var1, var2) = if
  lookup(env, var1) is Some(val1)
    and lookup(env, var2) is Some(val2)
    then val1 + val2
  // ... other cases
```

The main weakness of Haskell-style pattern guards is that the matching structure is still *one-level*: different pattern branches cannot share these intermediate values and matches[11]. Unlike the UCS, which may *nest* sub-matches, resulting in several **then** clauses instead of just one. Fig. 11 shows a comparison between the two approaches on a non-trivial example.

Pattern synonyms [Pickering et al. 2016] let users abstract over patterns in a similar way as functions abstract over values and type synonyms abstract over types. The early concept of pattern synonyms alongside transformational patterns, which serves as a generalization of pattern guards and pattern qualifiers, was proposed by Erwig and Peyton Jones [2001]. An earlier similar study on abstracting existing patterns on Standard ML had also been proposed by Aitken and Reppy [1992]. Despite the fact that the UCS seamlessly allows mixing patterns with computations—making abstracting through functions sufficient for most purposes—we may still consider adding a way to provide abstractions over patterns in the future. Other **Haskell pattern matching extensions** were proposed by Servadei [2018], among which nonlinear pattern variables.

Active patterns in F# [Erwig 1997] and Scala's extractors serve a similar purpose as view patterns and let programmers factor custom matching logic in the clothes of a normal pattern. We argue that the need for these abstractions is mainly due to the pattern matching syntax being too restrictive. Again, the need for view patterns and extractors can be somewhat obviated by interleaving normal computations within conditionals and performing cascading matches, as done with the UCS. Nevertheless, we are still considering adding user-defined patterns and extractors to MLscript for the extra concision and readability they can provide.

Pattern alternatives in languages like OCaml let programmers use disjunctions of sub-patterns which may contain pattern variables, a quite powerful capability. Currently, the UCS only includes operator '**and**'. We plan to support the '**or**' operator as pattern alternatives in the UCS, enabling more concise pattern matching. For example, this would allow rewriting the above `fib` definition to the slightly more concise:

```
fun fib(n) = if n is
  0 or 1 then 1
  m and np(2)(m) is Some(n') then fib(n' + 1) + fib(n')
```

Rust's[12] **and Swift's**[13] **'if-let' forms** let programmers write '**if**' statements that perform matching at the same time and bind the extracted values within the **true** branch, as in '**if let** `Some(v) = { ... v ... }`'. Moreover, Swift allows more than one match in '**if let**'. Rust also supports a "let-else" construct which allows implementing early return, to avoid the need for nested **if** statements. For instance, one can write the Rust equivalent of:

```
fun foo(x) =
  if let Some(value) = x else return None
  ... value ... // Here, `value` is in scope
```

[11] Interestingly, the Haskell community has been fruitlessly discussing the possibility of adding this feature for a long time, as this 2015 response on the StackOverflow question "Is it possible to nest guards in Haskell?" attests: "No, you can't. We all want it, but nobody can come up with a sensible syntax." https://stackoverflow.com/a/34168666/1518588

[12] https://doc.rust-lang.org/rust-by-example/flow_control/if_let.html

[13] https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html

A **Scala 3 proposed extension** to pattern matching[14] would allow nesting pattern matches in pattern guards, as shown in Fig. 11, fixing the limitation of Haskell's pattern guards. Compared with the UCS, this syntax does not allow the binding of intermediate values between cases.

Many **Lisp dialects like Racket**[15] have supported expressive conditional and pattern-matching structures, such as the `cond` and **`match`** macros. Racket also supports pattern abstractions through extending pattern matching definitions and defining transparent structs similar to pattern synonyms. The Racket community could take inspiration from some aspects of the flexible syntax of the UCS.

The popular "ssreflect" Coq library also implements an "if _ is _ then _ else _" notation that is quite related to the UCS.[16] Many object-oriented languages have also begun gradually adopting pattern matching, taking inspiration from multi-paradigm languages like Scala [Odersky et al. 2004]. C# extended **`switch`** expressions (statements) to pattern matching and introduced **`is`** operator for type-testing and cast as well as a variety of patterns [ECMA International 2023]. The most interesting one is the relational patterns, which allow partially applied built-in relational operators to be used as patterns and support logical combinations with other patterns. The conditional splits of the UCS can achieve a similar effect.

```
string Classify(double value) =>
  value switch {
    < -4.0     => "Too low",
    > 10.0     => "Too high",
    double.NaN => "Unknown",
    _          => "Acceptable" }
```

```
fun classify(value) =
  if value
    < -4.0 then "Too low"
    > 10.0 then "Too high"
    == NaN then "Unknown"
    else        "Acceptable"
```

Java 16[17] recently introduced a basic form of pattern matching through `instanceof`, as in '**`if`** `(obj instanceof String s) { ... s ... }`' and Java 17 extended the **`switch`** statement to support pattern matching[18]. Python introduced **`match`** statements for pattern matching since version 3.10[19]. TC39, the committee which defines ECMAScript (JavaScript), has a proposal currently at the earliest stage that aims to introduce a pattern-matching syntax including guards[20]. The C++ community has long been discussing how to incorporate pattern matching, as there are proposals within the community which introduce `inspect` statements[21] and **`as`**/**`is`** keywords[22] . Although C++ has been considering pattern matching a promising addition[23], it is still not officially a part of the C++ standard as of now. Solodkyy et al. [2013] presented a library for functional-style pattern matching through C++ concepts. Generally speaking, these newly proposed pattern matching syntaxes are similar to a large extent. Their main contribution is limited to providing functional-style flat pattern matching with a rich set of patterns and possibly guards, but they neither provide abstractions over patterns nor do they transcend the flexibility of the UCS.

5.2 Type System

Pattern matching provides a method to perform case analysis on values according to definitions in the type system. Additionally, pattern matching prevents the representation of cases that do not

[14] https://github.com/lampepfl/dotty-feature-requests/issues/301
[15] https://docs.racket-lang.org/reference/match.html
[16] https://coq.discourse.group/t/if-is-then-else-notation/437
[17] JEP 394 — Pattern Matching for **`instanceof`**: https://openjdk.org/jeps/394.
[18] JEP 406 — Pattern Matching for **`switch`** (Preview): https://openjdk.org/jeps/406.
[19] PEP-0622 — Structural Pattern Matching: https://peps.python.org/pep-0622/.
[20] ECMAScript Pattern Matching: https://tc39.es/proposal-pattern-matching/.
[21] P1371 — Pattern Matching: https://wg21.link/p1371r2.
[22] P2392 — Pattern matching using **`is`** and **`as`**: https://wg21.link/p2392r1.
[23] P2411 — Thoughts on pattern matching: https://wg21.link/p2411r0.

conform to the type system. Even though the method we propose in this article is agnostic to the type system, the language that supports the UCS still should possess a capable type system.

Occurrence typing was introduced by Tobin-Hochstadt and Felleisen [2008] for Typed Scheme and later incorporated in TypeScript and Flow, where it is known as *flow typing*. This approach allows the types of variables to be locally refined based on path conditions encountered in the program. The connection between this and our approach is in how they interact with conditionals and boolean operators to thread through the information that is discovered during the test. The difference is that in our approach, conditional tests may perform proper pattern matching with `is`, binding new variables, which are then available in the corresponding parts of the conditions.

Moreover, similarly to occurrence typing, MLscript enables matching one scrutinee against multiple types or patterns, refining the scrutinee's type to be the intersection of all corresponding types. Conversely, if a scrutinee fails to match against a constructor, the type of the scrutinee is intersected with the negation of the constructor. This kind of reasoning necessitates a type system that supports intersection and negation types. MLstruct, the core language of MLscript, supports ML-style principal type inference generalized with well-behaved forms of union and intersection types as well as pattern matching on single-inheritance class hierarchies [Parreaux and Chau 2022].

Semantic subtyping, unlike the methods mentioned above that define subtyping syntactically, defines types and the subtype relationships between them using a set-theoretic model [Castagna and Frisch 2005]. Type systems constructed with this method can form the Boolean algebra of intersection, union and negation types [Castagna 2024], and can implement the characteristics of occurrence typing [Castagna et al. 2022].

5.3 Compilation

The most straightforward approach to compiling pattern matching is to sequentially examine each case, testing all conditions within each case in order. The method is inefficient because the same condition may be tested multiple times. Practical implementations rely on factorizing patterns and compiling patterns into automata that perform fewer redundant tests. A common choice are tree automata, where the internal nodes correspond to flat pattern matching expressions (e.g., `case` expressions) and leaves correspond to the outcome terms of branches. Depending on whether backtracking is allowed, approaches can be categorized into two kinds. One is based on deterministic automata, often referred to as *decision trees*. The approach is more efficient because it ensures that each condition is tested at most once at the price of duplicated code. In the worst case, the size of deterministic automata is exponential in the size of patterns [Maranget 1994]. The other is based on backtracking automata, which detour in the automata and test the same conditions that occur at different places of the automata more than once before reaching the branch outcome.

The common practice of generating automata fall into three categories. The first approach regards a pattern matching expression as a *clause matrix*, where rows correspond to cases and columns correspond to scrutinees. Thus, each element represents the pattern that the corresponding scrutinee is matched against in the corresponding case. In each step, the algorithm selects a scrutinee and creates an internal node that matches the scrutinee against patterns in the corresponding column. Rows are grouped based on pattern distinctness in the chosen column. forming one or more sub-matrices, and the sub-patterns are appended to each sub-matrix. The operation is recursively applied to each sub-matrix until the matrix has one column, then a leaf node representing the outcome of the case is created. The approach was first implemented in the HOPE compiler and described by Cardelli [1984]. Based on this framework, Le Fessant and Maranget [2001] introduced several optimization techniques in compilation to backtracking automata, including reordering non-overlapping cases, using exhaustiveness information, and applying control flow optimization. Maranget [2008] introduced *necessity*, which indicates that a particular column of the input must

be examined in all possible decision trees to determine if a specific row matches. The necessity information can serve as a useful basis for designing heuristics.

The second one is based on tree transformations. Because pattern matching expressions can represent tree automata, pattern matching can be also compiled by directly transforming syntax trees. Augustsson [1984] proposed an algorithm that compiles nested patterns in Lazy ML [Friedman and Wise 1976] into backtracking automata made of **case** expressions that only have one-level constructor patterns. Barrett and Wadler [1987] derived a functional program to transform equational function definitions to decision trees made of **case** expressions, by using higher-order functions and mathematical properties. Wadler [1987a] further elaborated on a functional approach to compiling equational function definitions with several optimizations in Miranda.

The last one relies on direct compilation to directed acyclic graph (DAG) automata. Pettersson [1992]'s approach treats patterns as regular expressions and reduces duplicated branches and redundant tests by optimizing the finite automaton that is built to recognize the regular expression. Nedjah and de Macedo Mourelle [2002] presented a method that can directly generate minimised DAG matching automata without constructing the tree automaton first.

Balland and Moreau [2006]'s method differs from other approaches by separating optimization from compilation by first compiling pattern matching into a intermediate language in the most straightforward way, and then optimizing the intermediate program using transformation rules.

Previous approaches face a key decision: which scrutinee to inspect next. This choice affects both efficiency and automata size. MacQueen and Baudinet [1985] pointed out that building an optimal tree automaton with the minimal number of tests is equivalent to *the dispatching problem*, which is NP-complete. Scott and Ramsey [2000] found that while automata sizes were similar across heuristics, performance varied significantly. Most methods decide the order during compilation. Adpative automata, proposed by Sekar et al. [1995], employs a flexible order decided during runtime based on the state of inspected scrutinees.

Apart from the heuristics above, duplicated nodes are also a common problem in compiling to tree automata. The algorithm of Augustsson [1985] produces DAGs that share default cases. An alternative method is to generate a decision tree first, and then compress it using hash consing [Maranget 2008]. Nedjah [1998] proposed an efficient method to identify duplicated sub-automata and build graph automata without the need to construct tree automata first.

In our approach, the normalization stage builds non-backtracking tree automata that match scrutinees according to the textual order. However, unlike the traditional pattern matching, in different branches, the UCS can match the same set of scrutinees in different textual orders (see the example in Section 3.7). Thus, matches on the same scrutinee might be scattered. The post-processing stage can heuristically adjusts the order of matches without affecting their semantics, but due to interleaved computations, it may not always ensure that the generated automata always gather matches on the same scrutinee. Therefore, correctly compiling the UCS into more efficient automata remains an issue and we consider exploring more heuristics in our future work.

5.4 Practicality

Finally, we would like to emphasize three practical benefits that the UCS can easily achieve but are beyond the capabilities of other pattern matching syntaxes.

Fixing the Right Drift Problem. This problem often affects languages with pattern matching but unable to test several conditions in parallel. Consider the left-hand-sides of Fig. 12 and how the nesting of matches and conditionals on unrelated operands creates a shift to the right. The right-hand side of Fig. 12 shows more concise and non-drifting MLscript versions of the same code.

```
normalize(tp1) match
  case Bot => Bot
  case tp1_n =>
    normalize(tp2) match
      case Bot => Bot
      case tp2_n =>
        merge(tp1_n, tp2_n) match
          case Some(tp) => tp
          case None => glb(tp1_n, tp2_n)
```

```
if
  let tp1_n = normalize(tp1)
  tp1_n is Bot  then Bot
  let tp2_n = normalize(tp2)
  tp2_n is Bot  then Bot
  let m = merge(tp1_n, tp2_n)
  m is Some(tp) then tp
  m is None     then glb(tp1_n, tp2_n)
```

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

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

Fig. 12. Rightward drift examples and duplicated defaults in Scala 3 (left) and UCS versions (right).

Minimizing Repetition of "Else" Branches. This issue frequently occurs when people use pattern-matching and if-then-else expressions successively. Even though the "else" branches of those expressions are identical, their different syntactic structures prevent people from merging them. Consider the example on the left side of the second row of Fig. 12, which repeats `Left(name)` three times. The rewrite using the UCS sidesteps this issue by connecting those pattern matching and tests with '**and**', then the final "else" branch handles all failure cases.

Non-Disruptive Refactoring of Conditionals. In Section 2, we presented an example where we could use `Map.find_opt` but needed to rewrite the entire expression. This point is more evident in complex and larger expressions. Continuing with the second row of Fig. 12, assume that we allow the prefix of `name` to have more than one underscores. We would typically use `dropWhile` to remove the prefix. But `dropWhile` does not return `Option` values, we need to rewrite the second-level pattern matching in the left-hand Scala example accordingly. Specifically, we no longer need '`name.tailOption` **`match`** ...'. Instead, we introduce a new binding '**`val`** `tailOption = name.dropWhile(_ == '_')`', and an if expression to ensure that `name` is not an empty string. Rewriting the MLscript example on the right is much easier, as we only need to change the second line to:

```
and name.dropWhile(_ == '_') is nameTail
and nameTail.length > 0
```

The flexibility of the UCS allows us to refactor the code with minimal cost and structural impact.

6 Conclusions and Future Work

We presented the UCS, a novel conditional syntax that generalizes traditional pattern-matching and if-then-else expressions and supports the flexible binding of intermediate variables and computations. We defined a rich source abstract syntax and a concise core abstract syntax suitable for transformation and reasoning, along with a desugaring from the former to the latter. We also defined a normalized abstract syntax and a translation algorithm from core to normalized syntax. We formally proved the correctness of this translation in terms of semantics preservation. We also discussed coverage checking and post-processing aspects important to the UCS in practice.

As presented, our approach is type-system-agnostic: it does not require nor make any strong assumptions on an underlying type system of the host programming language. Indeed, our implementation in MLscript desugars and translates the UCS terms before the type checker even looks at them. In the future, we will investigate ways to more closely integrate type checking and type inference with the UCS transformation process, which could lead to better error messages and more efficient compilation. As part of future research goals, we also want to investigate suitable optimization algorithms for the UCS, which may be important given that the UCS has greater expressiveness than traditional conditional syntax, preventing the direct reuse of existing optimization techniques. Moreover, we want to continue expanding the supported UCS source syntax, further harnessing its expressiveness. For example, we intend to include support for the '**`or`**' operator in pattern flowing in a similar way to the '**`and`**' operator, and to support pattern alternatives.

We were delighted to learn that since our initial informal presentation of the UCS to the Programming Languages community, several researchers and language designers have expressed interest in adding support for it in their own programming languages. For instance, the Effekt language [Brachthäuser et al. 2022, 2020] has recently added support for pattern guards inspired by the UCS with '**`and`**' and '**`is`**' operators[24] and an OCaml maintainer has let us know that he had been investigating the implementation of a similar feature in OCaml.

Acknowledgement

We want to thank the anonymous reviewers for their precious feedback. Their detailed suggestions and candid comments have greatly helped us improve the quality and clarity of our work.

Data-Availability Statement

The implementation is available on Zenodo [Cheng and Parreaux 2024] and its latest version can be found at https://github.com/hkust-taco/ucs. This implementation includes a live programming environment that displays the intermediate terms from each stage as well as coverage checking, type checking, and evaluation results. This is also accessible as a web demo at https://ucs.mlscript.dev. A technical report version with appendices and complete proofs is available at https://lptk.github.io/ucs-paper.

[24] https://effekt-lang.org/examples/matching.html

References

William Aitken and John H. Reppy. 1992. *Abstract Value Constructors: Symbolic Constants for Standard ML.* Technical Report. Cornell University. https://hdl.handle.net/1813/7130

Lennart Augustsson. 1984. A Compiler for Lazy 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, 218–227. https://doi.org/10.1145/800055.802038

Lennart Augustsson. 1985. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 368–381.

Emilie Balland and Pierre-Etienne Moreau. 2006. *Optimizing pattern matching compilation by program transformation.* Technical Report. 19 pages. https://inria.hal.science/inria-00001127

Hendrik Pieter Barendregt. 1984. Chapter 2 - Conversion. In *The Lambda Calculus*, H.P. BARENDREGT (Ed.). Studies in Logic and the Foundations of Mathematics, Vol. 103. Elsevier, 22–49. https://doi.org/10.1016/B978-0-444-87508-2.50010-1

Geoff Barrett and Philip Wadler. 1987. Derivation of a Pattern-Matching Compiler. (1987). https://homepages.inf.ed.ac.uk/wadler/papers/pattern/pattern.pdf

Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 76 (Apr 2022), 30 pages. https://doi.org/10.1145/3527320

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (Nov 2020), 30 pages. https://doi.org/10.1145/3428194

R. M. Burstall. 1969. Proving Properties of Programs by Structural Induction. *Comput. J.* 12, 1 (02 1969), 41–48. https://doi.org/10.1093/comjnl/12.1.41 arXiv:https://academic.oup.com/comjnl/article-pdf/12/1/41/885019/12-1-41.pdf

R. M. Burstall and John Darlington. 1977. A Transformation System for Developing Recursive Programs. *J. ACM* 24, 1 (Jan 1977), 44–67. https://doi.org/10.1145/321992.321996

R. M. Burstall, D. B. MacQueen, and D. T. Sannella. 1980. HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming* (Stanford University, California, USA) *(LFP '80)*. Association for Computing Machinery, New York, NY, USA, 136–143. https://doi.org/10.1145/800087.802799

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

Luca Cardelli. 1984. Compiling a functional language. 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, 208–217. https://doi.org/10.1145/800055.802037

Giuseppe Castagna. 2024. Programming with union, intersection, and negation types. arXiv:2111.03354 [cs.PL]

Giuseppe Castagna and Alain Frisch. 2005. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Lisbon, Portugal) *(PPDP '05)*. Association for Computing Machinery, New York, NY, USA, 198–208. https://doi.org/10.1145/1069774.1069793

Giuseppe Castagna, Victor Lanvin, Mickaël Laurent, and Kim Nguyen. 2022. Revisiting occurrence typing. *Science of Computer Programming* 217 (May 2022), 102781. https://doi.org/10.1016/j.scico.2022.102781

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. 2015. Polymorphic Functions with Set-Theoretic Types: Part 2: Local Type Inference and Type Reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 289–302. https://doi.org/10.1145/2676726.2676991

Luyu Cheng and Lionel Parreaux. 2024. *The Ultimate Conditional Syntax (Artifact).* https://doi.org/10.5281/zenodo.13621222

Stephen Dolan. 2017. *Algebraic subtyping.* Ph. D. Dissertation.

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. *ACM SIGPLAN Notices* 52, 1 (Jan. 2017), 60–72. https://doi.org/10.1145/3093333.3009882

ECMA International. 2023. ECMA-334: C# Language Specification (7th Edition). https://ecma-international.org/publications-and-standards/standards/ecma-334/ Accessed: 2024-08-05.

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

Martin Erwig and Simon Peyton Jones. 2001. Pattern Guards and Transformational Patterns. *Electronic Notes in Theoretical Computer Science* 41, 1 (2001), 3. https://doi.org/10.1016/S1571-0661(05)80540-7 2000 ACM SIGPLAN Haskell Workshop (Satellite Event of PLI 2000).

Arthur Evans. 1968. PAL—a language designed for teaching programming linguistics. In *Proceedings of the 1968 23rd ACM National Conference (ACM '68)*. Association for Computing Machinery, New York, NY, USA, 395–403. https://doi.org/10.1145/800186.810604

Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*. ACM, New York, NY, USA, 268–277. https://doi.org/10.1145/113445.113468 event-place: Toronto, Ontario, Canada.

Daniel P. Friedman and David S. Wise. 1976. *CONS should not Evaluate its Arguments.* Technical Report TR44. https://help.luddy.indiana.edu/techreports/TRNNN.cgi?trnum=TR44 In I. S. Michaelson and R. Milner (eds.), Automata, Languages and Programming, Edinburgh University Press, Edinburgh (1976), 256–284.

Robert Harper. 2016. *Practical foundations for programming languages.* Cambridge University Press.

P. J. Landin. 1966. The next 700 programming languages. *Commun. ACM* 9, 3 (Mar 1966), 157–166. https://doi.org/10.1145/365230.365257

Fabrice Le Fessant and Luc Maranget. 2001. Optimizing Pattern Matching. *ACM SIGPLAN Notices* 36 (08 2001). https://doi.org/10.1145/507635.507641

David MacQueen and M Baudinet. 1985. Tree Pattern matching for ML. *Unpublished manuscript* (1985).

David MacQueen, Robert Harper, and John Reppy. 2020. The history of Standard ML. *Proc. ACM Program. Lang.* 4, HOPL, Article 86 (Jun 2020), 100 pages. https://doi.org/10.1145/3386336

Luc Maranget. 1994. *Two Techniques for Compiling Lazy Pattern Matching.* Research Report RR-2385. INRIA. https://inria.hal.science/inria-00074292 Projet PARA.

Luc Maranget. 2008. Compiling Pattern Matching to Good Decision Trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML* (Victoria, BC, Canada) *(ML '08)*. Association for Computing Machinery, New York, NY, USA, 35–46. https://doi.org/10.1145/1411304.1411311

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

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (Dec. 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

Nadia Nedjah. 1998. Minimal deterministic left-to-right pattern-matching automata. *SIGPLAN Not.* 33, 1 (jan 1998), 40–47. https://doi.org/10.1145/609742.609748

Nadia Nedjah and Luiza de Macedo Mourelle. 2002. Optimal Adaptive Pattern Matching. In *Developments in Applied Artificial Intelligence*, Tim Hendtlass and Moonis Ali (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 768–779.

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).

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

Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 141 (Oct 2022), 30 pages. https://doi.org/10.1145/3563304

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

Mikael Pettersson. 1992. A term pattern-match compiler inspired by finite automata theory. In *Compiler Construction*, Uwe Kastens and Peter Pfahler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 258–270.

Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern synonyms. In *Proceedings of the 9th International Symposium on Haskell* (Nara, Japan) *(Haskell 2016)*. Association for Computing Machinery, New York, NY, USA, 80–91. https://doi.org/10.1145/2976002.2976013

Martin Richards. 1967. BCPL Reference Manual. Designated Memorandum M-352 of MIT Project MAC. https://www.bell-labs.com/usr/dmr/www/bcpl.html Available at Bell Labs website.

Kevin Scott and Norman Ramsey. 2000. *When Do Match-compilation Heuristics Matter?* Report. University of Virginia, Department of Computer Science. https://doi.org/10.18130/V3GB4M

R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. 1995. Adaptive Pattern Matching. *SIAM J. Comput.* 24, 6 (1995), 1207–1234. https://doi.org/10.1137/S0097539793246252 arXiv:https://doi.org/10.1137/S0097539793246252

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

Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. 2013. Open pattern matching for C++. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences* (Indianapolis, Indiana, USA) *(GPCE '13)*. Association for Computing Machinery, New York, NY, USA, 33–42. https://doi.org/10.1145/2517208.2517222

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

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. 1979. A new implementation technique for applicative languages. *Software: Practice and Experience* 9, 1 (1979), 31–49. https://doi.org/10.1002/spe.4380090105 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380090105
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
Philip Wadler. 1987a. *The Implementation of Functional Programming Languages*. Prentice Hall, Chapter 5 — Efficient Compilation of Pattern-Matching. https://www.amazon.com/Implementation-Functional-Programming-Prentice-hall-International/dp/013453333X
P. Wadler. 1987b. 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
Hongwei Xi and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 249–257. https://doi.org/10.1145/277650.277732
Hongwei Xi and Frank Pfenning. 1999. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560

A Desugaring

In this section, we provide the complete definition of the desugaring algorithm and demonstrate some examples before and after desugaring.

A.1 Desugaring Algorithm

Fig. 13 and Fig. 14 shows the complete desugaring algorithm from source abstract syntax to core abstract syntax. $\mathcal{D}[\![e]\!]$ is desugaring algorithm's entry point, which is a straightforward term traversal function, except that it invokes $\mathcal{D}_{\mathcal{T}}$ for an **if** term. Partial term $\rho[\square]$ represents incomplete terms which contains a missing right-hand side term. Mutual recursive functions $\mathcal{D}_{\mathcal{T}}$, $\mathcal{D}_{\mathcal{P}}$, and $\mathcal{D}_{\mathcal{O}}$ desugar term splits, pattern splits, operator splits respectively. Function $\mathcal{D}_p$ desugars nested patterns to flat pattern matching. It relies on a helper function *zip* which constructs inner splits by pairing sub-scrutinees and sub-patterns.

$$\underline{\mathcal{D}\text{esugaring Terms}} \qquad \boxed{\mathcal{D}[\![e]\!] : t}$$

$$\begin{aligned}
\mathcal{D}[\![x]\!] &= x \\
\mathcal{D}[\![e_1\ e_2]\!] &= \mathcal{D}[\![e_1]\!]\ \mathcal{D}[\![e_2]\!] \\
\mathcal{D}[\![C(\overline{e_i}^{\,i})]\!] &= C(\overline{\mathcal{D}[\![e_i]\!]}^{\,i}) \\
\mathcal{D}[\![\lambda x.\ e]\!] &= \lambda x.\ \mathcal{D}[\![e]\!] \\
\mathcal{D}[\![e_1 \oplus e_2]\!] &= t_\oplus\ \mathcal{D}[\![e_1]\!]\ \mathcal{D}[\![e_2]\!] \qquad \text{if term } t_\oplus \text{ implements } \oplus \\
\mathcal{D}[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!] &= \textbf{let } x = \mathcal{D}[\![e_1]\!] \textbf{ in } \mathcal{D}[\![e_2]\!] \\
\mathcal{D}[\![\textbf{if } \{\, \mathcal{S}[\mathcal{T}] \,\}]\!] &= \textbf{if } \{\, \mathcal{D}_{\mathcal{T}}[\![\square \mid \mathcal{S}[\mathcal{T}]]\!] \,\}
\end{aligned}$$

Fig. 13. Desugar source abstract syntax to core abstract syntax.

A.2 Desugaring Examples

We provide several desugaring examples in Fig. 15. The table provides five examples of functions that are very common in actual programming tasks. Note that we use shorthands in Definition 3.3 to make examples compact. Meanwhile, considering that the '**let**' in our syntax is not recursive, we temporarily introduced the '**let rec**' syntax.

Partial Terms

$$\rho[\square] = \square \mid e \oplus \square$$

$\mathcal{D}$esugaring $\mathcal{T}$erm $\mathcal{S}$plits $\mathcal{S}[\mathcal{T}]$ with partial term ρ

$$\boxed{\mathcal{D}_{\mathcal{T}}[\![\rho \mid \mathcal{S}[\mathcal{T}]]\!] : \sigma}$$

$$\begin{aligned}
\mathcal{D}_{\mathcal{T}}[\![\rho \mid e_1 \textbf{ then } e_2 ;\ \mathcal{S}[\mathcal{T}]]\!] &= \mathcal{D}[\![\rho[e_1]]\!] \textbf{ is } \mathsf{True} \to \{ \textbf{ else } \mathcal{D}[\![e_2]\!] \} ;\ \mathcal{D}_{\mathcal{T}}[\![\mathcal{S}[\mathcal{T}]]\!] \\
\mathcal{D}_{\mathcal{T}}[\![\rho \mid e \textbf{ and } \{ \mathcal{S}[\mathcal{T}]_1 \} ;\ \mathcal{S}[\mathcal{T}]_2]\!] &= \mathcal{D}[\![\rho[e]]\!] \textbf{ is } \mathsf{True} \to \{ \mathcal{D}_{\mathcal{T}}[\![\mathcal{S}[\mathcal{T}]_1]\!] +\!\!+ \mathcal{D}_{\mathcal{T}}[\![\mathcal{S}[\mathcal{T}]_2]\!] \} ;\ \mathcal{D}_{\mathcal{T}}[\![\mathcal{S}[\mathcal{T}]_2]\!] \\
\mathcal{D}_{\mathcal{T}}[\![\rho \mid e \textbf{ is } \{ \mathcal{S}[\mathcal{P}] \} ;\ \mathcal{S}[\mathcal{T}]]\!] &= \mathcal{D}_{\mathcal{P}}[\![\mathcal{D}[\![\rho[e]]\!] \mid \mathcal{S}[\mathcal{P}]]\!] +\!\!+ \mathcal{D}_{\mathcal{T}}[\![\mathcal{S}[\mathcal{T}]]\!] \\
\mathcal{D}_{\mathcal{T}}[\![\rho \mid e \{ \mathcal{S}[O] \} ;\ \mathcal{S}[\mathcal{T}]]\!] &= \mathcal{D}_{O}[\![\mathcal{D}[\![\rho[e]]\!] \mid \mathcal{S}[O]]\!] +\!\!+ \mathcal{D}_{\mathcal{T}}[\![\mathcal{S}[\mathcal{T}]]\!] \\
\mathcal{D}_{\mathcal{T}}[\![\rho \mid e \oplus \{ \mathcal{S}[\mathcal{T}]_1 \} ;\ \mathcal{S}[\mathcal{T}]_2]\!] &= \mathcal{D}_{\mathcal{T}}[\![\rho[e] \oplus \square \mid \mathcal{S}[\mathcal{T}]_1]\!] +\!\!+ \mathcal{D}_{\mathcal{T}}[\![\mathcal{S}[\mathcal{T}]_2]\!]
\end{aligned}$$

$\mathcal{D}$esugaring $\mathcal{P}$attern $\mathcal{S}$plit $\mathcal{S}[\mathcal{P}]$ with scrutinee t

$$\boxed{\mathcal{D}_{\mathcal{P}}[\![t \mid \mathcal{S}[\mathcal{P}]]\!] : \sigma}$$

$$\begin{aligned}
\mathcal{D}_{\mathcal{P}}[\![t \mid p \textbf{ then } e ;\ \mathcal{S}[\mathcal{P}]]\!] &= \mathcal{D}_{p}[\![p \mid t \mid \{ \textbf{ else } \mathcal{D}[\![e]\!] \}]\!] +\!\!+ \mathcal{D}_{\mathcal{P}}[\![t \mid \mathcal{S}[\mathcal{P}]]\!] \\
\mathcal{D}_{\mathcal{P}}[\![t \mid p \textbf{ and } \{ \mathcal{S}[\mathcal{T}] \} ;\ \mathcal{S}[\mathcal{P}]]\!] &= \mathcal{D}_{p}[\![p \mid t \mid \mathcal{D}_{\mathcal{T}}[\![\mathcal{S}[\mathcal{T}]]\!]]\!] +\!\!+ \mathcal{D}_{\mathcal{P}}[\![t \mid \mathcal{S}[\mathcal{P}]]\!]
\end{aligned}$$

$\mathcal{D}$esugaring Operator $\mathcal{S}$plit $\mathcal{S}[O]$ with left-hand side term t

$$\boxed{\mathcal{D}_{O}[\![t \mid \mathcal{S}[O]]\!] : \sigma}$$

$$\begin{aligned}
\mathcal{D}_{O}[\![t \mid \textbf{is } \{ \mathcal{S}[\mathcal{P}] \} ;\ \mathcal{S}[O]]\!] &= \mathcal{D}_{\mathcal{P}}[\![t \mid \mathcal{S}[\mathcal{P}]]\!] +\!\!+ \mathcal{D}_{O}[\![t \mid \mathcal{S}[O]]\!] \\
\mathcal{D}_{O}[\![t \mid \oplus \{ \mathcal{S}[\mathcal{T}] \} ;\ \mathcal{S}[O]]\!] &= \mathcal{D}_{\mathcal{T}}[\![t \oplus \square \mid \mathcal{S}[\mathcal{T}]]\!] +\!\!+ \mathcal{D}_{O}[\![t \mid ;\ \mathcal{S}[O]]\!]
\end{aligned}$$

$\mathcal{D}$esugaring any $\mathcal{S}$plit $\mathcal{S}[\mathcal{X}]$

$$\boxed{\mathcal{D}_{\mathcal{X}}[\![\diamond \mid \mathcal{S}[\mathcal{X}]]\!] : \sigma} \qquad \diamond = \begin{cases} \rho & \text{if } \mathcal{X} = \mathcal{P} \\ t & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\mathcal{D}_{\mathcal{X}}[\![\diamond \mid \textbf{let } p = e ;\ \mathcal{S}[\mathcal{X}]]\!] &= \textbf{let } x = \mathcal{D}[\![e]\!] ;\ \mathcal{D}_{\mathcal{P}}[\![x \mid p \textbf{ and } \{ \mathcal{S}[\mathcal{X}] \} ;\ \varepsilon]\!] \\
\mathcal{D}_{\mathcal{X}}[\![\diamond \mid \textbf{else } e]\!] &= \textbf{else } \mathcal{D}[\![e]\!] \\
\mathcal{D}_{\mathcal{X}}[\![\varepsilon]\!] &= \varepsilon
\end{aligned}$$

$\mathcal{D}$esugar $p$atterns with scrutinee t and innermost split σ

$$\boxed{\mathcal{D}_{p}[\![p, t, \sigma]\!] : \sigma}$$

$$\begin{aligned}
\mathcal{D}_{p}[\![x, t, \sigma]\!] &= \textbf{let } x = t ;\ \sigma \\
\mathcal{D}_{p}[\![C(\overline{p_i}^{\,i}), t, \sigma]\!] &= \textbf{let } x = t ;\ x \textbf{ is } C(\overline{x_i}^{\,i}) \to zip(\overline{x_i}^{\,i}, \overline{p_i}^{\,i}, \sigma)x, \overline{x_i}^{\,i} \text{ fresh}
\end{aligned}$$

Zip sub-scrutinees $\overline{x_i}^{\,i}$ and sub-patterns $\overline{p_i}^{\,i}$ with innermost split σ

$$\boxed{zip(\overline{x_i}^{\,i}, \overline{p_i}^{\,i}, \sigma) : \sigma}$$

$$\begin{aligned}
zip((x_1, \overline{x_i}^{\,i}), (p_1, \overline{p_i}^{\,i}), \sigma) &= \mathcal{D}_{p}[\![p_1, x_1, zip(\overline{x_i}^{\,i}, \overline{p_i}^{\,i}, \sigma)]\!] \\
zip(\varepsilon, \varepsilon, \sigma) &= \sigma
\end{aligned}$$

Fig. 14. Desugar splits and patterns.

The first example FindFirst is to find the first element in a functional list (expressed by constructors 'Cons(*head*, *tail*)' and 'Nil(ε)') that satisfies the predicate p, and returns it in the Some constructor. If no such element is found, it returns None. There is not much difference before and after desugaring in this example, which meets our expectations: in most cases, users can intuitively imagine the results generated by the desugaring algorithm.

The second example, MapPartition, divides a list into two based on the results returned by function f. This example demonstrates that a non-variable scrutinee is desugared into a let binding, which is intended to prevent the scrutinee from being evaluated multiple times.

The next two examples, TreeInsert and TreeRemove, demonstrate two common operations on a binary search tree (without a balancing mechanism). They represent nodes and empty trees by constructors 'Node(*value*, *left*, *right*)' and 'Empty(ε)' respectively. In these two examples, the conditional split breaks off from variable v, and is connected with two different operators '<' and '>', as well as their corresponding right-hand side expressions. We can see that this organization of code is readable when combined with appropriate indentation.

The final example Expression implements a common evaluation function for expression syntax trees. We assume there are two constructors: 'NumLit(n)' represents numeric constants, and 'BinOp(*op*, *lhs*, *rhs*)' represents binary expressions. Together, they represent an algebraic data type of syntax trees of arithmetic expressions. The function returns Some for a successful evaluation, and None to indicate an error occurring during the evaluation. Therefore, when evaluating a binary expression, we must first evaluate the operands before proceeding. Here, the UCS can avoid handling the None branch, as the case is handled by the top-level '**else**' branch.

From these examples, we can see that the UCS works very well in practice with functional programming and immutable data types.

B Proofs

We now present the full correctness proofs outlined in the main text.

Definition B.1 (Normalization of terms). Normalization of terms is defined as follows.

$$
\begin{aligned}
&\mathcal{N}[\![x]\!] && = x \\
&\mathcal{N}[\![t_1\ t_2]\!] && = \mathcal{N}[\![t_1]\!]\ \mathcal{N}[\![t_2]\!] \\
&\mathcal{N}[\![C(\overline{t_i}^{\,i})]\!] && = C(\overline{\mathcal{N}[\![t_i]\!]}^{\,i}) \\
&\mathcal{N}[\![\lambda x.\, t_1]\!] && = \lambda x.\, \mathcal{N}[\![t_1]\!] \\
&\mathcal{N}[\![\textbf{let}\ x = t_1\ \textbf{in}\ t_2]\!] && = \textbf{let}\ x = \mathcal{N}[\![t_1]\!]\ \textbf{in}\ \mathcal{N}[\![t_2]\!] \\
&\mathcal{N}[\![\textbf{if}\ \{\,\sigma\,\}]\!] && = \textbf{if}\ \{\,\mathcal{M}[\![\sigma]\!]\,\}
\end{aligned}
$$

Lemma B.2 (Normalization of values). *For all terms t, if t is a value, then $\mathcal{N}[\![v]\!]$ is also a value. Otherwise, if t is not a value, then $\mathcal{N}[\![v]\!]$ is not a value.*

Definition B.3 (Shorthand for normalization of substitution). For all substitution $\phi = \overline{x_i \mapsto v_i}^{\,i}$, we denote $\overline{x_i \mapsto \mathcal{N}[\![v_i]\!]}^{\,i}$ by $\mathcal{N}[\![\phi]\!]$.

Lemma B.4 (Distributivity of normalization on substitution). *For all terms t and substitution $\phi = \overline{x_i \mapsto v_i}^{\,i}$, $\mathcal{N}[\![\,[\phi]\ t]\!] = [\mathcal{N}[\![\phi]\!]]\ \mathcal{N}[\![t]\!]$.*

Definition B.5 (Nested evaluation context of terms). We define $\mathscr{E}[\square]$ as the nested evaluation context of terms. Note that it does not participate in the evaluation of the UCS terms, its goal is to extract the next sub-term to be evaluated in the term.

$$
\mathscr{E}[\square] ::= \square \mid \mathscr{E}[\square]\ t \mid v\ \mathscr{E}[\square] \mid C(\overline{v}, \mathscr{E}[\square], \overline{t}) \mid \textbf{let}\ x = \mathscr{E}[\square]\ \textbf{in}\ t
$$

Name	Source Abstract Syntax Terms	$\mathcal{D}$esugared Terms

FindFirst	**let rec** *findFirst* = $\lambda p.\ \lambda xs.$ **if** { xs **is** {
	Nil **then** None ;
	Cons(x, xs) **and** {
	$p(x)$ **then** Some(x) ;
	else *findFirst*(p, xs) } } } **in** ...
		let rec *findFirst* = $\lambda p.\ \lambda xs.$ **if** {
		xs **is** Nil → None
		xs **is** Cons(x, xs) → {
		$p(x)$ **is** True → Some(x) ;
		else *findFirst*(p, xs) } } **in** ...

MapPartition	**let rec** *mapPartition* = $\lambda f.\ \lambda xs.$ **if** { xs **is** {
	Nil **then** Pair(Nil, Nil) ;
	Cons(x, xs) **and** $f(x)$ **is** {
	let Pair(l, r) = *mapPartition*(f, xs) ;
	Left(v) **then** Pair(Cons(v, l), r) ;
	Right(v) **then** Pair(l, Cons(v, r)) } } }
	in ...
		let rec *mapPartition* = $\lambda f.\ \lambda xs.$ **if** {
		xs **is** Nil → Pair(Nil, Nil) ;
		xs **is** Cons(x, xs) → {
		let x_0 = $f(x)$;
		let x_1 = *mapPartition*(f, xs) ;
		x_1 **is** Pair(l, r) → {
		x_0 **is** Left(v) → Pair(Cons(v, l), r) ;
		x_0 **is** Right(v) → Pair(l, Cons(v, r))
		} } } **in** ...

TreeInsert	**let rec** *insert* = $\lambda t.\ \lambda v.$ **if** { t **is** {
	Node(v', l, r) **and** v {
	$< v'$ **then** Node(v', *insert*(l, v), r) ;
	$> v'$ **then** Node(v', l, *insert*(r, v))
	else t } ;
	Empty **then** Node(v, Empty, Empty)} }
	in ...
		let rec *insert* = $\lambda t.\ \lambda v.$ **if** {
		t **is** Node(v', l, r) → {
		$< v\ v'$ → Node(v', *insert*(l, v), r) ;
		$> v\ v'$ → Node(v', l, *insert*(r, v)) ;
		else t } ;
		t **is** Empty → Node(v, Empty, Empty) }
		in ...

TreeRemove	**let rec** *remove* = $\lambda t.\ \lambda v.$ **if** { t **is** {
	Node(v', l, r) **and** v {
	$< v'$ **then** Node(v', *remove*(l, v), r) ;
	$> v'$ **then** Node(v', l, *remove*(r, v))
	minValue(r) **is** {
	None **then** l ;
	Some(v'') **then**
	Node(v'', l, *remove*(r, v'')) } } ;
	Empty **then** Empty } } **in** ...
		let rec *remove* = $\lambda t.\ \lambda v.$ **if** {
		t **is** Node(v', l, r) → {
		$< v\ v'$ → Node(v', *remove*(l, v), r) ;
		$> v\ v'$ → Node(v', l, *remove*(r, v)) ;
		let x_1 = *minValue*(r) ;
		x_1 **is** None → l ;
		x_1 **is** Some(v'') →
		Node(v'', l, *remove*(r, v'')) } ;
		t **is** Empty → Empty } **in** ...

Expression	**let rec** *eval* = $\lambda e.$ **if** { e **is** {
	NumLit(n) **then** Some(n)
	BinOp(op, l, r) **and** {
	eval l **is** Some(lhs) **and** {
	eval r **is** Some(rhs) **and** { op **is** {
	"+" **then** Some(+ lhs rhs) ;
	"*" **then** Some(× lhs rhs) } } } }
	else None } } **in** ...
		let rec *eval* = $\lambda e.$ **if** {
		e **is** NumLit(n) → Some(n)
		e **is** BinOp(op, l, r) → {
		let x_1 = *eval* l ;
		x_1 **is** Some(lhs) → {
		let x_2 = *eval* r ;
		x_2 **is** Some(rhs) → {
		op **is** "+" → Some(+ lhs rhs) ;
		op **is** "*" → Some(× lhs rhs) } } }
		else None } **in** ...

Fig. 15. Examples of source abstract syntax terms and corresponding core abstract syntax terms after desugaring. Note that the shorthand mentioned in Definition 3.3 is used in these examples. Considering that '**let**' is not recursive in our system, we use '**let rec**' syntax, which is common in many languages.

Lemma B.6 (Normalization of evaluation context of terms). *If a term t can be written as $\mathscr{E}[t_1]$, where t_1 is the sub-term to be reduced in the next small step of t, then there exists a $\mathscr{E}'$, such that $\mathcal{N}[\![t]\!]$ can be written as $\mathscr{E}'[\mathcal{N}[\![t_1]\!]]$, where $\mathcal{N}[\![t_1]\!]$ is also the sub-term to be reduced next in the next small step of $\mathcal{N}[\![t]\!]$.*

Definition B.7 (Abstraction of the UCS terms). The translation of term $C[\![t]\!]$ extracts the free variables of the UCS term in t and replaces every UCS term with a lambda abstraction containing the UCS term applied to those free variables. The definition of this translation is almost entirely congruence rules, except for the following one.

$$C[\![\textbf{if } \{ \sigma \}]\!] = (\lambda \overline{x_i}^{\,i}.\ \textbf{if } \{ C'[\![\sigma]\!] \})\ \overline{y_i}^{\,i} \qquad \text{where } \overline{y_i}^{\,i} = FV(\textbf{if } \{ C'[\![\sigma]\!] \})$$

This transformation does not affect the evaluation of the term. In the following correctness proof Theorem 3.5, we assume that all terms have undergone this transformation. When we mention the UCS term, we are referring to the lambda abstraction that contains the UCS term.

Definition B.8 (The finale of a UCS term). For all split σ, if $\textbf{if } \{ \sigma \} \overset{*}{\rightsquigarrow} v$ for some v, then there must exist an earliest UCS term $\textbf{if } \{ \textbf{else } t \}$ on this evaluation path such that $\textbf{if } \{ \sigma \} \overset{*}{\rightsquigarrow} t \overset{*}{\rightsquigarrow} v$. We call such $\textbf{if } \{ \textbf{else } t \}$ the *finale* of σ.

In the subsequent statements of proofs and lemmas, when we mention $\textbf{if } \{ \sigma \} \overset{*}{\rightsquigarrow} \textbf{if } \{ \textbf{else } t \}$, we always assume that $\textbf{if } \{ \textbf{else } t \}$ is the finale of $\textbf{if } \{ \sigma \}$. The purpose of this assumption is to confine us to discussing the reduction of a single UCS term at one time.

B.1 Semantic Preservation of Normalization of Terms

Theorem 3.5 (Semantic Preservation of Translation). *For all terms t, if $t \overset{*}{\rightsquigarrow} v$, then $\mathcal{N}[\![t]\!] \overset{*}{\rightsquigarrow} \mathcal{N}[\![v]\!]$.*

Proof. By induction on the number of evaluation small steps from t to v, the base case $t = v$ is immediate. Next, we consider the inductive case, we proceed with case analysis on the rule that applies to t. Based on the sub-term of t to be evaluated, we can divide it into two cases. In both scenarios, we will separately consider the evaluation paths of the original term and the term after normalization, and then regain this correspondence after one or several steps.

If the sub-term is a UCS term, that is, $t = \mathscr{E}[(\lambda \overline{x}.\ \textbf{if } \{ \sigma \})\ \overline{v}]$ for some σ and $\overline{v}$. Thus, we can see several key steps on the evaluation path of this sub-term.

$$(\lambda \overline{x_i}^{\,i}.\ \textbf{if } \{ \sigma \})\ \overline{v_i}^{\,i} \overset{*}{\rightsquigarrow} [\overline{x_i \mapsto v_i}^{\,i}]\ \textbf{if } \{ \sigma \} \overset{*}{\rightsquigarrow} \textbf{if } \{ \textbf{else } t_1 \} \overset{*}{\rightsquigarrow} v_1$$

Next, consider its normalization $\mathcal{N}[\![t]\!] = \mathscr{E}'[(\lambda \overline{x}.\ \textbf{if } \{ \mathcal{M}[\![\sigma]\!] \})\ \overline{\mathcal{N}[\![v]\!]}]$, where $\mathscr{E}'$ is by Lemma B.6. We can find the corresponding terms along the evaluation path of the normalized term.

$$\begin{aligned}
(\lambda \overline{x_i}^{\,i}.\ \textbf{if } \{ \mathcal{M}[\![\sigma]\!] \})\ \overline{\mathcal{N}[\![v_i]\!]}^{\,i} &\overset{*}{\rightsquigarrow} [\overline{x_i \mapsto \mathcal{N}[\![v_i]\!]}^{\,i}]\ \textbf{if } \{ \mathcal{M}[\![\sigma]\!] \} && \text{by E-AppAbs} \\
&\overset{*}{\rightsquigarrow} \textbf{if } \{ \textbf{else } \mathcal{N}[\![t_1]\!] \} && \text{by Lemma 3.6} \\
&\overset{*}{\rightsquigarrow} \mathcal{N}[\![v_1]\!] && \text{by the induction hypothesis}
\end{aligned}$$

After obtaining v_1, we consider $\mathscr{E}[v_1]$. If it is a value, then the argument is finished. If not, then we can repeatedly apply the induction hypothesis to the next sub-term of $\mathscr{E}[v_1]$ to be evaluated.

Otherwise, if the sub-term is not a UCS term. Suppose $t = \mathscr{E}[t_1]$ and t_1 is the next sub-term to be evaluated. Suppose $t_1 \overset{*}{\rightsquigarrow} t_1'$, thus $\mathscr{E}[t_1] \rightsquigarrow \mathscr{E}[t_1']$. Next, consider the normalization $\mathscr{E}'[\mathcal{N}[\![t_1]\!]] \rightsquigarrow \mathscr{E}'[\mathcal{N}[\![t_1']\!]]$. We can apply the induction hypothesis to the term $\mathscr{E}[t_1']$ and continue the proof until it is a value. □

B.2 Correctness of Normalization of Splits

Lemma 3.6 (Correctness of normalization of splits). *For all split σ and substitution $\phi = \overline{x_i \mapsto v_i}^i$, if $[\phi]$ **if** $\{\, \sigma \,\} \overset{*}{\rightsquigarrow}$ **if** $\{$ **else** $t \,\}$, then $[\mathcal{N}[\![\phi]\!]]$ **if** $\{\, \mathcal{M}[\![\sigma]\!] \,\} \overset{*}{\rightsquigarrow}$ **if** $\{$ **else** $\mathcal{N}[\![t]\!] \,\}$.*

Proof. By induction on the number of evaluation steps and case analysis on the shape of σ.

Case 1 Suppose $\sigma = \varepsilon$. Impossible because $[\phi]$ **if** $\{\, \varepsilon \,\}$ gets stuck.

Case 2 Suppose $\sigma =$ **else** t_1. Immediate by $[\phi]$ **if** $\{\, \sigma \,\} =$ **if** $\{$ **else** $[\phi]\, t_1 \,\}$ and

$$[\mathcal{N}[\![\phi]\!]] \textbf{ if } \{\, \mathcal{M}[\![\sigma]\!] \,\} = \textbf{if } \{\, \textbf{else } [\mathcal{N}[\![\phi]\!]]\, \mathcal{N}[\![t_1]\!] \,\} = \textbf{if } \{\, \textbf{else } \mathcal{N}[\![[\phi]\, t_1]\!] \,\}.$$

Case 3 Suppose $\sigma = (\textbf{let } x_1 = t_1 \,;\ \sigma_1)$. We have $[\phi]$ **if** $\{\, \sigma \,\} =$ **if** $\{\, \textbf{let } x_1 = [\phi]\, t_1 \,;\ [\phi]\, \sigma_1 \,\}$. By Lemma B.11, there exists v_1 such that $[\phi]\, t_1 \overset{*}{\rightsquigarrow} v_1$. Thus,

$$\begin{aligned} \textbf{if } \{\, \textbf{let } x_1 = [\phi]\, t_1 \,;\ [\phi]\, \sigma_1 \,\} &\overset{*}{\rightsquigarrow} \textbf{if } \{\, \textbf{let } x_1 = v_1 \,;\ [\phi]\, \sigma_1 \,\} && (1)\\ &\rightsquigarrow \textbf{if } \{\, [x_1 \mapsto v_1, \phi]\, \sigma_1 \,\}. && (2) \end{aligned}$$

By Theorem 3.5, $[\mathcal{N}[\![\phi]\!]]\, \mathcal{N}[\![t_1]\!] = \mathcal{N}[\![\, [\phi]\, t_1]\!] \overset{*}{\rightsquigarrow} \mathcal{N}[\![v_1]\!]$. Next, consider

$$\begin{aligned} [\mathcal{N}[\![\phi]\!]] \textbf{ if } \{\, \mathcal{M}[\![\sigma]\!] \,\} &= [\mathcal{N}[\![\phi]\!]] \textbf{ if } \{\, \mathcal{M}[\![\textbf{let } x_1 = t_1 \,;\ \sigma_1]\!] \,\} && (3)\\ &= [\mathcal{N}[\![\phi]\!]] \textbf{ if } \{\, \textbf{let } x_1 = \mathcal{N}[\![t_1]\!] \,;\ \mathcal{M}[\![\sigma_1]\!] \,\} && (4)\\ &= \textbf{if } \{\, \textbf{let } x_1 = [\mathcal{N}[\![\phi]\!]]\, \mathcal{N}[\![t_1]\!] \,;\ [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\sigma_1]\!] \,\} && (5)\\ &\overset{*}{\rightsquigarrow} \textbf{if } \{\, \textbf{let } x_1 = \mathcal{N}[\![v_1]\!] \,;\ [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\sigma_1]\!] \,\} && (6)\\ &\rightsquigarrow \textbf{if } \{\, [x_1 \mapsto \mathcal{N}[\![v_1]\!], \mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\sigma_1]\!] \,\}. && (7) \end{aligned}$$

Lastly, by the induction hypothesis, if *Eq.* (2) $\overset{*}{\rightsquigarrow}$ **if** $\{$ **else** $t_2 \,\}$ for some t_2, then

$$\textit{Eq. (7)} \overset{*}{\rightsquigarrow} \textbf{if } \{\, \textbf{else } \mathcal{N}[\![t_2]\!] \,\}.$$

Case 4 Suppose $\sigma = x_1$ **is** $C(\overline{x_j}^j) \to \{\, \sigma_1 \,\} \,;\ \sigma_2$. First, consider

$$[\phi] \textbf{ if } \{\, \sigma \,\} = \textbf{if } \{\, [\phi]\, x_1 \textbf{ is } C(\overline{x_j}^j) \to \{\, [\phi]\, \sigma_1 \,\} \,;\ [\phi]\, \sigma_2 \,\}.$$

By Lemma B.13, there exists a value v_1 such that $x_1 \mapsto v_1$ belongs to ϕ. We discuss the possible values of v_1. Note that we abbreviate $\mathcal{S}^{\pm}_{x_1 \textbf{ is } C(\overline{x_j}^j)}[\![\sigma]\!]$ to $\mathcal{S}^{\pm}[\![\sigma]\!]$ for any σ in this case for the sake of simplicity.

Case 4-1 Suppose v_1 is some $C(\overline{v_j}^j)$. By E-Match,

$$\begin{aligned} [\phi] \textbf{ if } \{\, \sigma \,\} &= \textbf{if } \{\, C(\overline{v_j}^j) \textbf{ is } C(\overline{x_j}^j) \to \{\, \sigma_1 \,\} \,;\ \sigma_2 \,\} && (8)\\ &\rightsquigarrow \textbf{if } \{\, [\overline{x_j \mapsto v_j}^j, \phi]\, \sigma_1 +\!\!+ [\phi]\, \sigma_2 \,\}. && (9) \end{aligned}$$

Because σ_2 does not have free variables in $\overline{x_j}^j$, the term is equivalent to

$$\textbf{if } \{\, [\overline{x_j \mapsto v_j}^j, \phi]\, (\sigma_1 +\!\!+ \sigma_2) \,\}. \quad (10)$$

Since ϕ contains $x_1 \mapsto C(\overline{v_j}^j)$, $\mathcal{N}[\![\phi]\!]$ contains $x_1 \mapsto C(\overline{\mathcal{N}[\![v_j]\!]}^j)$. Next, consider the normalization $[\mathcal{N}[\![\phi]\!]]$ **if** $\{\, \mathcal{M}[\![\sigma]\!] \,\}$

$$\begin{aligned} &= [\mathcal{N}[\![\phi]\!]] \textbf{ if } \{\, x_1 \textbf{ is } C(\overline{x_j}^j) \to \{\, \mathcal{M}[\![\mathcal{S}^+[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!] \,\} \,;\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma_2]\!]]\!] \,\} && (11)\\ &= \textbf{if } \{\, C(\overline{\mathcal{N}[\![v_j]\!]}^j) \textbf{ is } C(\overline{x_j}^j) \to \{\, [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^+[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!] \,\} \,;\ [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^-[\![\sigma_2]\!]]\!] \,\} && (12)\\ &\rightsquigarrow \textbf{if } \{\, [\overline{x_j \mapsto \mathcal{N}[\![v_j]\!]}^j, \mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^+[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!] \,\}. && (13) \end{aligned}$$

Next, we will show that if *Eq.* (10) $\overset{*}{\rightsquigarrow}$ **if** $\{$ **else** $t_1 \,\}$, then *Eq.* (13) $\overset{*}{\rightsquigarrow}$ **if** $\{$ **else** $\mathcal{N}[\![t_1]\!] \,\}$. By the induction hypothesis, we have if *Eq.* (10) $\overset{*}{\rightsquigarrow}$ **if** $\{$ **else** $t_1 \,\}$, then

$$[\overline{x_j \mapsto \mathcal{N}[\![v_j]\!]}^j, \mathcal{N}[\![\phi]\!]] \textbf{ if } \{\, \mathcal{M}[\![\sigma_1 +\!\!+ \sigma_2]\!] \,\} \overset{*}{\rightsquigarrow} \textbf{if } \{\, \textbf{else } \mathcal{N}[\![t_1]\!] \,\}. \quad (14)$$

Observe Eq. (13) and Eq. (14). We can apply Lemma B.9 to show if Eq. (14), then *Eq.* (13) $\overset{*}{\rightsquigarrow}$ **if** { **else** $\mathcal{N}[\![t_1]\!]$ }.

Case 4-2 Suppose v_1 does not match $C(\overline{x_j}^{\,j})$. From Eq. (11), we have

$$[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![\sigma]\!]\ \} \rightsquigarrow \textbf{if}\ \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma_2]\!]]\!]\ \}. \tag{15}$$

By the induction hypothesis, if $[\phi]$ **if** { σ_2 } $\overset{*}{\rightsquigarrow}$ **if** { **else** t_1 }, then

$$[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![\sigma_2]\!]\ \} \overset{*}{\rightsquigarrow} \textbf{if}\ \{\ \textbf{else}\ \mathcal{N}[\![t_1]\!]\ \}. \tag{16}$$

Finally, by Lemma B.10, if Eq. (16) holds, then *Eq.* (15) $\overset{*}{\rightsquigarrow}$ **if** { **else** $\mathcal{N}[\![t_1]\!]$ }.

Case 5 Suppose $\sigma = t_1$ **is** $C(\overline{x_j}^{\,j}) \to \{\ \sigma_1\ \};\ \sigma_2$. We have

$$[\phi]\ \textbf{if}\ \{\ \sigma\ \} = \textbf{if}\ \{\ [\phi]\ t_1\ \textbf{is}\ C(\overline{x_j}^{\,j}) \to \{\ [\phi]\ \sigma_1\ \};\ [\phi]\ \sigma_2\ \}. \tag{17}$$

By Lemma B.12, there exists v_1 such that $[\phi]\ t_1 \overset{*}{\rightsquigarrow} v_1$. Analogous to **Case 3**, by Theorem 3.5 we have $[\mathcal{N}[\![\phi]\!]]\ \mathcal{N}[\![t_1]\!] = \mathcal{N}[\![[\phi]\ t_1]\!] \overset{*}{\rightsquigarrow} \mathcal{N}[\![v_1]\!]$. Thus,

$$\textit{Eq.}\ (17) \overset{*}{\rightsquigarrow} \textbf{if}\ \{\ v_1\ \textbf{is}\ C(\overline{x_j}^{\,j}) \to \{\ [\phi]\ \sigma_1\ \};\ [\phi]\ \sigma_2\ \}. \tag{18}$$

Analogous to **Case 4**, we proceed by discussing the possible values of v_1. We can conclude in both cases.

□

B.3 Correctness of Specialization

Next, we demonstrate the correctness of specialization, which is achieved by showing that a UCS term that has been specialized and then normalized can be reduced to the same term as the UCS term that has been directly normalized does.

Lemma B.9 (Correctness of Positive Specialization in Normalization). *For all split σ, substitution $\phi_1 = \overline{x_i \mapsto v_i}^{\,i}$, variable x_1, value $v_1 = C(\overline{v_j}^{\,j})$, and pattern $C(\overline{x_j}^{\,j})$, with $\phi = \overline{x_j \mapsto v_j}^{\,j}, \phi_1$, if the following conditions hold:*

C1. ϕ_1 contains the substitution $x_1 \mapsto v_1$, and

C2. $[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![\sigma]\!]\ \} \overset{*}{\rightsquigarrow} \textbf{if}\ \{\ \textbf{else}\ \mathcal{N}[\![t]\!]\ \}$*;*

then $[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![\mathcal{S}^+_{x_1\ \textbf{is}\ C(\overline{x_j}^{\,j})}[\![\sigma]\!]]\!]\ \} \overset{*}{\rightsquigarrow} \textbf{if}\ \{\ \textbf{else}\ \mathcal{N}[\![t]\!]\ \}$.

For simplicity, for all σ, we abbreviate $\mathcal{S}^{\pm}_{x_1\ \textbf{is}\ C(\overline{x_j}^{\,j})}[\![\sigma]\!]$ to $\mathcal{S}^{\pm}[\![\sigma]\!]$ in the following proof.

Proof. By induction on the size of σ, we proceed with case analysis on σ.

Case 1 Suppose $\sigma = \varepsilon$. Impossible because $[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \varepsilon\ \}$ gets stuck.

Case 2 Suppose $\sigma =$ **else** t_1. Immediate by $[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![\sigma]\!]\ \} = \textbf{if}\ \{\ \textbf{else}\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{N}[\![t_1]\!]\ \}$ and $[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![\mathcal{S}^+[\![\sigma]\!]]\!]\ \} = \textbf{if}\ \{\ \textbf{else}\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{N}[\![t_1]\!]\ \}$.

Case 3 Suppose $\sigma = (\textbf{let}\ x_2 = v_2\ ;\ \sigma_1)$. Consider

$$\begin{aligned}
&[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![\sigma]\!]\ \} && && (19)\\
&= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \textbf{let}\ x_2 = \mathcal{N}[\![v_2]\!]\ ;\ \mathcal{M}[\![\sigma_1]\!]\ \} && \text{by } \mathcal{M}[\![\sigma]\!] && (20)\\
&\rightsquigarrow \textbf{if}\ \{\ [x_2 \mapsto \mathcal{N}[\![v_2]\!], \mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\sigma_1]\!]\ \} && \text{by E-IfLet, and} && (21)\\
&\quad [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma]\!]]\!]\ \} && && (22)\\
&= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \textbf{let}\ x_2 = \mathcal{N}[\![v_2]\!]\ ;\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma_1]\!]]\!]\ \} && \text{by } \mathcal{M}[\![\sigma]\!] \text{ and } \mathcal{S}^-[\![\sigma]\!] && (23)\\
&\rightsquigarrow \textbf{if}\ \{\ [x_2 \mapsto \mathcal{N}[\![v_2]\!], \mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma_1]\!]]\!]\ \} && \text{by E-IfLet.} && (24)
\end{aligned}$$

By the induction hypothesis, if *Eq.* (21) $\overset{*}{\rightsquigarrow}$ **if** $\{$ **else** $\mathcal{N}[\![t_1]\!]$ $\}$, then

$$\textit{Eq.}\ (24) \overset{*}{\rightsquigarrow} \textbf{if}\ \{\ \textbf{else}\ \mathcal{N}[\![t_1]\!]\ \}.$$

Case 4 Suppose $\sigma = x_2$ **is** $D(\overline{y_k}^k) \to \{\ \sigma_1\ \};\ \sigma_2$. For simplicity, for all splits σ, we abbreviate $\mathcal{S}^{\pm}_{x_2\ \textbf{is}\ D(\overline{y_k}^k)}[\![\sigma]\!]$ to $\mathcal{S}^{\pm}_{\star}[\![\sigma]\!]$ in **Case 4** and its sub-cases. Next, we proceed with further case analysis on $x_1 = x_2$ and $C = D$, and we will continue to discuss

$$[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![\sigma]\!]\ \} \tag{25}$$
$$= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![x_2\ \textbf{is}\ D(\overline{y_k}^k) \to \{\ \sigma_1\ \};\ \sigma_2]\!]\ \} \tag{26}$$
$$= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ x_2\ \textbf{is}\ D(\overline{y_k}^k) \to \{\ \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!]\ \};\ \mathcal{M}[\![\mathcal{S}^-_{\star}[\![\sigma_2]\!]]\!]\ \} \tag{27}$$
$$[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![\mathcal{S}^+[\![\sigma]\!]]\!]\ \} \tag{28}$$
$$= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![\mathcal{S}^+[\![x_2\ \textbf{is}\ D(\overline{y_k}^k) \to \{\ \sigma_1\ \};\ \sigma_2]\!]]\!]\ \} \tag{29}$$

in each case.

Case 4-1 Suppose $x_1 = x_2$ and $C = D$. That is, $\sigma = x_1$ **is** $C(\overline{y_j}^j) \to \{\ \sigma_1\ \};\ \sigma_2$. Thus,

$$\textit{Eq.}\ (27) = \textbf{if}\ \{\ [\mathcal{N}[\![\phi]\!]]\ x_1\ \textbf{is}\ C(\overline{y_j}^j) \to \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^+[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!]\ \};\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma_2]\!]]\!]\ \} \tag{30}$$
$$= \textbf{if}\ \{\ C(\overline{\mathcal{N}[\![v_j]\!]}^j)\ \textbf{is}\ C(\overline{y_j}^j) \to \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^+[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!]\ \};\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma_2]\!]]\!]\ \} \tag{31}$$
$$\rightsquigarrow \textbf{if}\ \{\ [\overline{y_j \mapsto \mathcal{N}[\![v_j]\!]}^j, \mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^+[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!]\ \} \qquad \text{by E-Match, and} \tag{32}$$
$$\textit{Eq.}\ (29) = [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![\mathcal{S}^+[\![[\overline{y_j \mapsto x_j}^j]\ \sigma_1 +\!\!+ \sigma_2]\!]]\!]\ \} \tag{33}$$

By Lemma B.15, Eq. (32) and Eq. (32) are equivalent.

Case 4-2 Suppose $x_1 = x_2$ and $C \neq D$. That is, $\sigma = x_1$ **is** $D(\overline{x_k}^k) \to \{\ \sigma_1\ \};\ \sigma_2$. Thus,

$$\begin{aligned}
\textit{Eq.}\ (27) &= \textbf{if}\ \{\ [\mathcal{N}[\![\phi]\!]]\ x_1\ \textbf{is}\ C(\overline{y_j}^j) \to \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^+[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!]\ \};\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma_2]\!]]\!]\ \} \\
&= \textbf{if}\ \{\ C(\overline{\mathcal{N}[\![v_j]\!]}^j)\ \textbf{is}\ C(\overline{y_j}^j) \to \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^+[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!]\ \};\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma_2]\!]]\!]\ \} \\
&\rightsquigarrow \textbf{if}\ \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma_2]\!]]\!]\ \} \qquad \text{by E-NotMatch, and} \\
\textit{Eq.}\ (29) &= \textbf{if}\ \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma_2]\!]]\!]\ \},
\end{aligned}$$

which are equivalent.

Case 4-3 Suppose $x_1 \neq x_2$. That is, $\sigma = x_2$ **is** $D(\overline{y_k}^k) \to \{\ \sigma_1\ \};\ \sigma_2$. By Lemma B.13, there exists v_2 such that ϕ contains $x_2 \mapsto v_2$, we proceed to a further case analysis on whether v_2 **is** $D(\overline{y_k}^k)$ holds.

Case 4-3-1 Suppose $v_2 = D(\overline{v_k}^k)$. Thus,

$$\begin{aligned}
\textit{Eq.}\ (27) &= \textbf{if}\ \{\ [\mathcal{N}[\![\phi]\!]]\ x_2\ \textbf{is}\ D(\overline{y_k}^k) \to \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!]\ \};\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-_{\star}[\![\sigma_2]\!]]\!]\ \} \\
&= \textbf{if}\ \{\ D(\overline{\mathcal{N}[\![v_k]\!]}^k)\ \textbf{is}\ D(\overline{y_k}^k) \to \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!]\ \};\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-_{\star}[\![\sigma_2]\!]]\!]\ \} \\
&\rightsquigarrow \textbf{if}\ \{\ [\overline{y_k \mapsto \mathcal{N}[\![v_k]\!]}^k, \mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!]\ \} \\
\textit{Eq.}\ (29) &= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![x_2\ \textbf{is}\ D(\overline{y_k}^k) \to \{\ \mathcal{S}^+[\![\sigma_1]\!]\ \};\ \mathcal{S}^+[\![\sigma_2]\!]]\!]\ \} \\
&= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ x_2\ \textbf{is}\ D(\overline{y_k}^k) \to \{\ \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\mathcal{S}^+[\![\sigma_1]\!] +\!\!+ \mathcal{S}^+[\![\sigma_2]\!]]\!]]\!]\ \};\ \mathcal{M}[\![\mathcal{S}^-_{\star}[\![\mathcal{S}^+[\![\sigma_2]\!]]\!]]\!]\ \} \\
&\rightsquigarrow \textbf{if}\ \{\ [\overline{y_k \mapsto \mathcal{N}[\![v_k]\!]}^k, \mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\mathcal{S}^+[\![\sigma_1]\!] +\!\!+ \mathcal{S}^+[\![\sigma_2]\!]]\!]]\!]\ \} \\
&= \textbf{if}\ \{\ [\overline{y_k \mapsto \mathcal{N}[\![v_k]\!]}^k, \mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\mathcal{S}^+[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!]]\!]\ \}
\end{aligned}$$

By applying the induction hypothesis to $\mathcal{S}^+[\![\sigma_1 +\!\!+ \sigma_2]\!]$, we can prove the case.

Case 4-3-2 Suppose v_2 does not match $D(\overline{y_k}^k)$. We have

$$
\begin{aligned}
\textit{Eq. (27)} &= \textbf{if}\ \{\, [\mathcal{N}[\![\phi]\!]]\ x_2\ \textbf{is}\ D(\overline{y_k}^k) \to \{\, [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^+_\star[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!]\,\};\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-_\star[\![\sigma_2]\!]]\!]\,\} \\
&= \textbf{if}\ \{\, v_2\ \textbf{is}\ D(\overline{y_k}^k) \to \{\, [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^+_\star[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!]\,\};\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-_\star[\![\sigma_2]\!]]\!]\,\} \\
&\rightsquigarrow \textbf{if}\ \{\, [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-_\star[\![\sigma_2]\!]]\!]\,\} \qquad \text{by E-NotMatch, and} \\
\textit{Eq. (29)} &= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \mathcal{M}[\![x_2\ \textbf{is}\ D(\overline{y_k}^k) \to \{\, \mathcal{S}^+[\![\sigma_1]\!]\,\};\ \mathcal{S}^+[\![\sigma_2]\!]]\!]\,\} \\
&= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, x_2\ \textbf{is}\ D(\overline{y_k}^k) \to \{\, \mathcal{M}[\![\mathcal{S}^+_\star[\![\mathcal{S}^+[\![\sigma_1]\!] +\!\!+ \mathcal{S}^+[\![\sigma_2]\!]]\!]]\!]\,\};\ \mathcal{M}[\![\mathcal{S}^-_\star[\![\mathcal{S}^+[\![\sigma_2]\!]]\!]]\!]\,\} \\
&\rightsquigarrow \textbf{if}\ \{\, [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-_\star[\![\mathcal{S}^+[\![\sigma_2]\!]]\!]]\!]\,\} \qquad \text{by E-NotMatch} \\
&= \textbf{if}\ \{\, [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^+[\![\mathcal{S}^-_\star[\![\sigma_2]\!]]\!]]\!]\,\} \qquad \text{by Lemma B.16}
\end{aligned}
$$

By Lemma B.18, we can apply the induction hypothesis to $\mathcal{S}^-_{x_2\ \textbf{is}\ C(\overline{y_k}^k)}[\![\sigma_2]\!]$.

□

Lemma B.10 (Correctness of Negative Specialization in Normalization). *For all split σ, substitution $\phi = \overline{x_i \mapsto v_i}^i$, variable x_1, value v_1, and pattern $C(\overline{x_j}^j)$, if the following conditions hold:*

C1. ϕ contains substitution $x_1 \mapsto v_1$ (i.e., $[\phi]\ x_1 = v_1$),
C2. v_1 is not $C(\overline{x_j}^j)$ (i.e., v_1 is some $\lambda x.\, t$ or some $D(\overline{v})$ where $C \neq D$), and
C3. $[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \mathcal{M}[\![\sigma]\!]\,\} \overset{*}{\rightsquigarrow} \textbf{if}\ \{\, \textbf{else}\ \mathcal{N}[\![t]\!]\,\}$;

then $[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \mathcal{M}[\![\mathcal{S}^-_{x_1\ \textbf{is}\ C(\overline{x_j}^j)}[\![\sigma]\!]]\!]\,\} \overset{*}{\rightsquigarrow} \textbf{if}\ \{\, \textbf{else}\ \mathcal{N}[\![t]\!]\,\}$.

For simplicity, for all σ, we abbreviate $\mathcal{S}^\pm_{x_1\ \textbf{is}\ C(\overline{x_j}^j)}[\![\sigma]\!]$ to $\mathcal{S}^\pm[\![\sigma]\!]$ in the following proof.

Proof. By induction on the size of σ, we proceed with a case analysis on σ.

Case 1 Suppose $\sigma = \varepsilon$. Analogous to the case in Lemma B.9.
Case 2 Suppose $\sigma = \textbf{else}\ t_1$. Analogous to the case in Lemma B.9.
Case 3 Suppose $\sigma = (\textbf{let}\ x_2 = v_2\,;\ \sigma_1)$. Consider

$$
\begin{aligned}
& [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \mathcal{M}[\![\sigma]\!]\,\} && && (34)\\
&= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \textbf{let}\ x_2 = \mathcal{N}[\![v_2]\!]\,;\ \mathcal{M}[\![\sigma_1]\!]\,\} && \text{by } \mathcal{M}[\![\sigma]\!] && (35)\\
&\rightsquigarrow \textbf{if}\ \{\, [x_2 \mapsto \mathcal{N}[\![v_2]\!], \mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\sigma_1]\!]\,\} && \text{by E-IfLet} && (36)\\
& [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \mathcal{M}[\![\mathcal{S}^-[\![\sigma]\!]]\!]\,\} && && (37)\\
&= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \textbf{let}\ x_2 = \mathcal{N}[\![v_2]\!]\,;\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma]\!]]\!]\,\} && \text{by } \mathcal{M}[\![\sigma]\!] \text{ and } \mathcal{S}^-[\![\sigma]\!] && (38)\\
&\rightsquigarrow \textbf{if}\ \{\, [x_2 \mapsto \mathcal{N}[\![v_2]\!], \mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma]\!]]\!]\,\} && \text{by E-IfLet.} && (39)
\end{aligned}
$$

By the induction hypothesis, we have if *Eq.* (36) $\overset{*}{\rightsquigarrow}$ **if** $\{\, \textbf{else}\ \mathcal{N}[\![t_1]\!]\,\}$ then *Eq.* (39) $\overset{*}{\rightsquigarrow}$ **if** $\{\, \textbf{else}\ \mathcal{N}[\![t_1]\!]\,\}$.
Case 4 Suppose $\sigma = x_2\ \textbf{is}\ D(\overline{x_k}^k) \to \{\, \sigma_1\,\};\ \sigma_2$. We proceed by further case analysis on whether $x_1 = x_2$ and $C = D$, which decides the result of negative specialization.

Case 4-1 Suppose $x_1 = x_2$ and $C = D$. That is $\sigma = x_1$ **is** $C(\overline{x_j}^{\,j}) \to \{\, \sigma_1 \,\};\ \sigma_2$. Consider

$$[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \mathcal{M}[\![\sigma]\!] \,\} \tag{40}$$
$$= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, x_1\ \textbf{is}\ C(\overline{x_j}^{\,j}) \to \{\, \mathcal{M}[\![\mathcal{S}^+[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!] \,\};\ \mathcal{M}[\![\mathcal{S}^+[\![\sigma_2]\!]]\!] \,\} \quad \text{by } \mathcal{M}[\![\sigma]\!] \tag{41}$$
$$= \textbf{if}\ \{\, [\mathcal{N}[\![\phi]\!]]\, x_1\ \textbf{is}\ C(\overline{x_j}^{\,j}) \to \{\, [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^+[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!] \,\};\ [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^+[\![\sigma_2]\!]]\!] \,\} \tag{42}$$
$$= \textbf{if}\ \{\, v_1\ \textbf{is}\ C(\overline{x_j}^{\,j}) \to \{\, [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^+[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!] \,\};\ [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^+[\![\sigma_2]\!]]\!] \,\} \quad \text{by C1} \tag{43}$$
$$\rightsquigarrow \textbf{if}\ \{\, [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\sigma_2]\!] \,\} \quad \text{by E-NotMatch} \tag{44}$$

$$[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \mathcal{M}[\![\mathcal{S}^-[\![\sigma]\!]]\!] \,\} \tag{45}$$
$$= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \mathcal{M}[\![\mathcal{S}^-[\![x_1\ \textbf{is}\ C(\overline{x_j}^{\,j}) \to \{\, \sigma_1 \,\};\ \sigma_2]\!]]\!] \,\} \tag{46}$$
$$= \textbf{if}\ \{\, [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^-[\![\sigma_2]\!]]\!] \,\} \quad \text{by } \mathcal{S}^-[\![\sigma]\!] \tag{47}$$

By the induction hypothesis, we have if *Eq.* (44) $\overset{*}{\rightsquigarrow}$ **if** $\{$ **else** $\mathcal{N}[\![t_1]\!]\ \}$, then *Eq.* (47) $\overset{*}{\rightsquigarrow}$ **if** $\{$ **else** $\mathcal{N}[\![t_1]\!]\ \}$.

Case 4-2 Suppose $x_1 \neq x_2$. That is, $\sigma = x_2$ **is** $D(\overline{x_k}^{\,k}) \to \{\, \sigma_1 \,\};\ \sigma_2$.
For simplicity, for all splits σ, we abbreviate $\mathcal{S}^{\pm}_{x_2\ \textbf{is}\ D(\overline{x_k}^{\,k})}[\![\sigma]\!]$ to $\mathcal{S}^{\pm}_{\star}[\![\sigma]\!]$ in **Case 4** and its sub-cases. Consider

$$[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \mathcal{M}[\![\sigma]\!] \,\} = [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \star \to \{\, \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!] \,\};\ \mathcal{M}[\![\mathcal{S}^-_{\star}[\![\sigma_2]\!]]\!] \,\} \tag{48}$$
$$= \textbf{if}\ \{\, [\mathcal{N}[\![\phi]\!]]\, \star \to \{\, [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!] \,\};\ [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^-_{\star}[\![\sigma_2]\!]]\!] \,\} \tag{49}$$
$$= \textbf{if}\ \{\, v_2\ \textbf{is}\ D(\overline{x_k}^{\,k}) \to \{\, [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!] \,\};\ [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^-_{\star}[\![\sigma_2]\!]]\!] \,\} \tag{50}$$

By applying Lemma B.13 to Eq. (50), $\mathcal{N}[\![\phi]\!]$ contains substitution $x_2 \mapsto v_2$ for some v_2. Thus,

$$[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \mathcal{M}[\![\mathcal{S}^-[\![\sigma]\!]]\!] \,\} \tag{51}$$
$$= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \mathcal{M}[\![\mathcal{S}^-[\![\star \to \{\, \sigma_1 \,\};\ \sigma_2]\!]]\!] \,\} \tag{52}$$
$$= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \mathcal{M}[\![\star \to \{\, \mathcal{S}^-[\![\sigma_1]\!] \,\};\ \mathcal{S}^-[\![\sigma_2]\!]]\!] \,\} \tag{53}$$
$$= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\, \star \to \{\, \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\mathcal{S}^-[\![\sigma_1]\!] +\!\!+ \mathcal{S}^-[\![\sigma_2]\!]]\!]]\!] \,\};\ \mathcal{M}[\![\mathcal{S}^-_{\star}[\![\mathcal{S}^-[\![\sigma_2]\!]]\!]]\!] \,\} \tag{54}$$
$$= \textbf{if}\ \{\, [\mathcal{N}[\![\phi]\!]]\, \star \to \{\, [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\mathcal{S}^-[\![\sigma_1]\!] +\!\!+ \mathcal{S}^-[\![\sigma_2]\!]]\!]]\!] \,\};\ [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^-_{\star}[\![\mathcal{S}^-[\![\sigma_2]\!]]\!]]\!] \,\} \tag{55}$$
$$= \textbf{if}\ \{\, v_2\ \textbf{is}\ D(\overline{x_j}^{\,j}) \to \{\, [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\mathcal{S}^-[\![\sigma_1]\!] +\!\!+ \mathcal{S}^-[\![\sigma_2]\!]]\!]]\!] \,\};\ [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^-_{\star}[\![\mathcal{S}^-[\![\sigma_2]\!]]\!]]\!] \,\} \tag{56}$$

We then discuss the possible values of v_2.

Case 4-2-1 Suppose $v_2 = D(\overline{v_k}^{\,k})$. That is, v_2 **is** $D(\overline{x_k}^{\,k})$ holds. Thus,

$$\textit{Eq.}\ (50) = \textbf{if}\ \{\, D(\overline{v_k}^{\,k})\ \textbf{is}\ D(\overline{x_k}^{\,k}) \to \{\, [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!] \,\};\ [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^-_{\star}[\![\sigma_2]\!]]\!] \,\} \tag{57}$$
$$\rightsquigarrow \textbf{if}\ \{\, [\overline{x_k \mapsto v_k}^{\,k}, \mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!] \,\} \tag{58}$$
$$\textit{Eq.}\ (56) = \textbf{if}\ \{\, D(\overline{v_k}^{\,k})\ \textbf{is}\ D(\overline{x_j}^{\,j}) \to \{\, [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\mathcal{S}^-[\![\sigma_1]\!] +\!\!+ \mathcal{S}^-[\![\sigma_2]\!]]\!]]\!] \,\};\ [\mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^-_{\star}[\![\mathcal{S}^-[\![\sigma_2]\!]]\!]]\!] \,\} \tag{59}$$
$$\rightsquigarrow \textbf{if}\ \{\, [\overline{x_k \mapsto v_k}^{\,k}, \mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\mathcal{S}^-[\![\sigma_1]\!] +\!\!+ \mathcal{S}^-[\![\sigma_2]\!]]\!]]\!] \,\} \tag{60}$$
$$= \textbf{if}\ \{\, [\overline{x_k \mapsto v_k}^{\,k}, \mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^+_{\star}[\![\mathcal{S}^-[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!]]\!] \,\} \tag{61}$$
$$= \textbf{if}\ \{\, [\overline{x_k \mapsto v_k}^{\,k}, \mathcal{N}[\![\phi]\!]]\, \mathcal{M}[\![\mathcal{S}^-[\![\mathcal{S}^+_{\star}[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!]]\!] \,\} \tag{62}$$

Observe Eq. (58) and Eq. (62). By applying the induction hypothesis to splits $\mathcal{S}^+_{\star}[\![\sigma_1 +\!\!+ \sigma_2]\!]$ and substitution $\overline{x_k \mapsto \mathcal{N}[\![v_k]\!]}^{\,k}, \mathcal{N}[\![\phi]\!]$, we have if *Eq.* (58) $\overset{*}{\rightsquigarrow}$ **if** $\{$ **else** $\mathcal{N}[\![t_1]\!]\ \}$ for some t_1, then *Eq.* (62) $\overset{*}{\rightsquigarrow}$ **if** $\{$ **else** $\mathcal{N}[\![t_1]\!]\ \}$.

Case 4-2-2 Suppose v_2 does not match $D(\overline{x_k}^k)$. Thus,

$$Eq.\ (50) \rightsquigarrow \textbf{if}\ \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-_{x_2\ \textbf{is}\ D(\overline{x_k}^k)}[\![\sigma_2]\!]]\!]\ \} \tag{63}$$

$$Eq.\ (56) \rightsquigarrow \textbf{if}\ \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-_\star[\![\mathcal{S}^-[\![\sigma_2]\!]]\!]]\!]\ \} \tag{64}$$

$$= \textbf{if}\ \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-[\![\mathcal{S}^-_\star[\![\sigma_2]\!]]\!]]\!]\ \} \tag{65}$$

By applying the induction hypothesis to $\mathcal{S}^-_\star[\![\sigma_2]\!]$ and $\mathcal{N}[\![\phi]\!]$, we have if $Eq.\ (63) \overset{*}{\rightsquigarrow}$ **if** $\{$ **else** $\mathcal{N}[\![t_1]\!]\ \}$ for some t_1, then $Eq.\ (65) \overset{*}{\rightsquigarrow}$ **if** $\{$ **else** $\mathcal{N}[\![t_1]\!]\ \}$.

Case 5 Suppose $\sigma = (\textbf{let}\ x_2 = t_2;\ \sigma_1)$. By Lemma B.11, there exists a value v_2 such that $[\mathcal{N}[\![\phi]\!]]\ t_2 \overset{*}{\rightsquigarrow} v_2$ and $[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \sigma\ \} \overset{*}{\rightsquigarrow} (\textbf{let}\ x_2 = v_2;\ [\mathcal{N}[\![\phi]\!]]\ \sigma_1)$. We proceed with a similar proof as in **Case 3**.

Case 6 Suppose $\sigma = t_2\ \textbf{is}\ D(\overline{x_k}^k) \to \{\ \sigma_1\ \};\ \sigma_2$. Consider $[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![\sigma]\!]\ \}$

$$= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ t_2\ \textbf{is}\ D(\overline{x_k}^k) \to \{\ \mathcal{M}[\![\sigma_1]\!]\ \};\ \mathcal{M}[\![\sigma_2]\!]\ \} \tag{66}$$

$$= \textbf{if}\ \{\ [\mathcal{N}[\![\phi]\!]]\ t_2\ \textbf{is}\ D(\overline{x_k}^k) \to \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\sigma_1]\!]\ \};\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\sigma_2]\!]\ \}. \tag{67}$$

By Lemma B.12, there exists a value v_2 such that $[\mathcal{N}[\![\phi]\!]]\ t_2 \overset{*}{\rightsquigarrow} v_2$ and

$$Eq.\ (67) \overset{*}{\rightsquigarrow} \textbf{if}\ \{\ v_2\ \textbf{is}\ D(\overline{x_k}^k) \to \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\sigma_1]\!]\ \};\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\sigma_2]\!]\ \}$$

Consider $[\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma]\!]]\!]\ \}$

$$= [\mathcal{N}[\![\phi]\!]]\ \textbf{if}\ \{\ t_2\ \textbf{is}\ D(\overline{x_k}^k) \to \{\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma_1]\!]]\!]\ \};\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma_2]\!]]\!]\ \} \tag{68}$$

$$= \textbf{if}\ \{\ [\mathcal{N}[\![\phi]\!]]\ t_2\ \textbf{is}\ D(\overline{x_k}^k) \to \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma_1]\!]]\!]\ \};\ \mathcal{M}[\![\,[\mathcal{N}[\![\phi]\!]]\ \mathcal{S}^-[\![\sigma_2]\!]]\!]\ \} \tag{69}$$

$$\overset{*}{\rightsquigarrow} \textbf{if}\ \{\ v_2\ \textbf{is}\ D(\overline{x_k}^k) \to \{\ [\mathcal{N}[\![\phi]\!]]\ \mathcal{M}[\![\mathcal{S}^-[\![\sigma_1]\!]]\!]\ \};\ \mathcal{M}[\![\,[\mathcal{N}[\![\phi]\!]]\ \mathcal{S}^-[\![\sigma_2]\!]]\!]\ \} \tag{70}$$

Analogous to **Case 4-2**, we proceed with case analysis on possible values of v_2.

Case 6-1 Suppose $v_2 = D(\overline{v_k}^k)$. That is, $v_2\ \textbf{is}\ D(\overline{x_k}^k)$ holds. We can rewrite terms in the same way as in **Case 4-2-1**. Then apply the IH to $\sigma_1 +\!\!+ \sigma_2$ and relevant substitution.

Case 6-2 Suppose v_2 does not match $D(\overline{x_k}^k)$. We can rewrite terms in the same way as in **Case 4-2-2**. Then apply the IH to σ_2 and substitution $\mathcal{N}[\![\phi]\!]$.

□

B.4 Inversion of Evaluation

Note that when we reduce an if term, sometimes the size of the top-level if term is not decreased because we evaluate the congruent rules. Specifically, it refers to the evaluation of let binding and scrutinee. Therefore, we need two auxiliary lemmas which skip the evaluation of congruent rules and extract the final value.

Lemma B.11 (Inversion of Let Bindings in Splits). *For all term* $t = \textbf{if}\ \{\ \textbf{let}\ x_1 = t_1;\ \sigma_1\ \}$ *if* $t \overset{*}{\rightsquigarrow} t' = \textbf{if}\ \{\ \textbf{else}\ t\ \}$*, then there exists* v_1 *such that* $t_1 \overset{*}{\rightsquigarrow} v_1$ *and* $t \overset{*}{\rightsquigarrow} \textbf{if}\ \{\ \textbf{let}\ x_1 = v_1;\ \sigma_1\ \}$.

Proof. Let $t = t_{(0)} \rightsquigarrow t_{(1)} \rightsquigarrow \ldots \rightsquigarrow t_{(n)} = t'$ be the evaluation sequence from t to t'. Observe the shape of t. Evaluation rule E-Ctx applies to t with evaluation context $E[\square] = \textbf{if}\ \{\ \textbf{let}\ x_1 = \square;\ \sigma_1\ \}$. By induction on the length of the evaluation sequence. Consider the base case. If t_1 is a value, then immeidate. Otherwise, if t_1 is not a value, then there exists t_1' such that $t_1 \rightsquigarrow t_1'$. Thus, $t_{(1)} = \textbf{if}\ \{\ \textbf{let}\ x_1 = t_1';\ \sigma_1\ \}$. By the induction hypothesis, there exists v_1 such that $t_1' \overset{*}{\rightsquigarrow} v_1$ and $t_{(1)} \overset{*}{\rightsquigarrow} \textbf{if}\ \{\ \textbf{let}\ x_1 = v_1;\ \sigma_1\ \}$. □

Lemma B.12 (Inversion of Scrutinees in Splits). *For all term* $t = \textbf{if}\ \{\, t_1\ \textbf{is}\ C(\overline{x_i}^{\,i}) \to \{\, \sigma_1 \,\};\ \sigma_2 \,\}$, *if* $t \overset{*}{\rightsquigarrow} t' = \textbf{if}\ \{\, \textbf{else}\ t \,\}$, *then there exists* v_1 *such that* $t_1 \overset{*}{\rightsquigarrow} v_1$ *and* $t \overset{*}{\rightsquigarrow} \textbf{if}\ \{\, v_1\ \textbf{is}\ C(\overline{x_i}^{\,i}) ;\ \sigma_1 \,\}$.

Proof. Similar to the proof of Lemma B.11. □

Lemma B.13 (Inversion of Variables in Splits). *For all term* $t = \textbf{if}\ \{\, x_1\ \textbf{is}\ C(\overline{x_i}^{\,i}) \to \{\, \sigma_1 \,\};\ \sigma_2 \,\}$ *and substitution* $\phi = \overline{x_j \mapsto v_j}^{\,j}$, *if* $[\phi]\ t \overset{*}{\rightsquigarrow} t' = \textbf{if}\ \{\, \textbf{else}\ t \,\}$, *then there exists* v_1 *such that* $x_1 \mapsto v_1$ *belongs to* ϕ.

Proof. From $[\phi]\ t = \textbf{if}\ \{\, [\phi]\ x_1\ \textbf{is}\ C(\overline{x_i}^{\,i}) \to \{\, [\phi]\ \sigma_1 \,\};\ [\phi]\ \sigma_2 \,\}$, we can see that only rule E-Ctx applies to $[\phi]\ t$ with evaluation context $E[\square] = \textbf{if}\ \{\, \square\ \textbf{is}\ C(\overline{x_i}^{\,i}) \to \{\, [\phi]\ \sigma_1 \,\};\ [\phi]\ \sigma_2 \,\}$. Assume ϕ does not contain x_1, then $[\phi]\ x_1 = x_1$ and $[\phi]\ t$ does not evaluate. Also, $t \neq t'$, thus the assumption leads to a contradiction. Therefore, ϕ contains a substitution $x_1 \mapsto v_1$ for some value v_1. □

B.5 Distributivity of Specialization

Lemma B.14 (Distributivity of Specialization). *For all splits* σ_1, σ_2, *constructor* C_1, *and variables* $x_1, \overline{x_i}^{\,i}$. *we have* $\mathcal{S}^{\pm}_{x_1\ \textbf{is}\ C_1(\overline{x_i}^{\,i})}[\![\sigma_1]\!] +\!\!+ \mathcal{S}^{\pm}_{x_1\ \textbf{is}\ C_1(\overline{x_i}^{\,i})}[\![\sigma_2]\!] = \mathcal{S}^{\pm}_{x_1\ \textbf{is}\ C_1(\overline{x_i}^{\,i})}[\![\sigma_1 +\!\!+ \sigma_2]\!]$.

Proof. For the sake of brevity, we denote $\mathcal{S}^{\pm}_{x_1\ \textbf{is}\ C_1(\overline{x_i}^{\,i})}[\![\sigma_1]\!]$ by $\mathcal{S}^{\pm}[\![\sigma_1]\!]$. We consider $\mathcal{S}^{+}$ and $\mathcal{S}^{-}$ separately. Both of them are proved by induction on the size of σ_1.

We first consider $\mathcal{S}^{+}$.

Case 1 Suppose $\sigma_1 = x_2\ \textbf{is}\ C_2(\overline{y_i}^{\,i}) \to \sigma_3\ ;\ \sigma_4$.

Case 1-1 Suppose $x_1 = x_2 \wedge C_1 = C_2$. By

$$\mathcal{S}^{+}[\![x_1\ \textbf{is}\ C_1(\overline{y_i}^{\,i}) \to \sigma_3\ ;\ \sigma_4]\!] +\!\!+ \mathcal{S}^{+}[\![\sigma_2]\!] = \mathcal{S}^{+}[\![[\overline{y_i \mapsto x_i}^{\,i}]\ \sigma_3 +\!\!+ \sigma_4]\!] +\!\!+ \mathcal{S}^{+}[\![\sigma_2]\!] \tag{71}$$

and

$$\begin{aligned}\mathcal{S}^{+}[\![(x_1\ \textbf{is}\ C_1(\overline{y_i}^{\,i}) \to \sigma_3\ ;\ \sigma_4) +\!\!+ \sigma_2]\!] &= \mathcal{S}^{+}[\![x_1\ \textbf{is}\ C_1(\overline{y_i}^{\,i}) \to \sigma_3\ ;\ (\sigma_4 +\!\!+ \sigma_2)]\!] &(72)\\ &= \mathcal{S}^{+}[\![[\overline{y_i \mapsto x_i}^{\,i}]\ \sigma_3 +\!\!+ (\sigma_4 +\!\!+ \sigma_2)]\!] &(73)\\ &= \mathcal{S}^{+}[\![([\overline{y_i \mapsto x_i}^{\,i}]\ \sigma_3 +\!\!+ \sigma_4) +\!\!+ \sigma_2]\!] &(74)\end{aligned}$$

The size of $[\overline{y_i \mapsto x_i}^{\,i}]\ \sigma_3$ is equal to the size of σ_3. By the induction hypothesis, *Eq.* (71) = *Eq.* (74).

Case 1-2 Suppose $x_1 = x_2 \wedge C_1 \neq C_2$. By

$$\mathcal{S}^{+}[\![x_1\ \textbf{is}\ C_2(\overline{y_i}^{\,i}) \to \sigma_3\ ;\ \sigma_4]\!] +\!\!+ \mathcal{S}^{+}[\![\sigma_2]\!] = \mathcal{S}^{+}[\![\sigma_4]\!] +\!\!+ \mathcal{S}^{+}[\![\sigma_2]\!] \tag{75}$$

and

$$\begin{aligned}\mathcal{S}^{+}[\![(x_1\ \textbf{is}\ C_2(\overline{y_i}^{\,i}) \to \sigma_3\ ;\ \sigma_4) +\!\!+ \sigma_2]\!] &= \mathcal{S}^{+}[\![x_1\ \textbf{is}\ C_2(\overline{y_i}^{\,i}) \to \sigma_3\ ;\ (\sigma_4 +\!\!+ \sigma_2)]\!] &(76)\\ &= \mathcal{S}^{+}[\![\sigma_4 +\!\!+ \sigma_2]\!] &(77)\end{aligned}$$

By the induction hypothesis, *Eq.* (75) = *Eq.* (77).

Case 1-3 Suppose $x_1 \neq x_2$. By

$$\begin{aligned}&\mathcal{S}^{+}[\![x_1\ \textbf{is}\ C_2(\overline{y_i}^{\,i}) \to \sigma_3\ ;\ \sigma_4]\!] +\!\!+ \mathcal{S}^{+}[\![\sigma_2]\!] &(78)\\ =\ &x_1\ \textbf{is}\ C_2(\overline{y_i}^{\,i}) \to \mathcal{S}^{+}[\![\sigma_3]\!]\ ;\ \mathcal{S}^{+}[\![\sigma_4]\!] +\!\!+ \mathcal{S}^{+}[\![\sigma_2]\!] &(79)\\ =\ &x_1\ \textbf{is}\ C_2(\overline{y_i}^{\,i}) \to \mathcal{S}^{+}[\![\sigma_3]\!]\ ;\ (\mathcal{S}^{+}[\![\sigma_4]\!] +\!\!+ \mathcal{S}^{+}[\![\sigma_2]\!]) &(80)\end{aligned}$$

and

$$\mathcal{S}^+[\![(x_1 \textbf{ is } C_2(\overline{y_i}^i) \to \sigma_3\,;\ \sigma_4) +\!\!+ \sigma_2]\!] = \mathcal{S}^+[\![x_1 \textbf{ is } C_2(\overline{y_i}^i) \to \sigma_3\,;\ (\sigma_4 +\!\!+ \sigma_2)]\!] \tag{81}$$

$$= x_1 \textbf{ is } C_2(\overline{y_i}^i) \to \mathcal{S}^+[\![\sigma_3]\!]\,;\ \mathcal{S}^+[\![\sigma_4 +\!\!+ \sigma_2]\!] \tag{82}$$

By the induction hypothesis, $\mathcal{S}^+[\![\sigma_4]\!] +\!\!+ \mathcal{S}^+[\![\sigma_2]\!] = \mathcal{S}^+[\![\sigma_4 +\!\!+ \sigma_2]\!]$, thus *Eq.* (75) = *Eq.* (77).

Case 2 Suppose $\sigma_1 = (\textbf{let } x_2 = t_1\,;\ \sigma_3)$. By

$$\mathcal{S}^+[\![\textbf{let } x_2 = t_1\,;\ \sigma_3]\!] +\!\!+ \mathcal{S}^+[\![\sigma_2]\!] = (\textbf{let } x_2 = t_1\,;\ \mathcal{S}^+[\![\sigma_3]\!]) +\!\!+ \mathcal{S}^+[\![\sigma_2]\!]$$
$$= \textbf{let } x_2 = t_1\,;\ (\mathcal{S}^+[\![\sigma_3]\!] +\!\!+ \mathcal{S}^+[\![\sigma_2]\!])$$

and

$$\mathcal{S}^+[\![(\textbf{let } x_2 = t_1\,;\ \sigma_3) +\!\!+ \sigma_2]\!] = \mathcal{S}^+[\![\textbf{let } x_2 = t_1\,;\ (\sigma_3 +\!\!+ \sigma_2)]\!]$$
$$= \textbf{let } x_2 = t_1\,;\ \mathcal{S}^+[\![\sigma_3 +\!\!+ \sigma_2]\!].$$

By the induction hypothesis, $\mathcal{S}^+[\![\sigma_3]\!] +\!\!+ \mathcal{S}^+[\![\sigma_2]\!] = \mathcal{S}^+[\![\sigma_3 +\!\!+ \sigma_2]\!]$, thus $\mathcal{S}^+[\![\sigma_1]\!] +\!\!+ \mathcal{S}^+[\![\sigma_2]\!] = \mathcal{S}^+[\![\sigma_1 +\!\!+ \sigma_2]\!]$.

Case 3 Suppose $\sigma_1 = \textbf{else } t_1$. By $\mathcal{S}^+[\![\textbf{else } t_1]\!] +\!\!+ \mathcal{S}^+[\![\sigma_2]\!] = \textbf{else } t_1 +\!\!+ \mathcal{S}^+[\![\sigma_2]\!] = \textbf{else } t_1$and $\mathcal{S}^+[\![\textbf{else } t_1 +\!\!+ \sigma_2]\!] = \mathcal{S}^+[\![\textbf{else } t_1]\!] = \textbf{else } t_1$.

Case 4 Suppose $\sigma_1 = \varepsilon$. By $\mathcal{S}^+[\![\varepsilon]\!] +\!\!+ \mathcal{S}^+[\![\sigma_2]\!] = \varepsilon +\!\!+ \mathcal{S}^+[\![\sigma_2]\!] = \mathcal{S}^+[\![\sigma_2]\!]$and $\mathcal{S}^+[\![\varepsilon +\!\!+ \sigma_2]\!] = \mathcal{S}^+[\![\sigma_2]\!]$.

We next consider $\mathcal{S}^-$. The cases where $\sigma_1 = (\textbf{let } x_2 = t_1\,;\ \sigma_3)$, $\sigma_1 = \textbf{else } t_1$, and $\sigma_1 = \varepsilon$ are analogous to $\mathcal{S}^+$. We only focus on the case of $\sigma_1 = x_2 \textbf{ is } C_2(\overline{y_i}^i)$ and proceed by a case analysis on x_2 and C_2.

Case 1 Suppose $x_1 = x_2 \wedge C_1 = C_2$. By

$$\mathcal{S}^-[\![x_1 \textbf{ is } C_1(\overline{y_i}^i) \to \sigma_3\,;\ \sigma_4]\!] +\!\!+ \mathcal{S}^-[\![\sigma_2]\!] = \mathcal{S}^-[\![\sigma_4]\!] +\!\!+ \mathcal{S}^-[\![\sigma_2]\!] \tag{83}$$

and

$$\mathcal{S}^-[\![(x_1 \textbf{ is } C_1(\overline{y_i}^i) \to \sigma_3\,;\ \sigma_4) +\!\!+ \sigma_2]\!] = \mathcal{S}^-[\![x_1 \textbf{ is } C_1(\overline{y_i}^i) \to \sigma_3\,;\ (\sigma_4 +\!\!+ \sigma_2)]\!] \tag{84}$$

$$= \mathcal{S}^-[\![\sigma_4 +\!\!+ \sigma_2]\!] \tag{85}$$

By the induction hypothesis, *Eq.* (83) = *Eq.* (85).

Case 2 Suppose $x_1 \neq x_2 \vee C_1 \neq C_2$. By

$$\mathcal{S}^-[\![x_2 \textbf{ is } C_2(\overline{y_i}^i) \to \sigma_3\,;\ \sigma_4]\!] +\!\!+ \mathcal{S}^-[\![\sigma_2]\!] \tag{86}$$

$$= x_2 \textbf{ is } C_2(\overline{y_i}^i) \to \mathcal{S}^-[\![\sigma_3]\!]\,;\ \mathcal{S}^-[\![\sigma_4]\!] +\!\!+ \mathcal{S}^-[\![\sigma_2]\!] \tag{87}$$

$$= x_2 \textbf{ is } C_2(\overline{y_i}^i) \to \mathcal{S}^-[\![\sigma_3]\!]\,;\ (\mathcal{S}^-[\![\sigma_4]\!] +\!\!+ \mathcal{S}^-[\![\sigma_2]\!]) \tag{88}$$

and

$$\mathcal{S}^-[\![(x_1 \textbf{ is } C_2(\overline{y_i}^i) \to \sigma_3\,;\ \sigma_4) +\!\!+ \sigma_2]\!] = \mathcal{S}^-[\![x_2 \textbf{ is } C_2(\overline{y_i}^i) \to \sigma_3\,;\ (\sigma_4 +\!\!+ \sigma_2)]\!] \tag{89}$$

$$= x_2 \textbf{ is } C_2(\overline{y_i}^i) \to \mathcal{S}^-[\![\sigma_3]\!]\,;\ \mathcal{S}^-[\![\sigma_4 +\!\!+ \sigma_2]\!] \tag{90}$$

By the induction hypothesis, $\mathcal{S}^-[\![\sigma_4]\!] +\!\!+ \mathcal{S}^-[\![\sigma_2]\!] = \mathcal{S}^-[\![\sigma_4 +\!\!+ \sigma_2]\!]$, thus *Eq.* (88) = *Eq.* (90).

□

B.6 Invariance of Specialization under Irrelevant Variable Substitutions

Lemma B.15. *For all split σ, constructor C_1, variables $x_1, \overline{x_i}^{i\in[1,n]}$ and variable substitution $\phi = \overline{x'_j \mapsto x''_j}^{j\in[1,m]}$, if $[\phi]\, x_1 = x_1$, then $\mathcal{S}^+_{x_1 \textbf{ is } C_1(\overline{x_i}^i)}[\![\,[\phi]\,\sigma\,]\!] = [\phi]\,\mathcal{S}^+_{x_1 \textbf{ is } C_1(\overline{x_i}^i)}[\![\,\sigma\,]\!]$.*

Proof. For the simplicity, for all σ, we abbreviate $\mathcal{S}^+_{x_1 \textbf{ is } C(\overline{x_i}^i)}[\![\,\sigma\,]\!]$ to $\mathcal{S}^+[\![\,\sigma\,]\!]$ in this proof. By induction on the size of σ, we proceed by a case analysis on σ.

Case 1 Suppose $\sigma = (\textbf{let } x_2 = t_1\,;\ \sigma_1)$. By

$$\mathcal{S}^+[\![\,[\phi]\ \textbf{let } x_2 = t_1\,;\ \sigma_1\,]\!] = \mathcal{S}^+[\![\,\textbf{let } x_2 = [\phi]\,t_1\,;\ [\phi]\,\sigma_1\,]\!] \tag{91}$$
$$= \textbf{let } x_2 = [\phi]\,t_1\,;\ \mathcal{S}^+[\![\,[\phi]\,\sigma_1\,]\!] \tag{92}$$

and

$$[\phi]\,\mathcal{S}^+[\![\,\textbf{let } x_2 = t_1\,;\ \sigma_1\,]\!] = [\phi]\ \textbf{let } x_2 = t_1\,;\ \mathcal{S}^+[\![\,\sigma_1\,]\!] \tag{93}$$
$$= \textbf{let } x_2 = [\phi]\,t_1\,;\ [\phi]\,\mathcal{S}^+[\![\,\sigma_1\,]\!]. \tag{94}$$

By the induction hypothesis, $\mathcal{S}^+[\![\,[\phi]\,\sigma_1\,]\!] = [\phi]\,\mathcal{S}^+[\![\,\sigma_1\,]\!]$. Therefore, *Eq.* (92) = *Eq.* (94).

Case 2 Suppose $\sigma = x_2 \textbf{ is } C_2(\overline{y_k}^{k\in[1,p]}) \to \{\,\sigma_1\,\}\,;\ \sigma_2$. We proceed by a further case analysis on whether $x_1 = x_2$ and $C_1 = C_2$.

Case 2-1 Suppose $x_1 = x_2 \wedge C_1 = C_2$. By

$$\mathcal{S}^+[\![\,[\phi]\,\sigma_1\,]\!] = \mathcal{S}^+[\![\,[\phi]\ (x_1 \textbf{ is } C_1(\overline{y_i}^{i\in[1,n]}) \to \{\,\sigma_1\,\}\,;\ \sigma_2)\,]\!] \tag{95}$$
$$= \mathcal{S}^+[\![\,[\phi]\,x_1 \textbf{ is } C_1(\overline{y_i}^{i\in[1,n]}) \to \{\,[\phi]\,\sigma_1\,\}\,;\ [\phi]\,\sigma_2\,]\!] \tag{96}$$
$$= \mathcal{S}^+[\![\,[\overline{y_i \mapsto x_i}^{i\in[1,n]}]\ [\phi]\,\sigma_1 +\!\!+ [\phi]\,\sigma_2\,]\!] \tag{97}$$
$$= \mathcal{S}^+[\![\,[\overline{y_i \mapsto x_i}^{i\in[1,n]}]\ [\phi]\ (\sigma_1 +\!\!+ \sigma_2)\,]\!] \tag{98}$$

and

$$[\phi]\,\mathcal{S}^+[\![\,\sigma_1\,]\!] = [\phi]\,\mathcal{S}^+[\![\,x_1 \textbf{ is } C_1(\overline{y_i}^{i\in[1,n]}) \to \{\,\sigma_1\,\}\,;\ \sigma_2\,]\!] \tag{99}$$
$$= [\phi]\,\mathcal{S}^+[\![\,[\overline{y_i \mapsto x_i}^{i\in[1,n]}]\,\sigma_1 +\!\!+ \sigma_2\,]\!] \tag{100}$$
$$= [\phi]\,\mathcal{S}^+[\![\,[\overline{y_i \mapsto x_i}^{i\in[1,n]}]\ (\sigma_1 +\!\!+ \sigma_2)\,]\!]. \tag{101}$$

Observe that variables $\overline{y_i}^{i\in[1,n]}$ are free in σ_1, while ϕ comes from outside splits. We can safely exchange the order of $\overline{y_i}^{i\in[1,n]}$ and ϕ. By the induction hypothesis,

$$\textit{Eq.}\ (98) = \mathcal{S}^+[\![\,[\phi]\ [\overline{y_i \mapsto x_i}^{i\in[1,n]}]\ (\sigma_1 +\!\!+ \sigma_2)\,]\!] \tag{102}$$
$$= [\phi]\,\mathcal{S}^+[\![\,[\overline{y_i \mapsto x_i}^{i\in[1,n]}]\ (\sigma_1 +\!\!+ \sigma_2)\,]\!] = \textit{Eq.}\ (101). \tag{103}$$

Case 2-2 Suppose $x_1 = x_2 \wedge C_1 \neq C_2$. By

$$\mathcal{S}^+[\![\,[\phi]\,\sigma_1\,]\!] = \mathcal{S}^+[\![\,[\phi]\ (x_1 \textbf{ is } C_2(\overline{y_k}^{k\in[1,p]}) \to \{\,\sigma_1\,\}\,;\ \sigma_2)\,]\!] \tag{104}$$
$$= \mathcal{S}^+[\![\,[\phi]\,x_1 \textbf{ is } C_2(\overline{y_k}^{k\in[1,p]}) \to \{\,[\phi]\,\sigma_1\,\}\,;\ [\phi]\,\sigma_2\,]\!] \tag{105}$$
$$= \mathcal{S}^+[\![\,[\phi]\,\sigma_2\,]\!] \tag{106}$$

and

$$[\phi]\,\mathcal{S}^+[\![\,\sigma_1\,]\!] = [\phi]\,\mathcal{S}^+[\![\,x_1 \textbf{ is } C_2(\overline{y_k}^{k\in[1,p]}) \to \{\,\sigma_1\,\}\,;\ \sigma_2\,]\!] = [\phi]\,\mathcal{S}^+[\![\,\sigma_2\,]\!]. \tag{107}$$

By the induction hypothesis, *Eq.* (106) = *Eq.* (107)

Case 2-3 Suppose $x_1 \neq x_2$. By

$$\mathcal{S}^+[\![[\phi]\ \sigma_1]\!] = \mathcal{S}^+[\![[\phi]\ (x_2 \textbf{ is } C_2(\overline{y_k}^{k\in[1,p]}) \to \{\ \sigma_1\ \};\ \sigma_2)]\!] \tag{108}$$

$$= \mathcal{S}^+[\![[\phi]\ x_2 \textbf{ is } C_2(\overline{y_k}^{k\in[1,p]}) \to \{\ [\phi]\ \sigma_1\ \};\ [\phi]\ \sigma_2]\!] \tag{109}$$

$$= [\phi]\ x_2 \textbf{ is } C_2(\overline{y_k}^{k\in[1,p]}) \to \{\ \mathcal{S}^+[\![[\phi]\ \sigma_1]\!]\ \};\ \mathcal{S}^+[\![[\phi]\ \sigma_2]\!] \tag{110}$$

and

$$[\phi]\ \mathcal{S}^+[\![\sigma_1]\!] = [\phi]\ \mathcal{S}^+[\![x_2 \textbf{ is } C_2(\overline{y_i}^{i\in[1,n]}) \to \{\ \sigma_1\ \};\ \sigma_2]\!] \tag{111}$$

$$= [\phi]\ (x_2 \textbf{ is } C_2(\overline{y_i}^{i\in[1,n]}) \to \{\ \mathcal{S}^+[\![\sigma_1]\!]\ \};\ \mathcal{S}^+[\![\sigma_2]\!]) \tag{112}$$

$$= [\phi]\ x_2 \textbf{ is } C_2(\overline{y_i}^{i\in[1,n]}) \to \{\ [\phi]\ \mathcal{S}^+[\![\sigma_1]\!]\ \};\ [\phi]\ \mathcal{S}^+[\![\sigma_2]\!]. \tag{113}$$

By the induction hypothesis, $\mathcal{S}^+[\![[\phi]\ \sigma_1 = [\phi]\ \mathcal{S}^+[\![\sigma_1]\!]]\!]$ and $\mathcal{S}^+[\![[\phi]\ \sigma_2]\!] = [\phi]\ \mathcal{S}^+[\![\sigma_2]\!]$. Thus, *Eq.* (110) = *Eq.* (113).

Case 3 Suppose $\sigma = \textbf{else } t_1$. By $[\phi]\ \mathcal{S}^+[\![\textbf{else } t_1]\!] = [\phi]\ \textbf{else } t_1 = \mathcal{S}^+[\![[\phi]\ \textbf{else } t_1]\!]$.

Case 4 Suppose $\sigma = \varepsilon$. By $\mathcal{S}^+[\![[\phi]\ \varepsilon]\!] = [\phi]\ \mathcal{S}^+[\![\varepsilon]\!]$

□

B.7 Commutativity of Specialization

Lemma B.16. *For all σ, x_1, x_2, $C(\overline{x_i}^i)$, and $D(\overline{y_i}^i)$, if $x_1 \neq x_2$ and $C \neq D$, then*

$$\mathcal{S}^\pm_{x_1 \textbf{ is } C(\overline{x_i}^i)}[\![\mathcal{S}^\pm_{x_2 \textbf{ is } D(\overline{x_j}^j)}[\![\sigma]\!]]\!] = \mathcal{S}^\pm_{x_2 \textbf{ is } D(\overline{x_j}^j)}[\![\mathcal{S}^\pm_{x_1 \textbf{ is } C(\overline{x_i}^i)}[\![\sigma]\!]]\!].$$

For the sake of simplicity, for all σ, we denote $\mathcal{S}^\pm_{x_1 \textbf{ is } C_1(\overline{x_i}^i)}[\![\sigma]\!]$ by $\mathcal{S}^\pm_\square[\![\sigma]\!]$ and denote $\mathcal{S}^\pm_{x_2 \textbf{ is } C_2(\overline{x_j}^j)}[\![\sigma]\!]$ by $\mathcal{S}^\pm_\diamond[\![\sigma]\!]$ in the following proof.

Proof. By induction on the size of σ, we proceed by case analysis on rules of $\mathcal{S}^\pm$. It is easy to check cases $\sigma = (\textbf{let } x = t;\ \sigma_1)$, $\sigma = (\textbf{else } t)$, and $\sigma = \varepsilon$. Next, we consider the case where $\sigma = y \textbf{ is } D(\overline{y_k}^k)$.

Case 1 Suppose $y = x_1$.

Case 1-1 Suppose $D = C_1$.

Case 1-1-1 Consider $\mathcal{S}^+_\square[\![\mathcal{S}^\pm_\diamond[\![\sigma]\!]]\!]$. By

$$\mathcal{S}^+_\square[\![\mathcal{S}^\pm_\diamond[\![\sigma]\!]]\!] = \mathcal{S}^+_\square[\![\mathcal{S}^\pm_\diamond[\![x_1 \textbf{ is } C_1(\overline{x_i}^i) \to \{\ \sigma_1\ \};\ \sigma_2]\!]]\!] \tag{114}$$

$$= \mathcal{S}^+_\square[\![x_1 \textbf{ is } C_1(\overline{x_i}^i) \to \{\ \mathcal{S}^\pm_\diamond[\![\sigma_1]\!]\ \};\ \mathcal{S}^\pm_\diamond[\![\sigma_2]\!]]\!] \tag{115}$$

$$= \mathcal{S}^+_\square[\![\mathcal{S}^\pm_\diamond[\![\sigma_1]\!] +\!\!+ \mathcal{S}^\pm_\diamond[\![\sigma_2]\!]]\!] \tag{116}$$

$$= \mathcal{S}^+_\square[\![\mathcal{S}^\pm_\diamond[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!] \tag{117}$$

and

$$\mathcal{S}^\pm_\diamond[\![\mathcal{S}^+_\square[\![\sigma]\!]]\!] = \mathcal{S}^\pm_\diamond[\![\mathcal{S}^+_\square[\![x_1 \textbf{ is } C_1(\overline{x_i}^i) \to \{\ \sigma_1\ \};\ \sigma_2]\!]]\!] \tag{118}$$

$$= \mathcal{S}^\pm_\diamond[\![\mathcal{S}^+_\square[\![\sigma_1 +\!\!+ \sigma_2]\!]]\!] \tag{119}$$

By the induction hypothesis, *Eq.* (117) = *Eq.* (119).

Case 1-1-2 Consider $\mathcal{S}^-_\square[\![\mathcal{S}^\pm_\diamond[\![\sigma]\!]]\!]$. By

$$\mathcal{S}^-_\square[\![\mathcal{S}^\pm_\diamond[\![\sigma]\!]]\!] = \mathcal{S}^-_\square[\![\mathcal{S}^\pm_\diamond[\![x_1 \textbf{ is } C_1(\overline{x_i}^i) \to \{\ \sigma_1\ \};\ \sigma_2]\!]]\!] \tag{120}$$

$$= \mathcal{S}^-_\square[\![x_1 \textbf{ is } C_1(\overline{x_i}^i) \to \{\ \mathcal{S}^\pm_\diamond[\![\sigma_1]\!]\ \};\ \mathcal{S}^\pm_\diamond[\![\sigma_2]\!]]\!] \tag{121}$$

$$= \mathcal{S}^-_\square[\![\mathcal{S}^\pm_\diamond[\![\sigma_2]\!]]\!] \tag{122}$$

and

$$\mathcal{S}^{\pm}_{\Diamond}[\![\mathcal{S}^{-}_{\Box}[\![\sigma]\!]]\!] = \mathcal{S}^{\pm}_{\Diamond}[\![\mathcal{S}^{-}_{\Box}[\![x_1 \textbf{ is } C_1(\overline{x_i}^{\,i}) \to \{\,\sigma_1\,\};\ \sigma_2]\!]]\!] \tag{123}$$
$$= \mathcal{S}^{\pm}_{\Diamond}[\![\mathcal{S}^{-}_{\Box}[\![\sigma_2]\!]]\!] \tag{124}$$

By the induction hypothesis, *Eq.* (122) = *Eq.* (124).

Case 1-2 Suppose $D \neq C_1$.

Case 1-2-1 Consider $\mathcal{S}^{+}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![\sigma]\!]]\!]$. By

$$\mathcal{S}^{+}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![\sigma]\!]]\!] = \mathcal{S}^{+}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![x_1 \textbf{ is } D(\overline{y_k}^{\,k}) \to \{\,\sigma_1\,\};\ \sigma_2]\!]]\!] \tag{125}$$
$$= \mathcal{S}^{+}_{\Box}[\![x_1 \textbf{ is } D(\overline{y_k}^{\,k}) \to \{\,\mathcal{S}^{\pm}_{\Diamond}[\![\sigma_1]\!]\,\};\ \mathcal{S}^{\pm}_{\Diamond}[\![\sigma_2]\!]]\!] \tag{126}$$
$$= \mathcal{S}^{+}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![\sigma_2]\!]]\!] \tag{127}$$

and

$$\mathcal{S}^{\pm}_{\Diamond}[\![\mathcal{S}^{+}_{\Box}[\![\sigma]\!]]\!] = \mathcal{S}^{\pm}_{\Diamond}[\![\mathcal{S}^{+}_{\Box}[\![x_1 \textbf{ is } D(\overline{y_k}^{\,k}) \to \{\,\sigma_1\,\};\ \sigma_2]\!]]\!] \tag{128}$$
$$= \mathcal{S}^{\pm}_{\Diamond}[\![\mathcal{S}^{+}_{\Box}[\![\sigma_2]\!]]\!] \tag{129}$$

By the induction hypothesis, *Eq.* (127) = *Eq.* (129).

Case 1-2-2 Consider $\mathcal{S}^{-}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![\sigma]\!]]\!]$. By

$$\mathcal{S}^{-}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![\sigma]\!]]\!] = \mathcal{S}^{-}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![x_1 \textbf{ is } D(\overline{y_k}^{\,k}) \to \{\,\sigma_1\,\};\ \sigma_2]\!]]\!] \tag{130}$$
$$= \mathcal{S}^{-}_{\Box}[\![x_1 \textbf{ is } D(\overline{y_k}^{\,k}) \to \{\,\mathcal{S}^{\pm}_{\Diamond}[\![\sigma_1]\!]\,\};\ \mathcal{S}^{\pm}_{\Diamond}[\![\sigma_2]\!]]\!] \tag{131}$$
$$= x_1 \textbf{ is } D(\overline{y_k}^{\,k}) \to \{\,\mathcal{S}^{-}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![\sigma_1]\!]]\!]\,\};\ \mathcal{S}^{-}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![\sigma_2]\!]]\!] \tag{132}$$

and

$$\mathcal{S}^{\pm}_{\Diamond}[\![\mathcal{S}^{-}_{\Box}[\![\sigma]\!]]\!] = \mathcal{S}^{\pm}_{\Diamond}[\![\mathcal{S}^{-}_{\Box}[\![x_1 \textbf{ is } D(\overline{y_k}^{\,k}) \to \{\,\sigma_1\,\};\ \sigma_2]\!]]\!] \tag{133}$$
$$= \mathcal{S}^{\pm}_{\Diamond}[\![x_1 \textbf{ is } D(\overline{y_k}^{\,k}) \to \{\,\mathcal{S}^{-}_{\Box}[\![\sigma_1]\!]\,\};\ \mathcal{S}^{-}_{\Box}[\![\sigma_2]\!]]\!] \tag{134}$$
$$= x_1 \textbf{ is } D(\overline{y_k}^{\,k}) \to \{\,\mathcal{S}^{\pm}_{\Diamond}[\![\mathcal{S}^{-}_{\Box}[\![\sigma_1]\!]]\!]\,\};\ \mathcal{S}^{\pm}_{\Diamond}[\![\mathcal{S}^{-}_{\Box}[\![\sigma_2]\!]]\!] \tag{135}$$

By the induction hypothesis, $\mathcal{S}^{-}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![\sigma_1]\!]]\!] = \mathcal{S}^{\pm}_{\Diamond}[\![\mathcal{S}^{-}_{\Box}[\![\sigma_1]\!]]\!]$ and $\mathcal{S}^{-}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![\sigma_2]\!]]\!] = \mathcal{S}^{\pm}_{\Diamond}[\![\mathcal{S}^{-}_{\Box}[\![\sigma_2]\!]]\!]$. Therefore, *Eq.* (132) = *Eq.* (135).

Case 2 Suppose $y = x_2$. We can proceed by a case analysis on whether $D = C_2$ or not. Every sub-case is symmetrical to the corresponding sub-case in **Case 1**, with the only difference being that x_2 can be reduced by $\mathcal{S}^{\pm}_{\Diamond}[\![\sigma]\!]$.

Case 3 Suppose $y \neq x_1 \wedge y \neq x_2$. We have

$$\mathcal{S}^{\pm}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![\sigma]\!]]\!] = \mathcal{S}^{\pm}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![y \textbf{ is } D(\overline{y_k}^{\,k}) \to \{\,\sigma_1\,\};\ \sigma_2]\!]]\!] \tag{136}$$
$$= \mathcal{S}^{\pm}_{\Box}[\![y \textbf{ is } D(\overline{y_k}^{\,k}) \to \{\,\mathcal{S}^{\pm}_{\Diamond}[\![\sigma_1]\!]\,\};\ \mathcal{S}^{\pm}_{\Diamond}[\![\sigma_2]\!]]\!] \tag{137}$$
$$= y \textbf{ is } D(\overline{y_k}^{\,k}) \to \{\,\mathcal{S}^{\pm}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![\sigma_1]\!]]\!]\,\};\ \mathcal{S}^{\pm}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![\sigma_2]\!]]\!] \tag{138}$$

By the induction hypothesis, we have $\mathcal{S}^{\pm}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![\sigma_1]\!]]\!] = \mathcal{S}^{\pm}_{\Diamond}[\![\mathcal{S}^{\pm}_{\Box}[\![\sigma_2]\!]]\!]$ and $\mathcal{S}^{\pm}_{\Box}[\![\mathcal{S}^{\pm}_{\Diamond}[\![\sigma_2]\!]]\!] = \mathcal{S}^{\pm}_{\Diamond}[\![\mathcal{S}^{\pm}_{\Box}[\![\sigma_2]\!]]\!]$. Then, we can rewrite Eq. (138) as $\mathcal{S}^{\pm}_{\Diamond}[\![\mathcal{S}^{\pm}_{\Box}[\![\sigma]\!]]\!]$.

□

B.8 Miscellaneous Lemmas

Lemma B.17 (Associativity of Split Concatenation). *For all splits σ_1, σ_2 and σ_3, $(\sigma_1 +\!\!+ \sigma_2) +\!\!+ \sigma_3 = \sigma_1 +\!\!+ (\sigma_2 +\!\!+ \sigma_3)$.*

Lemma B.18 (The Size of Specialized Splits). *For all split σ, variable x and pattern $C(\overline{x_i}^i)$, $size(\mathcal{S}^{\pm}_{x \text{ is } C(\overline{x_i}^i)}[\![\sigma]\!]) \leq size(\sigma)$.*

Received 2024-04-06; accepted 2024-08-18