Set-Theoretic Types for Polymorphic Variants

Giuseppe Castagna, Tommaso Petrucciani, Kim Nguyen

Set-Theoretic Types for Polymorphic Variants

Giuseppe Castagna[1] Tommaso Petrucciani[1,2] Kim Nguyễn[3]

[1]CNRS, Univ Paris Diderot, Sorbonne Paris Cité, Paris, France
[2]DIBRIS, Università degli Studi di Genova, Genova, Italy
[3]LRI, Université Paris-Sud, Orsay, France

Abstract

Polymorphic variants are a useful feature of the OCaml language whose current definition and implementation rely on kinding constraints to simulate a subtyping relation via unification. This yields an awkward formalization and results in a type system whose behaviour is in some cases unintuitive and/or unduly restrictive.

In this work, we present an alternative formalization of polymorphic variants, based on set-theoretic types and subtyping, that yields a cleaner and more streamlined system. Our formalization is more expressive than the current one (it types more programs while preserving type safety), it can internalize some meta-theoretic properties, and it removes some pathological cases of the current implementation resulting in a more intuitive and, thus, predictable type system. More generally, this work shows how to add full-fledged union types to functional languages of the ML family that usually rely on the Hindley-Milner type system. As an aside, our system also improves the theory of semantic subtyping, notably by proving completeness for the type reconstruction algorithm.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures; Polymorphism.

Keywords Type reconstruction, union types, type constraints.

1. Introduction

Polymorphic variants are a useful feature of OCaml, as they balance static safety and code reuse capabilities with a remarkable conciseness. They were originally proposed as a solution to add union types to Hindley-Milner (HM) type systems [17]. Union types have several applications and make it possible to deduce types that are finer grained than algebraic data types, especially in languages with pattern matching. Polymorphic variants cover several of the applications of union types, which explains their success; however they provide just a limited form of union types: although they offer some sort of subtyping and value sharing that ordinary variants do not, it is still not possible to form unions of values of generic types, but just finite enumerations of tagged values. This is obtained by superimposing on the HM type system a system of kinding constraints, which is used to simulate subtyping without actually introducing it. In general, the current system reuses the ML type system—including unification for type reconstruction—as much as possible. This is the source of several trade-offs which yield significant complexity, make polymorphic variants hard to understand (especially for beginners), and jeopardize expressiveness insofar as they forbid several useful applications that general union types make possible.

We argue that using a different system, one that departs drastically from HM, is advantageous. In this work we advocate the use of full-fledged union types (i.e., the original motivation of polymorphic variants) with standard set-theoretic subtyping. In particular we use *semantic subtyping* [15], a type system where (i) types are interpreted as set of values, (ii) they are enriched with unrestrained unions, intersections, and negations interpreted as the corresponding set-theoretic operations on sets of values, and (iii) subtyping corresponds to set containment. Using set-theoretic types and subtyping yields a much more natural and easy-to-understand system in which several key notions—e.g., bounded quantification and exhaustiveness and redundancy analyses of pattern matching—can be expressed directly by types; conversely, with the current formalization these notions need meta-theoretic constructions (in the case of kinding) or they are meta-theoretic properties not directly connected to the type theory (as for exhaustiveness and redundancy).

All in all, our proposal is not very original: in order to have the advantages of union types in an implicitly-typed language, we simply add them, instead of simulating them roughly and partially by polymorphic variants. This implies to generalize notions such as instantiation and generalization to cope with subtyping (and, thus, with unions). We chose not to start from scratch, but instead to build on the existing: therefore we show how to add unions as a modification of the type checker, that is, without disrupting the current syntax of OCaml. Nevertheless, our results can be used to add unions to other implicitly-typed languages of the ML family.

We said that the use of kinding constraints instead of full-fledged unions has several practical drawbacks and that the system may therefore result in unintuitive or overly restrictive behaviour. We illustrate this by the following motivating examples in OCaml.

EXAMPLE 1: ***loss of polymorphism.*** Let us consider the identity function and its application to a polymorphic variant in OCaml ("#" denotes the interactive toplevel prompt of OCaml, whose input is ended by a double semicolon and followed by the system response):

```
# let id x = x;;
val id : α → α = <fun>
# id `A;;
- : [> `A ] = `A
```

The identity function `id` has type[1] $\forall\alpha.\,\alpha \to \alpha$ (Greek letters denote type variables). Thus, when it is applied to the polymorphic variant value `` `A `` (polymorphic variants values are literals prefixed by a backquote), OCaml statically deduces that the result will be of

[1] Strictly speaking, it is a *type scheme*: cf. Definition 3.3.

A reduced version of this work will appear in the proceedings of *ICFP '16, the 21st ACM SIGPLAN International Conference on Functional Programming*, ACM, 2016.

ICFP '16 September 18–24, 2016, Nara, Japan

type "at least `A" (noted [> `A]), that is, of a type greater than or equal to the type whose only value is `A. Since the only value of type [> `A] is `A, then the value `A and the expression id `A are completely interchangeable.[2] For instance, we can use id `A where an expression of type "*at most* `A" (noted [< `A]) is expected:

```
# let f x = match x with `A → true;;
val f : [< `A ] → bool = <fun>
# f (id `A);;
- : bool = true
```

Likewise `A and id `A are equivalent in any context:

```
# [`A; `C];;
- : [> `A | `C ] list = [`A; `C]
# [(id `A); `C];;
- : [> `A | `C ] list = [`A; `C]
```

We now slightly modify the definition of the identity function:

```
# let id2 x = match x with `A | `B → x;;
val id2: ([< `A | `B ] as α) → α = <fun>
```

Since id2 maps x to x, it still is the identity function—so it has type $\alpha \to \alpha$—but since its argument is matched against `A | `B, this function can only be applied to arguments of type "at most `A | `B", where "|" denotes a union. Therefore, the type variable α must be constrained to be a "subtype" of `A | `B, that is, $\forall(\alpha \leq$ `A | `B$).\, \alpha \to \alpha$, expressed by the OCaml toplevel as val id2: ([< `A | `B] as α) → α.

A priori, this should not change the typing of the application of the (newly-defined) identity to `A, that is, id2 `A. It should still be statically known to have type "at least `A", and hence to be the value `A. However, this is not the case:

```
# id2 `A;;
- : [< `A | `B > `A ] = `A
```

id2 `A is given the type [< `A | `B > `A] which is parsed as [(< (`A|`B)) (> `A)] and means "at least `A" (i.e., [(> `A)]) and (without any practical justification) "at most `A|`B" (i.e., [(< (`A|`B))]). As a consequence `A and id2 `A are no longer considered statically equivalent:

```
# [(id2 `A); `C];;
Error: This expression has type [> `C ] but an
expression was expected of type [< `A | `B > `A ].
The second variant type does not allow tag(s) `C
```

Dealing with this problem requires the use of awkward explicit coercions that hinder any further use of subtype polymorphism.

EXAMPLE 2: ***roughly-typed pattern matching.*** The typing of pattern-matching expressions on polymorphic variants can prove to be imprecise. Consider:

```
# let f x = match id2 x with `A → `B | y → y;;
val f : [ `A | `B ] → [ `A | `B ] = <fun>
```

the typing of the function above is tainted by two approximations: (i) the domain of the function should be [< `A | `B], but—since the argument x is passed to the function id2—OCaml deduces the type [`A | `B] (a shorthand for [< `A | `B > `A | `B]), which is less precise: it loses subtype polymorphism; (ii) the return type states that f yields either `A or `B, while it is easy to see that only the latter is possible (when the argument is `A the function returns `B, and when the argument is `B the function returns the argument, that is, `B). So the type system deduces for f the type [`A | `B] → [`A | `B] instead of the more natural and precise [< `A | `B] → [> `B].

[2] Strictly speaking, `A is the only value *in all instances* of [> `A]: as shown in Section 3 the type [> `A] is actually a constrained type variable.

To recover the correct type, we need to state explicitly that the second pattern will only be used when y is `B, by using the alias pattern `B as y. This is a minor inconvenience here, but writing the type for y is not always possible and is often more cumbersome.

Likewise, OCaml unduly restricts the type of the function

```
# let g x = match x with `A → id2 x | _ → x;;
val g : ([< `A | `B > `A ] as α) → α = <fun>
```

as it states g can only be applied to `A or `B; actually, it can be applied safely to, say, `C or any variant value with any other tag. The system adds the upper bound `A | `B because id2 is applied to x. However, the application is evaluated only when x = `A: hence, this bound is unnecessary. The lower bound `A is unnecessary too.

The problem with these two functions is not specific to variant types. It is more general, and it stems from the lack of full-fledged connectives (union, intersection, and negation) in the types, a lack which neither allows the system to type a given pattern-matching branch by taking into account the cases the previous branches have already handled (e.g., the typing of the second branch in f), nor allows it to use the information provided by a pattern to refine the typing of the branch code (e.g., the typing of the first branch in g).

As a matter of fact, we can reproduce the same problem as for g, for instance, on lists:

```
# let rec map f l = match l with
  | [] → l
  | h::t → f h :: map f t;;
val map : (α → α) → α list → α list = <fun>
```

This is the usual map function, but it is given an overly restrictive type, accepting only functions with equal domain and codomain. The problem, again, is that the type system does not use the information provided by the pattern of the first branch to deduce that that branch always returns an empty list (rather than a generic α list). Also in this case alias patterns could be used to patch this specific example, but do not work in general.

EXAMPLE 3: ***rough approximations.*** During type reconstruction for pattern matching, OCaml uses the patterns themselves to determine the type of the matched expression. However, it might have to resort to approximations: there might be no type which corresponds precisely to the set of values matched by the patterns. Consider, for instance, the following function [from 18].

```
# let f x = match x with
    | (`A, _) → 1 | (`B, _) → 2
    | (_, `A) → 3 | (_, `B) → 4;;
val f : [> `A | `B ] * [> `A | `B ] → int
```

The type chosen by OCaml states that the function can be applied to any pair whose both components have a type greater than `A | `B. As a result, it can be applied to (`C, `C), whose components have type `A | `B | `C. This type therefore makes matching non-exhaustive: the domain also contains values that are not captured by any pattern (this is reported with a warning). Other choices could be made to ensure exhaustiveness, but they all pose different problems: choosing [< `A | `B ] * [< `A | `B ] → int makes the last two branches become redundant; choosing instead a type such as [> `A | `B ] * [< `A | `B] → int (or vice versa) is unintuitive as it breaks symmetry.

These rough approximations arise from the lack of full-fledged union types. Currently, OCaml only allows unions of variant types. If we could build a union of product types, then we could pick the type ([< `A | `B] * [>]) | ([>] * [< `A | `B]) (where [>] is "any variant"): exactly the set we need. More generally, true union types (and singleton types for constants) remove the need of any approximation for the set of values matched by the patterns

of a `match` expression, meaning we are never forced to choose—possibly inconsistently in different cases—between exhaustiveness and non-redundancy.

Although artificial, the three examples above provide a good overview of the kind of problems of the current formalization of polymorphic variants. Similar, but more "real life", examples of problems that our system solves can be found on the Web [e.g., 8–11, 21, 29].

Contributions. The main technical contribution of this work is the definition of a type system for a fragment of ML with polymorphic variants and pattern matching. Our system aims to replace the parts of the current type checker of OCaml that deal with these features. This replacement would result in a conservative extension of the current type checker (at least, for the parts that concern variants and pattern matching), since our system types (with the same or more specific types) all programs OCaml currently does; it would also be more expressive since it accepts more programs, while preserving type safety. The key of our solution is the addition of semantic subtyping—i.e., of unconstrained set-theoretic unions, intersections, and negations—to the type system. By adding it only in the type checker—thus, without touching the current syntax of types the OCaml programmer already knows—it is possible to solve all problems we illustrated in Examples 1 and 2. By a slight extension of the syntax of types—i.e., by permitting unions "|" not only of variants but of any two types—and no further modification we can also solve the problem described in Example 3. We also show that adding intersection and negation combinators, as well as singletons, to the syntax of types can be advantageous (cf. Sections 6.1 and 8). Therefore, the general contribution of our work is to show a way to add full-fledged union, intersection, and difference types to implicitly-typed languages that use HM type system.

Apart from the technical advantages and the gain in expressiveness, we think that the most important advantage of our system is that it is simpler, more natural, and arguably more intuitive than the current one (which uses a system of kinding constraints). Properties such as "*a given branch of a* `match` *expression will be executed for all values that can be produced by the matched expression, that can be captured by the pattern of the branch, and that cannot be captured by the patterns of the preceding branches*" can be expressed precisely and straightforwardly in terms of union, intersection, and negation types (i.e., the type of the matched expression, intersected by the type of the values matched by the pattern, minus the union of all the types of the values matched by any preceding pattern: see rule *Ts-Match* in Figure 2). The reason for this is that in our system we can express much more information at the level of types, which also means we can do without the system of kinding constraints. This is made possible by the presence of set-theoretic type connectives. Such a capability allows the type system to model pattern matching precisely and quite intuitively: we can describe exhaustiveness and non-redundancy checking in terms of subtype checking, whereas in OCaml they cannot be defined at the level of types. Likewise, unions and intersections allow us to encode bounded quantification—which is introduced in OCaml by structural polymorphism—without having to add it to the system. As a consequence, it is in general easy in our system to understand the origin of each constraint generated by the type checker.

Our work also presents several side contributions. First, it extends the type reconstruction of Castagna et al. [6] to pattern matching and let-polymorphism and, above all, proves it to be sound and complete with respect to our system (reconstruction in Castagna et al. [6] is only proven sound). Second, it provides a technique for a finer typing of pattern matching that applies to types other than polymorphic variants (e.g., the typing of `map` in Example 2) and languages other than OCaml (it is implemented in the development branch of CDuce [1, 7]). Third, the $\mathbb{K}$ system we define in Section 3 is a formalization of polymorphic variants and full-fledged pattern matching as they are currently implemented in OCaml: to our knowledge, no published formalization is as complete as $\mathbb{K}$.

Outline. Section 2 defines the syntax and semantics of the language we will study throughout this work. Sections 3 and 4 present two different type systems for this language.

In particular, Section 3 briefly describes the $\mathbb{K}$ type system we have developed as a formalization of how polymorphic variants are typed in OCaml. Section 4 describes the $\mathbb{S}$ type system, which employs set-theoretic types with semantic subtyping: we first give a deductive presentation of the system, and then we compare it to $\mathbb{K}$ to show that $\mathbb{S}$ can type every program that the $\mathbb{K}$ system can type. Section 5 defines a type reconstruction algorithm that is sound and complete with respect to the $\mathbb{S}$ type system.

Section 6 presents three extensions or modifications of the system: the first is the addition of overloaded functions; the second is a refinement of the typing of pattern matching, which we need to type precisely the functions `g` and `map` of Example 2; the third is a restriction which solves a discrepancy between our model and OCaml (the lack of type tagging at runtime in the OCaml implementation).

Finally, Section 7 compares our work with other formalizations of polymorphic variants and with previous work on systems with set-theoretic type connectives, and Section 8 concludes the presentation and points out some directions for future research.

For space reasons we omitted all the proofs as well as some definitions. They can be found in the Appendix.

2. The language of polymorphic variants

In this section, we define the syntax and semantics of the language with polymorphic variants and pattern matching that we study in this work. In the sections following this one we will define two different type systems for it (one with kinds in Section 3, the other with set-theoretic types in Section 4), as well as a type reconstruction algorithm (Section 5).

2.1 Syntax

We assume that there exist a countable set $\mathcal{X}$ of *expression variables*, ranged over by x, y, z, ..., a set $\mathcal{C}$ of constants, ranged over by c, and a set $\mathcal{L}$ of tags, ranged over by $\grave{}tag$. Tags are used to label variant expressions.

Definition 2.1 (Expressions)**.** *An* expression e *is a term inductively generated by the following grammar:*

$$e ::= x \mid c \mid \lambda x.\, e \mid e\, e \mid (e,e) \mid \grave{}tag(e) \mid \texttt{match}\ e\ \texttt{with}\ (p_i \rightarrow e_i)_{i\in I}$$

where p *ranges over the set* $\mathcal{P}$ *of patterns, defined below. We write* $\mathcal{E}$ *to denote the set of all expressions.*

We define $\mathsf{fv}(e)$ to be the set of expression variables occurring free in the expression e, and we say that e is *closed* if and only if $\mathsf{fv}(e)$ is empty. As customary, we consider expressions up to α-renaming of the variables bound by abstractions and by patterns.

The language is a λ-calculus with constants, pairs, variants, and pattern matching. Constants include a dummy constant `()` ('unit') to encode variants without arguments; multiple-argument variants are encoded with pairs. Matching expressions specify one or more branches (indexed by a set I) and can be used to encode `let`-expressions: $\texttt{let}\ x = e_0\ \texttt{in}\ e_1 \stackrel{\text{def}}{=} \texttt{match}\ e_0\ \texttt{with}\ x \rightarrow e_1$.

Definition 2.2 (Patterns)**.** *A* pattern p *is a term inductively generated by the following grammar:*

$$p ::= \_ \mid x \mid c \mid (p,p) \mid \grave{}tag(p) \mid p\&p \mid p|p$$

such that (i) *in a pair pattern* (p_1, p_2) *or an intersection pattern* $p_1\&p_2$, $\mathsf{capt}(p_1) \cap \mathsf{capt}(p_2) = \varnothing$; (ii) *in a union pattern* $p_1|p_2$,

$$
\begin{aligned}
v/\_ &= [\,] \\
v/x &= [v/x] \\
v/c &= \begin{cases} [\,] & \text{if } v = c \\ \Omega & \text{otherwise} \end{cases} \\
v/(p_1, p_2) &= \begin{cases} \varsigma_1 \cup \varsigma_2 & \text{if } v = (v_1, v_2) \text{ and } \forall i.\ v_i/p_i = \varsigma_i \\ \Omega & \text{otherwise} \end{cases} \\
v/\grave{}tag(p_1) &= \begin{cases} \varsigma_1 & \text{if } v = \grave{}tag(v_1) \text{ and } v_1/p_1 = \varsigma_1 \\ \Omega & \text{otherwise} \end{cases} \\
v/p_1 \& p_2 &= \begin{cases} \varsigma_1 \cup \varsigma_2 & \text{if } \forall i.\ v/p_i = \varsigma_i \\ \Omega & \text{otherwise} \end{cases} \\
v/p_1 | p_2 &= \begin{cases} v/p_1 & \text{if } v/p_1 \neq \Omega \\ v/p_2 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 1. Semantics of pattern matching.

$\mathsf{capt}(p_1) = \mathsf{capt}(p_2)$*, where* $\mathsf{capt}(p)$ *denotes the set of expression variables occurring as sub-terms in a pattern* p *(called the* capture variables *of* p*). We write* $\mathcal{P}$ *to denote the set of all patterns.*

Patterns have the usual semantics. A wildcard "_" accepts any value and generates no bindings; a variable pattern accepts any value and binds the value to the variable. Constants only accept themselves and do not bind. Pair patterns accept pairs if each sub-pattern accepts the corresponding component, and variant patterns accept variants with the same tag if the argument matches the inner pattern (in both cases, the bindings are those of the sub-patterns). Intersection patterns require the value to match both sub-patterns (they are a generalization of the alias patterns p `as` x of OCaml), while union patterns require it to match either of the two (the left pattern is tested first).

2.2 Semantics

We now define a small-step operational semantics for this calculus. First, we define the values of the language.

Definition 2.3 (Values)**.** *A* value v *is a closed expression inductively generated by the following grammar.*

$$v ::= c \mid \lambda x.\, e \mid (v, v) \mid \grave{}tag(v)$$

We now formalize the intuitive semantics of patterns that we have presented above.

Bindings are expressed in terms of *expression substitutions*, ranged over by ς: we write $[v_1/x_1, \ldots, v_n/x_n]$ for the substitution that replaces free occurrences of x_i with v_i, for each i. We write $e\varsigma$ for the application of the substitution ς to an expression e; we write $\varsigma_1 \cup \varsigma_2$ for the union of disjoint substitutions.

The semantics of pattern matching we have described is formalized by the definition of v/p given in Figure 1. In a nutshell, v/p is the result of matching a value v against a pattern p. We have either $v/p = \varsigma$, where ς is a substitution defined on the variables in $\mathsf{capt}(p)$, or $v/p = \Omega$. In the former case, we say that v matches p (or that p accepts v); in the latter, we say that matching fails.

Note that the unions of substitutions in the definition are always disjoint because of our linearity condition on pair and intersection patterns. The condition that sub-patterns of a union pattern $p_1|p_2$ must have the same capture variables ensures that v/p_1 and v/p_2 will be defined on the same variables.

Finally, we describe the reduction relation. It is defined by the following two notions of reduction

$$
\begin{array}{rcll}
(\lambda x.\, e)\, v & \rightsquigarrow & e[v/x] & \\
\texttt{match}\ v\ \texttt{with}\ (p_i \to e_i)_{i \in I} & \rightsquigarrow & e_j\varsigma & \begin{array}{l}\text{if } v/p_j = \varsigma \text{ and} \\ \forall i < j.\ v/p_i = \Omega\end{array}
\end{array}
$$

applied with a leftmost-outermost strategy which does not reduce inside λ-abstractions nor in the branches of `match` expressions.

The first reduction rule is the ordinary rule for call-by-value β-reduction. It states that the application of an abstraction $\lambda x.\, e$ to a value v reduces to the body e of the abstraction, where x is replaced by v. The second rule states that a `match` expression on a value v reduces to the branch e_j corresponding to the first pattern p_j for which matching is successful. The obtained substitution is applied to e_j, replacing the capture variables of p_j with sub-terms of v. If no pattern accepts v, the expression is stuck.

3. Typing variants with kinding constraints

In this section, we formalize $\mathbb{K}$, the type system with $\Bbbk$inding constraints for polymorphic variants as featured in OCaml; we will use it to gauge the merits of $\mathbb{S}$, our type system with $\mathbb{s}$et-theoretic types. This formalization is derived from, and extends, the published systems based on structural polymorphism [17, 19]. In our ken, no formalization in the literature includes polymorphic variants, let-polymorphism, and full-fledged pattern matching (see Section 7), which is why we give here a new one. While based on existing work, the formalization is far from being trivial (which with hindsight explains its absence), and thus we needed to prove all its properties from scratch. For space reasons we outline just the features that distinguish our formalization, namely variants, pattern matching, and type generalization for pattern capture variables. The Appendix presents the full definitions and proofs of all properties.

The system consists essentially of the core ML type system with the addition of a kinding system to distinguish normal type variables from *constrained* ones. Unlike normal variables, constrained ones cannot be instantiated into any type, but only into other constrained variables with compatible constraints. They are used to type variant expressions: there are no 'variant types' *per se*. Constraints are recorded in *kinds* and kinds in a *kinding environment* (i.e., a mapping from type variables to kinds) which is included in typing judgments. An important consequence of using kinding constraints is that they implicitly introduce (a limited form of) recursive types, since a constrained type variable may occur in its constraints.

We assume that there exists a countable set $\mathcal{V}$ of *type variables*, ranged over by α, β, γ, ... We also consider a finite set $\mathcal{B}$ of *basic types*, ranged over by b, and a function $b_{(\cdot)}$ from constants to basic types. For instance, we might take $\mathcal{B} = \{\texttt{bool}, \texttt{int}, \texttt{unit}\}$, with $b_{\texttt{true}} = \texttt{bool}$, $b_{\texttt{()}} = \texttt{unit}$, and so on.

Definition 3.1 (Types)**.** *A* type τ *is a term inductively generated by the following grammar.*

$$\tau ::= \alpha \mid b \mid \tau \to \tau \mid \tau \times \tau$$

The system only uses the types of core ML: all additional information is encoded in the kinds of type variables.

Kinds have two forms: the *unconstrained kind* "$\bullet$" classifies "normal" variables, while variables used to type variants are given a *constrained kind*. Constrained kinds are triples describing which tags may or may not appear (a *presence* information) and which argument types are associated to each tag (a *typing* information). The presence information is split in two parts, a lower and an upper bound. This is necessary to provide an equivalent to both covariant and contravariant subtyping—without actually having subtyping

in the system—that is, to allow both variant values and functions defined on variant values to be polymorphic.

Definition 3.2 (Kinds)**.** *A* kind κ *is either the* unconstrained kind *"$\bullet$" or a* constrained kind*, that is, a triple* (L, U, T) *where:*
- L *is a finite set of tags* $\{\text{`}tag_1, \ldots, \text{`}tag_n\}$*;*
- U *is either a finite set of tags or the set* $\mathcal{L}$ *of all tags;*
- T *is a finite set of pairs of a tag and a type, written* $\{\text{`}tag_1 \colon \tau_1, \ldots, \text{`}tag_n \colon \tau_n\}$ *(its domain* $\mathsf{dom}(T)$ *is the set of tags occurring in it);*

and where the following conditions hold:
- $L \subseteq U$*,* $L \subseteq \mathsf{dom}(T)$*, and, if* $U \neq \mathcal{L}$*,* $U \subseteq \mathsf{dom}(T)$*;*
- *tags in* L *have a single type in* T*, that is, if* $\text{`}tag \in L$*, whenever both* $\text{`}tag \colon \tau_1 \in T$ *and* $\text{`}tag \colon \tau_2 \in T$*, we have* $\tau_1 = \tau_2$*.*

In OCaml, kinds are written with the typing information inlined in the lower and upper bounds. These are introduced by `>` and `<` respectively and, if missing, $\varnothing$ is assumed for the lower bound and $\mathcal{L}$ for the upper. For instance, `[> `A of int | `B of bool ] as` α of OCaml is represented here by assigning to the variable α the kind $(\{\text{`A}, \text{`B}\}, \mathcal{L}, \{\text{`A: int}, \text{`B: bool}\})$; `[< `A of int | `B of bool ] as` β corresponds to β of kind $(\varnothing, \{\text{`A}, \text{`B}\}, \{\text{`A: int}, \text{`B: bool}\})$; finally `[< `A of int | `B of bool & unit > `A] as` $\gamma$ corresponds to $\gamma$ of kind $(\{\text{`A}\}, \{\text{`A}, \text{`B}\}, \{\text{`A: int}, \text{`B: bool}, \text{`B: unit}\})$.

Definition 3.3 (Type schemes)**.** *A* type scheme σ *is of the form* $\forall A.\, K \triangleright \tau$*, where:*
- A *is a finite set* $\{\alpha_1, \ldots, \alpha_n\}$ *of type variables;*
- K *is a kinding environment, that is, a map from type variables to kinds;*
- $\mathsf{dom}(K) = A$*.*

We identify a type scheme $\forall\varnothing.\, \varnothing \triangleright \tau$, which quantifies no variable, with the type τ itself. We consider type schemes up to renaming of the variables they bind and disregard useless quantification (i.e., quantification of variables that do not occur in the type).

Type schemes single out, by quantifying them, the variables of a type which can be instantiated. In ML without kinds, the quantified variables of a scheme can be instantiated with any type. The addition of kinds changes this: variables with constrained kinds may only be instantiated into other variables with *equally strong* or *stronger* constraints. This relation on constraints is formalized by the following entailment relation:

$$(L, U, T) \vDash (L', U', T') \iff L \supseteq L' \wedge U \subseteq U' \wedge T \supseteq T'\,,$$

where $\kappa_1 \vDash \kappa_2$ means that κ_1 is a constraint stronger than κ_2. This relation is used to select the type substitutions (ranged over by θ) that are *admissible*, that is, that are sound with respect to kinding.

Definition 3.4 (Admissibility of a type substitution)**.** *A type substitution* θ *is* admissible *between two kinding environments* K *and* K'*, written* $K \vdash \theta \colon K'$*, if and only if, for every type variable* α *such that* $K(\alpha) = (L, U, T)$*,* $\alpha\theta$ *is a type variable such that* $K'(\alpha\theta) = (L', U', T')$ *and* $(L', U', T') \vDash (L, U, T\theta)$*.*

In words, whenever α is constrained in K, then $\alpha\theta$ must be a type variable constrained in K' by a kind that entails the substitution instance of the kind of α in K.

The set of the instances of a type scheme are now obtained by applying only admissible substitutions.

Definition 3.5 (Instances of a type scheme)**.** *The set of* instances *of a type scheme* $\forall A.\, K' \triangleright \tau$ *in a kinding environment* K *is*

$$\mathsf{inst}_K(\forall A.\, K' \triangleright \tau) = \{\, \tau\theta \mid \mathsf{dom}(\theta) \subseteq A \;\wedge\; K, K' \vdash \theta \colon K \,\}\,.$$

As customary, this set is used in the type system rule to type expression variables:

$$\textit{Tk-Var}\ \frac{\tau \in \mathsf{inst}_K(\Gamma(x))}{K; \Gamma \vdash_{\mathbb{K}} x \colon \tau}$$

Notice that typing judgments are of the form $K; \Gamma \vdash_{\mathbb{K}} e \colon \tau$: the premises include a type environment Γ but also, which is new, a kinding environment K (the $\mathbb{K}$ subscript in the turnstile symbol is to distinguish this relation from $\vdash_{\mathbb{S}}$, the relation for the set-theoretic type system of the next section).

The typing rules for constants, abstractions, applications, and pairs are straightforward. There remain the rules for variants and for pattern matching, which are the only interesting ones.

$$\textit{Tk-Tag}\ \frac{K; \Gamma \vdash_{\mathbb{K}} e \colon \tau \qquad K(\alpha) \vDash (\{\text{`}tag\}, \mathcal{L}, \{\text{`}tag \colon \tau\})}{K; \Gamma \vdash_{\mathbb{K}} \text{`}tag(e) \colon \alpha}$$

The typing of variant expressions uses the kinding environment. Rule *Tk-Tag* states that $\text{`}tag(e)$ can be typed by any variable $\alpha$ such that $\alpha$ has a constrained kind in $K$ which entails the "minimal" kind for this expression. Specifically, if $K(\alpha) = (L, U, T)$, then we require $\text{`}tag \in L$ and $\text{`}tag \colon \tau \in T$, where $\tau$ is a type for $e$. Note that $T$ may not assign more than one type to $\text{`}tag$, since $\text{`}tag \in L$.

The typing of pattern matching is by far the most complex part of the type system and it is original to our system.

$$\textit{Tk-Match}\ \frac{\begin{array}{c} K; \Gamma \vdash_{\mathbb{K}} e_0 \colon \tau_0 \qquad \tau_0 \preccurlyeq_K \{\, p_i \mid i \in I \,\} \\ \forall i \in I \quad K \vdash p_i \colon \tau_0 \Rightarrow \Gamma_i \quad K; \Gamma, \mathsf{gen}_{K;\Gamma}(\Gamma_i) \vdash_{\mathbb{K}} e_i \colon \tau \end{array}}{K; \Gamma \vdash_{\mathbb{K}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I} \colon \tau}$$

Let us describe each step that the rule above implies. First the rule deduces the type τ_0 of the matched expression ($K; \Gamma \vdash_{\mathbb{K}} e_0 \colon \tau_0$). Second, for each pattern p_i, it generates the type environment Γ_i which assigns types to the capture variables of p_i, assuming p_i is matched against a value known to be of type τ_0. This is done by deducing the judgment $K \vdash p_i \colon \tau_0 \Rightarrow \Gamma_i$, whose inference system is mostly straightforward (see Figure 8 in the Appendix); for instance, for variable patterns we have:

$$\textit{TPk-Var}\ \frac{}{K \vdash x \colon \tau \Rightarrow \{x \colon \tau\}}$$

The only subtle point of this inference system is the rule for patterns of the form $\text{`}tag(p)$

$$\textit{TPk-Tag}\ \frac{\begin{array}{c} K \vdash p \colon \tau \Rightarrow \Gamma \qquad K(\alpha) = (L, U, T) \\ (\text{`}tag \in U \text{ implies } \text{`}tag \colon \tau \in T) \end{array}}{K \vdash \text{`}tag(p) \colon \alpha \Rightarrow \Gamma}$$

which—after generating the environment for the capture variables of p—checks whether the type of the matched expression is a variant type (i.e., a variable) with the right constraints for $\text{`}tag$.

Third, the rule *Tk-Match* types each branch e_i with type τ, in a type environment updated with $\mathsf{gen}_{K;\Gamma}(\Gamma_i)$, that is, with the generalization of the Γ_i generated by $K \vdash p_i \colon \tau_0 \Rightarrow \Gamma_i$. The definition of generalization is standard: it corresponds to quantifying all the variables that do not occur free in the environment Γ. The subtle point is the definition of the free variables of a type (and hence of an environment), which we omit for space reasons. It must navigate the kinding environment K to collect all variables which can be reached by following the constraints; hence, the gen function takes as argument K as well as Γ.

Finally, the premises of the rule also include the exhaustiveness condition $\tau_0 \preccurlyeq_K \{\, p_i \mid i \in I \,\}$, which checks whether every possible value that e_0 can produce matches at least one pattern p_i. The definition of exhaustiveness is quite convoluted.

Definition 3.6 (Exhaustiveness)**.** *We say that a set of patterns* P *is* exhaustive *with respect to a type* τ *in a kinding environment* K*,*

and we write $\tau \preccurlyeq_K P$, when, for every K', θ, and v,

$$(K \vdash \theta \colon K' \wedge K'; \varnothing \vdash_{\mathbb{K}} v \colon \tau\theta) \implies \exists p \in P, \varsigma.\ v/p = \varsigma\ .$$

In words, P is exhaustive when every value that can be typed with any admissible substitution of τ is accepted by at least one pattern in P. OCaml does not impose exhaustiveness—it just signals non-exhaustiveness with a warning—but our system does. We do so in order to have a simpler statement for soundness and to facilitate the comparison with the system of the next section. We do not discuss how exhaustiveness can be effectively computed; for more information on how OCaml checks it, see Garrigue [18] and Maranget [20].

We conclude this section by stating the type soundness property of the $\mathbb{K}$ type system.

Theorem 3.1 (Progress)**.** *Let e be a well-typed, closed expression. Then, either e is a value or there exists an expression e' such that $e \rightsquigarrow e'$.*

Theorem 3.2 (Subject reduction)**.** *Let e be an expression and τ a type such that $K; \Gamma \vdash_{\mathbb{K}} e \colon \tau$. If $e \rightsquigarrow e'$, then $K; \Gamma \vdash_{\mathbb{K}} e' \colon \tau$.*

Corollary 3.3 (Type soundness)**.** *Let e be a well-typed, closed expression, that is, such that $K; \varnothing \vdash_{\mathbb{K}} e \colon \tau$ holds for some τ. Then, either e diverges or it reduces to a value v such that $K; \varnothing \vdash_{\mathbb{K}} v \colon \tau$.*

4. Typing variants with set-theoretic types

We now describe $\mathbb{S}$, a type system for the language of Section 2 based on set-theoretic types. The approach we take in its design is drastically different from that followed for $\mathbb{K}$. Rather than adding a kinding system to record information that types cannot express, we directly enrich the syntax of types so they can express all the notions we need. Moreover, we add subtyping—using a semantic definition—rather than encoding it via instantiation. We exploit type connectives and subtyping to represent variant types as unions and to encode bounded quantification by union and intersection.

We argue that $\mathbb{S}$ has several advantages with respect to the previous system. It is more expressive: it is able to type some programs that $\mathbb{K}$ rejects though they are actually type safe, and it can derive more precise types than $\mathbb{K}$. It is arguably a simpler formalization: typing works much like in ML except for the addition of subtyping, we have explicit types for variants, and we can type pattern matching precisely and straightforwardly. Indeed, as regards pattern matching, an advantage of the $\mathbb{S}$ system is that it can express exhaustiveness and non-redundancy checking as subtyping checks, while they cannot be expressed at the level of types in $\mathbb{K}$.

Naturally, subtyping brings its own complications. We do not discuss its definition here, since we reuse the relation defined by Castagna and Xu [4]. The use of semantic subtyping makes the definition of a typing algorithm challenging: Castagna et al. [5, 6] show how to define one in an explicitly-typed setting. Conversely, we study here an implicitly-typed language and hence study the problem of type reconstruction (in the next section).

While this system is based on that described by Castagna et al. [5, 6], there are significant differences which we discuss in Section 7. Notably, intersection types play a more limited role in our system (no rule allows the derivation of an intersection of arrow types for a function), making our type reconstruction complete.

4.1 Types and subtyping

As before, we consider a set $\mathcal{V}$ of *type variables* (ranged over by α, β, γ, ...) and the sets $\mathcal{C}$, $\mathcal{L}$, and $\mathcal{B}$ of *language constants*, *tags*, and *basic types* (ranged over by c, $\grave{}tag$, and b respectively).

Definition 4.1 (Types)**.** *A* type t *is a term coinductively produced by the following grammar:*

$$t ::= \alpha \mid b \mid c \mid t \to t \mid t \times t \mid \grave{}tag(t) \mid t \vee t \mid \neg t \mid \mathbb{0}$$

which satisfies two additional constraints:

- *(*regularity*) the term must have a finite number of different subterms;*
- *(*contractivity*) every infinite branch must contain an infinite number of occurrences of atoms (i.e., a type variable or the immediate application of a type constructor: basic, constant, arrow, product, or variant).*

We introduce the following abbreviations:

$$t_1 \wedge t_2 \stackrel{\text{def}}{=} \neg(\neg t_1 \vee \neg t_2) \qquad t_1 \setminus t_2 \stackrel{\text{def}}{=} t_1 \wedge (\neg t_2) \qquad \mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}\ .$$

With respect to the types in Definition 3.1, we add several new forms. We introduce set-theoretic connectives (union, intersection, and negation), as well as bottom (the empty type $\mathbb{0}$) and top ($\mathbb{1}$) types. We add general (uniform) recursive types by interpreting the grammar *coinductively*, while $\mathbb{K}$ introduces recursion via kinds. Contractivity is imposed to bar out ill-formed types such as those fulfilling the equation $t = t \vee t$ (which does not give any information on the set of values it represents) or $t = \neg t$ (which cannot represent any set of values).

We introduce explicit types for variants. These types have the form $\grave{}tag(t)$: the type of variant expressions with tag $\grave{}tag$ and an argument of type t.[3] Type connectives allow us to represent all variant types of $\mathbb{K}$ by combining types of this form, as we describe in detail below. Finally, we add singleton types for constants (e.g., a type `true` which is a subtype of `bool`), which we use to type pattern matching precisely.

Variant types and bounded quantification. $\mathbb{K}$ uses constrained variables to type variants; when these variables are quantified in a type scheme, their kind constrains the possible instantiations of the scheme. This is essentially a form of bounded quantification: a variable of kind (L, U, T) may only be instantiated by other variables which fall within the bounds—the lower bound being determined by L and T, the upper one by U and T.

In $\mathbb{S}$, we can represent these bounds as unions of variant types $\grave{}tag(t)$. For instance, consider in $\mathbb{K}$ a constrained variable α of kind $(\{\grave{}\mathtt{A}\}, \{\grave{}\mathtt{A}, \grave{}\mathtt{B}\}, \{\grave{}\mathtt{A}\colon \mathtt{bool}, \grave{}\mathtt{B}\colon \mathtt{int}\})$. If we quantify α, we can then instantiate it with variables whose kinds entail that of α. Using our variant types and unions, we write the lower bound as $t_{\mathrm{L}} = \grave{}\mathtt{A}(\mathtt{bool})$ and the upper one as $t_{\mathrm{U}} = \grave{}\mathtt{A}(\mathtt{bool}) \vee \grave{}\mathtt{B}(\mathtt{int})$. In our system, α should be a variable with bounded quantification, which can only be instantiated by types t such that $t_{\mathrm{L}} \leq t \leq t_{\mathrm{U}}$.

However, we do not need to introduce bounded quantification as a feature of our language: we can use type connectives as proposed in [4] (cf. Footnote 4 therein) to encode it. The possible instantiations of α (with the bounds above) and the possible instantiations of $(t_{\mathrm{L}} \vee \beta) \wedge t_{\mathrm{U}}$, with no bound on β, are equivalent. We use the latter form: we internalize the bounds in the type itself by union and intersection. In this way, we need no system of constraints extraneous to types.

Subtyping. There exists a *subtyping* relation between types. We write $t_1 \leq t_2$ when t_1 is a subtype of t_2; we write $t_1 \simeq t_2$ when t_1 and t_2 are equivalent with respect to subtyping, that is, when $t_1 \leq t_2$ and $t_2 \leq t_1$. The definition and properties of this relation are studied in Castagna and Xu [4], except for variant types which, for this purpose, we encode as pairs (cf. Footnote 3).

[3] We could encode $\grave{}tag(t)$ by the product $\grave{}tag \times t$. Although we have preferred to add explicit variant types, we still use this encoding to derive their subtyping properties: see Petrucciani [23] for a detailed explanation.

$$\textit{Ts-Var}\ \frac{t \in \mathsf{inst}(\Gamma(x))}{\Gamma \vdash_{\mathbb{S}} x\colon t} \qquad \textit{Ts-Const}\ \frac{}{\Gamma \vdash_{\mathbb{S}} c\colon c} \qquad \textit{Ts-Abstr}\ \frac{\Gamma, \{x\colon t_1\} \vdash_{\mathbb{S}} e\colon t_2}{\Gamma \vdash_{\mathbb{S}} \lambda x.\, e\colon t_1 \to t_2} \qquad \textit{Ts-Appl}\ \frac{\Gamma \vdash_{\mathbb{S}} e_1\colon t' \to t \qquad \Gamma \vdash_{\mathbb{S}} e_2\colon t'}{\Gamma \vdash_{\mathbb{S}} e_1\, e_2\colon t}$$

$$\textit{Ts-Pair}\ \frac{\Gamma \vdash_{\mathbb{S}} e_1\colon t_1 \qquad \Gamma \vdash_{\mathbb{S}} e_2\colon t_2}{\Gamma \vdash_{\mathbb{S}} (e_1, e_2)\colon t_1 \times t_2} \qquad \textit{Ts-Tag}\ \frac{\Gamma \vdash_{\mathbb{S}} e\colon t}{\Gamma \vdash_{\mathbb{S}} \text{`}\mathit{tag}(e)\colon \text{`}\mathit{tag}(t)}$$

$$\textit{Ts-Match}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathbb{S}} e_0\colon t_0 \qquad t_0 \le \bigvee_{i\in I} \lbag p_i \rbag \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lbag p_j \rbag) \wedge \lbag p_i \rbag \\ \forall i \in I \qquad \Gamma, \mathsf{gen}_\Gamma(t_i // p_i) \vdash_{\mathbb{S}} e_i\colon t'_i\end{array}}{\Gamma \vdash_{\mathbb{S}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i\in I}\colon \bigvee_{i\in I} t'_i} \qquad \textit{Ts-Subsum}\ \frac{\Gamma \vdash_{\mathbb{S}} e\colon t' \qquad t' \le t}{\Gamma \vdash_{\mathbb{S}} e\colon t}$$

Figure 2. Typing relation of the $\mathbb{S}$ type system.

In brief, subtyping is given a semantic definition, in the sense that $t_1 \le t_2$ holds if and only if $[\![t_1]\!] \subseteq [\![t_2]\!]$, where $[\![\cdot]\!]$ is an interpretation function mapping types to sets of elements from some domain (intuitively, the set of values of the language). The interpretation is "set-theoretic" as it interprets union types as unions, negation as complementation, and products as Cartesian products.

In general, in the semantic-subtyping approach, we consider a type to denote the set of all values that have that type (we will say that some type "is" the set of values of that type). In particular, for arrow types, the type $t_1 \to t_2$ is that of function values (i.e., λ-abstractions) which, if they are given an argument in $[\![t_1]\!]$ and they do not diverge, yield a result in $[\![t_2]\!]$. Hence, all types of the form $\mathbb{0} \to t$, for any t, are equivalent (as only diverging expressions can have type $\mathbb{0}$); any of them is the type of all functions. Conversely, $\mathbb{1} \to \mathbb{0}$ is the type of functions that (provably) diverge on all inputs: a function of this type should yield a value in the empty type whenever it terminates, and that is impossible.

The presence of variables complicates the definition of semantic subtyping. Here, we just recall from Castagna and Xu [4] that subtyping is preserved by type substitutions: $t_1 \le t_2$ implies $t_1\theta \le t_2\theta$ for every type substitution θ.

4.2 Type system

We present $\mathbb{S}$ focusing on the differences with respect to the system of OCaml (i.e., $\mathbb{K}$); full definitions are in the Appendix. Unlike in $\mathbb{K}$, type schemes here are defined just as in ML as we no longer need kinding constraints.

Definition 4.2 (Type schemes)**.** *A* type scheme s *is of the form* $\forall A.\, t$*, where* A *is a finite set* $\{\alpha_1, \ldots, \alpha_n\}$ *of type variables.*

As in $\mathbb{K}$, we identify a type scheme $\forall\varnothing.\, t$ with the type t itself, we consider type schemes up to renaming of the variables they bind, and we disregard useless quantification.

We write $\mathsf{var}(t)$ for the set of type variables occurring in a type t; we say they are the *free variables* of t, and we say that t is *ground* or *closed* if and only if $\mathsf{var}(t)$ is empty. The (coinductive) definition of var can be found in Castagna et al. [5, Definition A.2].

Unlike in ML, types in our system can contain variables which are irrelevant to the meaning of the type. For instance, $\alpha \times \mathbb{0}$ is equivalent to $\mathbb{0}$ (with respect to subtyping), as we interpret product types into Cartesian products. Thus, α is irrelevant in $\alpha \times \mathbb{0}$. To capture this concept, we introduce the notion of *meaningful variables* in a type t. We define these to be the set

$$\mathsf{mvar}(t) = \{\, \alpha \in \mathsf{var}(t) \mid t[\mathbb{0}/\alpha] \not\simeq t \,\}\,,$$

where the choice of $\mathbb{0}$ to replace α is arbitrary (any other closed type yields the same definition). Equivalent types have exactly the same meaningful variables. To define generalization, we allow quantifying variables which are free in the type environment but are meaningless in it (intuitively, we act as if types were in a canonical form without irrelevant variables).

We extend var to type schemes as $\mathsf{var}(\forall A.\, t) = \mathsf{var}(t) \setminus A$, and do likewise for mvar.

Type substitutions are defined in a standard way by coinduction; there being no kinding system, we do not need the admissibility condition of $\mathbb{K}$.

We define type environments Γ as usual. The operations of generalization of types and instantiation of type schemes, instead, must account for the presence of irrelevant variables and of subtyping.

Generalization with respect to Γ quantifies all variables in a type except for those that are free *and meaningful* in Γ:

$$\mathsf{gen}_\Gamma(t) = \forall A.\, t\,, \quad \text{where } A = \mathsf{var}(t) \setminus \mathsf{mvar}(\Gamma)\,.$$

We extend gen pointwise to sets of bindings $\{x_1\colon t_1, \ldots, x_n\colon t_n\}$.

The set of instances of a type scheme is given by

$$\mathsf{inst}(\forall A.\, t) = \{\, t\theta \mid \mathsf{dom}(\theta) \subseteq A \,\}\,,$$

and we say that a type scheme s_1 is *more general* than a type scheme s_2—written $s_1 \sqsubseteq s_2$—if

$$\forall t_2 \in \mathsf{inst}(s_2).\ \exists t_1 \in \mathsf{inst}(s_1).\ t_1 \le t_2\,. \tag{1}$$

Notice that the use of subtyping in the definition above generalizes the corresponding definition of ML (which uses equality) and subsumes the notion of "admissibility" of $\mathbb{K}$ by a far simpler and more natural relation (cf. Definitions 3.4 and 3.5).

Figure 2 defines the typing relation $\Gamma \vdash_{\mathbb{S}} e\colon t$ of the $\mathbb{S}$ type system (we use the $\mathbb{S}$ subscript in the turnstile symbol to distinguish this relation from that for $\mathbb{K}$). All rules except that for pattern matching are straightforward. Note that *Ts-Const* is more precise than in $\mathbb{K}$ since we have singleton types, and that *Ts-Tag* uses the types we have introduced for variants.

The rule *Ts-Match* involves two new concepts that we present below. We start by typing the expression to be matched, e_0, with some type t_0. We also require every branch e_i to be well-typed with some type t'_i: the type of the whole `match` expression is the union of all t'_i. We type each branch in an environment expanded with types for the capture variables of p_i: this environment is generated by the function $t_i // p_i$ (described below) and is generalized.

The advantage of our richer types here is that, given any pattern, the set of values it accepts is always described precisely by a type.

Definition 4.3 (Accepted type)**.** *The* accepted type $\lbag p \rbag$ *of a pattern* p *is defined inductively as:*

$$\begin{array}{ll} \lbag \_ \rbag = \lbag x \rbag = \mathbb{1} & \lbag c \rbag = c \\ \lbag (p_1, p_2) \rbag = \lbag p_1 \rbag \times \lbag p_2 \rbag & \lbag \text{`}\mathit{tag}(p) \rbag = \text{`}\mathit{tag}(\lbag p \rbag) \\ \lbag p_1 \& p_2 \rbag = \lbag p_1 \rbag \wedge \lbag p_2 \rbag & \lbag p_1 | p_2 \rbag = \lbag p_1 \rbag \vee \lbag p_2 \rbag\,. \end{array}$$

For well-typed values v, we have $v/p \neq \Omega \iff \varnothing \vdash_{\mathbb{S}} v\colon \lbag p \rbag$. We use accepted types to express the condition of *exhaustiveness*:

$$
\begin{aligned}
[\![\alpha]\!]_K &= \begin{cases} \alpha & \text{if } K(\alpha) = \bullet \\ (\mathsf{low}_K(L,T) \vee \alpha) \wedge \mathsf{upp}_K(U,T) & \text{if } K(\alpha) = (L,U,T) \end{cases} \\
[\![b]\!]_K &= b \\
[\![\tau_1 \to \tau_2]\!]_K &= [\![\tau_1]\!]_K \to [\![\tau_2]\!]_K \\
[\![\tau_1 \times \tau_2]\!]_K &= [\![\tau_1]\!]_K \times [\![\tau_2]\!]_K
\end{aligned}
$$

where:
$$
\mathsf{low}_K(L,T) = \bigvee_{\text{`}tag \in L} \text{`}tag\Big(\bigwedge_{\text{`}tag:\ \tau \in T} [\![\tau]\!]_K\Big)
$$
$$
\mathsf{upp}_K(U,T) = \begin{cases} \bigvee_{\text{`}tag \in U} \text{`}tag\Big(\bigwedge_{\text{`}tag:\ \tau \in T} [\![\tau]\!]_K\Big) & \text{if } U \neq \mathcal{L} \\ \bigvee_{\text{`}tag \in \mathsf{dom}(T)} \text{`}tag\Big(\bigwedge_{\text{`}tag:\ \tau \in T} [\![\tau]\!]_K\Big) \ \vee\ \big(\mathbb{1}_{\mathrm{v}} \setminus \bigvee_{\text{`}tag \in \mathsf{dom}(T)} \text{`}tag(\mathbb{1})\big) & \text{if } U = \mathcal{L} \end{cases}
$$

Figure 3. Translation of $\Bbbk$-types to $\mathbb{s}$-types.

$t_0 \leq \bigvee_{i \in I} \lbag p_i \rbag$ ensures that every value e_0 can reduce to (i.e., every value in t_0) will match at least one pattern (i.e., is in the accepted type of some pattern). We also use them to compute precisely the subtypes of t_0 corresponding to the values which will trigger each branch. In the rule, t_i is the type of all values which will be selected by the i-th branch: those in t_0 (i.e., generated by e_0), not in any $\lbag p_j \rbag$ for $j < i$ (i.e., not captured by any previous pattern), and in $\lbag p_i \rbag$ (i.e., accepted by p_i). These types t_i allow us to express *non-redundancy* checks: if $t_i \leq \mathbb{0}$ for some i, then the corresponding pattern will never be selected (which likely means the programmer has made some mistake and should receive a warning).[4]

The last element we must describe is the generation of types for the capture variables of each pattern by the $t_i // p_i$ function. Here, our use of t_i means we exploit the shape of the pattern p_i and of the previous ones to generate more precise types; environment generation in $\mathbb{K}$ essentially uses only t_0 and is therefore less precise.

Environment generation relies on two functions π_1 and π_2 which extract the first and second component of a type $t \leq \mathbb{1} \times \mathbb{1}$. For instance, if $t = (\alpha \times \beta) \vee (\mathtt{bool} \times \mathtt{int})$, we have $\pi_1(t) = \alpha \vee \mathtt{bool}$ and $\pi_2(t) = \beta \vee \mathtt{int}$. Given any tag $\text{`}tag$, $\pi_{\text{`}tag}$ does likewise for variant types with that tag. See Castagna et al. [5, Appendix C.2.1] and Petrucciani [23] for the full details.

Definition 4.4 (Pattern environment generation)**.** *Given a pattern p and a type $t \leq \lbag p \rbag$, the type environment $t // p$ generated by pattern matching is defined inductively as:*

$$
\begin{aligned}
t//\_ &= \varnothing & t//(p_1, p_2) &= \pi_1(t)//p_1 \ \cup\ \pi_2(t)//p_2 \\
t//x &= \{x \colon t\} & t//\text{`}tag(p) &= \pi_{\text{`}tag}(t)//p \\
t//c &= \varnothing & t//p_1 \& p_2 &= t//p_1 \ \cup\ t//p_2 \\
t//p_1|p_2 &= (t \wedge \lbag p_1 \rbag)//p_1 \ \between\ (t \setminus \lbag p_1 \rbag)//p_2
\end{aligned}
$$

where $(\Gamma \between \Gamma')(x) = \Gamma(x) \vee \Gamma'(x)$.

The $\mathbb{S}$ type system is sound, as stated by the following properties.

Theorem 4.1 (Progress)**.** *Let e be a well-typed, closed expression (i.e., $\varnothing \vdash_{\mathbb{S}} e \colon t$ holds for some t). Then, either e is a value or there exists an expression e' such that $e \rightsquigarrow e'$.*

Theorem 4.2 (Subject reduction)**.** *Let e be an expression and t a type such that $\Gamma \vdash_{\mathbb{S}} e \colon t$. If $e \rightsquigarrow e'$, then $\Gamma \vdash_{\mathbb{S}} e' \colon t$.*

Corollary 4.3 (Type soundness)**.** *Let e be a well-typed, closed expression, that is, such that $\varnothing \vdash_{\mathbb{S}} e \colon t$ holds for some t. Then, either e diverges or it reduces to a value v such that $\varnothing \vdash_{\mathbb{S}} v \colon t$.*

[4] We can also exploit redundancy information to exclude certain branches from typing (see Section 6.1), though it is not always possible during type reconstruction.

4.3 Comparison with $\mathbb{K}$

Our type system $\mathbb{S}$ extends $\mathbb{K}$ in the sense that every well-typed program of $\mathbb{K}$ is also well-typed in $\mathbb{S}$: we say that $\mathbb{S}$ is *complete* with respect to $\mathbb{K}$.

To show completeness, we define a translation $[\![\cdot]\!]_K$ which maps $\Bbbk$-types (i.e., types of $\mathbb{K}$) to $\mathbb{s}$-types (types of $\mathbb{S}$). The translation is parameterized by a kinding environment to make sense of type variables.

Definition 4.5 (Translation of types)**.** *Given a $\Bbbk$-type τ in a non-recursive kinding environment K, its* translation *is the $\mathbb{s}$-type $[\![\tau]\!]_K$ defined inductively by the rules in Figure 3.*

We define the translation of type schemes as $[\![\forall A.\, K' \rhd \tau]\!]_K = \forall A.\, [\![\tau]\!]_{K,K'}$ and that of type environments by translating each type scheme pointwise.

The only complex case is the translation of a constrained variable. We translate it to the same variable, in union with its lower bound and in intersection with its upper bound. Lower bounds and finite upper ones (i.e., those where $U \neq \mathcal{L}$) are represented by a union of variant types. In $\mathbb{K}$, a tag in U may be associated with more than one argument type, in which case its argument should have all these types. This is a somewhat surprising feature of the type system in OCaml—for details, see Garrigue [17, 19]—but here we can simply take the intersection of all argument types. For instance, the OCaml type `[< `A of int | `B of unit > `A] as` $\alpha$, represented in $\mathbb{K}$ by the type variable $\alpha$ with kind $(\{\text{`}\mathtt{A}\}, \{\text{`}\mathtt{A}, \text{`}\mathtt{B}\}, \{\text{`}\mathtt{A}\colon \mathtt{int}, \text{`}\mathtt{B}\colon \mathtt{unit}\})$, is translated into $(\text{`}\mathtt{A}(\mathtt{int}) \vee \alpha) \wedge (\text{`}\mathtt{A}(\mathtt{int}) \vee \text{`}\mathtt{B}(\mathtt{unit}))$.

The translation of an upper bound $U = \mathcal{L}$ is more involved. Ideally, we need the type

$$
\bigvee_{\text{`}tag \in \mathsf{dom}(T)} \text{`}tag\Big(\bigwedge_{\text{`}tag:\ \tau \in T} [\![\tau]\!]_K\Big) \ \vee\ \bigvee_{\text{`}tag \notin \mathsf{dom}(T)} \text{`}tag(\mathbb{1})
$$

which states that tags mentioned in T can only appear with arguments of the proper type, whereas tags not in T can appear with any argument. However, the union on the right is infinite and cannot be represented in our system; hence, in the definition in Figure 3 we use its complement with respect to the top type of variants $\mathbb{1}_{\mathrm{v}}$.[5]

In practice, a type $(t_{\mathrm{L}} \vee \alpha) \wedge t_{\mathrm{U}}$ can be replaced by its lower (respectively, upper) bound if α only appears in covariant (resp., contravariant) position.

We state the completeness property as follows.

[5] The type $\mathbb{1}_{\mathrm{v}}$ can itself be defined by complementation as

$$
\neg\big((\bigvee_{b \in \mathcal{B}} b) \vee (\mathbb{0} \to \mathbb{1}) \vee (\mathbb{1} \times \mathbb{1})\big)\ :
$$

the type of values which are not constants, nor abstractions, nor pairs.

$$\textit{TRs-Var}\ \frac{}{x\colon t \Rightarrow \{x \mathrel{\dot{\leq}} t\}} \qquad \textit{TRs-Const}\ \frac{}{c\colon t \Rightarrow \{c \mathrel{\dot{\leq}} t\}} \qquad \textit{TRs-Abstr}\ \frac{e\colon \beta \Rightarrow C}{\lambda x.\, e\colon t \Rightarrow \{\mathtt{def}\ \{x\colon \alpha\}\ \mathtt{in}\ C,\ \alpha \to \beta \mathrel{\dot{\leq}} t\}}$$

$$\textit{TRs-Appl}\ \frac{e_1\colon \alpha \to \beta \Rightarrow C_1 \qquad e_2\colon \alpha \Rightarrow C_2}{e_1\, e_2\colon t \Rightarrow C_1 \cup C_2 \cup \{\beta \mathrel{\dot{\leq}} t\}} \qquad \textit{TRs-Pair}\ \frac{e_1\colon \alpha_1 \Rightarrow C_1 \qquad e_2\colon \alpha_2 \Rightarrow C_2}{(e_1, e_2)\colon t \Rightarrow C_1 \cup C_2 \cup \{\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t\}}$$

$$\textit{TRs-Tag}\ \frac{e\colon \alpha \Rightarrow C}{\grave{}\mathit{tag}(e)\colon t \Rightarrow C \cup \{\grave{}\mathit{tag}(\alpha) \mathrel{\dot{\leq}} t\}} \qquad \textit{TRs-Match}\ \frac{\begin{array}{c} e_0\colon \alpha \Rightarrow C_0 \qquad t_i = (\alpha \setminus \bigvee_{j<i} \lbag p_j \rbag) \wedge \lbag p_i \rbag \\ \forall i \in I \qquad t_i /\!/\!/ p_i \Rightarrow (\Gamma_i, C_i) \qquad e_i\colon \beta \Rightarrow C_i' \\ C_0' = C_0 \cup (\bigcup_{i\in I} C_i) \cup \{\alpha \mathrel{\dot{\leq}} \bigvee_{i\in I} \lbag p_i \rbag\} \end{array}}{\mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i\in I}\colon t \Rightarrow \{\mathtt{let}\ [C_0'](\Gamma_i\ \mathtt{in}\ C_i')_{i\in I},\ \beta \mathrel{\dot{\leq}} t\}}$$

Figure 4. Constraint generation rules.

Theorem 4.4 (Preservation of typing)**.** *Let e be an expression, K a non-recursive kinding environment, Γ a $\Bbbk$-type environment, and τ a $\Bbbk$-type. If $K; \Gamma \vdash_{\Bbbk} e\colon \tau$, then $[\![\Gamma]\!]_K \vdash_{\mathbb{S}} e\colon [\![\tau]\!]_K$.*

Notice that we have defined $[\![\cdot]\!]_K$ by induction. Therefore, strictly speaking, we have only proved that $\mathbb{S}$ deduces all the judgements provable for non-recursive types in $\mathbb{K}$. Indeed, in the statement we require the kinding environment K to be non-recursive[6]. We conjecture that the result holds also with recursive kindings and that it can be proven by coinductive techniques.

5. Type reconstruction

In this section, we study type reconstruction for the $\mathbb{S}$ type system. We build on the work of Castagna et al. [6], who study local type inference and type reconstruction for the polymorphic version of $\mathbb{C}$Duce. In particular, we reuse their work on the resolution of the *tallying problem*, which plays in our system the same role as unification in ML.

Our contribution is threefold: (i) we prove type reconstruction for our system to be both sound and complete, while in Castagna et al. [6] it is only proven to be sound for $\mathbb{C}$Duce (indeed, we rely on the restricted role of intersection types in our system to obtain this result); (ii) we describe reconstruction with let-polymorphism and use structured constraints to separate constraint generation from constraint solving; (iii) we define reconstruction for full pattern matching. Both let-polymorphism and pattern matching are omitted in Castagna et al. [6].

Type reconstruction for a program (a closed expression) e consists in finding a type t such that $\varnothing \vdash_{\mathbb{S}} e\colon t$ can be derived: we see it as finding a type substitution θ such that $\varnothing \vdash_{\mathbb{S}} e\colon \alpha\theta$ holds for some fresh variable α. We generalize this to non-closed expressions and to reconstruction of types that are partially known. Thus, we say that type reconstruction consists—given an expression e, a type environment Γ, and a type t—in computing a type substitution θ such that $\Gamma\theta \vdash_{\mathbb{S}} e\colon t\theta$ holds, if any such θ exists.

Reconstruction in our system proceeds in two main phases. In the first, *constraint generation* (Section 5.1), we generate from an expression e and a type t a set of constraints that record the conditions under which e may be given type t. In the second phase, *constraint solving* (Sections 5.2–5.3), we solve (if possible) these constraints to obtain a type substitution θ.

We keep these two phases separate following an approach inspired by presentations of HM(X) [25]: we use structured constraints which contain expression variables, so that constraint generation does not depend on the type environment Γ that e is to be typed in. Γ is used later for constraint solving.

[6] We say K is non-recursive if it does not contain any cycle $\alpha, \alpha_1, \ldots, \alpha_n, \alpha$ such that the kind of each variable α_i contains α_{i+1}.

Constraint solving is itself made up of two steps: *constraint rewriting* (Section 5.2) and *type-constraint solving* (Section 5.3). In the former, we convert a set of structured constraints into a simpler set of subtyping constraints. In the latter, we solve this set of subtyping constraints to obtain a set of type substitutions; this latter step is analogous to unification in ML and is computed using the tallying algorithm of Castagna et al. [6]. Constraint rewriting also uses type-constraint solving internally; hence, these two steps are actually intertwined in practice.

5.1 Constraint generation

Given an expression e and a type t, constraint generation computes a finite set of constraints of the form defined below.

Definition 5.1 (Constraints)**.** *A* constraint *c is a term inductively generated by the following grammar:*

$$c ::= t \mathrel{\dot{\leq}} t \mid x \mathrel{\dot{\leq}} t \mid \mathtt{def}\ \Gamma\ \mathtt{in}\ C \mid \mathtt{let}\ [C](\Gamma_i\ \mathtt{in}\ C_i)_{i\in I}$$

where C ranges over constraint sets*, that is, finite sets of constraints, and where the range of every type environment Γ in constraints of the form* `def` *or* `let` *only contains types (i.e., trivial type schemes).*

A constraint of the form $t \mathrel{\dot{\leq}} t'$ requires $t\theta \leq t'\theta$ to hold for the final substitution θ. One of the form $x \mathrel{\dot{\leq}} t$ constrains the type of x (actually, an instantiation of its type scheme with fresh variables) in the same way. A definition constraint $\mathtt{def}\ \Gamma\ \mathtt{in}\ C$ introduces new expression variables, as we do in abstractions; these variables may then occur in C. We use `def` constraints to introduce monomorphic bindings (environments with types and not type schemes).

Finally, `let` constraints introduce polymorphic bindings. We use them for pattern matching: hence, we define them with multiple branches (the constraint sets C_i's), each with its own environment (binding the capture variables of each pattern to types). To solve a constraint $\mathtt{let}\ [C_0](\Gamma_i\ \mathtt{in}\ C_i)_{i\in I}$, we first solve C_0 to obtain a substitution θ; then, we apply θ to all types in each Γ_i and we generalize the resulting types; finally, we solve each C_i (in an environment expanded with the generalization of $\Gamma_i\theta$).

We define constraint generation as a relation $e\colon t \Rightarrow C$, given by the rules in Figure 4. We assume all variables introduced by the rules to be fresh (see the Appendix for the formal treatment of freshness: cf. Definition A.39 and Figures 13 and 14). Constraint generation for variables and constants (rules *TRs-Var* and *TRs-Const*) just yields a subtyping constraint. For an abstraction $\lambda x.\, e$ (rule *TRs-Abstr*), we generate constraints for the body and wrap them into a definition constraint binding x to a fresh variable α; we add a subtyping constraint to ensure that $\lambda x.\, e$ has type t by subsumption. The rules for applications, pairs, and tags are similar.

For pattern-matching expressions (rule *TRs-Match*), we use an auxiliary relation $t /\!/\!/ p \Rightarrow (\Gamma, C)$ to generate the pattern type

$$\frac{\forall i \in I \qquad \Gamma \vdash c_i \rightsquigarrow D_i}{\Gamma \vdash \{\, c_i \mid i \in I \,\} \rightsquigarrow \bigcup_{i \in I} D_i} \qquad \frac{}{\Gamma \vdash t \mathrel{\dot{\leq}} t' \rightsquigarrow \{t \mathrel{\dot{\leq}} t'\}} \qquad \frac{\Gamma(x) = \forall\{\alpha_1, \ldots, \alpha_n\}.\, t_x}{\Gamma \vdash x \mathrel{\dot{\leq}} t \rightsquigarrow \{t_x[{}^{\beta_1}/_{\alpha_1}, \ldots, {}^{\beta_n}/_{\alpha_n}] \mathrel{\dot{\leq}} t\}}$$

$$\frac{\Gamma, \Gamma' \vdash C \rightsquigarrow D}{\Gamma \vdash \mathtt{def}\ \Gamma'\ \mathtt{in}\ C \rightsquigarrow D} \qquad \frac{\begin{array}{c}\Gamma \vdash C_0 \rightsquigarrow D_0 \qquad \theta_0 \in \mathsf{tally}(D_0) \\ \forall i \in I \qquad \Gamma, \mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0) \vdash C_i \rightsquigarrow D_i\end{array}}{\Gamma \vdash \mathtt{let}\ [C_0](\Gamma_i\ \mathtt{in}\ C_i)_{i \in I} \rightsquigarrow \mathsf{equiv}(\theta_0) \cup \bigcup_{i \in I} D_i}$$

Figure 5. Constraint rewriting rules.

environment Γ, together with a set of constraints C in case the environment contains new type variables. The full definition is in the Appendix; as an excerpt, consider the rules for variable and tag patterns.

$$\frac{}{t /\!/\!/ x \Rightarrow (\{x \colon t\}, \varnothing)} \qquad \frac{\alpha /\!/\!/ p \Rightarrow (\Gamma, C)}{t /\!/\!/ \grave{}tag(p) \Rightarrow (\Gamma, C \cup \{t \mathrel{\dot{\leq}} \grave{}tag(\alpha)\})}$$

The rule for variable patterns produces no constraints (and the empty environment). Conversely, the rule for tags must introduce a new variable α to stand for the argument type: the constraint produced mirrors the use of the projection operator $\pi_{\grave{}tag}$ in the deductive system. To generate constraints for a pattern-matching expression, we generate them for the expression to be matched and for each branch separately. All these are combined in a `let` constraint, together with the constraints generated by patterns and with $\alpha \mathrel{\dot{\leq}} \bigvee_{i \in I} \lceil p_i \rfloor$, which ensures exhaustiveness.

5.2 Constraint rewriting

The first step of constraint solving consists in rewriting the constraint set into a simpler form that contains only subtyping constraints, that is, into a set of the form $\{t_1 \mathrel{\dot{\leq}} t'_1, \ldots, t_n \mathrel{\dot{\leq}} t'_n\}$ (i.e., no `let`, `def`, or expression variable). We call such sets *type-constraint sets* (ranged over by D).

Constraint rewriting is defined as a relation $\Gamma \vdash C \rightsquigarrow D$: between type environments, constraints or constraint sets, and type-constraint sets. It is given by the rules in Figure 5.

We rewrite constraint sets pointwise. We leave subtyping constraints unchanged. In variable type constraints, we replace the variable x with an instantiation of the type scheme $\Gamma(x)$ with the variables $\beta_1, \ldots, \beta_n$, which we assume to be fresh. We rewrite `def` constraints by expanding the environment and rewriting the inner constraint set.

The complex case is that of `let` constraints, which is where rewriting already performs type-constraint solving. We first rewrite the constraint set C_0. Then we extract a solution θ_0—if any exists—by the tally algorithm (described below). The algorithm can produce multiple alternative solutions: hence, this step is nondeterministic. Finally, we rewrite each of the C_i in an expanded environment. We perform generalization, so `let` constraints may introduce polymorphic bindings. The resulting type-constraint set is the union of the type-constraint sets obtained for each branch plus $\mathsf{equiv}(\theta_0)$, which is defined as

$$\mathsf{equiv}(\theta_0) = \textstyle\bigcup_{\alpha \in \mathsf{dom}(\theta_0)} \{\alpha \mathrel{\dot{\leq}} \alpha\theta_0,\ \alpha\theta_0 \mathrel{\dot{\leq}} \alpha\}\ .$$

We add the constraints of $\mathsf{equiv}(\theta_0)$ because tallying might generate multiple incompatible solutions for the constraints in D_0. The choice of θ_0 is arbitrary, but we must force subsequent steps of constraint solving to abide by it. Adding $\mathsf{equiv}(\theta_0)$ ensures that every solution θ to the resulting type-constraint set will satisfy $\alpha\theta \simeq \alpha\theta_0\theta$ for every α, and hence will not contradict our choice.

5.3 Type-constraint solving

Castagna et al. [6] define the *tallying problem* as the problem—in our terminology—of finding a substitution that satisfies a given type-constraint set.

Definition 5.2. *We say that a type substitution θ satisfies a type-constraint set D, written $\theta \Vdash D$, if $t\theta \leq t'\theta$ holds for every $t \mathrel{\dot{\leq}} t'$ in D. When θ satisfies D, we say it is a solution to the* tallying problem *of D.*

The tallying problem is the analogue in our system of the unification problem in ML. However, there is a very significant difference: while unification admits principal solutions, tallying does not. Indeed, the algorithm to solve the tallying problem for a type-constraint set produces a finite set of type substitutions. The algorithm is sound in that all substitutions it generates are solutions. It is complete in the sense that any other solution is less general than one of those in the set: we have a finite number of solutions which are principal when taken together, but not necessarily a single solution that is principal on its own.

This is a consequence of our semantic definition of subtyping. As an example, consider subtyping for product types: with a straightforward syntactic definition, a constraint $t_1 \times t'_1 \leq t_2 \times t'_2$ would simplify to the conjunction of two constraints $t_1 \leq t_2$ and $t'_1 \leq t'_2$. With semantic subtyping—where products are seen as Cartesian products—that simplification is sound, but it is not the only possible choice: either $t_1 \leq \mathbb{0}$ or $t'_1 \leq \mathbb{0}$ is also enough to ensure $t_1 \times t'_1 \leq t_2 \times t'_2$, since both ensure $t_1 \times t'_1 \simeq \mathbb{0}$. The three possible choices can produce incomparable solutions.

Castagna et al. [6, Section 3.2 and Appendix C.1] define a sound, complete, and terminating algorithm to solve the tallying problem, which can be adapted to our types by encoding variants as pairs. We refer to this algorithm here as tally (it is $\mathsf{Sol}_\varnothing$ in the referenced work) and state its properties.

Property 5.3 (Tallying algorithm)**.** *There exists a terminating algorithm* tally *such that, for any type-constraint set D,* $\mathsf{tally}(D)$ *is a finite, possibly empty, set of type substitutions.*

Theorem 5.1 (Soundness and completeness of tally)**.** *Let D be a type-constraint set. For any type substitution θ:*
- *if $\theta \in \mathsf{tally}(D)$, then $\theta \Vdash D$;*
- *if $\theta \Vdash D$, then $\exists \theta' \in \mathsf{tally}(D), \theta''. \forall \alpha \in \mathsf{dom}(\theta). \alpha\theta \simeq \alpha\theta'\theta''$.*

Hence, given a type-constraint set, we can use tally to either find a set of solutions or determine it has no solution: $\mathsf{tally}(D) = \varnothing$ occurs if and only if there exists no θ such that $\theta \Vdash D$.

5.3.1 Properties of type reconstruction

Type reconstruction as a whole consists in generating a constraint set C from an expression, rewriting this set into a type-constraint set D (which can require solving intermediate type-constraint sets) and finally solving D by the tally algorithm. Type reconstruction is both sound and complete with respect to the deductive type system $\mathbb{S}$. We state these properties in terms of constraint rewriting.

Theorem 5.2 (Soundness of constraint generation and rewriting)**.** *Let e be an expression, t a type, and Γ a type environment. If $e\colon t \Rightarrow C$, $\Gamma \vdash C \rightsquigarrow D$, and $\theta \Vdash D$, then $\Gamma\theta \vdash_{\mathbb{S}} e\colon t\theta$.*

Theorem 5.3 (Completeness of constraint generation and rewriting)**.** *Let e be an expression, t a type, and Γ a type environment. Let θ be a type substitution such that $\Gamma\theta \vdash_{\mathbb{S}} e\colon t\theta$.*

Let $e\colon t \Rightarrow C$. There exist a type-constraint set D and a type substitution θ', with $\mathsf{dom}(\theta) \cap \mathsf{dom}(\theta') = \varnothing$, such that $\Gamma \vdash C \rightsquigarrow D$ and $(\theta \cup \theta') \Vdash D$.

These theorems and the properties above express soundness and completeness for the reconstruction system. Decidability is a direct consequence of the termination of the tallying algorithm.

5.3.2 Practical issues

As compared to reconstruction in ML, our system has the disadvantage of being non-deterministic: in practice, an implementation should check every solution that tallying generates at each step of type-constraint solving until it finds a choice of solution which makes the whole program well-typed. This should be done at every step of generalization (that is, for every `match` expression) and might cripple efficiency. Whether this is significant in practice or not is a question that requires further study and experimentation. Testing multiple solutions cannot be avoided since our system does not admit principal types. For instance the function

```
let f(x,y) = (function (`A,`A)|(`B,`B)→`C)(x,y)
```

has both type `(`A,`A)→`C` and type `(`B,`B)→`C` (and neither is better than the other) but it is not possible to deduce for it their least upper bound `(`A,`A)∨(`B,`B)→`C` (which would be principal).

Multiple solutions often arise by instantiating some type variables by the empty type. Such solutions are in many cases subsumed by other more general solutions, but not always. For instance, consider the α `list` data-type (encoded as the recursive type `X = (α,X)∨[]`) together with the classic `map` function over lists (the type of which is $(\alpha \to \beta) \to \alpha\,\texttt{list} \to \beta\,\texttt{list}$). The application of `map` to the successor function $\texttt{succ} : \texttt{int} \to \texttt{int}$ has type $\texttt{int list} \to \texttt{int list}$, but also type `[]→[]` (obtained by instantiating all the variables of the type of `map` by the empty type). The latter type is correct, cannot be derived (by instantiation and/or subtyping) from the former, but it is seldom useful (it just states that `map(succ)` maps the empty list into the empty list). As such, it should be possible to define some preferred choice of solution (i.e., the solution that does not involve empty types) which is likely to be the most useful in practice. As it happens, we would like to try to restrict the system so that it only considers solutions without empty types. While it would make us lose completeness with respect to $\mathbb{S}$, it would be interesting to compare the restricted system with ML (with respect to which it could still be complete).

6. Extensions

In this section, we present three extensions or modifications to the $\mathbb{S}$ type system; the presentation is just sketched for space reasons: the details of all three can be found in the Appendix.

The first is the introduction of overloaded functions typed via intersection types, as done in ℂDuce. The second is a refinement of the typing of pattern matching, which we have shown as part of Example 2 (the function `g` and our definition of `map`). Finally, the third is a restriction of our system to adapt it to the semantics of the OCaml implementation which, unlike our calculus, cannot compare safely untagged values of different types at runtime.

6.1 Overloaded functions

ℂDuce allows the use of intersection types to type overloaded functions precisely: for example, it can type the negation function

$$\texttt{not} \stackrel{\text{def}}{=} \lambda x.\,\texttt{match}\ x\ \texttt{with}\ \texttt{true} \to \texttt{false} \mid \texttt{false} \to \texttt{true}$$

with the type $(\texttt{true} \to \texttt{false}) \wedge (\texttt{false} \to \texttt{true})$, which is more precise than $\texttt{bool} \to \texttt{bool}$. We can add this feature by changing the rule to type λ-abstractions to

$$\frac{\forall j \in J.\ \Gamma, \{x\colon t'_j\} \vdash e\colon t_j}{\Gamma \vdash \lambda x.e\colon \bigwedge_{j\in J} t'_j \to t_j}$$

which types the abstraction with an intersection of arrow types, provided each of them can be derived for it. The rule above roughly corresponds to the one introduced by Reynolds for the language Forsythe [26]. With this rule alone, however, one has only the so-called *coherent overloading* [24], that is, the possibility of assigning different types to the same piece of code, yielding an intersection type. In full-fledged overloading, instead, different pieces of code are executed for different types of the input. This possibility was first introduced by ℂDuce [1, 14] and it is obtained by typing pattern matching without taking into account the type of the branches that cannot be selected for a given input type. Indeed, the function "`not`" above cannot be given the type we want if we just add the rule above: it can neither be typed as $\texttt{true} \to \texttt{false}$ nor as $\texttt{false} \to \texttt{true}$.

To use intersections effectively for pattern matching, we need to exclude redundant patterns from typing. We do so by changing the rule *Ts-Match* (in Figure 2): when for some branch i we have $t_i \leq \mathbb{0}$, we do not type that branch at all, and we do not consider it in the result type (that is, we set $t'_i = \mathbb{0}$). In this way, if we take $t'_j = \texttt{true}$, we can derive $t_j = \texttt{false}$ (and vice versa). Indeed, if we assume that the argument is `true`, the second branch will never be selected: it is therefore sound not to type it at all. This typing technique is peculiar to ℂDuce's overloading. However, functions in ℂDuce are explicitly typed. As type reconstruction is undecidable for unrestricted intersection type systems, this extension would make annotations necessary in our system as well. We plan to study the extension of our system with intersection types for functions and to adapt reconstruction to also consider explicit annotations.

6.2 Refining the type of expressions in pattern matching

Two of our motivating examples concerning pattern matching (from Section 1, Example 2) involved a refinement of the typing of pattern matching that we have not described yet, but which can be added as a small extension of our $\mathbb{S}$ system.

Recall the function `g` defined as $\lambda x.\,\texttt{match}\ x\ \texttt{with}\ \texttt{`A} \to \texttt{id2}\ x \mid \_ \to x$, where `id2` has domain $\texttt{`A} \vee \texttt{`B}$. Like OCaml, $\mathbb{S}$ requires the type of $x$ to be a subtype of $\texttt{`A} \vee \texttt{`B}$, but this constraint is unnecessary because $\texttt{id2}\ x$ is only computed when $x = \texttt{`A}$. To capture this, we need pattern matching to introduce more precise types for variables in the matched expression; this is a form of *occurrence typing* [28] or *flow typing* [22].

We first consider pattern matching on a variable. In an expression $\texttt{match}\ x\ \texttt{with}\ (p_i \to e_i)_{i\in I}$ we can obtain this increased precision by using the type t_i—actually, its generalization—for x while typing the i-th branch. In the case of `g`, the first branch is typed assuming x has type $t_0 \wedge \texttt{`A}$, where $t_0$ is the type we have derived for $x$. As a result, the constraint $t_0 \wedge \texttt{`A} \leq \texttt{`A} \vee \texttt{`B}$ does not restrict t_0.

We can express this so as to reuse pattern environment generation. Let $(\!|\cdot|\!)\colon \mathcal{E} \to \mathcal{P}$ be a function such that $(\!|x|\!) = x$ and $(\!|e|\!) = \_$ when e is not a variable. Then, we obtain the typing above if we use

$$\Gamma, \mathsf{gen}_\Gamma(t_i // (\!|e_0|\!)), \mathsf{gen}_\Gamma(t_i // p_i)$$

as the type environment in which we type the i-th branch, rather than $\Gamma, \mathsf{gen}_\Gamma(t_i // p_i)$.

We generalize this approach to refine types also for variables occurring inside pairs and variants. To do so, we redefine $(\!|\cdot|\!)$. On variants, we let $(\!|\texttt{`}tag(e)|\!) = \texttt{`}tag((\!|e|\!))$. On pairs, ideally we

want $(\!|(e_1, e_2)|\!) = ((\!|e_1|\!), (\!|e_2|\!))$: however, pair patterns cannot have repeated variables, while (e_1, e_2) might. We therefore introduce a new form of pair pattern $\langle p_1, p_2 \rangle$ (only for internal use) which admits repeated variables: environment generation for such patterns intersects the types it obtains for each occurrence of a variable.

6.3 Applicability to OCaml

A thesis of this work is that the type system of OCaml—specifically, the part dealing with polymorphic variants and pattern matching—could be profitably replaced by an alternative, set-theoretic system. Of course, we need the set-theoretic system to be still type safe.

In Section 4, we stated that $\mathbb{S}$ is sound with respect to the semantics we gave in Section 2. However, this semantics is not precise enough, as it does not correspond to the behaviour of the OCaml implementation on ill-typed terms.[7]

Notably, OCaml does not record type information at runtime: values of different types cannot be compared safely and constants of different basic types might have the same representation (as, for instance, `1` and `true`). Consider as an example the two functions

$$\lambda x.\, \texttt{match}\ x\ \texttt{with}\ \texttt{true} \to \texttt{true} \mid \_ \to \texttt{false}$$
$$\lambda x.\, \texttt{match}\ x\ \texttt{with}\ (\texttt{true}, \texttt{true}) \to \texttt{true} \mid \_ \to \texttt{false}\ .$$

Both can be given the type $\mathbb{1} \to \texttt{bool}$ in $\mathbb{S}$, which is indeed safe in our semantics. Hence, we can apply both of them to `1`, and both return `false`. In OCaml, conversely, the first would return `true` and the second would cause a crash. The types $\texttt{bool} \to \texttt{bool}$ and $\texttt{bool} \times \texttt{bool} \to \texttt{bool}$, respectively, would be safe for these functions in OCaml.

To model OCaml more faithfully, we define an alternative semantics where matching a value v against a pattern p can have three outcomes rather than two: it can succeed ($v/p = \varsigma$), fail ($v/p = \Omega$), or be undefined ($v/p = \mho$). Matching is undefined whenever it is unsafe in OCaml: for instance, $1/\texttt{true} = 1/(\texttt{true}, \texttt{true}) = \mho$ (see Appendix A.5.3 for the full definition).

We use the same definition as before for reduction (see Section 2.2). Note that a `match` expression on a value reduces to the first branch for which matching is successful *if the result is* Ω *for all previous branches*. If matching for a branch is undefined, no branch after it can be selected; hence, there are fewer possible reductions with this semantics.

Adapting the type system requires us to restrict the typing of pattern matching so that undefined results cannot arise. We define the *compatible type* $\lceil p \rceil$ of a pattern p as the type of values v which can be safely matched with it: those for which $v/p \neq \mho$. For instance, $\lceil 1 \rceil = \texttt{int}$. The rule for pattern matching should require that the type t_0 of the matched expression be a subtype of all $\lceil p_i \rceil$.

Note that this restricts the use of union types in the system. For instance, if we have a value of type $\texttt{bool} \vee \texttt{int}$, we can no longer use pattern matching to discriminate between the two cases. This is to be expected in a language without runtime type tagging: indeed, union types are primarily used for variants, which reintroduce tagging explicitly. Nevertheless, having unions of non-variant types in the system is still useful, both internally (to type pattern matching) and externally (see Example 3 in Section 1, for instance).

7. Related work

We discuss here the differences between our system and other formalizations of variants in ML. We also compare our work with the work on $\mathbb{C}$Duce and other union/intersection type systems.

[7] We can observe this if we bypass type-checking, for instance by using `Obj.magic` for unsafe type conversions.

7.1 Variants in ML: formal models and OCaml

$\mathbb{K}$ is based on the framework of *structural polymorphism* and more specifically on the presentations by Garrigue [17, 19]. There exist several other systems with structural polymorphism: for instance, the earlier one by Garrigue [16] and more expressive constraint-based frameworks, like the presentation of HM(X) by Pottier and Rémy [25]. We have chosen as a starting point the system which corresponds most closely to the actual implementation in OCaml.

With respect to the system in Garrigue [17, 19], $\mathbb{K}$ differs mainly in three respects. First, Garrigue's system describes constraints more abstractly and can accommodate different forms of polymorphic typing of variants and of records. We only consider variants and, as a result, give a more concrete presentation. Second, we model full pattern matching instead of "shallow" case analysis. To our knowledge, pattern matching on polymorphic variants in OCaml is only treated in Garrigue [18] and only as concerns some problems with type reconstruction. We have chosen to formalize it to compare $\mathbb{K}$ to our set-theoretic type system $\mathbb{S}$, which admits a simpler formalization and more precise typing. However, we have omitted a feature of OCaml that allows refinement of variant types in alias patterns and which is modeled in Garrigue [17] by a `split` construct. While this feature makes OCaml more precise than $\mathbb{K}$, it is subsumed in $\mathbb{S}$ by the precise typing of capture variables. Third, we did not study type inference for $\mathbb{K}$. Since $\mathbb{S}$ is more expressive than $\mathbb{K}$ and since we describe complete reconstruction for it, extending Garrigue's inference system to pattern matching was unnecessary for the goals of this work.

As compared to OCaml itself (or, more precisely, to the fragment we consider) our formalization is different because it requires exhaustiveness; this might not always by practical in $\mathbb{K}$, but non-exhaustive pattern matching is no longer useful once we introduce more precise types, as in $\mathbb{S}$. Other differences include not considering variant refinement in alias patterns, as noted above, and the handling of conjunctive types, where OCaml is more restrictive than we are in order to infer more intuitive types [as discussed in 18, Section 4.1].

7.2 $\mathbb{S}$ and the $\mathbb{C}$Duce calculus

$\mathbb{S}$ reuses the subtyping relation defined by Castagna and Xu [4] and some of the work described in Castagna et al. [5, 6] (notably, the encoding of bounded polymorphism via type connectives and the algorithm to solve the tallying problem). Here, we explore the application of these elements to a markedly different language.

Castagna et al. [5, 6] study polymorphic typing for the $\mathbb{C}$Duce language, which features type-cases. Significantly, such type-cases can discriminate between functions of different types; pattern matching in ML cannot (indeed, it cannot distinguish between functions and non-functional values). As a result, the runtime semantics of $\mathbb{C}$Duce is quite involved and, unlike ours, not type-erasing; our setting has allowed us to simplify the type system too. Moreover, most of the work in Castagna et al. [5, 6] studies an explicitly-typed language (where functions can be typed with intersection types). In contrast, our language is implicitly typed. We focus our attention on type reconstruction and prove it sound and complete, thanks to the limited use we make of intersections. We have also introduced differences in presentation to conform our system to standard descriptions of the Hindley-Milner system.

7.3 Union types and pattern matching

The use of union and intersection types in ML has been studied in the literature of *refinement type* systems. For example, the theses of Davies [12] and Dunfield [13] describe systems where declared datatypes (such as the ordinary variants of OCaml) are refined by finite discriminated unions. Here we study a very different setting, because we consider *polymorphic* variants and, above all, we focus

on providing complete type reconstruction, while the cited works describe forms of bidirectional type checking which require type annotations. Conversely, our system makes a more limited use of intersection types, since it does not allow the derivation of intersection types for functions. Refinement type systems are closer in spirit to the work on ℂDuce which is why we refer the reader to Section 7 on related work in Castagna et al. [5] for a comprehensive comparison.

For what concerns programming languages we are not aware of any implicitly-typed language with full-fledged union types. The closest match to our work is probably Typed Racket [27, 28] which represents datatypes as unions of tagged types, as we do. However it does not perform type reconstruction: it is an explicitly-typed language with local type inference, that is, the very same setting studied for ℂDuce in Castagna et al. [6] whose Section 6 contains a thorough comparison with the type system of Typed Racket. Typed Racket also features *occurrence typing*, which refines the types of variables according to the results of tests (combinations of predicates on base types and selectors) to give a form of flow sensitivity. We introduced a similar feature in Section 6.2: we use pattern matching and hence consider tests which are as expressive as theirs, but we do not allow them to be abstracted out as functions.

8. Conclusion

This work shows how to add general union, intersection and difference types in implicitly-typed languages that traditionally use the HM type system. Specifically, we showed how to improve the current OCaml type system of polymorphic variants in four different aspects: its formalization, its meta-theoretic properties, the expressiveness of the system, and its practical ramifications. These improvements are obtained by a drastic departure from the current unification-based approach and by the injection in the system of set-theoretic types and semantic subtyping.

Our approach arguably improves the formalization of polymorphic variants: in our system we directly encode all meta-theoretic notions in a core—albeit rich—type theory, while the current OCaml system must introduce sophisticated "ad hoc" constructions (e.g., the definition of constrained kind, cf. Definition 3.2) to simulate subtyping. This is why, in our approach, bounded polymorphism can be encoded in terms of union and intersection types, and meta-theoretic properties such as exhaustiveness and redundancy in pattern matching can be internalized and expressed in terms of types and subtyping. Likewise, the most pleasant surprise of our formalization is the definition of the generality relation $\sqsubseteq$ on type schemes (cf. equation (1)): the current OCaml formalization requires complicated definitions such as the admissibility of type substitutions, while in our system it turns out to be the straightforward and natural generalization to subtyping of the usual relation of ML. A similar consideration can be done for unification, which is here generalized by the notion of tallying.

It could be objected that what we have done is just to push complexities down in the theory of semantic subtyping. However, the theory of semantic subtyping is well developed and tested, and, above all, it was defined independently from polymorphic variants. The fact that polymorphic variants can be encoded in semantic subtyping without any real modification of the latter looks to us as a further proof of the relevance of our solution. In the end we obtain a type system which is very natural: if we abstract the technicalities of the rule for pattern matching, the type system really is what one expects it to be: all (and only) the classic typing rules plus a subsumption rule. And even the rule *Ts-Match*, the most complicated one, is at the end what one should expect it to be: (1) type the matched expression e_0, (2) check whether the patterns are exhaustive, (3) for each branch (3.i) compute the set of the results of e_0 that are captured by the pattern of the branch, (3.ii) use them to deduce the type of the capture variables of the pattern (3.iii) generalize the types of these variables in order to type the body of the branch, and (4) return the union of the types of the branches.

The advantages of our approach are not limited to the formalization. The resulting system is more expressive—it types more programs while preserving static type safety—and natural, insofar as it removes the pathological behaviours we outlined in the introduction as well as problems found in real life [e.g., 21, 29]. The solution can be even more satisfactory if we extend the current syntax of OCaml types. For instance, Nicollet [21] shows the OCaml function `function `A → `B | x → x` which transforms `` `A `` into `` `B `` and leaves any other constructor unchanged. OCaml gives to this function the somewhat nonsensical type `([> `A | `B ] as α) → α`. Our reconstruction algorithm deduces instead the type `α → (`B | (α\`A))`: it correctly deduces that the result can be either `` `B `` or the variant in input, but can never be `` `A `` [for further examples of the use of difference types see 8, 11]. If we want to preserve the current syntax of OCaml types, this type should be approximated as `([> `B] as α) → α`; however, if we extend the syntax with differences (that in our system come for free), we gain the expressiveness that the kinding approach can only achieve with explicit row variables and that is needed, for instance, to encode exceptions [2]. But we can do more: by allowing also intersections in the syntax of OCaml types we could type Nicollet's function by the type `(`A → `B) & ((α\`A) → (α\`A))`, which is exact since it states that the function maps `` `A `` to `` `B `` and leaves any argument other than `` `A `` unchanged. As an aside, notice that types of this form provide an exact typing of exception handlers as intended by Blume et al. [2] (Nicollet's function can be seen as a handler that catches the exception `` `A `` yielding `` `B `` and lets all other values pass through).

Finally, our work improves some aspects of the theory of semantic subtyping as well: our type reconstruction copes with let-polymorphism and pattern matching and it is proven to be not only sound but also complete, all properties that the system in Castagna et al. [6] does not possess. Furthermore, the refinement we proposed in Section 6.2 applies to ℂDuce patterns as well, and it has already been implemented in the development version of ℂDuce.

This work is just the first step of a long-term research. Our short-term plan is to finish an ongoing implementation and test it, especially as concerns messages to show to the programmer. We also need to extend the subtyping relation used here to cope with types containing cyclic values (e.g., along the lines of the work of Bonsangue et al. [3]): the subtyping relation of Castagna and Xu [4] assumes that types contain only finite values, but cyclic values can be defined in OCaml.

The interest of this work is not limited to polymorphic variants. In the long term we plan to check whether building on this work it is possible to extend the syntax of OCaml patterns and types, so as to encode XML document types and provide the OCaml programmer with processing capabilities for XML documents like those that can be found in XML-centred programming languages such as ℂDuce. Likewise we want to explore the addition of intersection types to OCaml (or Haskell) in order to allow the programmer to define refinement types and check how such an integration blends with existing features, notably GADTs.

Acknowledgments

We want to thank Jacques Garrigue for his invaluable feedback on an early version of this work.

References

[1] V. Benzaken, G. Castagna, and A. Frisch. ℂDuce: an XML-centric general-purpose language. In *ACM SIGPLAN International Confer-*

ence on Functional Programming (ICFP), pages 51–63, 2003.

[2] M. Blume, U. A. Acar, and W. Chae. Exception handlers as extensible cases. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems (APLAS)*, LNCS, pages 273–289. Springer, 2008.

[3] M. Bonsangue, J. Rot, D. Ancona, F. de Boer, and J. Rutten. A coalgebraic foundation for coinductive union types. In *Automata, Languages, and Programming - 41st International Colloquium (ICALP)*, volume 8573 of *Lecture Notes in Computer Science*, pages 62–73. Springer, 2014.

[4] G. Castagna and Z. Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 94–106, 2011.

[5] G. Castagna, K. Nguyễn, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic functions with set-theoretic types. part 1: Syntax, semantics, and evaluation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 5–17, 2014.

[6] G. Castagna, K. Nguyễn, Z. Xu, and P. Abate. Polymorphic functions with set-theoretic types. part 2: Local type inference and type reconstruction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Mumbai, India*, pages 289–302, 2015.

[7] CDuce. `http://www.cduce.org`.

[8] [CAML-LIST 1]. Polymorphic variant difference. `https://goo.gl/WlPgdY`, May 2007. OCaml mailing list post.

[9] [CAML-LIST 2]. Variant filtering. `https://goo.gl/d7DQhU`, Feb. 2000. OCaml mailing list post.

[10] [CAML-LIST 3]. Polymorphic variant typing. `https://goo.gl/O054v1`, Feb. 2005. OCaml mailing list post.

[11] [CAML-LIST 4]. Getting rid of impossible polymorphic variant tags from infered types. `https://goo.gl/ELougz`, Mar. 2004. OCaml mailing list post.

[12] R. Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, May 2005.

[13] J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, Aug. 2007.

[14] A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping. In *LICS '02, 17th Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.

[15] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):1–67, 2008.

[16] J. Garrigue. Programming with polymorphic variants. In *ACM SIGPLAN Workshop on ML, Baltimore, Maryland, USA*, 1998. Informal proceedings.

[17] J. Garrigue. Simple type inference for structural polymorphism. In *International Workshop on Foundations of Object-Oriented Languages (FOOL), Portland, Oregon, USA*, 2002. Informal proceedings.

[18] J. Garrigue. Typing deep pattern-matching in presence of polymorphic variants. In *JSSST Workshop on Programming and Programming Languages, Gamagori, Japan*, 2004.

[19] J. Garrigue. A certified implementation of ML with structural polymorphism and recursive types. *Mathematical Structures in Computer Science*, 25:867–891, 2015.

[20] L. Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17(3):387–421, 2007.

[21] V. Nicollet. Do variant types in OCaml suck? `http://goo.gl/FOOwal`, Mar. 2011. Blog post.

[22] D. J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 335–354, Jan. 2013.

[23] T. Petrucciani. A set-theoretic type system for polymorphic variants in ML. Master's thesis, Università degli studi di Genova, 2015.

[24] B. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

[25] F. Pottier and D. Rémy. The essence of ML type inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[26] J. C. Reynolds. *Algol-like Languages*, chapter Design of the Programming Language Forsythe, pages 173–233. Birkhäuser, Boston, MA, 1997. ISBN 978-1-4612-4118-8. Peter W. O'Hearn and Robert D. Tennent (eds.).

[27] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California, USA*, pages 395–406, 2008.

[28] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 117–128, 2010.

[29] T. Wegrzanowski. Variant types in OCaml suck. `http://goo.gl/bY0bMA`, May 2006. Blog post.

A. Appendix

In this Appendix, we present full definitions of the language and type systems we have described, together with complete proofs of all results.

A.1 The language of polymorphic variants

A.1.1 Syntax

We assume that there exist a countable set $\mathcal{X}$ of *expression variables*, ranged over by x, y, z, ..., a set $\mathcal{C}$ of constants, ranged over by c, and a set $\mathcal{L}$ of tags, ranged over by $`tag$.

Definition A.1 (Expressions)**.** *An* expression e *is a term inductively generated by the following grammar:*

$$e ::= x \mid c \mid \lambda x.\, e \mid e\, e \mid (e, e) \mid `tag(e) \mid \mathtt{match}\ e\ \mathtt{with}\ (p_i \to e_i)_{i \in I}$$

where p ranges over the set $\mathcal{P}$ of patterns, defined below. We write $\mathcal{E}$ to denote the set of all expressions.

We define $\mathsf{fv}(e)$ to be the set of expression variables occurring free in the expression e, and we say that e is *closed* if and only if $\mathsf{fv}(e)$ is empty.

As customary, we consider expressions up to α-renaming of the variables bound by abstractions and by patterns.

Definition A.2 (Patterns)**.** *A* pattern p *is a term inductively generated by the following grammar:*

$$p ::= \_ \mid x \mid c \mid (p, p) \mid `tag(p) \mid p \& p \mid p | p$$

such that

- *in a pair pattern (p_1, p_2) or an intersection pattern $p_1 \& p_2$, $\mathsf{capt}(p_1) \cap \mathsf{capt}(p_2) = \varnothing$;*
- *in a union pattern $p_1 | p_2$, $\mathsf{capt}(p_1) = \mathsf{capt}(p_2)$,*

where $\mathsf{capt}(p)$ denotes the set of expression variables occurring as sub-terms in a pattern p (called the capture variables *of p).*

We write $\mathcal{P}$ to denote the set of all patterns.

A.1.2 Semantics

Definition A.3 (Values)**.** *A* value v *is a closed expression inductively generated by the following grammar.*

$$v ::= c \mid \lambda x.\, e \mid (v, v) \mid `tag(v)$$

Definition A.4 (Expression substitution)**.** *An* expression substitution ς *is a partial mapping of expression variables to values. We write $[\, {}^{v_i}/_{x_i} \mid i \in I \,]$ for the substitution which replaces free occurrences of x_i with v_i, for each $i \in I$. We write $e\varsigma$ for the application of the substitution to an expression e. We write $\varsigma_1 \cup \varsigma_2$ for the union of disjoint substitutions.*

Definition A.5 (Semantics of pattern matching)**.** *We write v/p for the result of matching a value v against a pattern p. We have either $v/p = \varsigma$, where ς is a substitution defined on the variables in $\mathsf{capt}(p)$, or $v/p = \Omega$. In the former case, we say that v matches p (or that p accepts v); in the latter, we say that matching fails.*

The definition of v/p is given inductively in Figure 6.

Definition A.6 (Evaluation contexts)**.** *Let the symbol $[\,]$ denote a hole. An* evaluation context E *is a term inductively generated by the following grammar.*

$$E ::= [\,] \mid E\, e \mid v\, E \mid (E, e) \mid (v, E) \mid `tag(E) \mid \mathtt{match}\ E\ \mathtt{with}\ (p_i \to e_i)_{i \in I}$$

We write $E[\, e \,]$ for the expression obtained by replacing the hole in E with the expression e.

Definition A.7 (Reduction)**.** *The reduction relation $\rightsquigarrow$ between expressions is given by the rules in Figure 7.*

A.2 Typing variants with kinding constraints

A.2.1 Definition of the $\mathbb{K}$ type system

We assume that there exists a countable set $\mathcal{V}$ of *type variables*, ranged over by α, β, γ, ... We also consider a finite set $\mathcal{B}$ of *basic types*, ranged over by b, and a function $b_{(\cdot)}$ from constants to basic types.

Definition A.8 (Types)**.** *A* type τ *is a term inductively generated by the following grammar.*

$$\tau ::= \alpha \mid b \mid \tau \to \tau \mid \tau \times \tau$$

$$
\begin{aligned}
v/\_ &= [\,] \\
v/x &= [{}^{v}/_{x}] \\
v/c &= \begin{cases} [\,] & \text{if } v = c \\ \Omega & \text{otherwise} \end{cases} \\
v/(p_1, p_2) &= \begin{cases} \varsigma_1 \cup \varsigma_2 & \text{if } v = (v_1, v_2) \text{ and } \forall i.\ v_i/p_i = \varsigma_i \\ \Omega & \text{otherwise} \end{cases} \\
v/\text{`}tag(p_1) &= \begin{cases} \varsigma_1 & \text{if } v = \text{`}tag(v_1) \text{ and } v_1/p_1 = \varsigma_1 \\ \Omega & \text{otherwise} \end{cases} \\
v/p_1 \& p_2 &= \begin{cases} \varsigma_1 \cup \varsigma_2 & \text{if } \forall i.\ v/p_i = \varsigma_i \\ \Omega & \text{otherwise} \end{cases} \\
v/p_1 | p_2 &= \begin{cases} v/p_1 & \text{if } v/p_1 \neq \Omega \\ v/p_2 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 6. Semantics of pattern matching.

$$
\textit{R-Appl}\ \frac{}{(\lambda x.\, e)\ v\ \rightsquigarrow\ e[{}^{v}/_{x}]} \qquad \textit{R-Match}\ \frac{v/p_j = \varsigma \qquad \forall i < j.\ v/p_i = \Omega}{\texttt{match}\ v\ \texttt{with}\ (p_i \to e_i)_{i \in I}\ \rightsquigarrow\ e_j \varsigma}\ j \in I
$$

$$
\textit{R-Ctx}\ \frac{e\ \rightsquigarrow\ e'}{E[\,e\,]\ \rightsquigarrow\ E[\,e'\,]}
$$

Figure 7. Small-step reduction relation.

Definition A.9 (Kinds)**.** *A* kind κ *is either the* unconstrained kind *"$\bullet$" or a* constrained kind*, that is, a triple (L, U, T) where:*

- *L is a finite set of tags $\{\text{`}tag_1, \ldots, \text{`}tag_n\}$;*
- *U is either a finite set of tags or the set $\mathcal{L}$ of all tags;*
- *T is a finite set of pairs of a tag and a type, written $\{\text{`}tag_1 \colon \tau_1, \ldots, \text{`}tag_n \colon \tau_n\}$ (its domain $\mathsf{dom}(T)$ is the set of tags occurring in it);*

and where the following conditions hold:

- *$L \subseteq U$, $L \subseteq \mathsf{dom}(T)$, and, if $U \neq \mathcal{L}$, $U \subseteq \mathsf{dom}(T)$;*
- *tags in L have a single type in T, that is, if $\text{`}tag \in L$, whenever both $\text{`}tag \colon \tau_1 \in T$ and $\text{`}tag \colon \tau_2 \in T$, we have $\tau_1 = \tau_2$.*

Definition A.10 (Kind entailment)**.** *The entailment relation $\cdot \vDash \cdot$ between constrained kinds is defined as*

$$
(L, U, T) \vDash (L', U', T') \iff L \supseteq L' \wedge U \subseteq U' \wedge T \supseteq T'\,.
$$

Definition A.11 (Kinding environments)**.** *A* kinding environment *K is a partial mapping from type variables to kinds. We write kinding environments as $K = \{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}$. We write K, K' for the updating of the kinding environment K with the new bindings in K'. It is defined as follows.*

$$
(K, K')(\alpha) = \begin{cases} K'(\alpha) & \textit{if } \alpha \in \mathsf{dom}(K') \\ K(\alpha) & \textit{otherwise} \end{cases}
$$

We say that a kinding environment is closed *if all the type variables that appear in the types in its range also appear in its domain. We say it is* canonical *if it is infinite and contains infinitely many variables of every kind.*

Definition A.12 (Type schemes)**.** *A* type scheme *σ is of the form $\forall A.\ K \triangleright \tau$, where:*

- *A is a finite set $\{\alpha_1, \ldots, \alpha_n\}$ of type variables;*
- *K is a kinding environment such that $\mathsf{dom}(K) = A$.*

We identify a type scheme $\forall\varnothing.\, \varnothing \triangleright \tau$, which quantifies no variable, with the type τ itself. We consider type schemes up to renaming of the variables they bind and disregard useless quantification (i.e., quantification of variables that do not occur in the type).

Definition A.13 (Free variables)**.** *The set of* free variables $\mathsf{var}_K(\sigma)$ *of a type scheme* σ *with respect to a kinding environment* K *is the minimum set satisfying the following equations.*

$$\begin{aligned}
\mathsf{var}_K(\forall A.\, K' \triangleright \tau) &= \mathsf{var}_{K,K'}(\tau) \setminus A \\
\mathsf{var}_K(\alpha) &= \begin{cases} \{\alpha\} \cup \bigcup_{\mathit{tag}:\, \tau \in T} \mathsf{var}_K(\tau) & \text{if } K(\alpha) = (L, U, T) \\ \{\alpha\} & \text{otherwise} \end{cases} \\
\mathsf{var}_K(b) &= \varnothing \\
\mathsf{var}_K(\tau_1 \to \tau_2) &= \mathsf{var}_K(\tau_1) \cup \mathsf{var}_K(\tau_2) \\
\mathsf{var}_K(\tau_1 \times \tau_2) &= \mathsf{var}_K(\tau_1) \cup \mathsf{var}_K(\tau_2)
\end{aligned}$$

We say that a type τ *is* ground *or* closed *if and only if* $\mathsf{var}_\varnothing(\tau)$ *is empty. We say that a type or a type scheme is* closed in a kinding environment K *if all its free variables are in the domain of* K.

Definition A.14 (Type substitutions)**.** *A* type substitution θ *is a finite mapping of type variables to types. We write* $[\,{}^{\tau_i}/_{\alpha_i} \mid i \in I\,]$ *for the type substitution which simultaneously replaces* α_i *with* τ_i*, for each* $i \in I$*. We write* $\tau\theta$ *for the application of the substitution* θ *to the type* τ*, which is defined as follows.*

$$\begin{aligned}
\alpha\theta &= \begin{cases} \tau' & \text{if } {}^{\tau'}/_{\alpha} \in \theta \\ \alpha & \text{otherwise} \end{cases} \\
b\theta &= b \\
(\tau_1 \to \tau_2)\theta &= (\tau_1\theta) \to (\tau_2\theta) \\
(\tau_1 \times \tau_2)\theta &= (\tau_1\theta) \times (\tau_2\theta)
\end{aligned}$$

We extend the var *operation to substitutions as*

$$\mathsf{var}_K(\theta) = \bigcup_{\alpha \in \mathsf{dom}(\theta)} \mathsf{var}_K(\alpha\theta)\,.$$

We extend application of substitutions to the typing component of a constrained kind (L, U, T)*:* $T\theta$ *is given by the pointwise application of* θ *to all types in* T*. We extend it to kinding environments:* $K\theta$ *is given by the pointwise application of* θ *to the typing component of every constrained kind in the range of* K*. We extend it to type schemes* $\forall A.\, K \triangleright \tau$*: by renaming quantified variables, we assume* $A \cap (\mathsf{dom}(\theta) \cup \mathsf{var}_\varnothing(\theta)) = \varnothing$*, and we have* $(\forall A.\, K \triangleright \tau)\theta = \forall A.\, K\theta \triangleright \tau\theta$*.*

We write $\theta_1 \cup \theta_2$ *for the union of disjoint substitutions and* $\theta_1 \circ \theta_2$ *for the composition of substitutions.*

Definition A.15 (Admissibility of a type substitution)**.** *A type substitution* θ *is* admissible *between two kinding environments* K *and* K'*, written* $K \vdash \theta \colon K'$*, if and only if, for every type variable* α *such that* $K(\alpha) = (L, U, T)$*,* $\alpha\theta$ *is a type variable such that* $K'(\alpha\theta) = (L', U', T')$ *and* $(L', U', T') \vDash (L, U, T\theta)$*.*

Definition A.16 (Type environments)**.** *A* type environment Γ *is a partial mapping from expression variables to type schemes. We write type environments as* $\Gamma = \{x_1 \colon \sigma_1, \ldots, x_n \colon \sigma_n\}$*.*

We write Γ, Γ' *for the updating of the type environment* Γ *with the new bindings in* Γ'*. It is defined as follows.*

$$(\Gamma, \Gamma')(x) = \begin{cases} \Gamma'(x) & \text{if } x \in \mathsf{dom}(\Gamma') \\ \Gamma(x) & \text{otherwise} \end{cases}$$

We extend the var *operation to type environments as*

$$\mathsf{var}_K(\Gamma) = \bigcup_{\sigma \in \mathsf{range}(\Gamma)} \mathsf{var}_K(\sigma)\,.$$

Definition A.17 (Generalization)**.** *We define the* generalization *of a type* τ *with respect to a kinding environment* K *and a type environment* Γ *as the type scheme*

$$\mathsf{gen}_{K;\Gamma}(\tau) = \forall A.\, K' \triangleright \tau$$

where $A = \mathsf{var}_K(\tau) \setminus \mathsf{var}_K(\Gamma)$ *and* $K' = \{\, \alpha :: K(\alpha) \mid \alpha \in A \,\}$*.*

We extend this definition to type environments which only contain types (i.e., trivial type schemes) as

$$\mathsf{gen}_{K;\Gamma}(\{\, x_i \colon \tau_i \mid i \in I \,\}) = \{\, x_i \colon \mathsf{gen}_{K;\Gamma}(\tau_i) \mid i \in I \,\}\,.$$

$$\textit{TPk-Wildcard}\ \frac{}{K \vdash \_ : \tau \Rightarrow \varnothing} \qquad \textit{TPk-Var}\ \frac{}{K \vdash x : \tau \Rightarrow \{x : \tau\}}$$

$$\textit{TPk-Const}\ \frac{}{K \vdash c : b_c \Rightarrow \varnothing} \qquad \textit{TPk-Pair}\ \frac{K \vdash p_1 : \tau_1 \Rightarrow \Gamma_1 \qquad K \vdash p_2 : \tau_2 \Rightarrow \Gamma_2}{K \vdash (p_1, p_2) : \tau_1 \times \tau_2 \Rightarrow \Gamma_1 \cup \Gamma_2}$$

$$\textit{TPk-Tag}\ \frac{K \vdash p : \tau \Rightarrow \Gamma \qquad K(\alpha) = (L, U, T) \qquad (\text{`}\mathit{tag} \in U \text{ implies } \text{`}\mathit{tag} : \tau \in T)}{K \vdash \text{`}\mathit{tag}(p) : \alpha \Rightarrow \Gamma}$$

$$\textit{TPk-And}\ \frac{K \vdash p_1 : \tau \Rightarrow \Gamma_1 \qquad K \vdash p_2 : \tau \Rightarrow \Gamma_2}{K \vdash p_1 \& p_2 : \tau \Rightarrow \Gamma_1 \cup \Gamma_2}$$

$$\textit{TPk-Or}\ \frac{K \vdash p_1 : \tau \Rightarrow \Gamma \qquad K \vdash p_2 : \tau \Rightarrow \Gamma}{K \vdash p_1 | p_2 : \tau \Rightarrow \Gamma}$$

Figure 8. Pattern environment generation for $\mathbb{K}$.

$$\textit{Tk-Var}\ \frac{\tau \in \mathsf{inst}_K(\Gamma(x))}{K; \Gamma \vdash_{\mathbb{K}} x : \tau} \qquad \textit{Tk-Const}\ \frac{}{K; \Gamma \vdash_{\mathbb{K}} c : b_c} \qquad \textit{Tk-Abstr}\ \frac{K; \Gamma, \{x : \tau_1\} \vdash_{\mathbb{K}} e : \tau_2}{K; \Gamma \vdash_{\mathbb{K}} \lambda x.\, e : \tau_1 \to \tau_2}$$

$$\textit{Tk-Appl}\ \frac{K; \Gamma \vdash_{\mathbb{K}} e_1 : \tau' \to \tau \qquad K; \Gamma \vdash_{\mathbb{K}} e_2 : \tau'}{K; \Gamma \vdash_{\mathbb{K}} e_1\ e_2 : \tau} \qquad \textit{Tk-Pair}\ \frac{K; \Gamma \vdash_{\mathbb{K}} e_1 : \tau_1 \qquad K; \Gamma \vdash_{\mathbb{K}} e_2 : \tau_2}{K; \Gamma \vdash_{\mathbb{K}} (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\textit{Tk-Tag}\ \frac{K; \Gamma \vdash_{\mathbb{K}} e : \tau \qquad K(\alpha) \vDash (\{\text{`}\mathit{tag}\}, \mathcal{L}, \{\text{`}\mathit{tag} : \tau\})}{K; \Gamma \vdash_{\mathbb{K}} \text{`}\mathit{tag}(e) : \alpha}$$

$$\textit{Tk-Match}\ \frac{\begin{array}{c} K; \Gamma \vdash_{\mathbb{K}} e_0 : \tau_0 \qquad \tau_0 \preccurlyeq_K \{\, p_i \mid i \in I \,\} \\ \forall i \in I \qquad K \vdash p_i : \tau_0 \Rightarrow \Gamma_i \qquad K; \Gamma, \mathsf{gen}_{K;\Gamma}(\Gamma_i) \vdash_{\mathbb{K}} e_i : \tau \end{array}}{K; \Gamma \vdash_{\mathbb{K}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I} : \tau}$$

Figure 9. Typing relation for $\mathbb{K}$.

Definition A.18 (Instances of a type scheme)**.** *The set of* instances *of a type scheme $\forall A.\, K' \triangleright \tau$ in a kinding environment K is defined as*

$$\mathsf{inst}_K(\forall A.\, K' \triangleright \tau) = \{\, \tau\theta \mid \mathsf{dom}(\theta) \subseteq A \ \wedge\ K, K' \vdash \theta : K \,\}\,.$$

We say that a type scheme σ_1 is more general *than a type scheme σ_2 in K, and we write $\sigma_1 \sqsubseteq_K \sigma_2$, if $\mathsf{inst}_K(\sigma_1) \supseteq \mathsf{inst}_K(\sigma_2)$.*

We extend this notion to type environments as

$$\Gamma_1 \sqsubseteq_K \Gamma_2 \iff \mathsf{dom}(\Gamma_1) = \mathsf{dom}(\Gamma_2) \wedge \forall x \in \mathsf{dom}(\Gamma_1).\, \Gamma_1(x) \sqsubseteq_K \Gamma_2(x)\,.$$

Definition A.19 (Pattern environment generation)**.** *The environment generated by pattern matching is given by the relation $K \vdash p : \tau \Rightarrow \Gamma$ (*the pattern p can match type τ in K, producing the bindings in Γ*), defined by the rules in Figure 8.*

Definition A.20 (Exhaustiveness)**.** *We say that a set of patterns P is* exhaustive *with respect to a type τ in a kinding environment K, and we write $\tau \preccurlyeq_K P$, when*

$$\forall K', \theta, v.\, (K \vdash \theta : K' \ \wedge\ K'; \varnothing \vdash_{\mathbb{K}} v : \tau\theta) \implies \exists p \in P, \varsigma.\, v/p = \varsigma\,.$$

Definition A.21 (Typing relation)**.** *The typing relation $K; \Gamma \vdash_{\mathbb{K}} e : \tau$ (e* is given type τ in the kinding environment K and the type environment Γ*) is defined by the rules in Figure 9, where we require K to be closed and Γ and τ to be closed with respect to K. We also assume that K is canonical.*

A.2.2 Properties of the $\mathbb{K}$ type system

Lemma A.1 (Generation for values)**.** *Let v be a value. Then:*

- *if $K; \Gamma \vdash_{\mathbb{K}} v : b$, then $v = c$ for some constant c such that $b_c = b$;*
- *if $K; \Gamma \vdash_{\mathbb{K}} v : \tau_1 \to \tau_2$, then v is of the form $\lambda x.\, e$ and $K; \Gamma, \{x : \tau_1\} \vdash_{\mathbb{K}} e : \tau_2$;*
- *if $K; \Gamma \vdash_{\mathbb{K}} v : \tau_1 \times \tau_2$, then v is of the form (v_1, v_2), $K; \Gamma \vdash_{\mathbb{K}} v_1 : \tau_1$, and $K; \Gamma \vdash_{\mathbb{K}} v_2 : \tau_2$;*

- *if* $K; \Gamma \vdash_{\mathbb{K}} v \colon \alpha$*, then* v *is of the form* $\grave{}tag(v_1)$*,* $K(\alpha) = (L, U, T)$*,* $\grave{}tag \in L$*, and* $K; \Gamma \vdash_{\mathbb{K}} v_1 \colon \tau_1$ *for the only type* τ_1 *such that* $\grave{}tag \colon \tau_1 \in T$*.*

Proof. The typing rules are syntax-directed, so the last rule applied to type a value is fixed by its form. All these rules derive types of different forms, thus the form of the type assigned to a value determines the last rule used. In each case the premises of the rule entail the consequences above. □

Lemma A.2 (Correctness of environment generation)**.** *Let* p *be a pattern and* v *a value such that* $v/p = \varsigma$*. If* $K; \Gamma \vdash_{\mathbb{K}} v \colon \tau$ *and* $K \vdash p \colon \tau \Rightarrow \Gamma'$*, then, for all* $x \in \mathsf{capt}(p)$*,* $K; \Gamma \vdash_{\mathbb{K}} x\varsigma \colon \Gamma'(x)$*.*

Proof. By induction on the derivation of $K \vdash p \colon \tau \Rightarrow \Gamma'$. We reason by cases on the last applied rule.

Cases TPk-Wildcard and TPk-Const There is nothing to prove since $\mathsf{capt}(p) = \varnothing$.

Case TPk-Var We have

$$v/x = [v/x] \qquad K \vdash x \colon \tau \Rightarrow \{x \colon \tau\}$$

and must prove $K; \Gamma \vdash_{\mathbb{K}} x[v/x] \colon \{x \colon \tau\}(x)$, which we know by hypothesis.

Case TPk-Pair We have

$$K \vdash (p_1, p_2) \colon \tau_1 \times \tau_2 \Rightarrow \Gamma_1' \cup \Gamma_2' \qquad K \vdash p_1 \colon \tau_1 \Rightarrow \Gamma_1' \qquad K \vdash p_2 \colon \tau_2 \Rightarrow \Gamma_2' \,.$$

By Lemma A.1, $K; \Gamma \vdash_{\mathbb{K}} v \colon \tau_1 \times \tau_2$ implies $v = (v_1, v_2)$ and $K; \Gamma \vdash_{\mathbb{K}} v_i \colon \tau_i$ for both i. Furthermore, $(v_1, v_2)/(p_1, p_2) = \varsigma = \varsigma_1 \cup \varsigma_2$, and $v_i/p_i = \varsigma_i$ for both i. For each capture variable x, we can apply the induction hypothesis to the sub-pattern which contains x and conclude.

Case TPk-Tag We have

$$K \vdash \grave{}tag(p_1) \colon \alpha \Rightarrow \Gamma' \qquad K \vdash p_1 \colon \tau_1 \Rightarrow \Gamma'$$
$$K(\alpha) = (L, U, T) \qquad (\grave{}tag \in U \text{ implies } \grave{}tag \colon \tau_1 \in T) \,.$$

Since $v/\grave{}tag(p_1) = \varsigma$, we know $v = \grave{}tag(v_1)$. Hence, by Lemma A.1, we have $\grave{}tag \in L$ and $K; \Gamma \vdash_{\mathbb{K}} v_1 \colon \tau_1'$ with $\grave{}tag \colon \tau_1' \in T$. Since $\grave{}tag \in U$, we also have $\grave{}tag \colon \tau_1 \in T$ and hence $\tau_1 = \tau_1'$ (as $\grave{}tag$ is also in L and can only have a single type in T).

We therefore know $K \vdash p_1 \colon \tau_1 \Rightarrow \Gamma'$ and $K; \Gamma \vdash_{\mathbb{K}} v_1 \colon \tau_1$, as well as $v_1/p_1 = \varsigma$. We can apply the induction hypothesis to conclude.

Cases TPk-And and TPk-Or Straightforward application of the induction hypothesis, to both sub-patterns for intersections and to the one that is actually selected for unions. □

Lemma A.3 (Stability of environment generation under type substitutions)**.** *If* $K \vdash p \colon \tau \Rightarrow \Gamma$*, then* $K' \vdash p \colon \tau\theta \Rightarrow \Gamma\theta$ *for every type substitution* θ *such that* $K \vdash \theta \colon K'$*.*

Proof. By induction on the derivation of $K \vdash p \colon \tau \Rightarrow \Gamma$. We reason by cases on the last applied rule.

Cases TPk-Wildcard, TPk-Var, and TPk-Const Straightforward.

Case TPk-Pair We have

$$K \vdash (p_1, p_2) \colon \tau_1 \times \tau_2 \Rightarrow \Gamma_1 \cup \Gamma_2 \qquad K \vdash p_1 \colon \tau_1 \Rightarrow \Gamma_1 \qquad K \vdash p_2 \colon \tau_2 \Rightarrow \Gamma_2 \,.$$

By the induction hypothesis we derive both $K' \vdash p_1 \colon \tau_1\theta \Rightarrow \Gamma_1\theta$ and $K' \vdash p_2 \colon \tau_2\theta \Rightarrow \Gamma_2\theta$, then we apply *TPk-Pair* again to conclude.

Case TPk-Tag We have

$$K \vdash \grave{}tag(p_1) \colon \alpha \Rightarrow \Gamma \qquad K \vdash p_1 \colon \tau_1 \Rightarrow \Gamma$$
$$K(\alpha) = (L, U, T) \qquad (\grave{}tag \in U \text{ implies } \grave{}tag \colon \tau_1 \in T) \,.$$

By the induction hypothesis we derive $K' \vdash p_1 \colon \tau_1\theta \Rightarrow \Gamma\theta$. Since $K \vdash \theta \colon K'$, $\alpha\theta$ must be a variable β such that $K'(\beta) = (L', U', T')$. To apply *TPk-Tag* and conclude, we must establish that, if $\grave{}tag \in U'$, then $\grave{}tag \colon \tau_1\theta \in T'$. Since admissibility also implies $(L', U', T') \vDash (L, U, T\theta)$, we have $U' \subseteq U$ and $T\theta \subseteq T'$. Hence, if $\grave{}tag \in U'$, then $\grave{}tag \in U$, in which case $\grave{}tag \colon \tau_1 \in T$ and therefore $\grave{}tag \colon \tau_1\theta \in T\theta$, and $\grave{}tag \colon \tau_1\theta \in T'$.

Cases TPk-And and TPk-Or Straightforward application of the induction hypothesis, analogously to the case of pair patterns. □

Lemma A.4 (Stability of exhaustiveness under type substitutions)**.** *If $\tau \preccurlyeq_K P$, then $\tau\theta \preccurlyeq_{K'} P$ for any type substitution θ such that $K \vdash \theta \colon K'$.*

Proof. We must prove, for every K'', θ' such that $K' \vdash \theta' \colon K''$ and every v such that $K''; \varnothing \vdash_{\mathbb{K}} v \colon \tau\theta\theta'$, that there exists a $p \in P$ which accepts v. This holds because $\theta' \circ \theta$ is such that $K \vdash \theta' \circ \theta \colon K''$: for any α such that $K(\alpha) = (L, U, T)$, we have $K'(\alpha\theta) = (L', U', T')$ and hence $K''(\alpha\theta\theta') = (L'', U'', T'')$; we have $(L', U', T') \vDash (L, U, T\theta)$ and $(L'', U'', T'') \vDash (L', U', T'\theta')$ and therefore $(L'', U'', T'') \vDash (L, U, T\theta\theta')$. The conclusion follows by the definition of $\tau \preccurlyeq_K P$. □

Lemma A.5. *If $\mathsf{var}_K(\Gamma_1) \subseteq \mathsf{var}_K(\Gamma_2)$, then, for every type τ, $\mathsf{gen}_{K;\Gamma_1}(\tau) \sqsubseteq_K \mathsf{gen}_{K;\Gamma_2}(\tau)$.*

Proof. An instance of $\mathsf{gen}_{K;\Gamma_2}(\tau)$ is a type $\tau\theta$ such that $\mathsf{dom}(\theta) \subseteq \mathsf{var}_K(\tau) \setminus \mathsf{var}_K(\Gamma_2)$ and $K \vdash \theta \colon K$. It is also an instance of $\mathsf{gen}_{K;\Gamma_1}(\tau)$, with the same θ, since $\mathsf{var}_K(\tau) \setminus \mathsf{var}_K(\Gamma_2) \subseteq \mathsf{var}_K(\tau) \setminus \mathsf{var}_K(\Gamma_1)$. □

Lemma A.6 (Weakening)**.** *Let K be a kinding environment and Γ_1, Γ_2 two type environments such that $\Gamma_1 \sqsubseteq_K \Gamma_2$ and $\mathsf{var}_K(\Gamma_1) \subseteq \mathsf{var}_K(\Gamma_2)$. If $K; \Gamma_2 \vdash_{\mathbb{K}} e \colon \tau$, then $K; \Gamma_1 \vdash_{\mathbb{K}} e \colon \tau$.*

Proof. By induction on the derivation of $K; \Gamma_2 \vdash_{\mathbb{K}} e \colon \tau$. We reason by cases on the last applied rule.

Case Tk-Var We have:

$$K; \Gamma_2 \vdash_{\mathbb{K}} x \colon \tau \qquad \tau \in \mathsf{inst}_K(\Gamma_2(x))$$

and hence, since $\Gamma_1 \sqsubseteq_K \Gamma_2$, we have $\tau \in \mathsf{inst}_K(\Gamma_1(x))$ and apply *Tk-Var* to conclude.

Case Tk-Const Straightforward.

Case Tk-Abstr We have:

$$K; \Gamma_2 \vdash_{\mathbb{K}} \lambda x.\, e_1 \colon \tau_1 \to \tau_2 \qquad K; \Gamma_2, \{x \colon \tau_1\} \vdash_{\mathbb{K}} e_1 \colon \tau_2 \,.$$

Since $\Gamma_1 \sqsubseteq_K \Gamma_2$, we have $\Gamma_1, \{x \colon \tau_1\} \sqsubseteq_K \Gamma_2, \{x \colon \tau_1\}$, and, since $\mathsf{var}_K(\Gamma_1) \subseteq \mathsf{var}_K(\Gamma_2)$, we have $\mathsf{var}_K(\Gamma_1, \{x \colon \tau_1\}) \subseteq \mathsf{var}_K(\Gamma_2, \{x \colon \tau_1\})$. Thus we may derive $K; \Gamma_1, \{x \colon \tau_1\} \vdash_{\mathbb{K}} e_1 \colon \tau_2$ by the induction hypothesis and apply *Tk-Abstr* to conclude.

Cases Tk-Appl, Tk-Pair, and Tk-Tag Straightforward application of the induction hypothesis.

Case Tk-Match We have

$$\begin{gathered} K; \Gamma_2 \vdash_{\mathbb{K}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I} \colon \tau \\ K; \Gamma_2 \vdash_{\mathbb{K}} e_0 \colon \tau_0 \qquad \tau_0 \preccurlyeq_K \{\, p_i \mid i \in I \,\} \\ \forall i \in I.\ K \vdash p_i \colon \tau_0 \Rightarrow \Gamma_i \qquad K; \Gamma_2, \mathsf{gen}_{K;\Gamma_2}(\Gamma_i) \vdash_{\mathbb{K}} e_i \colon \tau \ . \end{gathered}$$

By the induction hypothesis, we derive $K; \Gamma_1 \vdash_{\mathbb{K}} e_0 \colon \tau_0$.

For every branch, note that by Lemma A.5 $\mathsf{var}_K(\Gamma_1) \subseteq \mathsf{var}_K(\Gamma_2)$ implies $\mathsf{gen}_{K;\Gamma_1}(\tau) \sqsubseteq_K \mathsf{gen}_{K;\Gamma_2}(\tau)$ for any τ. Hence, we have $\Gamma_1, \mathsf{gen}_{K;\Gamma_1}(\Gamma_i) \sqsubseteq_K \Gamma_2, \mathsf{gen}_{K;\Gamma_2}(\Gamma_i)$. Additionally, since $\mathsf{var}_K(\mathsf{gen}_{K;\Gamma_1}(\Gamma_i)) \subseteq \mathsf{var}_K(\Gamma_1)$, we have $\mathsf{var}_K(\Gamma_1, \mathsf{gen}_{K;\Gamma_1}(\Gamma_i)) \subseteq \mathsf{var}_K(\Gamma_2, \mathsf{gen}_{K;\Gamma_2}(\Gamma_i))$.

Hence we may apply the induction hypothesis for all i to derive $K; \Gamma_1, \mathsf{gen}_{K;\Gamma_1}(\Gamma_i) \vdash_{\mathbb{K}} e_i \colon \tau$ and then apply *Tk-Match* to conclude. □

Lemma A.7 (Stability of typing under type substitutions)**.** *Let K, K' be two closed, canonical kinding environments and θ a type substitution such that $K \vdash \theta \colon K'$. If $K; \Gamma \vdash_{\mathbb{K}} e \colon \tau$, then $K'; \Gamma\theta \vdash_{\mathbb{K}} e \colon \tau\theta$.*

Proof. By induction on the derivation of $K; \Gamma \vdash_{\mathbb{K}} e \colon \tau$. We reason by cases on the last applied rule.

Case Tk-Var We have

$$\begin{gathered} K; \Gamma \vdash_{\mathbb{K}} x \colon \tau \qquad \tau \in \mathsf{inst}_K(\Gamma(x)) \\ \Gamma(x) = \forall A.\, K_x \triangleright \tau_x \qquad \tau = \tau_x\theta_x \qquad \mathsf{dom}(\theta_x) \subseteq A \qquad K, K_x \vdash \theta_x \colon K \end{gathered}$$

and must show

$$K'; \Gamma\theta \vdash_{\mathbb{K}} x \colon \tau\theta \,.$$

By α-renaming we can assume that θ does not involve A, that is, $A \cap \mathsf{dom}(\theta) = \varnothing$ and $A \cap \mathsf{var}_\varnothing(\theta) = \varnothing$, and also that $A \cap (\mathsf{dom}(K') \cup \mathsf{var}_\varnothing(K')) = \varnothing$, that is, that the variables in A are not assigned a kind in K' nor do they appear in the types in the typing component of the kinds in K'.

Under these assumptions, $(\Gamma\theta)(x) = \forall A.\, K_x\theta \triangleright \tau_x\theta$. We must show that $\tau\theta = \tau_x\theta\theta'_x$ for a substitution θ'_x such that $\mathsf{dom}(\theta'_x) \subseteq A$ and $K', K_x\theta \vdash \theta'_x \colon K'$.

Let $\theta'_x = [\,{}^{\alpha\theta_x\theta}/_{\alpha} \mid \alpha \in A\,]$. First, we show that $\tau_x\theta\theta'_x = \tau_x\theta_x\theta = \tau\theta$, by showing that, for any α, $\alpha\theta\theta'_x = \alpha\theta_x\theta$. If $\alpha \in A$, then $\alpha\theta\theta'_x = \alpha\theta'_x = \alpha\theta_x\theta$ (θ is not defined on the variables in A). If $\alpha \notin A$, then $\alpha\theta\theta'_x = \alpha\theta$ (θ never produces any variable in A) and $\alpha\theta_x\theta = \alpha\theta$ as $\alpha \notin \mathsf{dom}(\theta_x)$.

Since $\mathsf{dom}(\theta'_x) \subseteq A$ holds, we only need to establish that $K', K_x\theta \vdash \theta'_x \colon K'$. This requires proving, for each α such that $(K', K_x\theta)(\alpha) = (L, U, T)$, that $\alpha\theta'_x$ is a type variable such that $K'(\alpha\theta'_x) = (L', U', T')$ and $(L', U', T') \vDash (L, U, T\theta'_x)$.

Such an α can either be in the domain of $K_x\theta$ (if and only if it is in A) or in the domain of K'. In the latter case, we have $\alpha\theta'_x = \alpha$, since $\alpha \notin A$, and hence its kind in K' is the same as in $K', K_x\theta$. We must prove $(L, U, T) \vDash (L, U, T\theta'_x)$, which holds because the variables in A do not appear in T since $(L, U, T) \in \mathsf{range}(K')$.

In the former case, we have $(K_x\theta)(\alpha) = (L, U, T)$ and hence $K_x(\alpha) = (L, U, T_1)$, with $T = T_1\theta$. Also, $\alpha\theta'_x = \alpha\theta_x\theta$. Since $K, K_x \vdash \theta_x \colon K$, $K(\alpha\theta_x) = (L_2, U_2, T_2)$. Then, since $K \vdash \theta \colon K'$, $K'(\alpha\theta_x\theta) = (L', U', T')$. We know $(L_2, U_2, T_2) \vDash (L, U, T_1\theta_x)$ and $(L', U', T') \vDash (L_2, U_2, T_2\theta)$. Both $L' \supseteq L$ and $U' \subseteq U$ hold by transitivity. We show $T' \supseteq T\theta'_x$ holds as well. If $\grave{}\mathit{tag} \colon \tau \in T\theta'_x$, since $T = T_1\theta$, then $\grave{}\mathit{tag} \colon \tau_1 \in T_1$ and $\tau = \tau_1\theta\theta'_x = \tau_1\theta_x\theta$. We thus have $\grave{}\mathit{tag} \colon \tau_1\theta_x \in T_1\theta_x$ and therefore $\grave{}\mathit{tag} \colon \tau_1\theta_x \in T_2$ and $\grave{}\mathit{tag} \colon \tau_1\theta_x\theta \in T'$.

Case Tk-Const Straightforward.

Case Tk-Abstr We have:

$$K; \Gamma \vdash_{\mathbb{k}} \lambda x.\, e_1 \colon \tau_1 \to \tau_2 \qquad K; \Gamma, \{x \colon \tau_1\} \vdash_{\mathbb{k}} e_1 \colon \tau_2\ .$$

By the induction hypothesis we have $K'; \Gamma\theta, \{x \colon \tau_1\theta\} \vdash_{\mathbb{k}} e_1 \colon \tau_2\theta$. Then by *Tk-Abstr* we derive $K'; \Gamma\theta \vdash_{\mathbb{k}} \lambda x.\, e_1 \colon (\tau_1 \to \tau_2)\theta$, since $(\tau_1 \to \tau_2)\theta = (\tau_1\theta) \to (\tau_2\theta)$.

Cases Tk-Appl and Tk-Pair Straightforward application of the induction hypothesis.

Case Tk-Match For the sake of clarity, we first prove the simpler case corresponding to (the encoding of) `let`, where—simplifying environment generation—we have

$$K; \Gamma \vdash_{\mathbb{k}} \mathtt{match}\ e_0\ \mathtt{with}\ x \to e_1 \colon \tau \qquad K; \Gamma \vdash_{\mathbb{k}} e_0 \colon \tau_0 \qquad K; \Gamma, \mathsf{gen}_{K;\Gamma}(\{x \colon \tau_0\}) \vdash_{\mathbb{k}} e_1 \colon \tau$$

and must show

$$K'; \Gamma\theta \vdash_{\mathbb{k}} \mathtt{match}\ e_0\ \mathtt{with}\ x \to e_1 \colon \tau\theta$$

which we prove by establishing, for some type $\hat{\tau_0}$, that

$$K'; \Gamma\theta \vdash_{\mathbb{k}} e_0 \colon \hat{\tau_0} \qquad K'; \Gamma\theta, \mathsf{gen}_{K';\Gamma\theta}(\{x \colon \hat{\tau_0}\}) \vdash_{\mathbb{k}} e_1 \colon \tau\theta\ .$$

Let $A = \{\alpha_1, \ldots, \alpha_n\} = \mathsf{var}_K(\tau_0) \setminus \mathsf{var}_K(\Gamma)$. We assume that the variables in A do not appear in the kinds of variables not in A, that is, that if $K(\alpha) = (L, U, T)$ and $\alpha \notin A$, then $\mathsf{var}_K(T) \cap A = \varnothing$.

This assumption is justified by the following observations. The variables in A only appear quantified in the environment used for the typing derivation for e_1. Therefore we may assume that they do not appear in τ: if they do, it is because they have been chosen when instantiating some type scheme and, since K is canonical, we might have chosen some other variable of the same kind. As for the occurrences of the variables in A in the derivation for e_0, a similar reasoning applies. These variables do not appear free in the environment (neither directly in a type in Γ, nor in the kinds of variables which appear free in Γ). Therefore, if they occur in τ_0 it is because they have been chosen either during instantiation of a type scheme or when typing an abstraction, and in both cases we might have chosen a different variable.

Now we rename these variables so that θ will not have effect on them. Let $B = \{\beta_1, \ldots, \beta_n\}$ be a set of type variables such that $B \cap (\mathsf{dom}(\theta) \cup \mathsf{var}_\varnothing(\theta)) = \varnothing$ and $B \cap \mathsf{var}_\varnothing(\Gamma) = \varnothing$. Let $\theta_0 = [{}^{\beta_1}/_{\alpha_1}, \ldots, {}^{\beta_n}/_{\alpha_n}]$ and $\theta' = \theta \circ \theta_0$. Since K' is canonical, we can choose each β_i so that, if $K(\alpha_i) = \bullet$, then $K'(\beta_i) = \bullet$, and if $K(\alpha_i) = (L, U, T)$, then $K(\beta_i) = (L, U, T\theta')$. As for A, we choose B so that the kinds in K' for variables not in B do not contain variables of B.

We show $K \vdash \theta' \colon K'$. For each α such that $K(\alpha) = (L, U, T)$, if $\alpha \in A$ then $\alpha = \alpha_i$ for some i, $\alpha\theta' = \beta_i$ and kind entailment holds straightforwardly by our choice of β_i. If $\alpha \notin A$, then $\alpha\theta' = \alpha\theta$ and the admissibility of θ implies $K'(\alpha\theta) = (L', U', T')$ and $(L', U', T') \vDash (L, U, T\theta)$. We have $T\theta = T\theta'$ because of our assumption on A.

Since θ' is admissibile, by the induction hypothesis applied to θ', we derive $K; \Gamma\theta' \vdash_{\mathbb{k}} e_0 \colon \tau_0\theta'$. Since the variables in A do not appear in Γ, we have $\Gamma\theta' = \Gamma\theta$. We choose $\hat{\tau_0}$ to be $\tau_0\theta'$.

We apply the induction hypothesis to the derivation for e_1, this time using θ as the substitution. Now we have:

$$K'; \Gamma\theta \vdash_{\mathbb{k}} e_0 \colon \tau_0\theta' \qquad K'; \Gamma\theta, (\mathsf{gen}_{K;\Gamma}(\{x \colon \tau_0\}))\theta \vdash_{\mathbb{k}} e_1 \colon \tau\theta\ .$$

We apply weakening (Lemma A.6) to derive from the latter the typing we need, that is,

$$K'; \Gamma\theta, \mathsf{gen}_{K';\Gamma\theta}(\{x : \tau_0\theta'\}) \vdash_{\mathbb{k}} e_1 \colon \tau\theta \,.$$

To do so we must show

$$\Gamma\theta, \mathsf{gen}_{K';\Gamma\theta}(\{x \colon \tau_0\theta'\}) \sqsubseteq_{K'} \Gamma\theta, (\mathsf{gen}_{K;\Gamma}(\{x \colon \tau_0)\})\theta$$
$$\mathsf{var}_{K'}(\Gamma\theta, \mathsf{gen}_{K';\Gamma\theta}(\{x \colon \tau_0\theta'\})) \subseteq \mathsf{var}_{K'}(\Gamma\theta, (\mathsf{gen}_{K;\Gamma}(\{x \colon \tau_0)\})\theta) \,.$$

The latter holds because $\mathsf{var}_{K'}(\Gamma\theta, \mathsf{gen}_{K';\Gamma\theta}(\{x \colon \tau_0\theta'\})) \subseteq \mathsf{var}_{K'}(\Gamma\theta)$.
As for the former, we prove $\mathsf{gen}_{K';\Gamma\theta}(\{x \colon \tau_0\theta'\}) \sqsubseteq_{K'} (\mathsf{gen}_{K;\Gamma}(\{x \colon \tau_0)\})\theta$. We have

$$\mathsf{gen}_{K;\Gamma}(\{x \colon \tau_0\}) = \forall A.\, K_x \triangleright \tau_0 \qquad K_x = \{\, \alpha :: K(\alpha) \mid \alpha \in A \,\} \,.$$

By α-renaming of the quantified variables we can write

$$\mathsf{gen}_{K;\Gamma}(\{x \colon \tau_0\}) = \forall B.\, K^\star_x \triangleright \tau_0\theta_0$$
$$K^\star_x = \{\, \beta_i :: \bullet \mid \alpha_i :: \bullet \in K_x \,\} \cup \{\, \beta_i :: (L, U, T\theta_0) \mid \alpha_i :: (L, U, T) \in A \,\}$$

and, since θ does not involve B,

$$(\mathsf{gen}_{K;\Gamma}(\{x \colon \tau_0\}))\theta = \forall B.\, K^\star_x\theta \triangleright \tau_0\theta_0\theta = \forall B.\, K'_x \triangleright \tau_0\theta'$$
$$K'_x = \{\, \beta :: K'(\beta) \mid \beta \in B \,\} \,.$$

The other type scheme is

$$\mathsf{gen}_{K';\Gamma\theta}(\tau_0\theta') = \forall C.\, K'_C \triangleright \tau_0\theta'$$
$$C = \mathsf{var}_{K'}(\tau_0\theta') \setminus \mathsf{var}_{K'}(\Gamma\theta) \qquad K'_C = \{\, \beta :: K'(\beta) \mid \beta \in C \,\} \,.$$

We show $B \subseteq C$, which concludes the proof (because the kinding environments are both restrictions of K'). Consider $\beta_i \in B$. We have $\alpha_i \in \mathsf{var}_K(\tau_0) \setminus \mathsf{var}_K(\Gamma)$. Then $\beta_i = \alpha_i\theta' \in \mathsf{var}_{K'}(\tau_0\theta')$. Furthermore $\beta_i \notin \mathsf{var}_{K'}(\Gamma\theta)$ holds because $\Gamma\theta$ does not contain variables in B (Γ does not contain them and θ does not introduce them) and variables in B do not appear in the kinds of other variables which are not themselves in B.

We now consider the rule *Tk-Match* in its generality. We have

$$K; \Gamma \vdash_{\mathbb{k}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I} \colon \tau$$
$$K; \Gamma \vdash_{\mathbb{k}} e_0 \colon \tau_0 \qquad \tau_0 \preccurlyeq_K \{\, p_i \mid i \in I \,\}$$
$$\forall i \in I.\ K \vdash p_i \colon \tau_0 \Rightarrow \Gamma_i \qquad K; \Gamma, \mathsf{gen}_{K;\Gamma}(\Gamma_i) \vdash_{\mathbb{k}} e_i \colon \tau$$

and must show

$$K'; \Gamma\theta \vdash_{\mathbb{k}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I} \colon \tau\theta$$

which we prove by establishing, for some $\hat{\tau_0}$ and $\{\, \hat{\Gamma_i} \mid i \in I \,\}$, that

$$K'; \Gamma\theta \vdash_{\mathbb{k}} e_0 \colon \hat{\tau_0} \qquad \hat{\tau_0} \preccurlyeq_{K'} \{\, p_i \mid i \in I \,\}$$
$$\forall i \in I.\ K' \vdash p_i \colon \hat{\tau_0} \Rightarrow \hat{\Gamma_i} \qquad K'; \Gamma\theta, \mathsf{gen}_{K';\Gamma\theta}(\hat{\Gamma_i}) \vdash_{\mathbb{k}} e_i \colon \tau\theta \ .$$

For the derivation for e_0 we proceed as above and have $\hat{\tau_0} = \tau_0\theta'$. By Lemma A.4 we have $\tau_0\theta' \preccurlyeq_{K'} \{\, p_i \mid i \in I \,\}$. By Lemma A.3, we have $K' \vdash p_i \colon \tau_0\theta' \Rightarrow \Gamma_i\theta'$ and thus take $\hat{\Gamma_i} = \Gamma_i\theta'$.

We proceed as before also for the derivations for each branch. The difference is that, to apply weakening, we must prove the two premises for the environments and not for τ_0 alone. The condition on variables is straightforward, as before. For the other we prove, for each $x \in \mathsf{capt}(p_i)$ and assuming $\Gamma_i(x) = \tau_x$,

$$\Gamma\theta, \mathsf{gen}_{K';\Gamma\theta}(\tau_x\theta') \sqsubseteq_{K'} \Gamma\theta, (\mathsf{gen}_{K;\Gamma}(\tau_x))\theta \,.$$

We show it as for τ_0 above: $\mathsf{var}_K(\tau_x)$ is always a subset of $\mathsf{var}_K(\tau_0)$ because environment generation does not introduce new variables. □

Lemma A.8 (Expression substitution)**.** *Let $x_1, \ldots, x_n$ be distinct variables and $v_1, \ldots, v_n$ values. Let $\Gamma' = \{x_1 \colon \sigma_1, \ldots, x_n \colon \sigma_n\}$ and $\varsigma = [{}^{v_1}/_{x_1}, \ldots, {}^{v_n}/_{x_n}]$.*

If $K; \Gamma, \Gamma' \vdash_{\mathbb{k}} e \colon \tau$ and, for all $k \in \{1, \ldots, n\}$ and for all $\tau_k \in \mathsf{inst}_K(\sigma_k)$, $K; \Gamma \vdash_{\mathbb{k}} v_k \colon \tau_k$, then $K; \Gamma \vdash_{\mathbb{k}} e\varsigma \colon \tau$.

Proof. By induction on the derivation of $K; \Gamma, \Gamma' \vdash_{\mathbb{k}} e \colon \tau$. We reason by cases on the last applied rule.

Case Tk-Var We have

$$K; \Gamma, \Gamma' \vdash_{\mathbb{k}} x \colon \tau \qquad \tau \in \mathsf{inst}_K((\Gamma, \Gamma')(x)) \,.$$

Either $x = x_k$ for some k or not. In the latter case, $x\varsigma = x$, $x \notin \mathsf{dom}(\Gamma')$ and hence $(\Gamma, \Gamma')(x) = \Gamma(x)$. Then, since $\tau \in \mathsf{inst}_K((\Gamma, \Gamma')(x))$, $\tau \in \mathsf{inst}_K(\Gamma(x))$ and *Tk-Var* can be applied.

If $x = x_k$, then $(\Gamma, \Gamma')(x) = \Gamma'(x) = \sigma_k$. We must then prove $K; \Gamma \vdash_{\mathbb{k}} v_k \colon \tau$, which we know by hypothesis since $\tau \in \mathsf{inst}_K(\sigma_k)$.

Case Tk-Const Straightforward.

Case Tk-Abstr We have

$$K; \Gamma, \Gamma' \vdash_{\mathbb{k}} \lambda x.\, e_1 \colon \tau_1 \to \tau_2 \qquad K; \Gamma, \Gamma', \{x \colon \tau_1\} \vdash_{\mathbb{k}} e_1 \colon \tau_2 \,.$$

By α-renaming we can assume $x \notin \mathsf{dom}(\Gamma')$; then $(\lambda x.\, e_1)\varsigma = \lambda x.\, (e_1\varsigma)$ and $\Gamma, \Gamma', \{x \colon \tau_1\} = \Gamma, \{x \colon \tau_1\}, \Gamma'$. Therefore we have $K; \Gamma, \{x \colon \tau_1\}, \Gamma' \vdash_{\mathbb{k}} e_1 \colon \tau_2$ and, by the induction hypothesis, $K; \Gamma, \{x \colon \tau_1\} \vdash_{\mathbb{k}} e_1\varsigma \colon \tau_2$. We apply *Tk-Abstr* to conclude.

Cases Tk-Appl, Tk-Pair, and Tk-Tag Straightforward application of the induction hypothesis.

Case Tk-Match We have

$$\begin{gathered} K; \Gamma, \Gamma' \vdash_{\mathbb{k}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I} \colon \tau \\ K; \Gamma, \Gamma' \vdash_{\mathbb{k}} e_0 \colon \tau_0 \qquad \tau_0 \preccurlyeq_K \{\, p_i \mid i \in I \,\} \\ \forall i \in I.\ K \vdash p_i \colon \tau_0 \Rightarrow \Gamma_i \qquad K; \Gamma, \Gamma', \mathsf{gen}_{K;\Gamma,\Gamma'}(\Gamma_i) \vdash_{\mathbb{k}} e_i \colon \tau \ . \end{gathered}$$

We assume by α-renaming that no capture variable of any pattern is in the domain of Γ'. Then, $(\mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I})\varsigma = \mathtt{match}\ e_0\varsigma\ \mathtt{with}\ (p_i \to e_i\varsigma)_{i \in I}$ and $\Gamma, \Gamma', \mathsf{gen}_{K;\Gamma,\Gamma'}(\Gamma_i) = \Gamma, \mathsf{gen}_{K;\Gamma,\Gamma'}(\Gamma_i), \Gamma'$ for any i.

By the induction hypothesis, we derive $K; \Gamma \vdash_{\mathbb{k}} e_0\varsigma \colon \tau_0$ and $K; \Gamma, \mathsf{gen}_{K;\Gamma,\Gamma'}(\Gamma_i) \vdash_{\mathbb{k}} e_i\varsigma \colon \tau$ for all i. From the latter, we prove $K; \Gamma, \mathsf{gen}_{K;\Gamma}(\Gamma_i) \vdash_{\mathbb{k}} e_i\varsigma \colon \tau$ by weakening (Lemma A.6): we have $\mathsf{gen}_{K;\Gamma}(\Gamma_i) \sqsubseteq_K \mathsf{gen}_{K;\Gamma,\Gamma'}(\Gamma_i)$ by Lemma A.5—since $\mathsf{var}_K(\Gamma) \subseteq \mathsf{var}_K(\Gamma, \Gamma')$—and clearly we have $\mathsf{var}_K(\Gamma, \mathsf{gen}_{K;\Gamma}(\Gamma_i)) \subseteq \mathsf{var}_K(\Gamma, \mathsf{gen}_{K;\Gamma,\Gamma'}(\Gamma_i))$ since $\mathsf{var}_K(\mathsf{gen}_{K;\Gamma}(\Gamma_i)) \subseteq \mathsf{var}_K(\Gamma)$. □

Theorem A.9 (Progress)**.** *Let e be a well-typed, closed expression. Then, either e is a value or there exists an expression e' such that $e \rightsquigarrow e'$.*

Proof. By hypothesis we have $K; \varnothing \vdash_{\mathbb{k}} e \colon \tau$. The proof is by induction on its derivation; we reason by cases on the last applied rule.

Case Tk-Var This case does not occur because variables are not closed.

Case Tk-Const In this case e is a constant c and therefore a value.

Case Tk-Abstr In this case e is an abstraction $\lambda x.\, e_1$. Since it is also closed, it is a value.

Case Tk-Appl We have

$$K; \varnothing \vdash_{\mathbb{k}} e_1\ e_2 \colon \tau \qquad K; \varnothing \vdash_{\mathbb{k}} e_1 \colon \tau' \to \tau \qquad K; \varnothing \vdash_{\mathbb{k}} e_2 \colon \tau' \,.$$

By the induction hypothesis, each of e_1 and e_2 either is a value or may reduce. If $e_1 \rightsquigarrow e_1'$, then $e_1\ e_2 \rightsquigarrow e_1'\ e_2$. If e_1 is a value and $e_2 \rightsquigarrow e_2'$, then $e_1\ e_2 \rightsquigarrow e_1\ e_2'$.

If both are values then, by Lemma A.1, e_1 has the form $\lambda x.\, e_3$ for some e_3. Then, we can apply *R-Appl* and $e_1\ e_2 \rightsquigarrow e_3[{}^{e_2}/_{x}]$.

Case Tk-Pair We have

$$K; \varnothing \vdash_{\mathbb{k}} (e_1, e_2) \colon \tau_1 \times \tau_2 \qquad K; \varnothing \vdash_{\mathbb{k}} e_1 \colon \tau_1 \qquad K; \varnothing \vdash_{\mathbb{k}} e_2 \colon \tau_2 \,.$$

By the induction hypothesis, each of e_1 and e_2 either is a value or may reduce. If $e_1 \rightsquigarrow e_1'$, then $(e_1, e_2) \rightsquigarrow (e_1', e_2)$. If e_1 is a value and $e_2 \rightsquigarrow e_2'$, then $(e_1, e_2) \rightsquigarrow (e_1, e_2')$. If both are values, then (e_1, e_2) is also a value.

Case Tk-Tag We have

$$K; \varnothing \vdash_{\mathbb{k}} \text{`}\mathit{tag}(e_1) \colon \alpha \qquad K; \varnothing \vdash_{\mathbb{k}} e_1 \colon \tau_1 \,.$$

Analogously to the previous case, by the induction hypothesis we have that either e_1 is a value or $e_1 \rightsquigarrow e_1'$. In the former case, $\text{`}\mathit{tag}(e_1)$ is a value as well. In the latter, we have $\text{`}\mathit{tag}(e_1) \rightsquigarrow \text{`}\mathit{tag}(e_1')$.

Case Tk-Match We have

$$K; \varnothing \vdash_{\mathbb{k}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I} \colon \tau \qquad K; \varnothing \vdash_{\mathbb{k}} e_0 \colon \tau_0 \qquad \tau_0 \preccurlyeq_K \{\, p_i \mid i \in I \,\} \,.$$

By the inductive hypothesis, either e_0 is a value or it may reduce. In the latter case, if $e_0 \rightsquigarrow e_0'$, then $\mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I} \rightsquigarrow \mathtt{match}\ e_0'\ \mathtt{with}\ (p_i \to e_i)_{i \in I}$.

If e_0 is a value, on the other hand, the expression may reduce by application of *R-Match*. Since $\tau_0 \preccurlyeq_K \{\, p_i \mid i \in I \,\}$ and e_0 is a value of type τ_0 (and therefore satisfies the premises of the definition of exhaustiveness, with $\theta = [\,]$ and $K = K'$), there exists at least an $i \in I$ such that $e_0/p_i = \varsigma$ for some substitution ς. Let j be the least of these i and ς_j the corresponding substitution; then $\texttt{match}\ e_0\ \texttt{with}\ (p_i \to e_i)_{i\in I} \rightsquigarrow e_j\varsigma_j$. □

Theorem A.10 (Subject reduction)**.** *Let e be an expression and τ a type such that $K; \Gamma \vdash_{\mathbb{K}} e : \tau$. If $e \rightsquigarrow e'$, then $K; \Gamma \vdash_{\mathbb{K}} e' : \tau$.*

Proof. By induction on the derivation of $K; \Gamma \vdash_{\mathbb{K}} e : \tau$. We reason by cases on the last applied rule.

Cases Tk-Var, Tk-Const, and Tk-Abstr These cases may not occur: variables, constants, and abstractions never reduce.

Case Tk-Appl We have

$$K; \Gamma \vdash_{\mathbb{K}} e_1\ e_2 : \tau \qquad K; \Gamma \vdash_{\mathbb{K}} e_1 : \tau' \to \tau \qquad K; \Gamma \vdash_{\mathbb{K}} e_2 : \tau'\,.$$

$e_1\ e_2 \rightsquigarrow e'$ occurs in any of three ways: (i) $e_1 \rightsquigarrow e_1'$ and $e' = e_1'\ e_2$; (ii) e_1 is a value, $e_2 \rightsquigarrow e_2'$ and $e' = e_1\ e_2'$; (iii) both e_1 and e_2 are values, e_1 is of the form $\lambda x.\, e_3$, and $e' = e_3[e_2/x]$.

In the first case, we derive by the induction hypothesis that $K; \Gamma \vdash_{\mathbb{K}} e_1' : \tau' \to \tau$ and conclude by applying *Tk-Appl* again. The second case is analogous.

In the third case, we know by Lemma A.1 that $K; \Gamma, \{x : \tau'\} \vdash_{\mathbb{K}} e_3 : \tau$. We also know that e_2 is a value such that $K; \Gamma \vdash_{\mathbb{K}} e_2 : \tau'$. Then, by Lemma A.8, $K; \Gamma \vdash_{\mathbb{K}} e_3[e_2/x] : \tau$.

Case Tk-Pair We have

$$K; \Gamma \vdash_{\mathbb{K}} (e_1, e_2) : \tau_1 \times \tau_2 \qquad K; \Gamma \vdash_{\mathbb{K}} e_1 : \tau_1 \qquad K; \Gamma \vdash_{\mathbb{K}} e_2 : \tau_2\,.$$

$(e_1, e_2) \rightsquigarrow e'$ occurs either because $e_1 \rightsquigarrow e_1'$ and $e' = (e_1', e_2)$, or because e_1 is a value, $e_2 \rightsquigarrow e_2'$, and $e' = (e_1, e_2')$. In either case, the induction hypothesis allows us to derive that the type of the component that reduces is preserved; therefore, we can apply *Tk-Pair* again to conclude.

Case Tk-Tag Analogously to the previous case, a variant expression only reduces if its argument does, so we apply the induction hypothesis and *Tk-Tag* to conclude.

Case Tk-Match We have

$$\frac{K; \Gamma \vdash_{\mathbb{K}} \texttt{match}\ e_0\ \texttt{with}\ (p_i \to e_i)_{i\in I} : \tau \qquad K; \Gamma \vdash_{\mathbb{K}} e_0 : \tau_0 \qquad \forall i \in I.\ K \vdash p_i : \tau_0 \Rightarrow \Gamma_i \qquad K; \Gamma, \mathsf{gen}_{K;\Gamma}(\Gamma_i) \vdash_{\mathbb{K}} e_i : \tau}{}\ .$$

$\texttt{match}\ e_0\ \texttt{with}\ (p_i \to e_i)_{i\in I} \rightsquigarrow e'$ occurs either because $e_0 \rightsquigarrow e_0'$ and $e' = \texttt{match}\ e_0'\ \texttt{with}\ (p_i \to e_i)_{i\in I}$ or because e_0 is a value and $e' = e_j\varsigma$, where $e_0/p_j = \varsigma$ and, for all $i < j$, $e_0/p_i = \Omega$. In the former case, we apply the induction hypothesis and conclude by *Tk-Match*.

In the latter case, ς is a substitution from the capture variables of p_j to values, and we know by Lemma A.2 that, for all $x \in \mathsf{capt}(p_j)$, $K; \Gamma \vdash_{\mathbb{K}} x\varsigma : \Gamma_j(x)$. We show that, additionally, $K; \Gamma \vdash_{\mathbb{K}} x\varsigma : \tau_x$ holds for every $\tau_x \in \mathsf{inst}_K(\mathsf{gen}_{K;\Gamma}(\Gamma_j(x)))$. Every such τ_x is equal to $\Gamma_j(x)\theta$ for a θ such that $\mathsf{dom}(\theta) \subseteq \mathsf{var}_K(\Gamma_j(x)) \setminus \mathsf{var}_K(\Gamma)$ and $K \vdash \theta : K$ (the kinding environment captured by generalization is just a subset of K). Then, $K; \Gamma \vdash_{\mathbb{K}} x\varsigma : \Gamma_j(x)\theta$ holds by Lemma A.14, since $\Gamma\theta = \Gamma$ (the substitution does not change any free variable of Γ).

From $K; \Gamma, \mathsf{gen}_{K;\Gamma}(\Gamma_j) \vdash_{\mathbb{K}} e_j : \tau$ and from the fact that we have $K; \Gamma \vdash_{\mathbb{K}} x\varsigma : \tau_x$ for all $x \in \mathsf{dom}(\Gamma_j)$ and all $\tau_x \in \mathsf{inst}_K(\mathsf{gen}_{K;\Gamma}(\Gamma_j(x)))$, we derive $K; \Gamma \vdash_{\mathbb{K}} e_j\varsigma : \tau$ by Lemma A.8. □

Corollary A.11 (Type soundness)**.** *Let e be a well-typed, closed expression, that is, such that $K; \varnothing \vdash_{\mathbb{K}} e : \tau$ holds for some τ. Then, either e diverges or it reduces to a value v such that $K; \varnothing \vdash_{\mathbb{K}} v : \tau$.*

Proof. Consequence of Theorem A.9 and Theorem A.10. □

A.3 Typing variants with set-theoretic types

A.3.1 Definition of the $\mathbb{S}$ type system

We consider a set $\mathcal{V}$ of *type variables* (ranged over by α, β, γ, …) and the sets $\mathcal{C}$, $\mathcal{L}$, and $\mathcal{B}$ of *language constants*, *tags*, and *basic types* (ranged over by c, $`tag$, and b respectively).

Definition A.22 (Types)**.** *A* type t *is a term coinductively produced by the following grammar:*

$$t ::= \alpha \mid b \mid c \mid t \to t \mid t \times t \mid `tag(t) \mid t \vee t \mid \neg t \mid \mathbb{0}$$

which satisfies two additional constraints:

- *(*regularity*) the term must have a finite number of different sub-terms;*
- *(*contractivity*) every infinite branch must contain an infinite number of occurrences of atoms (i.e., a type variable or the immediate application of a type constructor: basic, constant, arrow, product, or variant).*

We introduce the following abbreviations:

$$t_1 \wedge t_2 \stackrel{\text{def}}{=} \neg(\neg t_1 \vee \neg t_2) \qquad t_1 \setminus t_2 \stackrel{\text{def}}{=} t_1 \wedge (\neg t_2) \qquad \mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}\ .$$

Definition A.23 (Type schemes)**.** *A* type scheme s *is of the form* $\forall A.\, t$*, where* A *is a finite set* $\{\alpha_1, \ldots, \alpha_n\}$ *of type variables.*

We identify a type scheme $\forall\varnothing.\, t$ with the type t itself. Furthermore, we consider type schemes up to renaming of the variables they bind, and we disregard useless quantification.

Definition A.24 (Free variables)**.** *We write* $\mathsf{var}(t)$ *for the set of type variables occurring in a type* t*; we say they are the* free variables *of* t*, and we say that* t *is* ground *or* closed *if and only if* $\mathsf{var}(t)$ *is empty.*

We extend the definition to type schemes as $\mathsf{var}(\forall A.\, t) = \mathsf{var}(t) \setminus A$*.*

The (coinductive) definition of var can be found in Castagna et al. [5, Definition A.2].

Definition A.25 (Meaningful variables)**.** *We define the set* $\mathsf{mvar}(t)$ *of* meaningful variables *of a type* t *as*

$$\mathsf{mvar}(t) = \{\, \alpha \in \mathsf{var}(t) \mid t[{}^{\mathbb{0}}/_{\alpha}] \not\simeq t \,\}\ .$$

We extend the definition to type schemes as $\mathsf{mvar}(\forall A.\, t) = \mathsf{mvar}(t) \setminus A$*.*

Definition A.26 (Type substitutions)**.** *A* type substitution θ *is a finite mapping of type variables to types. We write* $[\, {}^{t_i}/_{\alpha_i} \mid i \in I \,]$ *for the type substitution which simultaneously replaces* α_i *with* t_i*, for each* $i \in I$*. We write* $t\theta$ *for the application of the substitution* θ *to the type* t*; application is defined coinductively by the following equations.*

$$\begin{aligned}
\alpha\theta &= \begin{cases} t' & \text{if } {}^{t'}/_{\alpha} \in \theta \\ \alpha & \text{otherwise} \end{cases} \\
b\theta &= b \\
c\theta &= c \\
(t_1 \to t_2)\theta &= (t_1\theta) \to (t_2\theta) \\
(t_1 \times t_2)\theta &= (t_1\theta) \times (t_2\theta) \\
(\text{`}tag(t))\theta &= \text{`}tag(t\theta) \\
(t_1 \vee t_2)\theta &= (t_1\theta) \vee (t_2\theta) \\
(\neg t)\theta &= \neg(t\theta) \\
\mathbb{0}\theta &= \mathbb{0}
\end{aligned}$$

We extend the var *operation to substitutions as*

$$\mathsf{var}(\theta) = \bigcup_{\alpha \in \mathsf{dom}(\theta)} \mathsf{var}(\alpha\theta)\ .$$

and we extend mvar *likewise.*

We extend application of substitutions to type schemes $\forall A.\, t$*: by renaming quantified variables, we assume* $A \cap (\mathsf{dom}(\theta) \cup \mathsf{var}(\theta)) = \varnothing$*, and we have* $(\forall A.\, t)\theta = \forall A.\, t\theta$*.*

We write $\theta_1 \cup \theta_2$ *for the union of disjoint substitutions and* $\theta_1 \circ \theta_2$ *for the composition of substitutions.*

Definition A.27 (Type environments)**.** *A* type environment Γ *is a partial mapping from expression variables to type schemes. We write type environments as* $\Gamma = \{x_1 \colon s_1, \ldots, x_n \colon s_n\}$*.*

We write Γ, Γ' *for the updating of the type environment* Γ *with the new bindings in* Γ'*. It is defined as follows.*

$$(\Gamma, \Gamma')(x) = \begin{cases} \Gamma'(x) & \text{if } x \in \mathsf{dom}(\Gamma') \\ \Gamma(x) & \text{otherwise} \end{cases}$$

We extend the var *operation to type environments as*

$$\mathsf{var}(\Gamma) = \bigcup_{s \in \mathsf{range}(\Gamma)} \mathsf{var}(s)\ ,$$

and we extend mvar *likewise.*

Definition A.28 (Generalization)**.** *We define the* generalization *of a type t with respect to the type environment Γ as the type scheme*

$$\mathsf{gen}_\Gamma(t) = \forall A.\, t$$

where $A = \mathsf{var}(t) \setminus \mathsf{mvar}(\Gamma)$.

We extend this definition to type environments which only contain types (i.e., trivial type schemes) as

$$\mathsf{gen}_\Gamma(\{\, x_i \colon t_i \mid i \in I \,\}) = \{\, x_i \colon \mathsf{gen}_\Gamma(t_i) \mid i \in I \,\}\,.$$

Definition A.29 (Instances of a type scheme)**.** *The set of* instances *of a type scheme $\forall A.\, t$ is defined as*

$$\mathsf{inst}(\forall A.\, t) = \{\, t\theta \mid \mathsf{dom}(\theta) \subseteq A \,\}\,.$$

We say that a type scheme s_1 is more general *than a type scheme s_2, and we write $s_1 \sqsubseteq s_2$, if*

$$\forall t_2 \in \mathsf{inst}(s_2).\, \exists t_1 \in \mathsf{inst}(s_1).\, t_1 \leq t_2\,.$$

We extend this notion to type environments as

$$\Gamma_1 \sqsubseteq \Gamma_2 \iff \mathsf{dom}(\Gamma_1) = \mathsf{dom}(\Gamma_2) \wedge \forall x \in \mathsf{dom}(\Gamma_1).\, \Gamma_1(x) \sqsubseteq \Gamma_2(x)\,.$$

Definition A.30 (Accepted type)**.** *The* accepted type $\lfloor p \rfloor$ *of a pattern p is defined inductively as:*

$$\begin{aligned} \lfloor \_ \rfloor = \lfloor x \rfloor &= \mathbb{1} & \lfloor c \rfloor &= c \\ \lfloor (p_1, p_2) \rfloor &= \lfloor p_1 \rfloor \times \lfloor p_2 \rfloor & \lfloor \grave{}\mathit{tag}(p) \rfloor &= \grave{}\mathit{tag}(\lfloor p \rfloor) \\ \lfloor p_1 \& p_2 \rfloor &= \lfloor p_1 \rfloor \wedge \lfloor p_2 \rfloor & \lfloor p_1 | p_2 \rfloor &= \lfloor p_1 \rfloor \vee \lfloor p_2 \rfloor\,. \end{aligned}$$

The projection operators π_1 and π_2 for product types are defined by Castagna et al. [5, Appendix C.2.1]. We do not repeat the definition, but we state below the properties we need in the proofs. The projection operators for variant types correspond to π_2 if we encode variant types as pairs; we therefore rephrase the same properties for them.

Property A.31 (Projections of product types)**.** *There exist two functions π_1 and π_2 which, given a type $t \leq \mathbb{1} \times \mathbb{1}$, yield types $\pi_1(t)$ and $\pi_2(t)$ such that:*

- $t \leq \pi_1(t) \times \pi_2(t)$*;*
- *if $t \leq t_1 \times t_2$, then $\pi_i(t) \leq t_i$;*
- *if $t \leq t' \leq \mathbb{1} \times \mathbb{1}$, then $\pi_i(t) \leq \pi_i(t')$;*
- *for all type substitutions θ, $\pi_i(t\theta) \leq \pi_i(t)\theta$.*

Property A.32 (Projections of variant arguments)**.** *For every tag $\grave{}\mathit{tag}$ there exists a function $\pi_{\grave{}\mathit{tag}}$ which, given a type $t \leq \grave{}\mathit{tag}(\mathbb{1})$, yields a type $\pi_{\grave{}\mathit{tag}}(t)$ such that:*

- $t \leq \grave{}\mathit{tag}(\pi_{\grave{}\mathit{tag}}(t))$*;*
- *if $t \leq \grave{}\mathit{tag}(t')$, then $\pi_{\grave{}\mathit{tag}}(t) \leq t'$;*
- *if $t \leq t' \leq \grave{}\mathit{tag}(\mathbb{1})$, then $\pi_{\grave{}\mathit{tag}}(t) \leq \pi_{\grave{}\mathit{tag}}(t')$;*
- *for all type substitutions θ, $\pi_{\grave{}\mathit{tag}}(t\theta) \leq \pi_{\grave{}\mathit{tag}}(t)\theta$.*

Definition A.33 (Pattern environment generation)**.** *Given a pattern p and a type $t \leq \lfloor p \rfloor$, the type environment $t//p$ generated by pattern matching is defined inductively as:*

$$\begin{aligned} t//\_ &= \varnothing \\ t//x &= \{x \colon t\} \\ t//c &= \varnothing \\ t//(p_1, p_2) &= \pi_1(t)//p_1 \;\cup\; \pi_2(t)//p_2 \\ t//\grave{}\mathit{tag}(p) &= \pi_{\grave{}\mathit{tag}}(t)//p \\ t//p_1 \& p_2 &= t//p_1 \;\cup\; t//p_2 \\ t//p_1 | p_2 &= (t \wedge \lfloor p_1 \rfloor)//p_1 \;\curlyvee\; (t \setminus \lfloor p_1 \rfloor)//p_2\,, \end{aligned}$$

where $(\Gamma \curlyvee \Gamma')(x) = \Gamma(x) \vee \Gamma'(x)$.

Definition A.34 (Typing relation)**.** *The typing relation $\Gamma \vdash_{\mathbb{S}} e \colon t$ (e* is given type t in the type environment Γ*) is defined by the rules in Figure 10.*

A.3.2 Properties of the $\mathbb{S}$ type system

Lemma A.12 (Generation for values)**.** *Let v be a value. Then:*

- *if $\Gamma \vdash_{\mathbb{S}} v \colon c$, then $v = c$;*
- *if $\Gamma \vdash_{\mathbb{S}} v \colon b$, then $v = c$ for some c such that $b_c \leq b$;*
- *if $\Gamma \vdash_{\mathbb{S}} v \colon t_1 \to t_2$, then v is of the form $\lambda x.\, e$ and $\Gamma, \{x \colon t_1\} \vdash_{\mathbb{S}} e \colon t_2$;*

$$\textit{Ts-Var}\ \frac{t \in \mathsf{inst}(\Gamma(x))}{\Gamma \vdash_{\mathbb{S}} x\colon t} \qquad \textit{Ts-Const}\ \frac{}{\Gamma \vdash_{\mathbb{S}} c\colon c} \qquad \textit{Ts-Abstr}\ \frac{\Gamma, \{x\colon t_1\} \vdash_{\mathbb{S}} e\colon t_2}{\Gamma \vdash_{\mathbb{S}} \lambda x.\, e\colon t_1 \to t_2}$$

$$\textit{Ts-Appl}\ \frac{\Gamma \vdash_{\mathbb{S}} e_1\colon t' \to t \qquad \Gamma \vdash_{\mathbb{S}} e_2\colon t'}{\Gamma \vdash_{\mathbb{S}} e_1\, e_2\colon t} \qquad \textit{Ts-Pair}\ \frac{\Gamma \vdash_{\mathbb{S}} e_1\colon t_1 \qquad \Gamma \vdash_{\mathbb{S}} e_2\colon t_2}{\Gamma \vdash_{\mathbb{S}} (e_1, e_2)\colon t_1 \times t_2}$$

$$\textit{Ts-Tag}\ \frac{\Gamma \vdash_{\mathbb{S}} e\colon t}{\Gamma \vdash_{\mathbb{S}} \grave{}\mathit{tag}(e)\colon \grave{}\mathit{tag}(t)}$$

$$\textit{Ts-Match}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathbb{S}} e_0\colon t_0 \qquad t_0 \leq \bigvee_{i\in I} \lbag p_i \rbag \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lbag p_j \rbag) \wedge \lbag p_i \rbag \\ \forall i \in I \qquad \Gamma, \mathsf{gen}_\Gamma(t_i // p_i) \vdash_{\mathbb{S}} e_i\colon t'_i\end{array}}{\Gamma \vdash_{\mathbb{S}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i\in I}\colon \bigvee_{i\in I} t'_i}$$

$$\textit{Ts-Subsum}\ \frac{\Gamma \vdash_{\mathbb{S}} e\colon t' \qquad t' \leq t}{\Gamma \vdash_{\mathbb{S}} e\colon t}$$

Figure 10. Typing relation of the $\mathbb{S}$ type system.

- *if* $\Gamma \vdash_{\mathbb{S}} v\colon t_1 \times t_2$*, then* v *is of the form* (v_1, v_2)*,* $\Gamma \vdash_{\mathbb{S}} v_1\colon t_1$*, and* $\Gamma \vdash_{\mathbb{S}} v_2\colon t_2$*;*
- *if* $\Gamma \vdash_{\mathbb{S}} v\colon \grave{}\mathit{tag}(t_1)$*, then* v *is of the form* $\grave{}\mathit{tag}(v_1)$ *and* $\Gamma \vdash_{\mathbb{S}} v_1\colon t_1$*.*

Proof. By induction on the typing derivation: values must be typed by an application of the rule corresponding to their form to appropriate premises, possibly followed by applications of *Ts-Subsum*.

The base cases are straightforward. In the inductive step, we just apply the induction hypothesis; for abstractions, the result follows from the behaviour of subtyping on arrow types. □

We state the next three lemmas without proof, as they rely on the model of types which we have not discussed. Details can be found in Frisch et al. [15] and Castagna and Xu [4], as well as in Alain Frisch's PhD thesis.[8]

Lemma A.13. *For each* $i \in I$*, let* p_i *be a pattern. If* $\Gamma \vdash_{\mathbb{S}} v\colon \bigvee_{i\in I} \lbag p_i \rbag$*, then there exists an* $i \in I$ *such that* $\Gamma \vdash_{\mathbb{S}} v\colon \lbag p_i \rbag$*.*

Lemma A.14. *Let* t *be a type. Let* t' *be a type such that either* $t' = \lbag p \rbag$ *or* $t' = \neg\lbag p \rbag$*, for some pattern* p*. If* $\Gamma \vdash_{\mathbb{S}} v\colon t$ *and* $\Gamma \vdash_{\mathbb{S}} v\colon t'$*, then* $\Gamma \vdash_{\mathbb{S}} v\colon t \wedge t'$*.*

Lemma A.15. *Let* v *be a well-typed value (i.e.,* $\varnothing \vdash_{\mathbb{S}} v\colon t$ *holds for some* t*) and* p *a pattern. Then:*

- $\varnothing \vdash_{\mathbb{S}} v\colon \lbag p \rbag$ *holds if and only if* $v/p = \varsigma$ *for some substitution* ς*;*
- $\varnothing \vdash_{\mathbb{S}} v\colon \neg\lbag p \rbag$ *holds if and only if* $v/p = \Omega$*.*

Lemma A.16. *Let* p *be a pattern and* t*,* t' *two types. If* $t \leq t' \leq \lbag p \rbag$*, then, for all* $x \in \mathsf{capt}(p)$*,* $(t//p)(x) \leq (t'//p)(x)$*.*

Proof. By structural induction on p.

Cases $p = \_$ *and* $p = c$ There is nothing to prove since $\mathsf{capt}(p) = \varnothing$.

Case $p = x$ We must prove $(t//x)(x) \leq (t'//x)(x)$, that is, $t \leq t'$, which we know by hypothesis.

Case $p = (p_1, p_2)$ Each $x \in \mathsf{capt}(p)$ is either in $\mathsf{capt}(p_1)$ or in $\mathsf{capt}(p_2)$. Assume $x \in \mathsf{capt}(p_i)$; then, $(t//p)(x) = (\pi_i(t)//p_i)(x)$ and $(t'//p)(x) = (\pi_i(t')//p_i)(x)$. Since $t \leq t'$ implies $\pi_i(t) \leq \pi_i(t')$ by Property A.31, we can apply the induction hypothesis to conclude.

Case $p = \grave{}\mathit{tag}(p)$ Analogous to the previous case, because $t \leq t'$ implies $\pi_{\grave{}\mathit{tag}}(t) \leq \pi_{\grave{}\mathit{tag}}(t')$ by Property A.32.

Case $p = p_1 \& p_2$ Each $x \in \mathsf{capt}(p)$ is either in $\mathsf{capt}(p_1)$ or in $\mathsf{capt}(p_2)$. Assume $x \in \mathsf{capt}(p_i)$; then, $(t//p)(x) = (t//p_i)(x)$ and $(t'//p)(x) = (t'//p_i)(x)$. We apply the induction hypothesis to conclude.

[8] A. Frisch. *Théorie, conception et réalisation d'un langage de programmation adapté à XML*. PhD thesis, Université Paris 7 – Denis Diderot, 2004.

Case $p = p_1|p_2$ Every $x \in \mathsf{capt}(p)$ is both in $\mathsf{capt}(p_1)$ and in $\mathsf{capt}(p_2)$. We have that $(t//p)(x) = (t \wedge \lfloor p_1 \rfloor // p_1)(x) \vee (t \setminus \lfloor p_1 \rfloor // p_2)(x)$ and likewise for t'. Since $t \wedge \lfloor p_1 \rfloor \leq t' \wedge \lfloor p_1 \rfloor$ and $t \setminus \lfloor p_1 \rfloor \leq t' \setminus \lfloor p_1 \rfloor$, we can apply the induction hypothesis to both sub-patterns to derive $(t \wedge \lfloor p_1 \rfloor // p_1)(x) \leq (t' \wedge \lfloor p_1 \rfloor // p_1)(x)$ and $(t \setminus \lfloor p_1 \rfloor // p_2)(x) \leq (t' \setminus \lfloor p_1 \rfloor // p_2)(x)$. Then we have $(t \wedge \lfloor p_1 \rfloor // p_1)(x) \vee (t \setminus \lfloor p_1 \rfloor // p_2)(x) \leq (t' \wedge \lfloor p_1 \rfloor // p_1)(x) \vee (t' \setminus \lfloor p_1 \rfloor // p_2)(x)$. □

Lemma A.17 (Correctness of environment generation)**.** *Let p be a pattern and v a value such that $\Gamma \vdash_{\mathbb{S}} v \colon t$ for some $t \leq \lfloor p \rfloor$. Then, for all $x \in \mathsf{capt}(p)$, $\Gamma \vdash_{\mathbb{S}} x(v/p) \colon (t//p)(x)$.*

Proof. By structural induction on p.

Cases $p = \_$ and $p = c$ There is nothing to prove since $\mathsf{capt}(p) = \varnothing$.

Case $p = x$ We must prove $\Gamma \vdash_{\mathbb{S}} x[^{v}/_{x}] \colon (t//x)(x)$, which is the hypothesis $\Gamma \vdash_{\mathbb{S}} v \colon t$.

Case $p = (p_1, p_2)$ We have $t \leq \mathbb{1} \times \mathbb{1}$, hence $t \leq \pi_1(t) \times \pi_2(t)$; then, since $\Gamma \vdash_{\mathbb{S}} v \colon \pi_1(t) \times \pi_2(t)$ by subsumption, we have by Lemma A.12 that $v = (v_1, v_2)$ and that $\Gamma \vdash_{\mathbb{S}} v_i \colon \pi_1(t)$ for both i. Moreover, $t \leq \lfloor (p_1, p_2) \rfloor = \lfloor p_1 \rfloor \times \lfloor p_2 \rfloor$. Hence, by Property A.31, $\pi_i(t) \leq \lfloor p_i \rfloor$ for both i.

Each $x \in \mathsf{capt}(p)$ is either in $\mathsf{capt}(p_1)$ or in $\mathsf{capt}(p_2)$. Assume $x \in \mathsf{capt}(p_i)$; then, $x(v/p) = x(v_i/p_i)$ and $(t//p)(x) = (\pi_i(t)//p_i)(x)$. We apply the induction hypothesis to conclude.

Case $p = \text{`}tag(p)$ Analogous to the previous case.

Case $p = p_1 \& p_2$ Each $x \in \mathsf{capt}(p)$ is either in $\mathsf{capt}(p_1)$ or in $\mathsf{capt}(p_2)$. Assume $x \in \mathsf{capt}(p_i)$; then, we can directly apply the induction hypothesis since $t \leq \lfloor p_1 \& p_2 \rfloor$ implies $t \leq \lfloor p_1 \rfloor$ and $t \leq \lfloor p_2 \rfloor$.

Case $p = p_1|p_2$ Either $v/p = v/p_1$ or $v/p = v/p_2$ (in which case $v/p_1 = \Omega$).

Case $v/p = v/p_1$ By Lemma A.15 we have $\Gamma \vdash_{\mathbb{S}} v \colon \lfloor p_1 \rfloor$; by Lemma A.14 we have $\Gamma \vdash_{\mathbb{S}} v \colon t \wedge \lfloor p_1 \rfloor$. Since $t \wedge \lfloor p_1 \rfloor \leq \lfloor p_1 \rfloor$, by the induction hypothesis we have, for all $x \in \mathsf{capt}(p_1) = \mathsf{capt}(p)$, $\Gamma \vdash_{\mathbb{S}} x(v/p) \colon (t \wedge \lfloor p_1 \rfloor // p_1)(x)$ and, by subsumption, $\Gamma \vdash_{\mathbb{S}} x(v/p) \colon (t \wedge \lfloor p_1 \rfloor // p_1)(x) \vee (t \setminus \lfloor p_1 \rfloor // p_2)(x)$.

Case $v/p = v/p_2$ By Lemma A.15 and Lemma A.14, we have $\Gamma \vdash_{\mathbb{S}} v \colon t \setminus \lfloor p_1 \rfloor$. Additionally, $t \setminus \lfloor p_1 \rfloor \leq \lfloor p_2 \rfloor$ holds because it is equivalent to $t \leq \lfloor p_1 \rfloor \vee \lfloor p_2 \rfloor$. Therefore by the induction hypothesis we have, for all $x \in \mathsf{capt}(p_1) = \mathsf{capt}(p)$, $\Gamma \vdash_{\mathbb{S}} x(v/p) \colon (t \setminus \lfloor p_1 \rfloor // p_2)(x)$ and, by subsumption, $\Gamma \vdash_{\mathbb{S}} x(v/p) \colon (t \wedge \lfloor p_1 \rfloor // p_1)(x) \vee (t \setminus \lfloor p_1 \rfloor // p_2)(x)$. □

Lemma A.18. *Let p be a pattern, t a type such that $t \leq \lfloor p \rfloor$, and θ a type substitution. Then, for all $x \in \mathsf{capt}(p)$, $(t\theta//p)(x) \leq ((t//p)(x))\theta$.*

Proof. By structural induction on p.

Cases $p = \_$ and $p = c$ There is nothing to prove since $\mathsf{capt}(p) = \varnothing$.

Case $p = x$ We must prove $(t\theta//x)(x) \leq (t//x)(x)\theta$, which is $t\theta \leq t\theta$.

Case $p = (p_1, p_2)$ Each $x \in \mathsf{capt}(p)$ is either in $\mathsf{capt}(p_1)$ or in $\mathsf{capt}(p_2)$. Assume $x \in \mathsf{capt}(p_i)$; then, $(t\theta//p)(x) = (\pi_i(t\theta)//p_i)(x)$ and $(t//p)(x)\theta = (\pi_i(t)//p_i)(x)\theta$.

Since $\pi_i(t\theta) \leq \pi_i(t)\theta$, by Lemma A.16 we have $(\pi_i(t\theta)//p_i)(x) \leq (\pi_i(t)\theta//p_i)(x)$. By the induction hypothesis we have $(\pi_i(t)\theta//p_i)(x) \leq (\pi_i(t)//p_i)(x)\theta$.

*Case $p = \text{`}tag(p)$* Analogous to the previous case, since $\pi_{\text{`}tag}(t\theta) \leq \pi_{\text{`}tag}(t)\theta$.

Case $p = p_1 \& p_2$ Each $x \in \mathsf{capt}(p)$ is either in $\mathsf{capt}(p_1)$ or in $\mathsf{capt}(p_2)$. Assume $x \in \mathsf{capt}(p_i)$; then, $(t\theta//p)(x) = (t\theta//p_i)(x)$ and $(t//p)(x)\theta = (t//p_i)(x)\theta$. We conclude by the induction hypothesis.

Case $p = p_1|p_2$ Every $x \in \mathsf{capt}(p)$ is both in $\mathsf{capt}(p_1)$ and in $\mathsf{capt}(p_2)$. We have $(t\theta//p)(x) = ((t \wedge \lfloor p_1 \rfloor)\theta//p_1)(x) \vee ((t \setminus \lfloor p_1 \rfloor)\theta//p_2)(x)$—pattern types are closed, so we can apply θ to them too—and $(t//p)(x)\theta = (t \wedge \lfloor p_1 \rfloor // p_1)(x)\theta \vee (t \setminus \lfloor p_1 \rfloor // p_2)(x)\theta$. We conclude by applying the induction hypothesis to both members of the union. □

Lemma A.19. *Let t_1 and t_2 be equivalent types ($t_1 \simeq t_2$). Then, $\mathsf{mvar}(t_1) = \mathsf{mvar}(t_2)$.*

Proof. Since subtyping is preserved by type substitutions, for every α we have $t_1[^{\mathbb{0}}/_{\alpha}] \simeq t_2[^{\mathbb{0}}/_{\alpha}]$. If $\alpha \in \mathsf{mvar}(t_1)$, we have $t_1[^{\mathbb{0}}/_{\alpha}] \not\simeq t_1$ by the definition of mvar. This necessarily implies $t_2[^{\mathbb{0}}/_{\alpha}] \not\simeq t_2$, otherwise we would have $t_1[^{\mathbb{0}}/_{\alpha}] \simeq t_1$ by transitivity. □

Lemma A.20. *Let t be a type and θ a type substitution such that $\mathsf{dom}(\theta) \cap \mathsf{mvar}(t) = \varnothing$. Then $t\theta \simeq t$.*

Proof. Let $t' = t[{}^{\mathbb{0}}/_{\alpha_1}, \ldots, {}^{\mathbb{0}}/_{\alpha_n}]$ where $\{\alpha_1, \ldots, \alpha_n\} = \mathsf{var}(t) \setminus \mathsf{mvar}(t)$. We have $t \simeq t'$ and $\mathsf{var}(t') = \mathsf{mvar}(t)$. Since substitutions preserve subtyping (and hence equivalence), we have also $t\theta \simeq t'\theta$. But $t'\theta = t' \simeq t$; hence, we reach the conclusion by the transitivity of equivalence. □

Lemma A.21. *Let Γ_1, Γ_2 be two type environments such that $\mathsf{mvar}(\Gamma_1) \subseteq \mathsf{mvar}(\Gamma_2)$ and t_1, t_2 two types such that $t_1 \leq t_2$. Then, $\mathsf{gen}_{\Gamma_1}(t_1) \sqsubseteq \mathsf{gen}_{\Gamma_2}(t_2)$.*

Proof. An instance of $\mathsf{gen}_{\Gamma_2}(t_2)$ is a type $t_2\theta_2$ such that $\mathsf{dom}(\theta_2) \subseteq \mathsf{var}(t_2) \setminus \mathsf{mvar}(\Gamma_2)$. Let θ_1 be the restriction of θ_2 to the variables in $\mathsf{var}(t_1) \setminus \mathsf{mvar}(\Gamma_1)$. Then, $t_1\theta_1$ is an instance of $\mathsf{gen}_{\Gamma_1}(t_1)$.

We have $t_1\theta_1 = t_1\theta_2$ because the two substitutions differ only on variables in $\mathsf{var}(t_2) \setminus \mathsf{var}(t_1)$ (which do not appear in t_1 at all) or in $\mathsf{mvar}(\Gamma_1) \setminus \mathsf{mvar}(\Gamma_2)$ (which is empty). Finally, we have $t_1\theta_2 \leq t_2\theta_2$ because subtyping is preserved by substitutions. □

Lemma A.22 (Weakening)**.** *Let Γ_1, Γ_2 be two type environments such that $\Gamma_1 \sqsubseteq \Gamma_2$ and $\mathsf{mvar}(\Gamma_1) \subseteq \mathsf{mvar}(\Gamma_2)$. If $\Gamma_2 \vdash_{\mathbb{S}} e : t$, then $\Gamma_1 \vdash_{\mathbb{S}} e : t$.*

Proof. By induction on the derivation of $\Gamma_2 \vdash_{\mathbb{S}} e : t$. We reason by cases on the last applied rule.

Case Ts-Var We have

$$\Gamma_2 \vdash_{\mathbb{S}} x : t \qquad t \in \mathsf{inst}(\Gamma_2(x))$$

and hence, since $\Gamma_1 \sqsubseteq \Gamma_2$, there exists a $t' \in \mathsf{inst}(\Gamma_1(x))$ such that $t' \leq t$. We apply *Ts-Var* to derive $\Gamma_1 \vdash_{\mathbb{S}} x : t'$ and *Ts-Subsum* to conclude.

Case Ts-Const Straightforward.

Case Ts-Abstr We have

$$\Gamma_2 \vdash_{\mathbb{S}} \lambda x.\, e_1 : t_1 \to t_2 \qquad \Gamma_2, \{x : t_1\} \vdash_{\mathbb{S}} e_1 : t_2\,.$$

Since $\Gamma_1 \sqsubseteq \Gamma_2$, we have $\Gamma_1, \{x : t_1\} \sqsubseteq \Gamma_2, \{x : t_1\}$; since $\mathsf{mvar}(\Gamma_1) \subseteq \mathsf{mvar}(\Gamma_2)$, we have $\mathsf{mvar}(\Gamma_1, \{x : t_1\}) \subseteq \mathsf{mvar}(\Gamma_2, \{x : t_1\})$. We derive $\Gamma_1, \{x : t_1\} \vdash_{\mathbb{S}} e_1 : t_2$ by the induction hypothesis and apply *Ts-Abstr* to conclude.

Cases Ts-Appl, Ts-Pair, Ts-Tag, and Ts-Subsum Straightforward application of the induction hypothesis.

Case Tk-Match We have

$$\Gamma_2 \vdash_{\mathbb{S}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I} : t$$
$$\Gamma_2 \vdash_{\mathbb{S}} e_0 : t_0 \qquad t_0 \leq \bigvee_{i \in I} \lbag p_i \rbag \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lbag p_j \rbag) \wedge \lbag p_i \rbag$$
$$\forall i \in I.\ \ \Gamma_2, \mathsf{gen}_{\Gamma_2}(t_i /\!/ p_i) \vdash_{\mathbb{S}} e_i : t_i' \qquad t = \bigvee_{i \in I} t_i'\ .$$

By the induction hypothesis, we derive $\Gamma_1 \vdash_{\mathbb{S}} e_0 : t_0$.

For any branch, note that $\mathsf{mvar}(\Gamma_1) \subseteq \mathsf{mvar}(\Gamma_2)$ implies $\mathsf{gen}_{\Gamma_1}(t) \sqsubseteq \mathsf{gen}_{\Gamma_2}(t)$ for any t by Lemma A.21. Hence, we have $\Gamma_1, \mathsf{gen}_{\Gamma_1}(t_i /\!/ p_i) \sqsubseteq \Gamma_2, \mathsf{gen}_{\Gamma_2}(t_i /\!/ p_i)$. Additionally, since $\mathsf{mvar}(\mathsf{gen}_{\Gamma_1}(t_i /\!/ p_i)) \subseteq \mathsf{mvar}(\Gamma_1) \subseteq \mathsf{mvar}(\Gamma_2)$, we have $\mathsf{mvar}(\Gamma_1, \mathsf{gen}_{\Gamma_1}(t_i /\!/ p_i)) \subseteq \mathsf{mvar}(\Gamma_2, \mathsf{gen}_{\Gamma_2}(t_i /\!/ p_i))$.

Hence we may apply the induction hypothesis for all i to derive $\Gamma_1, \mathsf{gen}_{\Gamma_1}(t_i /\!/ p_i) \vdash_{\mathbb{S}} e_i : t_i'$ and then apply *Ts-Match* to conclude. □

Lemma A.23 (Stability of typing under type substitutions)**.** *Let θ be a type substitution. If $\Gamma \vdash_{\mathbb{S}} e : t$, then $\Gamma\theta \vdash_{\mathbb{S}} e : t\theta$.*

Proof. By induction on the derivation of $\Gamma \vdash_{\mathbb{S}} e : t$. We reason by cases on the last applied rule.

Case Ts-Var We have

$$\Gamma \vdash_{\mathbb{S}} x : t \qquad t \in \mathsf{inst}(\Gamma(x)) \qquad \Gamma(x) = \forall A.\, t_x \qquad t = t_x\theta_x \qquad \mathsf{dom}(\theta_x) \subseteq A$$

and must show $\Gamma\theta \vdash_{\mathbb{S}} x : t\theta$.

By α-renaming we assume $A \cap (\mathsf{dom}(\theta) \cup \mathsf{var}(\theta)) = \varnothing$. Under this assumption, $(\Gamma\theta)(x) = \forall A.\, t_x\theta$. We must show that $t\theta = t_x\theta\theta_x'$ for a substitution θ_x' such that $\mathsf{dom}(\theta_x') \subseteq A$.

Let $\theta_x' = [\,{}^{\alpha\theta_x\theta}/_{\alpha} \mid \alpha \in A\,]$. We show that $t\theta\theta_x' = t_x\theta_x\theta = t\theta$, by showing that, for every α, $\alpha\theta\theta_x' = \alpha\theta_x\theta$. If $\alpha \in A$, then $\alpha\theta\theta_x' = \alpha\theta_x' = \alpha\theta_x\theta$ (θ is not defined on the variables in A). If $\alpha \notin A$, then $\alpha\theta\theta_x' = \alpha\theta$ (θ never produces any variable in A) and $\alpha\theta_x\theta = \alpha\theta$ as $\alpha \notin \mathsf{dom}(\theta_x)$.

Case Ts-Const Straightforward.

Case Ts-Abstr We have

$$\Gamma \vdash_{\mathbb{S}} \lambda x.\, e_1 : t_1 \to t_2 \qquad \Gamma, \{x : t_1\} \vdash_{\mathbb{S}} e_1 : t_2 \,.$$

By the induction hypothesis we have $\Gamma\theta, \{x : t_1\theta\} \vdash_{\mathbb{S}} e_1 : t_2\theta$. Then by *Ts-Abstr* we derive $\Gamma\theta \vdash_{\mathbb{S}} \lambda x.\, e_1 : (t_1\theta) \to (t_2\theta)$, which is $\Gamma\theta \vdash_{\mathbb{S}} \lambda x.\, e_1 : (t_1 \to t_2)\theta$.

Cases Ts-Appl, Ts-Pair, and Ts-Tag Straightforward application of the induction hypothesis.

Case Ts-Match We have

$$\Gamma \vdash_{\mathbb{S}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I} : t$$
$$\Gamma \vdash_{\mathbb{S}} e_0 : t_0 \qquad t_0 \leq \bigvee_{i \in I} \lbag p_i \rbag \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lbag p_j \rbag) \wedge \lbag p_i \rbag$$
$$\forall i \in I.\ \ \Gamma, \mathsf{gen}_\Gamma(t_i // p_i) \vdash_{\mathbb{S}} e_i : t'_i \qquad t = \bigvee_{i \in I} t'_i$$

and must show $\Gamma\theta \vdash_{\mathbb{S}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I} : t\theta$.

We prove it by establishing, for some types $\hat{t_0}$ and $\hat{t_i}$, $\hat{t'_i}$ for each i, that

$$\Gamma\theta \vdash_{\mathbb{S}} e_0 : \hat{t_0} \qquad \hat{t_0} \leq \bigvee_{i \in I} \lbag p_i \rbag \qquad \hat{t_i} = (\hat{t_0} \setminus \bigvee_{j<i} \lbag p_j \rbag) \wedge \lbag p_i \rbag$$
$$\forall i \in I.\ \ \Gamma\theta, \mathsf{gen}_{\Gamma\theta}(\hat{t_i} // p_i) \vdash_{\mathbb{S}} e_i : \hat{t'_i} \qquad \bigvee_{i \in I} \hat{t'_i} \leq t\theta \ .$$

Let $A = \{\alpha_1, \ldots, \alpha_n\} = \mathsf{var}(t_0) \setminus \mathsf{mvar}(\Gamma)$. Let $B = \{\beta_1, \ldots, \beta_n\}$ be a set of type variables such that $B \cap (\mathsf{dom}(\theta) \cup \mathsf{var}(\theta) \cup \mathsf{var}(\Gamma)) = \varnothing$. Let $\theta_0 = [{}^{\beta_1}/_{\alpha_1}, \ldots, {}^{\beta_n}/_{\alpha_n}]$ and $\theta' = \theta \circ \theta_0$.

By the induction hypothesis, using θ', we derive $\Gamma\theta' \vdash_{\mathbb{S}} e_0 : t_0\theta'$. From it, we derive $\Gamma\theta \vdash_{\mathbb{S}} e_0 : t_0\theta'$ by weakening (Lemma A.22); we prove the required premises below. We take $\hat{t_0} = t_0\theta'$: note that the exhaustiveness condition is satisfied because substitutions preserve subtyping (and all accepted types of patterns are closed). We have $\hat{t_i} = t_i\theta'$ for all i.

For all branches, we have $\Gamma, \mathsf{gen}_\Gamma(t_i // p_i) \vdash_{\mathbb{S}} e_i : t'_i$ and, by the induction hypothesis using θ, we can derive $\Gamma\theta, (\mathsf{gen}_\Gamma(t_i // p_i))\theta \vdash_{\mathbb{S}} e_i : t'_i\theta$.

We apply Lemma A.22 to derive $\Gamma\theta, \mathsf{gen}_{\Gamma\theta}(t_i\theta' // p_i) \vdash_{\mathbb{S}} e_i : t'_i\theta$ (we prove the required premises below). We take $\hat{t'_i} = t'_i\theta$.

Proof of $\Gamma\theta \vdash_{\mathbb{S}} e_0 : t_0\theta'$ *from* $\Gamma\theta' \vdash_{\mathbb{S}} e_0 : t_0\theta'$ We prove this by Lemma A.22, which requires us to show $\Gamma\theta \sqsubseteq \Gamma\theta'$ and $\mathsf{mvar}(\Gamma\theta) \subseteq \mathsf{mvar}(\Gamma\theta')$. We show this by showing, for every $(x : \forall A_x.\, t_x) \in \Gamma$—assume by α-renaming $A_x \cap (\mathsf{dom}(\theta) \cup \mathsf{var}(\theta) \cup A \cup B) = \varnothing$—, $t_x\theta \simeq t_x\theta'$, which implies both $\forall A_x.\, t_x\theta \sqsubseteq \forall A_x.\, t_x\theta'$ and $\mathsf{mvar}(\forall A_x.\, t_x\theta) \subseteq \mathsf{mvar}(\forall A_x.\, t_x\theta')$ (by Lemma A.21 and Lemma A.19).

We have $t_x\theta_0 \simeq t_x$ by Lemma A.20: $\mathsf{dom}(\theta_0) \cap \mathsf{mvar}(t_x) = \varnothing$ because every $\alpha \in \mathsf{mvar}(t_x)$ is either in A_x or $\mathsf{mvar}(\Gamma)$, and in both cases this means it cannot be in $\mathsf{dom}(\theta_0)$. Hence—since substitutions preserve subtyping—we have also $t_x\theta' = t_x\theta_0\theta \simeq t_x\theta$.

Proof of $\Gamma\theta, \mathsf{gen}_{\Gamma\theta}(t_i\theta' // p_i) \vdash_{\mathbb{S}} e_i : t'_i\theta$ *from* $\Gamma\theta, (\mathsf{gen}_\Gamma(t_i // p_i))\theta \vdash_{\mathbb{S}} e_i : t'_i\theta$ To apply Lemma A.22, we must show

$$\mathsf{gen}_{\Gamma\theta}(t_i\theta' // p_i) \sqsubseteq (\mathsf{gen}_\Gamma(t_i // p_i))\theta$$
$$\mathsf{mvar}(\Gamma\theta, \mathsf{gen}_{\Gamma\theta}(t_i\theta' // p_i)) \subseteq \mathsf{mvar}(\Gamma\theta, (\mathsf{gen}_\Gamma(t_i // p_i))\theta) \,.$$

The latter holds because every variable in $\mathsf{mvar}(\Gamma\theta, \mathsf{gen}_{\Gamma\theta}(t_i\theta' // p_i))$ is in $\mathsf{mvar}(\Gamma\theta)$. For the former, we prove that, for every $x \in \mathsf{capt}(p_i)$,

$$\mathsf{gen}_{\Gamma\theta}((t_i\theta' // p_i)(x)) \sqsubseteq (\mathsf{gen}_\Gamma((t_i // p_i)(x)))\theta \,.$$

Let

$$t'_x = (t_i\theta' // p_i)(x) \qquad t_x = (t_i // p_i)(x)\,;$$

the statement becomes

$$\mathsf{gen}_{\Gamma\theta}(t'_x) \sqsubseteq (\mathsf{gen}_\Gamma(t_x))\theta \,.$$

We have $\mathsf{gen}_\Gamma(t_x) = \forall A_x.\, t_x$, where $A_x = \mathsf{var}(t_x) \setminus \mathsf{mvar}(\Gamma)$. Since $\mathsf{var}(t_x) \subseteq \mathsf{var}(t_i) = \mathsf{var}(t_0)$, $A_x \subseteq A$. Let $J = \{\, j \mid \alpha_j \in A_x \,\}$; thus $J \subseteq \{1, \ldots, n\}$ and $A_x = A|_J = \{\, \alpha_j \mid j \in J \,\}$. Let $B|_J = \{\, \beta_j \mid j \in J \,\}$. We have $\mathsf{gen}_\Gamma(t_x) = \forall B|_J.\, t_x\theta_0$ by α-renaming (we are substituting also the α_i such that $i \notin J$, but it makes no difference as they not in t_x). Thus—since $B \cap (\mathsf{dom}(\theta) \cup \mathsf{var}(\theta)) = \varnothing$—we have

$$(\mathsf{gen}_\Gamma(t_x))\theta = \forall B|_J.\, t_x\theta_0\theta = \forall B|_J.\, t_x\theta' \,.$$

The instances of this type scheme are all types $t_x\theta'\theta_x$, with $\mathsf{dom}(\theta_x) \subseteq B|_J$. Given such a type, we must construct an instance of $\mathsf{gen}_{\Gamma\theta}(t'_x)$ that is a subtype of it. Let θ'_x be

the restriction of θ_x to variables in $\mathsf{var}(t'_x) \setminus \mathsf{mvar}(\Gamma\theta)$. Then $t'_x\theta'_x$ is a valid instance of $\mathsf{gen}_{\Gamma\theta}(t'_x)$. We prove $t'_x\theta'_x \leq t_x\theta'\theta_x$.

We have $t'_x\theta'_x = t'_x\theta_x$: the two substitutions differ only on variables in $B|_J \setminus \mathsf{var}(t'_x)$ (variables which do not appear in the type at all) and on variables in $B|_J \cap \mathsf{mvar}(\Gamma\theta)$ (which is empty, because B was chosen fresh). By Lemma A.16, we have $t'_x = (t_i\theta'/\!/p_i)(x) \leq (t_i/\!/p_i)(x)\theta'$: hence, $t'_x\theta_x = (t_i\theta'/\!/p_i)(x)\theta_x \leq (t_i/\!/p_i)(x)\theta'\theta_x = t_x\theta'\theta_x$.

Case Ts-Subsum The conclusion follows from the induction hypothesis since substitutions preserve subtyping. □

Corollary A.24. *Let Γ be a type environment and θ a type substitution such that $\mathsf{dom}(\theta) \cap \mathsf{mvar}(\Gamma) = \varnothing$. If $\Gamma \vdash_{\mathbb{S}} e\colon t$, then $\Gamma \vdash_{\mathbb{S}} e\colon t\theta$.*

Proof. From $\Gamma \vdash_{\mathbb{S}} e\colon t$ we derive $\Gamma\theta \vdash_{\mathbb{S}} e\colon t\theta$ by Lemma A.23. Then, we show $\Gamma \sqsubseteq \Gamma\theta$ and $\mathsf{mvar}(\Gamma) \subseteq \mathsf{mvar}(\Gamma\theta)$, which allow us to apply Lemma A.22 to derive $\Gamma \vdash_{\mathbb{S}} e\colon t\theta$.

To show the two conditions above, we show that, for every $(x\colon \forall A.\, t) \in \Gamma$—assume by α-renaming $A \cap (\mathsf{dom}(\theta) \cup \mathsf{var}(\theta)) = \varnothing$—, $\forall A.\, t \sqsubseteq \forall A.\, t\theta$ and $\mathsf{mvar}(\forall A.\, t) \subseteq \mathsf{mvar}(\forall A.\, t\theta)$.

We show $t \simeq t\theta$, which implies both (by Lemma A.21 and Lemma A.19). The equivalence holds by Lemma A.20: $\mathsf{dom}(\theta) \cap \mathsf{mvar}(t) = \varnothing$ because every $\alpha \in \mathsf{mvar}(t)$ is either in A or $\mathsf{mvar}(\Gamma)$, and in both cases this means it cannot be in $\mathsf{dom}(\theta)$. □

Lemma A.25 (Expression substitution)**.** *Let $x_1, \ldots, x_n$ be distinct variables and $v_1, \ldots, v_n$ values. Let $\Gamma' = \{x_1\colon s_1, \ldots, x_n\colon s_n\}$ and $\varsigma = [{}^{v_1}/_{x_1}, \ldots, {}^{v_n}/_{x_n}]$.*

If $\Gamma, \Gamma' \vdash_{\mathbb{S}} e\colon t$ and, for all $k \in \{1, \ldots, n\}$ and for all $t_k \in \mathsf{inst}(s_k)$, $\Gamma \vdash_{\mathbb{S}} v_k\colon t_k$, then $\Gamma \vdash_{\mathbb{S}} e\varsigma\colon t$.

Proof. By induction on the derivation of $\Gamma, \Gamma' \vdash_{\mathbb{S}} e\colon t$. We reason by cases on the last applied rule.

Case Ts-Var We have

$$\Gamma, \Gamma' \vdash_{\mathbb{S}} x\colon t \qquad t \in \mathsf{inst}((\Gamma, \Gamma')(x))\,.$$

Either $x = x_k$ for some k or not. In the latter case, $x\varsigma = x$, $x \notin \mathsf{dom}(\Gamma')$ and hence $(\Gamma, \Gamma')(x) = \Gamma(x)$. Then, since $t \in \mathsf{inst}((\Gamma, \Gamma')(x))$, $t \in \mathsf{inst}(\Gamma(x))$ and we can apply *Ts-Var*.

If $x = x_k$, then $(\Gamma, \Gamma')(x) = \Gamma'(x) = s_k$. We must then prove $\Gamma \vdash_{\mathbb{S}} v_k\colon t$, which we know by hypothesis since $t \in \mathsf{inst}(s_k)$.

Case Ts-Const Straightforward.

Case Ts-Abstr We have

$$\Gamma, \Gamma' \vdash_{\mathbb{S}} \lambda x.\, e_1\colon t_1 \to t_2 \qquad \Gamma, \Gamma', \{x\colon t_1\} \vdash_{\mathbb{S}} e_1\colon t_2\,.$$

By α-renaming we can assume $x \notin \mathsf{dom}(\Gamma, \Gamma')$; then $(\lambda x.\, e_1)\varsigma = \lambda x.\, (e_1\varsigma)$ and $\Gamma, \Gamma', \{x\colon t_1\} = \Gamma, \{x\colon t_1\}, \Gamma'$. Therefore we have $\Gamma, \{x\colon t_1\}, \Gamma' \vdash_{\mathbb{S}} e_1\colon t_2$ and hence $\Gamma, \{x\colon t_1\} \vdash_{\mathbb{S}} e_1\varsigma\colon t_2$ by the induction hypothesis. We apply *Ts-Abstr* to conclude.

Cases Ts-Appl, Ts-Pair, Ts-Tag, and Ts-Subsum Straightforward application of the induction hypothesis.

Case Ts-Match We have

$$\Gamma, \Gamma' \vdash_{\mathbb{S}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i\in I}\colon t$$
$$\Gamma, \Gamma' \vdash_{\mathbb{S}} e_0\colon t_0 \qquad t_0 \leq \bigvee_{i\in I} \lfloor p_i \rfloor \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lfloor p_j \rfloor) \wedge \lfloor p_i \rfloor$$
$$\forall i \in I.\ \Gamma, \Gamma', \mathsf{gen}_{\Gamma,\Gamma'}(t_i/\!/p_i) \vdash_{\mathbb{S}} e_i\colon t'_i \qquad t = \bigvee_{i\in I} t'_i\ .$$

We assume by α-renaming that no capture variable of any pattern is in the domain of Γ or Γ'. Then, $(\mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i\in I})\varsigma = \mathtt{match}\ e_0\varsigma\ \mathtt{with}\ (p_i \to e_i\varsigma)_{i\in I}$ and $\Gamma, \Gamma', \mathsf{gen}_{\Gamma,\Gamma'}(t_i/\!/p_i) = \Gamma, \mathsf{gen}_{\Gamma,\Gamma'}(t_i/\!/p_i), \Gamma'$ for any i.

By the induction hypothesis, we derive $\Gamma \vdash_{\mathbb{S}} e_0\varsigma\colon t_0$ and $\Gamma, \mathsf{gen}_{\Gamma,\Gamma'}(t_i/\!/p_i) \vdash_{\mathbb{S}} e_i\varsigma\colon t'_i$ for all i. From the latter, we prove $\Gamma, \mathsf{gen}_{\Gamma}(t_i/\!/p_i) \vdash_{\mathbb{S}} e_i\varsigma\colon t'_i$ by weakening (Lemma A.22): we have $\mathsf{gen}_{\Gamma}(t_i/\!/p_i) \sqsubseteq \mathsf{gen}_{\Gamma,\Gamma'}(t_i/\!/p_i)$ by Lemma A.21—since $\mathsf{mvar}(\Gamma) \subseteq \mathsf{mvar}(\Gamma, \Gamma')$ – and clearly we have $\mathsf{mvar}(\Gamma, \mathsf{gen}_{\Gamma}(t_i/\!/p_i)) \subseteq \mathsf{mvar}(\Gamma, \mathsf{gen}_{\Gamma,\Gamma'}(t_i/\!/p_i))$ since $\mathsf{mvar}(\mathsf{gen}_{\Gamma}(t_i/\!/p_i)) \subseteq \mathsf{mvar}(\Gamma)$. □

Theorem A.26 (Progress)**.** *Let e be a well-typed, closed expression (i.e., $\varnothing \vdash_{\mathbb{S}} e\colon t$ holds for some t). Then, either e is a value or there exists an expression e' such that $e \rightsquigarrow e'$.*

Proof. By hypothesis we have $\varnothing \vdash_{\mathbb{S}} e\colon t$. The proof is by induction on its derivation; we reason by cases on the last applied rule.

Case Ts-Var This case does not occur because variables are not closed.

Case Ts-Const In this case e is a constant c and therefore a value.

Case Ts-Abstr In this case e is an abstraction $\lambda x.\, e_1$. Since it is also closed, it is a value.

Case Ts-Appl We have

$$\varnothing \vdash_{\mathsf{s}} e_1\, e_2 \colon t \qquad \varnothing \vdash_{\mathsf{s}} e_1 \colon t' \to t \qquad \varnothing \vdash_{\mathsf{s}} e_2 \colon t' \,.$$

By the induction hypothesis, each of e_1 and e_2 either is a value or may reduce. If $e_1 \rightsquigarrow e_1'$, then $e_1\, e_2 \rightsquigarrow e_1'\, e_2$. If e_1 is a value and $e_2 \rightsquigarrow e_2'$, then $e_1\, e_2 \rightsquigarrow e_1\, e_2'$.

If both are values then, by Lemma A.12, e_1 has the form $\lambda x.\, e_3$ for some e_3. Then, we can apply *R-Appl* and $e_1\, e_2 \rightsquigarrow e_3[{}^{e_2}\!/_{x}]$.

Case Ts-Pair We have

$$\varnothing \vdash_{\mathsf{s}} (e_1, e_2) \colon t_1 \times t_2 \qquad \varnothing \vdash_{\mathsf{s}} e_1 \colon t_1 \qquad \varnothing \vdash_{\mathsf{s}} e_2 \colon t_2 \,.$$

By the induction hypothesis, each of e_1 and e_2 either is a value or may reduce. If $e_1 \rightsquigarrow e_1'$, then $(e_1, e_2) \rightsquigarrow (e_1', e_2)$. If e_1 is a value and $e_2 \rightsquigarrow e_2'$, then $(e_1, e_2) \rightsquigarrow (e_1, e_2')$. If both are values, then (e_1, e_2) is also a value.

Case Ts-Tag We have

$$\varnothing \vdash_{\mathsf{s}} \text{`}\mathit{tag}(e_1) \colon \text{`}\mathit{tag}(t_1) \qquad \varnothing \vdash_{\mathsf{s}} e_1 \colon t_1 \,.$$

Analogously to the previous case, by the induction hypothesis we have that either e_1 is a value or $e_1 \rightsquigarrow e_1'$. In the former case, $\text{`}\mathit{tag}(e_1)$ is a value as well. In the latter, we have $\text{`}\mathit{tag}(e_1) \rightsquigarrow \text{`}\mathit{tag}(e_1')$.

Case Ts-Match We have

$$\varnothing \vdash_{\mathsf{s}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I} \colon t \qquad \varnothing \vdash_{\mathsf{s}} e_0 \colon t_0 \qquad t_0 \leq \textstyle\bigvee_{i \in I} \jmath p_i \jsl \,.$$

By the inductive hypothesis, either e_0 is a value or it may reduce. In the latter case, if $e_0 \rightsquigarrow e_0'$, then $\mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I} \rightsquigarrow \mathtt{match}\ e_0'\ \mathtt{with}\ (p_i \to e_i)_{i \in I}$.

If e_0 is a value, on the other hand, the expression may reduce by application of *R-Match*. Since $t_0 \leq \bigvee_{i \in I} \jmath p_i \jsl$, $\varnothing \vdash_{\mathsf{s}} e_0 \colon \bigvee_{i \in I} \jmath p_i \jsl$ holds by subsumption. Hence, since e_0 is a value, $\varnothing \vdash_{\mathsf{s}} e_0 \colon \jmath p_i \jsl$ holds for at least one i (by Lemma A.13); for each such i we have $e_0/p_i = \varsigma_i$ (by Lemma A.15). Let j be the least of these i; then $\mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I} \rightsquigarrow e_j \varsigma_j$.

Case Ts-Subsum Straightforward application of the induction hypothesis. □

Theorem A.27 (Subject reduction)**.** *Let e be an expression and t a type such that $\Gamma \vdash_{\mathsf{s}} e \colon t$. If $e \rightsquigarrow e'$, then $\Gamma \vdash_{\mathsf{s}} e' \colon t$.*

Proof. By induction on the derivation of $\Gamma \vdash_{\mathsf{s}} e \colon t$. We reason by cases on the last applied rule.

Cases Ts-Var, Ts-Const, and Ts-Abstr These cases do not occur: variables, constants, and abstractions never reduce.

Case Ts-Appl We have

$$\Gamma \vdash_{\mathsf{s}} e_1\, e_2 \colon t \qquad \Gamma \vdash_{\mathsf{s}} e_1 \colon t' \to t \qquad \Gamma \vdash_{\mathsf{s}} e_2 \colon t' \,.$$

$e_1\, e_2 \rightsquigarrow e'$ occurs in any of three ways: (i) $e_1 \rightsquigarrow e_1'$ and $e' = e_1'\, e_2$; (ii) e_1 is a value, $e_2 \rightsquigarrow e_2'$ and $e' = e_1\, e_2'$; (iii) both e_1 and e_2 are values, e_1 is of the form $\lambda x.\, e_3$, and $e' = e_3[{}^{e_2}\!/_{x}]$.

In the first case, we derive by the induction hypothesis that $\Gamma \vdash_{\mathsf{s}} e_1' \colon t' \to t$ and conclude by applying *Ts-Appl* again. The second case is analogous.

In the third case, we know by Lemma A.12 that $\Gamma, \{x \colon t'\} \vdash_{\mathsf{s}} e_3 \colon t$. We also know that e_2 is a value such that $\Gamma \vdash_{\mathsf{s}} e_2 \colon t'$. Then, by Lemma A.25, $\Gamma \vdash_{\mathsf{s}} e_3[{}^{e_2}\!/_{x}] \colon t$.

Case Ts-Pair We have

$$\Gamma \vdash_{\mathsf{s}} (e_1, e_2) \colon t_1 \times t_2 \qquad \Gamma \vdash_{\mathsf{s}} e_1 \colon t_1 \qquad \Gamma \vdash_{\mathsf{s}} e_2 \colon t_2 \,.$$

$(e_1, e_2) \rightsquigarrow e'$ occurs either because $e_1 \rightsquigarrow e_1'$ and $e' = (e_1', e_2)$, or because e_1 is a value, $e_2 \rightsquigarrow e_2'$, and $e' = (e_1, e_2')$. In either case, the induction hypothesis allows us to derive that the type of the component that reduces is preserved; therefore, we can apply *Ts-Pair* again to conclude.

Case Ts-Tag Analogously to the previous case, a variant expression only reduces if its argument does, so we apply the induction hypothesis and *Ts-Tag* to conclude.

$$\textit{R-Fix}\ \frac{}{\Upsilon\,(\lambda x.\,e)\ \rightsquigarrow\ e[{}^{\Upsilon\,(\lambda x.\,e)}/_{x}]}$$

$$\textit{Tk-Fix}\ \frac{K;\Gamma \vdash_{\Bbbk} e\colon \tau \to \tau}{K;\Gamma \vdash_{\Bbbk} \Upsilon\, e\colon \tau} \qquad \textit{Ts-Fix}\ \frac{\Gamma \vdash_{\mathbb{S}} e\colon t \to t}{\Gamma \vdash_{\mathbb{S}} \Upsilon\, e\colon t}$$

Figure 11. Rules for the fixed-point combinator.

Case Ts-Match We have

$$\Gamma \vdash_{\mathbb{S}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i\in I}\colon t$$
$$\Gamma \vdash_{\mathbb{S}} e_0\colon t_0 \qquad t_0 \leq \bigvee_{i\in I} \wr p_i \int \qquad t_i = (t_0 \setminus \bigvee_{j<i} \wr p_j \int) \wedge \wr p_i \int$$
$$\forall i \in I.\ \Gamma, \mathsf{gen}_{\Gamma}(t_i /\!/ p_i) \vdash_{\mathbb{S}} e_i\colon t_i' \qquad t = \bigvee_{i\in I} t_i'\ .$$

The reduction $\mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i\in I} \rightsquigarrow e'$ occurs either because $e_0 \rightsquigarrow e_0'$ and $e' = \mathtt{match}\ e_0'\ \mathtt{with}\ (p_i \to e_i)_{i\in I}$ or because e_0 is a value and $e' = e_j\varsigma$, where $e_0/p_j = \varsigma$ and, for all $i < j$, $e_0/p_i = \Omega$. In the former case, we apply the induction hypothesis and conclude by *Ts-Match*.

In the latter case, ς is a substitution from the capture variables of p_j to values. We can derive

$$\Gamma \vdash_{\mathbb{S}} e_0\colon \wr p_j \int \qquad \forall i < j.\ \Gamma \vdash_{\mathbb{S}} e_0\colon \neg \wr p_i \int$$

by Lemma A.15 and thence $\Gamma \vdash_{\mathbb{S}} e_0\colon t_j$ by Lemma A.14. Therefore, by Lemma A.17, we have that, for all $x \in \mathsf{capt}(p_j)$, $\Gamma \vdash_{\mathbb{S}} x\varsigma\colon (t_j /\!/ p_j)(x)$. Let $\Gamma' = t_j /\!/ p_j$.

We show that, additionally, $\Gamma \vdash_{\mathbb{S}} x\varsigma\colon t_x$ holds for every $t_x \in \mathsf{inst}(\mathsf{gen}_{\Gamma}(\Gamma'(x)))$. Every such t_x is equal to $\Gamma'(x)\theta$ for a θ such that $\mathsf{dom}(\theta) \subseteq \mathsf{var}(\Gamma'(x)) \setminus \mathsf{mvar}(\Gamma)$. Then, $\Gamma \vdash_{\mathbb{S}} x\varsigma\colon \Gamma'(x)\theta$ holds by Corollary A.24, since $\mathsf{dom}(\theta) \cap \mathsf{mvar}(\Gamma) = \varnothing$ (the substitution does not change any meaningful variable of Γ).

From $\Gamma, \mathsf{gen}_{\Gamma}(\Gamma') \vdash_{\mathbb{S}} e_j\colon t_j'$ and from the fact that we have $\Gamma \vdash_{\mathbb{S}} x\varsigma\colon t_x$ for all $x \in \mathsf{capt}(p_j)$ and all $t_x \in \mathsf{inst}(\mathsf{gen}_{\Gamma}(\Gamma'(x)))$, we derive $\Gamma \vdash_{\mathbb{S}} e_j\varsigma\colon t_j'$ by Lemma A.25 and then conclude by subsumption.

Case Ts-Subsum Straightforward application of the induction hypothesis. □

Corollary A.28 (Type soundness)**.** *Let e be a well-typed, closed expression, that is, such that $\varnothing \vdash_{\mathbb{S}} e\colon t$ holds for some t. Then, either e diverges or it reduces to a value v such that $\varnothing \vdash_{\mathbb{S}} v\colon t$.*

Proof. Consequence of Theorem A.26 and Theorem A.27. □

A.3.3 Completeness of $\mathbb{S}$ with respect to $\mathbb{K}$

In the proof of completeness, we consider a calculus and type systems extended with the addition of a fixed-point combinator Υ: this simplifies the proof (as it allows us to assume that all arrow types are inhabited) and it would be desirable anyway in order to use the system in pratice. We add a new production $\Upsilon\, e$ to the grammar defining expressions, a new production $\Upsilon\, E$ to the grammar of evaluation contexts, and the new reduction rule *R-Fix* in Figure 11. We extend $\mathbb{K}$ and $\mathbb{S}$ with the addition, respectively, of the rules *Tk-Fix* and *Ts-Fix* in Figure 11.

As mentioned in Section 4.3, we prove completeness of $\mathbb{S}$ with respect to $\mathbb{K}$ using inductive techniques which do not account for the presence of recursion in kinds: we therefore have to restrict ourselves to only consider kinding environments which do not feature recursion, (the *non-recursive* environments defined below). We conjecture that coinductive techniques could be used instead to prove the result for general kinding environments.

Definition A.35 (Non-recursive kinding environments)**.** *We say that a kinding environment K is* non-recursive *if, for all α such that $K(\alpha) = (L, U, T)$, we have $\alpha \notin \bigcup_{\tau\mathit{ag}\colon\ \tau\in T} \mathsf{var}_K(\tau)$.*

Definition A.36. *We define a function w which, given a $\Bbbk$-type τ in a non-recursive kinding environment K, yields the* measure $w(\tau, K)$ *of τ in K. It is defined by the following equations.*

$$w(\alpha, K) = \begin{cases} 1 + \sum_{\tau\mathit{ag}\colon\ \tau\in T} w(\tau, K) & \text{if } K(\alpha) = (L, U, T) \\ 1 & \text{otherwise} \end{cases}$$
$$w(b, K) = 1$$
$$w(\tau_1 \to \tau_2, K) = w(\tau_1, K) + w(\tau_2, K) + 1$$
$$w(\tau_1 \times \tau_2, K) = w(\tau_1, K) + w(\tau_2, K) + 1$$

Definition A.37 (Translation of types)**.** *Given a $\Bbbk$-type τ in a non-recursive kinding environment K, its* translation *is the $\mathbb{s}$-type $[\![\tau]\!]_K$ defined inductively by the rules in Figure 12.*

$$[\![\alpha]\!]_K = \begin{cases} \alpha & \text{if } K(\alpha) = \bullet \\ (\mathsf{low}_K(L,T) \vee \alpha) \wedge \mathsf{upp}_K(U,T) & \text{if } K(\alpha) = (L,U,T) \end{cases}$$
$$[\![b]\!]_K = b$$
$$[\![\tau_1 \to \tau_2]\!]_K = [\![\tau_1]\!]_K \to [\![\tau_2]\!]_K$$
$$[\![\tau_1 \times \tau_2]\!]_K = [\![\tau_1]\!]_K \times [\![\tau_2]\!]_K$$

where:

$$\mathsf{low}_K(L,T) = \bigvee_{\text{`}tag \in L} \text{`}tag(\bigwedge_{\text{`}tag\colon\, \tau \in T} [\![\tau]\!]_K)$$
$$\mathsf{upp}_K(U,T) = \begin{cases} \bigvee_{\text{`}tag \in U} \text{`}tag(\bigwedge_{\text{`}tag\colon\, \tau \in T} [\![\tau]\!]_K) & \text{if } U \neq \mathcal{L} \\ \bigvee_{\text{`}tag \in \mathsf{dom}(T)} \text{`}tag(\bigwedge_{\text{`}tag\colon\, \tau \in T} [\![\tau]\!]_K) \vee (\mathbb{1}_{\mathbb{v}} \setminus \bigvee_{\text{`}tag \in \mathsf{dom}(T)} \text{`}tag(\mathbb{1})) & \text{if } U = \mathcal{L} \end{cases}$$

Figure 12. Translation of $\Bbbk$-types to $\mathbb{s}$-types.

We define the translation of type schemes as $[\![\forall A.\, K' \triangleright \tau]\!]_K = \forall A.\, [\![\tau]\!]_{K,K'}$ *and of type environments by translating each type scheme pointwise.*

Lemma A.29. *For any $\Bbbk$-type τ in a non-recursive kinding environment K, we have* $\mathsf{var}([\![\tau]\!]_K) \subseteq \mathsf{var}_K(\tau)$. *Likewise, for any $\Bbbk$-scheme σ and $\Bbbk$-type environment Γ, we have* $\mathsf{var}([\![\sigma]\!]_K) \subseteq \mathsf{var}_K(\sigma)$ *and* $\mathsf{var}([\![\Gamma]\!]_K) \subseteq \mathsf{var}_K(\Gamma)$.

Proof. The translation does not introduce new variables, therefore we can show $\mathsf{var}([\![\tau]\!]_K) \subseteq \mathsf{var}_K(\tau)$ by induction on $w(\tau, K)$. We extend this straightforwardly to type schemes and environments. □

Lemma A.30. *Let p be a pattern and $t \leq \lbag p \rbag$ an $\mathbb{s}$-type. If $K \vdash p\colon \tau \Rightarrow \Gamma$ and $t \leq [\![\tau]\!]_K$, then, for all $x \in \mathsf{capt}(p)$, $(t//p)(x) \leq [\![\Gamma(x)]\!]_K$.*

Proof. By structural induction on p.

Cases $p = \_$ and $p = c$ There is nothing to prove since $\mathsf{capt}(p) = \varnothing$.

Case $p = x$ We have

$$K \vdash p\colon \tau \Rightarrow \{x\colon \tau\} \qquad t//x = \{x\colon t\}$$

and must prove $\{x\colon t\}(x) \leq [\![\{x\colon \tau\}(x)]\!]_K$, that is, $t \leq [\![\tau]\!]_K$, which is true by hypothesis.

Case $p = (p_1, p_2)$ We have

$$K \vdash p\colon \tau_1 \times \tau_2 \Rightarrow \Gamma \qquad \Gamma = \Gamma_1 \cup \Gamma_2 \qquad \forall i.\ K \vdash p_i\colon \tau_i \Rightarrow \Gamma_i$$
$$t//p = \pi_1(t)//p_1 \cup \pi_2(t)//p_2\ .$$

Since $t \leq [\![\tau_1]\!]_K \times [\![\tau_2]\!]_K$, by Property A.31 we have $\pi_i(t) \leq [\![\tau_i]\!]_K$. Likewise, $\pi_i(t) \leq \lbag p_i \rbag$. We apply the induction hypothesis to conclude.

Case $p = \text{`}tag(p_1)$ We have

$$K \vdash p\colon \alpha \Rightarrow \Gamma \qquad K \vdash p_1\colon \tau_1 \Rightarrow \Gamma$$
$$K(\alpha) = (L,U,T) \qquad (\text{`}tag \in U \text{ implies } \text{`}tag\colon \tau_1 \in T) \qquad t//p = \pi_{\text{`}tag}(t)//p_1\ .$$

Since $t \leq \lbag \text{`}tag(p_1) \rbag = \text{`}tag(\lbag p_1 \rbag)$, by Property A.32 we have $\pi_{\text{`}tag}(t) \leq \lbag p_1 \rbag$. We next prove $\pi_{\text{`}tag}(t) \leq [\![\tau_1]\!]_K$, which allows us to apply the induction hypothesis and conclude.

The translation of α is $[\![\alpha]\!]_K = (\mathsf{low}_K(L,T) \vee \alpha) \wedge \mathsf{upp}_K(U,T)$. We have $t \leq [\![\alpha]\!]_K$ and hence $t \leq \mathsf{upp}_K(U,T)$. Since $t \leq \text{`}tag(\mathbb{1})$, $t \leq \mathsf{upp}_K(U,T) \wedge \text{`}tag(\mathbb{1})$. We distribute the intersection over the summands of $\mathsf{upp}_K(U,T)$, which is a union.

If $\text{`}tag \notin U$ (in which case $U \neq \mathcal{L}$), then all summands have the form $\text{`}tag_1(\tau')$ and for each $\text{`}tag_1$ we have $\text{`}tag_1 \neq \text{`}tag$: hence, the intersection is empty and thus we have $t \leq \mathbb{0} \simeq \text{`}tag(\mathbb{0})$. Then $\pi_{\text{`}tag}(t) \leq \mathbb{0} \leq [\![\tau_1]\!]_K$.

If $\text{`}tag \in U$, then necessarily $\text{`}tag \in \mathsf{dom}(T)$ holds as well. In that case the intersection $\mathsf{upp}_K(U,T) \wedge \text{`}tag(\mathbb{1})$ is equivalent to $\text{`}tag(\bigwedge_{\text{`}tag\colon\, \tau' \in T} [\![\tau']\!]_K)$. Hence $t \leq \text{`}tag(\bigwedge_{\text{`}tag\colon\, \tau' \in T} [\![\tau']\!]_K)$ and $\pi_{\text{`}tag}(t) \leq \bigwedge_{\text{`}tag\colon\, \tau' \in T} [\![\tau']\!]_K$. Since $\text{`}tag\colon \tau_1 \in T$, $\bigwedge_{\text{`}tag\colon\, \tau' \in T} [\![\tau']\!]_K \leq [\![\tau_1]\!]_K$, from which follows $\pi_{\text{`}tag}(t) \leq [\![\tau_1]\!]_K$.

Case $p = p_1 \& p_2$ We directly apply the induction hypothesis to both sub-patterns and conclude.

Case $p = p_1|p_2$ We have

$$K \vdash p\colon \tau \Rightarrow \Gamma \qquad \forall i.\ K \vdash p_i\colon \tau \Rightarrow \Gamma$$
$$t//p = (t \wedge \lbag p_1 \rbag)//p_1 \mathbin{\curlyvee} (t \setminus \lbag p_1 \rbag)//p_2\ .$$

Since $t \wedge \lbag p_1 \rbag$ and $t \setminus \lbag p_1 \rbag$ are subtypes of t, they are also subtypes of $[\![\tau]\!]_K$. We can apply the induction hypothesis and, for each x, derive both that $(t \wedge \lbag p_1 \rbag//p_1)(x) \leq [\![\Gamma(x)]\!]_K$ and that $(t \setminus \lbag p_1 \rbag//p_2)(x) \leq [\![\Gamma(x)]\!]_K$. Hence, $(t//p)(x) \leq [\![\Gamma(x)]\!]_K$. □

Lemma A.31 (Translation of type substitutions)**.** *Let K, K' be two non-recursive kinding environments such that $\mathsf{dom}(K') \cap (\mathsf{dom}(K) \cup \mathsf{var}_\varnothing(K)) = \varnothing$. Let θ be a $\Bbbk$-type substitution such that $\mathsf{dom}(\theta) \subseteq \mathsf{dom}(K')$ and $K, K' \vdash \theta\colon K$.*

Let θ' be the $\mathbb{s}$-type substitution defined as $[\, {}^{[\![\alpha\theta]\!]_K}/_\alpha \mid \alpha \in \mathsf{dom}(K')\,]$. For every $\Bbbk$-type τ, we have $[\![\tau]\!]_{K,K'}\theta' \simeq [\![\tau\theta]\!]_K$.

Proof. By complete induction on $w(\tau, (K, K'))$. We proceed by cases on τ and assume that the lemma holds for all τ' such that $w(\tau', (K, K')) < w(\tau, (K, K'))$.

Case $\tau = \alpha$, with $(K, K')(\alpha) = \bullet$ We have $[\![\alpha]\!]_{K,K'} = \alpha$, hence $[\![\alpha]\!]_{K,K'}\theta' = \alpha\theta'$. Either $\alpha \in \mathsf{dom}(K)$ or $\alpha \in \mathsf{dom}(K')$ (the domains are disjoint). In the former case, $\alpha\theta = \alpha$ and $\alpha\theta' = \alpha$. Thus we have $[\![\alpha]\!]_{K,K'}\theta' = \alpha = [\![\alpha\theta]\!]_K$. In the latter, $\alpha\theta' = [\![\alpha\theta]\!]_K$ holds by definition of θ'.

Case $\tau = \alpha$, with $K(\alpha) = (L, U, T)$ and $\alpha \notin \mathsf{dom}(K')$ We have $[\![\alpha]\!]_{K,K'} = [\![\alpha]\!]_K$ because no variable in the kind of α is in $\mathsf{dom}(K')$. For the same reason, since the translation does not add variables, $[\![\alpha]\!]_{K,K'}\theta' = [\![\alpha]\!]_K$. Additionally, $\alpha\theta = \alpha$, so also $[\![\alpha\theta]\!]_K = [\![\alpha]\!]_K$.

Case $\tau = \alpha$, with $K'(\alpha) = (L', U', T')$ Because $K, K' \vdash \theta\colon K$, we know that $\alpha\theta$ is some variable β such that $K(\beta) = (L, U, T)$ and $(L, U, T) \vDash (L', U', T'\theta)$.

We have

$$[\![\alpha\theta]\!]_K = [\![\beta]\!]_K = (\mathsf{low}_K(L, T) \vee \beta) \wedge \mathsf{upp}_K(U, T)$$

and

$$\begin{aligned}
[\![\alpha]\!]_{K,K'}\theta' &= \big((\mathsf{low}_{K,K'}(L', T') \vee \alpha) \wedge \mathsf{upp}_{K,K'}(U', T')\big)\theta' \\
&= (\mathsf{low}_{K,K'}(L', T')\theta' \vee \alpha\theta') \wedge \mathsf{upp}_{K,K'}(U', T')\theta' \\
&= \Big(\mathsf{low}_{K,K'}(L', T')\theta' \vee \big((\mathsf{low}_K(L, T) \vee \beta) \wedge \mathsf{upp}_K(U, T)\big)\Big) \wedge \mathsf{upp}_{K,K'}(U', T')\theta'\ .
\end{aligned}$$

Let us define

$$l = \mathsf{low}_K(L, T) \qquad u = \mathsf{upp}_K(U, T)$$
$$l' = \mathsf{low}_{K,K'}(L', T')\theta' \qquad u' = \mathsf{upp}_{K,K'}(U', T')\theta'$$

and assume that the following hold (we prove them below):

$$l \leq u \qquad l' \leq u' \qquad l' \leq l \qquad u \leq u'\ .$$

Then we have also $l' \leq u$ by transitivity. Whenever $t \leq t'$, we have $t \wedge t' \simeq t$ and $t \vee t' \simeq t'$.

Thus we have the following equivalences:

$$\begin{aligned}
[\![\alpha]\!]_{K,K'}\theta' &= (l' \vee ((l \vee \beta) \wedge u)) \wedge u' \\
&\simeq (l' \wedge u') \vee ((l \vee \beta) \wedge u \wedge u') && \text{distributivity} \\
&\simeq l' \vee ((l \vee \beta) \wedge u) && l' \leq u' \text{ and } u \leq u' \\
&\simeq (l' \vee l \vee \beta) \wedge (l' \vee u) && \text{distributivity} \\
&\simeq (l \vee \beta) \wedge u && l' \leq l \text{ and } l' \leq u
\end{aligned}$$

by which we conclude.

We now prove our four assumptions. The first, $l \leq u$, holds because $L \subseteq U$ and $L \subseteq \mathsf{dom}(T)$: hence each branch of l appears in u as well. The second is analogous.

For the other assumptions, note that $[\![\tau']\!]_{K,K'}\theta' \simeq [\![\tau'\theta]\!]_K$ holds for all τ' in the range of T'. To prove $l' \leq l$, note that $L' \subseteq L$ and $T'\theta \subseteq T$. In l', we distribute the application of θ' over all the summands of the union and inside all variant type constructors. Then, we show $\grave{}tag(\bigwedge_{\grave{}tag\colon\, \tau' \in T'} [\![\tau']\!]_{K,K'}\theta') \leq l$ for each $\grave{}tag \in L'$. We have $\grave{}tag(\bigwedge_{\grave{}tag\colon\, \tau' \in T'} [\![\tau']\!]_{K,K'}\theta') \simeq \grave{}tag(\bigwedge_{\grave{}tag\colon\, \tau' \in T'} [\![\tau'\theta]\!]_K) = \grave{}tag(\bigwedge_{\grave{}tag\colon\, \tau'\theta \in T'\theta} [\![\tau'\theta]\!]_K)$. Since $L' \subseteq L$, there is a summand of l with the same tag. Since $\grave{}tag$ is in the lower bound, it has a single type in both T and T' and, since $T'\theta \subseteq T$, the type it has in T must be $\tau'\theta$.

To prove $u \leq u'$, note that $U \subseteq U'$. If $U = \mathcal{L}$, then $U' = \mathcal{L}$. Then both u and u' are unions of two types: the union of tags mentioned respectively in T and T' and the rest. For each $\grave{}tag$, if

$\grave{}tag \notin \mathsf{dom}(T)$, then $\grave{}tag \notin \mathsf{dom}(T')$, in which case both u and u' admit it with any argument type. If $\grave{}tag \in \mathsf{dom}(T)$, either $\grave{}tag \in \mathsf{dom}(T')$ or not. In the former case, u admits a smaller argument type than u' because $T'\theta \subseteq T$. The same occurs in the latter case, since u' admits $\grave{}tag$ with any argument type.

If $U \neq \mathcal{L}$, then U' could be $\mathcal{L}$ or not. In either case we can prove, for each $\grave{}tag \in U$, that u' admits $\grave{}tag$ with a larger argument type than u does.

Case $\tau = b$ Straightforward, since a basic type is translated into itself and is never affected by substitutions.

Case $\tau = \tau_1 \to \tau_2$ By the induction hypothesis we have $[\![\tau_i]\!]_{K,K'}\theta' \simeq [\![\tau_i\theta]\!]_K$ for both i. Then

$$\begin{aligned}[\![\tau_1 \to \tau_2]\!]_{K,K'}\theta' &= ([\![\tau_1]\!]_{K,K'}\theta') \to ([\![\tau_2]\!]_{K,K'}\theta') \simeq [\![\tau_1\theta]\!]_K \to [\![\tau_2\theta]\!]_K \\ &= [\![(\tau_1\theta) \to (\tau_2\theta)]\!]_K = [\![(\tau_1 \to \tau_2)\theta]\!]_K\,.\end{aligned}$$

Case $\tau = \tau_1 \times \tau_2$ Analogous to the previous case. □

Lemma A.32. *If $\varnothing \vdash_{\mathbb{S}} v\colon [\![\tau]\!]_K$, then there exists a value v' such that $K; \varnothing \vdash_{\Bbbk} v'\colon \tau$ and, for every pattern p, $v/p = \Omega \iff v'/p = \Omega$.*

Proof. By structural induction on v.

Note that values are always typed by an application of the typing rule corresponding to their form (*Ts-Const*, *Ts-Abstr*, *Ts-Pair*, or *Ts-Tag*) to appropriate premises, possibly followed by applications of *Ts-Subsum*. Hence, if $\varnothing \vdash_{\mathbb{S}} v\colon t$, there is a type $t' \leq t$ such that $\varnothing \vdash_{\mathbb{S}} v\colon t'$ and that the last typing rule used to derive $\varnothing \vdash_{\mathbb{S}} v\colon t'$ is one of the four above, given by the form of v.

Case $v = c$ We have $[\![\tau]\!]_K \geq c$. Hence $\tau = b_c$, as the translation of any other τ is disjoint from c. Then we can take $v' = v$.

Case $v = (v_1, v_2)$ We have $[\![\tau]\!]_K \geq t_1 \times t_2$ for some t_1 and t_2. Hence $\tau = \tau_1 \times \tau_2$: any other τ would translate to a type disjoint from all products. Therefore $\varnothing \vdash_{\mathbb{S}} v\colon [\![\tau_1]\!]_K \times [\![\tau_2]\!]_K$. By Lemma A.12 we have $\varnothing \vdash_{\mathbb{S}} v_i\colon [\![\tau_i]\!]_K$ for both i; then by the induction hypothesis we find v_i' for both i and let $v' = (v_1', v_2')$.

Case $v = \grave{}tag(v_1)$ We have $[\![\tau]\!]_K \geq \grave{}tag(t_1)$ and $\varnothing \vdash_{\mathbb{S}} v\colon \grave{}tag(t_1)$ for some $t_1 \not\leq \mathbb{0}$ (since t_1 types the value v_1). Therefore, by the same reasoning as above, $\tau = \alpha$ with $K(\alpha) = (L, U, T)$. Since $[\![\tau]\!]_K \geq \grave{}tag(t_1)$, we have $\grave{}tag \in L$ and therefore $\grave{}tag\colon \tau_1 \in T$ for some τ_1 such that $t_1 \leq [\![\tau_1]\!]_K$. Then we have $\varnothing \vdash_{\mathbb{S}} v_1\colon [\![\tau_1]\!]_K$; we may apply the induction hypothesis to find a value v_1' and let $v' = \grave{}tag(v_1')$.

Case $v = \lambda x.\, e$ Note that an abstraction is only accepted by patterns which accept any value, so any two abstractions fail to match exactly the same patterns.

We have $\varnothing \vdash_{\mathbb{S}} v\colon t_1 \to t_2$ for some $t_1 \to t_2 \leq [\![\tau]\!]_K$. Hence we know τ is of the form $\tau_1 \to \tau_2$; thus we have $\varnothing \vdash_{\mathbb{S}} v\colon [\![\tau_1]\!]_K \to [\![\tau_2]\!]_K$. We take v' to be the function $\lambda x.\ \curlyvee\,(\lambda f.\, \lambda x.\, f\ x)\ x$, which never terminates and can be assigned any arrow type. □

Lemma A.33. *Let K be a kinding environment, τ a $\Bbbk$-type, and P a set of patterns. If $\tau \preccurlyeq_K P$, then $[\![\tau]\!]_K \leq \bigvee_{p \in P} \wr p \int$.*

Proof. By contradiction, assume that $\tau \preccurlyeq_K P$ holds but $[\![\tau]\!]_K \not\leq \bigvee_{p \in P} \wr p \int$. The latter condition implies that there exists a value v in the interpretation of $[\![\tau]\!]_K$ which is not in the interpretation of $\bigvee_{p \in P} \wr p \int$. Because the definition of accepted type is exact with respect to the semantics of pattern matching, we have $v/p = \Omega$ for all $p \in P$. We also have $\varnothing \vdash_{\mathbb{S}} v\colon [\![\tau]\!]_K$ since v is in the interpretation of that type (typing is complete with respect to the interpretation if we restrict ourselves to translations of $\Bbbk$-types).

By Lemma A.32, from v we can build a value v' such that $K; \varnothing \vdash_{\Bbbk} v'\colon \tau$ and, for every pattern p, $v/p = \Omega \iff v'/p = \Omega$. We reach a contradiction, since $\tau \preccurlyeq_K P$ and $K; \varnothing \vdash_{\Bbbk} v'\colon \tau$ imply that there exists a $p \in P$ such that $v'/p \neq \Omega$, whereas we have $v/p = \Omega$ for all $p \in P$. □

Theorem A.34 (Preservation of typing)**.** *Let e be an expression, K a non-recursive kinding environment, Γ a $\Bbbk$-type environment, and τ a $\Bbbk$-type. If $K; \Gamma \vdash_{\Bbbk} e\colon \tau$, then $[\![\Gamma]\!]_K \vdash_{\mathbb{S}} e\colon [\![\tau]\!]_K$.*

Proof. By induction on the derivation of $K; \Gamma \vdash_{\Bbbk} e\colon \tau$. We reason by cases on the last applied rule.

Case Tk-Var We have

$$K; \Gamma \vdash_{\Bbbk} x\colon \tau \qquad \tau \in \mathsf{inst}_K(\Gamma(x)) \qquad \text{hence}$$
$$\Gamma(x) = \forall A.\, K_x \triangleright \tau_x \qquad \tau = \tau_x\theta \qquad \mathsf{dom}(\theta) \subseteq A \qquad K, K_x \vdash \theta\colon K$$

and must show $[\![\Gamma]\!]_K \vdash_{\mathbb{S}} x\colon [\![\tau]\!]_K$. Since $[\![\Gamma]\!]_K(x) = \forall A.\, [\![\tau_x]\!]_{K,K_x}$, by *Ts-Var* we can derive $[\![\tau_x]\!]_{K,K_x}\theta'$ for any $\mathbb{S}$-type substitution θ' with $\mathsf{dom}(\theta') \subseteq A$.

Consider the $\mathbb{S}$-type substitution $\theta' = [\, {}^{[\![\alpha\theta]\!]_K}\!/_{\alpha} \mid \alpha \in A\,]$. We have $[\![\tau_x]\!]_{K,K_x}\theta' \simeq [\![\tau_x\theta]\!]_K$ by Lemma A.31 (we can assume the conditions on the domain of K_x to hold by renaming the variables in A). Hence, we derive $[\![\tau_x]\!]_{K,K_x}\theta'$ by *Ts-Var* and then $[\![\tau_x\theta]\!]_K$ by subsumption.

Case Tk-Const We have

$$K;\Gamma \vdash_{\mathbb{K}} c\colon b_c \qquad [\![b_c]\!]_K = b_c$$

and may derive $[\![\Gamma]\!]_K \vdash_{\mathbb{S}} c\colon c$ by *Ts-Const* and $[\![\Gamma]\!]_K \vdash_{\mathbb{S}} c\colon b_c$ by subsumption.

Case Tk-Abstr We have

$$K;\Gamma \vdash_{\mathbb{K}} \lambda x.\, e_1\colon \tau_1 \to \tau_2 \qquad K;\Gamma, \{x\colon \tau_1\} \vdash_{\mathbb{K}} e_1\colon \tau_2 \qquad [\![\tau_1 \to \tau_2]\!]_K = [\![\tau_1]\!]_K \to [\![\tau_2]\!]_K\,.$$

By the induction hypothesis we derive $[\![\Gamma]\!]_K, \{x\colon [\![\tau_1]\!]_K\} \vdash_{\mathbb{S}} e_1\colon [\![\tau_2]\!]_K$, then we apply *Ts-Abstr*.

Cases Tk-Appl, Tk-Pair, and Tk-Fix Straightforward application of the induction hypothesis.

Case Tk-Tag We have

$$K;\Gamma \vdash_{\mathbb{K}} \text{`}tag(e_1)\colon \alpha \qquad K;\Gamma \vdash_{\mathbb{K}} e_1\colon \tau_1 \qquad K(\alpha) = (L,U,T) \qquad \text{`}tag \in L \qquad \text{`}tag\colon \tau_1 \in T$$
$$[\![\alpha]\!]_K = (\mathsf{low}_K(L,T) \vee \alpha) \wedge \mathsf{upp}_K(U,T)\,.$$

We derive $[\![\Gamma]\!]_K \vdash_{\mathbb{S}} e_1\colon [\![\tau_1]\!]_K$ by the induction hypothesis, then $[\![\Gamma]\!]_K \vdash_{\mathbb{S}} \text{`}tag(e_1)\colon \text{`}tag([\![\tau_1]\!]_K)$ by *Ts-Tag*. We show that $\text{`}tag([\![\tau_1]\!]_K) \leq [\![\alpha]\!]_K$ holds: hence, we may derive the supertype by subsumption.

Since $\text{`}tag \in L$ and hence $\text{`}tag \in \mathsf{dom}(T)$, both $\mathsf{low}_K(L,T)$ and $\mathsf{upp}_K(U,T)$ contain a summand $\text{`}tag(\bigwedge_{\text{`}tag\colon \tau' \in T} [\![\tau']\!]_K)$. Since $\text{`}tag\colon \tau_1 \in T$ and no other type may be associated to $\text{`}tag$, the intersection has a single factor $[\![\tau_1]\!]_K$. Thus we have both $\text{`}tag([\![\tau_1]\!]_K) \leq \mathsf{low}_K(L,T)$ and $\text{`}tag([\![\tau_1]\!]_K) \leq \mathsf{upp}_K(U,T)$; hence, $\text{`}tag([\![\tau_1]\!]_K) \leq [\![\alpha]\!]_K$.

Case Tk-Match We have

$$K;\Gamma \vdash_{\mathbb{K}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i\in I}\colon \tau$$
$$K;\Gamma \vdash_{\mathbb{K}} e_0\colon \tau_0 \qquad \tau_0 \preccurlyeq_K \{\, p_i \mid i \in I\,\}$$
$$\forall i \in I.\ K \vdash p_i\colon \tau_0 \Rightarrow \Gamma_i \qquad K;\Gamma, \mathsf{gen}_{K;\Gamma}(\Gamma_i) \vdash_{\mathbb{K}} e_i\colon \tau$$

and must show

$$[\![\Gamma]\!]_K \vdash_{\mathbb{S}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i\in I}\colon [\![\tau]\!]_K$$

which we prove by establishing, for some types t_0 and t_i, t_i' for each i, that

$$[\![\Gamma]\!]_K \vdash_{\mathbb{S}} e_0\colon t_0 \qquad t_0 \leq \bigvee_{i\in I} \wr p_i \int \qquad t_i = (t_0 \setminus \bigvee_{j<i} \wr p_j \int) \wedge \wr p_i \int$$
$$\forall i \in I.\ [\![\Gamma]\!]_K, \mathsf{gen}_{[\![\Gamma]\!]_K}(t_i /\!/ p_i) \vdash_{\mathbb{S}} e_i\colon t_i' \qquad \bigvee_{i\in I} t_i' \leq [\![\tau]\!]_K\,.$$

and then applying *Ts-Match*, followed by *Ts-Subsum* if necessary.

By the induction hypothesis we derive $[\![\Gamma]\!]_K \vdash_{\mathbb{S}} e_0\colon [\![\tau_0]\!]_K$ and hence have $t_0 = [\![\tau_0]\!]_K$. By Lemma A.33, we have $t_0 \leq \bigvee_{i\in I} \wr p_i \int$. For every branch, $t_i \leq t_0$ and $t_i \leq \wr p_i \int$: therefore, we can apply Lemma A.30 and derive that $(t_i /\!/ p_i)(x) \leq [\![\Gamma_i(x)]\!]_K$ holds for every $x \in \mathsf{capt}(p_i)$.

For each branch, we derive $[\![\Gamma]\!]_K, [\![\mathsf{gen}_{K;\Gamma}(\Gamma_i)]\!]_K \vdash_{\mathbb{S}} e_i\colon [\![\tau]\!]_K$ by the induction hypothesis. We derive $[\![\Gamma]\!]_K, \mathsf{gen}_{[\![\Gamma]\!]_K}(t_i /\!/ p_i) \vdash_{\mathbb{S}} e_i\colon [\![\tau]\!]_K$ by Lemma A.22 by proving $[\![\Gamma]\!]_K, \mathsf{gen}_{[\![\Gamma]\!]_K}(t_i /\!/ p_i) \sqsubseteq [\![\Gamma]\!]_K, [\![\mathsf{gen}_{K;\Gamma}(\Gamma_i)]\!]_K$ and $\mathsf{mvar}([\![\Gamma]\!]_K, \mathsf{gen}_{[\![\Gamma]\!]_K}(t_i /\!/ p_i)) \subseteq \mathsf{mvar}([\![\Gamma]\!]_K, [\![\mathsf{gen}_{K;\Gamma}(\Gamma_i)]\!]_K)$. The latter is straightforward. For the former, for each $x \in \mathsf{capt}(p_i)$—say $\Gamma_i(x) = \tau_x$ and $(t_i /\!/ p_i)(x) = t_x$—we must show $\mathsf{gen}_{[\![\Gamma]\!]_K}(t_x) \sqsubseteq [\![\mathsf{gen}_{K;\Gamma}(\tau_x)]\!]_K$. This holds because $t_x \leq [\![\tau_x]\!]_K$ and because, by Lemma A.29, $\mathsf{var}([\![\Gamma]\!]_K) \subseteq \mathsf{var}_K(\Gamma)$.

We can thus choose $t_i' = [\![\tau]\!]_K$ for all branches, satisfying $\bigvee_{i\in I} t_i' \leq [\![\tau]\!]_K$. □

A.4 Type reconstruction

A.4.1 Definition of type reconstruction for $\mathbb{S}$

Definition A.38 (Constraints)**.** *A* constraint c *is a term inductively generated by the following grammar:*

$$c ::= t \dot{\leq} t \mid x \dot{\leq} t \mid \mathtt{def}\ \Gamma\ \mathtt{in}\ C \mid \mathtt{let}\ [C](\Gamma_i\ \mathtt{in}\ C_i)_{i\in I}$$

where C *ranges over* constraint sets*, that is, finite sets of constraints, and where the range of every type environment* Γ *in constraints of the form* `def` *or* `let` *only contains types (i.e., trivial type schemes).*

We give different definitions of constraint generation and rewriting here than those in Section 5, because we keep track explicitly of the new variables introduced during the derivation, rather than

$$\frac{}{t///\_ \Rightarrow_\varnothing (\varnothing, \varnothing)} \qquad \frac{}{t///x \Rightarrow_\varnothing (\{x\colon t\}, \varnothing)} \qquad \frac{}{t///c \Rightarrow_\varnothing (\varnothing, \varnothing)}$$

$$\frac{\alpha_1///p_1 \Rightarrow_{A_1} (\Gamma_1, C_1) \qquad \alpha_2///p_2 \Rightarrow_{A_2} (\Gamma_2, C_2)}{t///(p_1, p_2) \Rightarrow_{A_1 \uplus A_2 \uplus \{\alpha_1, \alpha_2\}} (\Gamma_1 \cup \Gamma_2, C_1 \cup C_2 \cup \{t \dot{\leq} \alpha_1 \times \alpha_2\})} \; A_1, A_2, \alpha_1, \alpha_2 \sharp t$$

$$\frac{\alpha///p \Rightarrow_A (\Gamma, C)}{t///\text{`}tag(p) \Rightarrow_{A \uplus \{\alpha\}} (\Gamma, C \cup \{t \dot{\leq} \text{`}tag(\alpha)\})} \; A, \alpha \sharp t$$

$$\frac{t///p_1 \Rightarrow_{A_1} (\Gamma_1, C_1) \qquad t///p_2 \Rightarrow_{A_2} (\Gamma_2, C_2)}{t///p_1 \& p_2 \Rightarrow_{A_1 \uplus A_2} (\Gamma_1 \cup \Gamma_2, C_1 \cup C_2)}$$

$$\frac{(t \wedge \lbag p_1 \rbag)///p_1 \Rightarrow_{A_1} (\Gamma_1, C_1) \qquad (t \setminus \lbag p_1 \rbag)///p_2 \Rightarrow_{A_2} (\Gamma_2, C_2)}{t///p_1 | p_2 \Rightarrow_{A_1 \uplus A_2} (\{\, x\colon \Gamma_1(x) \vee \Gamma_2(x) \mid x \in \mathsf{capt}(p_1) \,\}, C_1 \cup C_2)}$$

Figure 13. Constraint generation for pattern environments.

$$\textit{TRs-Var}\ \frac{}{x\colon t \Rightarrow_\varnothing \{x \dot{\leq} t\}} \qquad \textit{TRs-Const}\ \frac{}{c\colon t \Rightarrow_\varnothing \{c \dot{\leq} t\}}$$

$$\textit{TRs-Abstr}\ \frac{e\colon \beta \Rightarrow_A C}{\lambda x.\, e\colon t \Rightarrow_{A \uplus \{\alpha, \beta\}} \{\mathtt{def}\ \{x\colon \alpha\}\ \mathtt{in}\ C,\ \alpha \to \beta \dot{\leq} t\}} \; A, \alpha, \beta \sharp t$$

$$\textit{TRs-Appl}\ \frac{e_1\colon \alpha \to \beta \Rightarrow_{A_1} C_1 \qquad e_2\colon \alpha \Rightarrow_{A_2} C_2}{e_1\, e_2\colon t \Rightarrow_{A_1 \uplus A_2 \uplus \{\alpha, \beta\}} C_1 \cup C_2 \cup \{\beta \dot{\leq} t\}} \; A_1, A_2, \alpha, \beta \sharp t$$

$$\textit{TRs-Pair}\ \frac{e_1\colon \alpha_1 \Rightarrow_{A_1} C_1 \qquad e_2\colon \alpha_2 \Rightarrow_{A_2} C_2}{(e_1, e_2)\colon t \Rightarrow_{A_1 \uplus A_2 \uplus \{\alpha_1, \alpha_2\}} C_1 \cup C_2 \cup \{\alpha_1 \times \alpha_2 \dot{\leq} t\}} \; A_1, A_2, \alpha_1, \alpha_2 \sharp t$$

$$\textit{TRs-Tag}\ \frac{e\colon \alpha \Rightarrow_A C}{\text{`}tag(e)\colon t \Rightarrow_{A \uplus \{\alpha\}} C \cup \{\text{`}tag(\alpha) \dot{\leq} t\}} \; A, \alpha \sharp t$$

$$\textit{TRs-Match}\ \frac{\begin{array}{c} e_0\colon \alpha \Rightarrow_{A_0} C_0 \qquad t_i = (\alpha \setminus \bigvee_{j<i} \lbag p_j \rbag) \wedge \lbag p_i \rbag \\ \forall i \in I \qquad t_i///p_i \Rightarrow_{A_i} (\Gamma_i, C_i) \qquad e_i\colon \beta \Rightarrow_{A_i'} C_i' \\ C_0' = C_0 \cup (\bigcup_{i \in I} C_i) \cup \{\alpha \dot{\leq} \bigvee_{i \in I} \lbag p_i \rbag\} \\ A = A_0 \uplus (\biguplus_{i \in I} A_i) \uplus (\biguplus_{i \in I} A_i') \uplus \{\alpha, \beta\} \end{array}}{\mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I}\colon t \Rightarrow_A \{\mathtt{let}\ [C_0'](\Gamma_i\ \mathtt{in}\ C_i')_{i \in I},\ \beta \dot{\leq} t\}} \; A \sharp t$$

Figure 14. Constraint generation rules with explicit variable introduction.

informally requiring them to be fresh. For instance, in $e\colon t \Rightarrow_A C$, A is the set of variables which appear in C but not in t. We will omit it for soundness proofs, where it is not relevant.

We use the symbol $\uplus$ to denote the union of two disjoint sets. Therefore, when we write $A_1 \uplus A_2$, we require A_1 and A_2 to be disjoint. When we require this for sets of type variables, the condition is always satisfiable by an appropriate choice of variables, since there is an infinite supply to choose from.

Definition A.39 (Freshness)**.** *We say that a type variable α is* fresh *with respect to a set of type variables A, and write $\alpha \sharp A$, if $\alpha \notin A$. We write $A \sharp A'$ if $\forall \alpha \in A.\ \alpha \sharp A'$.*

We extend this to define freshness with respect to types, type environments, and type substitutions: we write $\alpha \sharp t$ if $\alpha \sharp \mathsf{var}(t)$, $\alpha \sharp \Gamma$ if $\alpha \sharp \mathsf{var}(\Gamma)$, and $\alpha \sharp \theta$ if $\alpha \sharp (\mathsf{dom}(\theta) \cup \mathsf{var}(\theta))$.

Definition A.40 (Environment generation for pattern matching)**.** *The environment generation relation for pattern matching $t///p \Rightarrow_A (\Gamma, C)$ is defined by the rules in Figure 13.*

Definition A.41 (Constraint generation)**.** *The constraint generation relation $e\colon t \Rightarrow_A C$ is defined by the rules in Figure 14.*

$$\frac{\forall i \in I \qquad \Gamma \vdash c_i \rightsquigarrow_{A_i} D_i}{\Gamma \vdash \{\, c_i \mid i \in I \,\} \rightsquigarrow_{\uplus_{i\in I} A_i} \bigcup_{i\in I} D_i} \qquad \frac{}{\Gamma \vdash t \mathrel{\dot\leq} t' \rightsquigarrow_{\varnothing} \{t \mathrel{\dot\leq} t'\}}$$

$$\frac{\Gamma(x) = \forall\{\alpha_1, \ldots, \alpha_n\}.\, t_x}{\Gamma \vdash x \mathrel{\dot\leq} t \rightsquigarrow_{\{\beta_1,\ldots,\beta_n\}} \{t_x[{}^{\beta_1}/_{\alpha_1}, \ldots, {}^{\beta_n}/_{\alpha_n}] \mathrel{\dot\leq} t\}} \qquad \frac{\Gamma, \Gamma' \vdash C \rightsquigarrow_A D}{\Gamma \vdash \mathtt{def}\ \Gamma'\ \mathtt{in}\ C \rightsquigarrow_A D}$$

$$\frac{\begin{array}{c}\Gamma \vdash C_0 \rightsquigarrow_{A_0} D_0 \qquad \theta_0 \in \mathsf{tally}(D_0) \\ \forall i \in I \qquad \Gamma, \mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0) \vdash C_i \rightsquigarrow_{A_i} D_i \\ A = A_0 \uplus (\biguplus_{i\in I} A_i) \uplus \mathsf{var}(\theta_0)\end{array}}{\Gamma \vdash \mathtt{let}\ [C_0](\Gamma_i\ \mathtt{in}\ C_i)_{i\in I} \rightsquigarrow_A \mathsf{equiv}(\theta_0) \cup \bigcup_{i\in I} D_i}$$

Figure 15. Constraint rewriting rules with explicit variable introduction.

Note that all rules include a constraint of the form $(\cdot) \mathrel{\dot\leq} t$. We add this constraint everywhere to streamline the proofs; in practice, it can be dropped from *TRs-Appl* and *TRs-Match* by using directly t instead of β to generate constraints for the sub-expressions.

Definition A.42 (Type-constraint set)**.** *A* type-constraint set D *is a set of constraints of the form* $t \mathrel{\dot\leq} t'$*, where* t *and* t' *are types.*

We say that a type substitution θ *satisfies a type-constraint set* D*, written* $\theta \Vdash D$*, if* $t\theta \leq t'\theta$ *holds for every* $t \mathrel{\dot\leq} t'$ *in* D*.*

Definition A.43 (Equivalent type-constraint set)**.** *The* equivalent type-constraint set $\mathsf{equiv}(\theta)$ *of a type substitution* θ *is defined as*

$$\mathsf{equiv}(\theta) = \textstyle\bigcup_{\alpha\in\mathsf{dom}(\theta)} \{\alpha \mathrel{\dot\leq} \alpha\theta,\ \alpha\theta \mathrel{\dot\leq} \alpha\}\ .$$

Definition A.44 (Constraint rewriting)**.** *The constraint rewriting relation* $\Gamma \vdash c \rightsquigarrow_A D$ *between type environments, constraints or constraint sets, and type-constraint sets is defined by the rules in Figure 15.*

Definition A.45 (Tallying problem)**.** *Let* D *be a type-constraint set. A type substitution* θ *is a solution to the* tallying problem *of* D *if it satisfies* D*, that is, if* $\theta \Vdash D$*.*

Property A.46 (Tallying algorithm)**.** *There exists a terminating algorithm* tally *such that, for any type-constraint set* D*,* $\mathsf{tally}(D)$ *is a finite, possibly empty, set of type substitutions.*

Theorem A.35 (Soundness and completeness of tally)**.** *Let* D *be a type-constraint set. For any type substitution* θ*:*

- *if* $\theta \in \mathsf{tally}(D)$*, then* $\theta \Vdash D$*;*
- *if* $\theta \Vdash D$*, then* $\exists\theta' \in \mathsf{tally}(D), \theta''.\ \forall\alpha \notin \mathsf{var}(\theta').\ \alpha\theta \simeq \alpha\theta'\theta''$*.*

Furthermore, if $\theta \in \mathsf{tally}(D)$*, then* $\mathsf{dom}(\theta)$ *is the set of variables appearing in* D *and* $\mathsf{var}(\theta)$ *is a set of fresh variables of the same cardinality. In the completeness property above, for* θ'' *we can take* $\theta \cup \theta'''$ *where* $\mathsf{dom}(\theta''') = \mathsf{var}(\theta')$*.*

A.4.2 Properties of type reconstruction for $\mathbb{S}$

Lemma A.36. *Given a constraint set* C*, we write* $\mathsf{var}(C)$ *for the set of variables appearing in it. The following properties hold:*

- *whenever* $t///p \Rightarrow_A (\Gamma, C)$*, we have* $\mathsf{var}(C) \subseteq \mathsf{var}(t) \cup A$*,* $\mathsf{var}(\Gamma) \subseteq \mathsf{var}(t) \cup A$*, and* $A \sharp t$*;*
- *whenever* $e\colon t \Rightarrow_A C$*, we have* $\mathsf{var}(C) \subseteq \mathsf{var}(t) \cup A$ *and* $A \sharp t$*;*
- *whenever* $\Gamma \vdash C \rightsquigarrow_A D$*, we have* $\mathsf{var}(D) \subseteq \mathsf{var}(C) \cup \mathsf{var}(\Gamma) \cup A$*.*

Proof. Straightforward proofs by induction on the derivations. □

Lemma A.37 (Correctness of environment reconstruction)**.** *Let* p *be a pattern and* t*,* t' *two types, with* $t' \leq \wr p \int$*. Let* $t///p \Rightarrow (\Gamma, C)$*. If* θ *is a type substitution such that* $\theta \Vdash C$ *and* $t' \leq t\theta$*, then, for all* $x \in \mathsf{capt}(p)$*,* $(t'//p)(x) \leq \Gamma(x)\theta$*.*

Proof. By structural induction on p.

Cases $p = \_$ *and* $p = c$ There is nothing to prove since $\mathsf{capt}(p) = \varnothing$.

Case $p = x$ We have

$$t///x \Rightarrow (\{x\colon t\}, \varnothing) \qquad (t'//x)(x) = t'$$

and must show $t' \leq t\theta$, which we know by hypothesis.

Case $p = (p_1, p_2)$ We have

$$t///p \Rightarrow (\Gamma_1 \cup \Gamma_2, C_1 \cup C_2 \cup \{t \dot{\leq} \alpha_1 \times \alpha_2\}) \qquad \forall i.\ \alpha_i///p_i \Rightarrow (\Gamma_i, C_i)\ .$$

Each $x \in \mathsf{capt}(p)$ is either in $\mathsf{capt}(p_1)$ or in $\mathsf{capt}(p_2)$. Let $x \in \mathsf{capt}(p_i)$; then, we must show $(\pi_i(t')//p_i)(x) \leq \Gamma_i(x)\theta$. This follows from the induction hypothesis, since $t' \leq t\theta \leq \alpha_1\theta \times \alpha_2\theta$ implies $\pi_i(t') \leq \alpha_i\theta$ by Property A.31.

Case $p = \grave{}\mathit{tag}(p_1)$ We have

$$t///\grave{}\mathit{tag}(p) \Rightarrow (\Gamma, C \cup \{t \dot{\leq} \grave{}\mathit{tag}(\alpha)\}) \qquad \alpha///p \Rightarrow (\Gamma, C)\ .$$

Analogous to the previous case. We can apply the induction hypothesis, because $t' \leq t\theta \leq \grave{}\mathit{tag}(\alpha)\theta$ implies $\pi_{\grave{}\mathit{tag}}(t') \leq \alpha\theta$ by Property A.32.

Case $p = p_1 \& p_2$ Every $x \in \mathsf{capt}(p)$ is either in $\mathsf{capt}(p_1)$ or in $\mathsf{capt}(p_2)$. Let $x \in \mathsf{capt}(p_i)$; then, we apply the induction hypothesis to p_i to conclude.

Case $p = p_1 | p_2$ We have

$$t///p_1|p_2 \Rightarrow (\{\, x \colon \Gamma_1(x) \vee \Gamma_2(x) \mid x \in \mathsf{capt}(p_1)\,\}, C_1 \cup C_2)$$
$$(t \wedge \lbag p_1 \rbag)///p_1 \Rightarrow (\Gamma_1, C_1) \qquad (t \setminus \lbag p_1 \rbag)///p_2 \Rightarrow (\Gamma_2, C_2)\ .$$

By the induction hypothesis applied to both p_1 and p_2 we derive, for all x,

$$(t' \wedge \lbag p_1 \rbag//p_1)(x) \leq \Gamma_1(x)\theta \qquad (t' \setminus \lbag p_1 \rbag//p_2)(x) \leq \Gamma_2(x)\theta$$

from which we can conclude

$$(t'//p)(x) = (t' \wedge \lbag p_1 \rbag//p_1)(x) \vee (t' \setminus \lbag p_1 \rbag//p_2)(x) \leq \Gamma_1(x)\theta \vee \Gamma_2(x)\theta\ . \qquad \square$$

Lemma A.38 (Precise solution to environment reconstruction constraints)**.** *Let p be a pattern, t a type, and θ a type substitution such that $t\theta \leq \lbag p \rbag$. Let $t///p \Rightarrow_A (\Gamma, C)$, with $A \sharp \mathsf{dom}(\theta)$.*

There exists a type substitution θ' such that $\mathsf{dom}(\theta') = A$, that $(\theta \cup \theta') \Vdash C$, and that, for all $x \in \mathsf{capt}(p)$, $\Gamma(x)(\theta \cup \theta') \leq (t\theta//p)(x)$.

Proof. By structural induction on p.

Cases $p = \_$ *and* $p = c$ In both cases we take $\theta' = [\,]$.

Case $p = x$ We have

$$t///x \Rightarrow_\varnothing (\{x \colon t\}, \varnothing)\ .$$

We take $\theta' = [\,]$ and have $t(\theta \cup \theta') \leq t\theta$.

Case $p = (p_1, p_2)$ We have

$$t///(p_1, p_2) \Rightarrow_{A_1 \uplus A_2 \uplus \{\alpha_1, \alpha_2\}} (\Gamma_1 \cup \Gamma_2, C_1 \cup C_2 \cup \{t \dot{\leq} \alpha_1 \times \alpha_2\})$$
$$\alpha_1///p_1 \Rightarrow_{A_1} (\Gamma_1, C_1) \qquad \alpha_2///p_2 \Rightarrow_{A_2} (\Gamma_2, C_2) \qquad A_1, A_2, \alpha_1, \alpha_2 \sharp t\ .$$

Let $\theta^\star = \theta \cup [{}^{\pi_1(t\theta)}/_{\alpha_1}, {}^{\pi_2(t\theta)}/_{\alpha_2}]$. We have $t\theta' = t\theta$ and $t\theta' \leq \lbag (p_1, p_2) \rbag = \lbag p_1 \rbag \times \lbag p_2 \rbag$; thus, by Property A.31, $\pi_i(t\theta') \leq \lbag p_i \rbag$. We also have $A_i \sharp \mathsf{dom}(\theta^\star), \alpha_i$ for both i, since $\{\alpha_1, \alpha_2\}$ is disjoint from each A_i.

We can therefore apply the induction hypothesis to p_i, α_i, and $\theta^\star$, for both i. We derive from each that there is a substitution θ_i' with domain A_i, such that $(\theta^\star \cup \theta_i') \Vdash C_i$ and, for all $x \in \mathsf{capt}(p_i)$, $\Gamma_i(x)(\theta^\star \cup \theta_i') \leq (\alpha_i\theta^\star//p_i)(x)$.

We take $\theta' = [{}^{\pi_1(t\theta)}/_{\alpha_1}, {}^{\pi_2(t\theta)}/_{\alpha_2}] \cup \theta_1' \cup \theta_2'$. We have $(\theta \cup \theta') \Vdash C_1 \cup C_2 \cup \{t \dot{\leq} \alpha_1 \times \alpha_2\}$ since it satisfies C_1 and C_2 and since $t\theta \leq (\alpha_1 \times \alpha_2)\theta' = \pi_1(t\theta) \times \pi_2(t\theta)$.

Case $p = \grave{}\mathit{tag}(p_1)$ We have

$$t///\grave{}\mathit{tag}(p_1) \Rightarrow_{A_1 \uplus \{\alpha\}} (\Gamma_1, C_1 \cup \{t \dot{\leq} \grave{}\mathit{tag}(\alpha)\}) \qquad \alpha///p_1 \Rightarrow_{A_1} (\Gamma_1, C_1) \qquad A_1, \alpha \sharp t\ .$$

Analogously to the previous case, we construct $\theta^\star = \theta \cup [{}^{\pi_{\grave{}\mathit{tag}}(t\theta)}/_{\alpha}]$ and apply the induction hypothesis to p_1, α, and $\theta^\star$. We derive θ_1' and take $\theta' = [{}^{\pi_{\grave{}\mathit{tag}}(t\theta)}/_{\alpha}] \cup \theta_1'$.

Case $p = p_1 \& p_2$ We have

$$t///p_1 \& p_2 \Rightarrow_{A_1 \uplus A_2} (\Gamma_1 \cup \Gamma_2, C_1 \cup C_2) \qquad t///p_1 \Rightarrow_{A_1} (\Gamma_1, C_1) \qquad t///p_2 \Rightarrow_{A_2} (\Gamma_2, C_2)\ .$$

For both i, we apply the induction hypothesis to p_i, t, and θ to derive θ_i'. We take $\theta' = \theta_1' \cup \theta_2'$.

Case $p = p_1|p_2$ We have

$$t///p_1|p_2 \Rightarrow_{A_1 \uplus A_2} (\{\, x \colon \Gamma_1(x) \vee \Gamma_2(x) \mid x \in \mathsf{capt}(x) \,\}, C_1 \cup C_2)$$
$$(t \wedge \lbag p_1 \rbag)///p_1 \Rightarrow_{A_1} (\Gamma_1, C_1) \qquad (t \setminus \lbag p_1 \rbag)///p_2 \Rightarrow_{A_2} (\Gamma_2, C_2)\,.$$

We apply the induction hypothesis to p_1, $t \wedge \lbag p_1 \rbag$, and θ to derive θ_1'. We apply it to p_2, $t \setminus \lbag p_1 \rbag$, and θ to derive θ_2'; here, note that $t\theta \leq \lbag p_1 \rbag \vee \lbag p_2 \rbag$ implies $t\theta \setminus \lbag p_1 \rbag \leq \lbag p_2 \rbag$.

We take $\theta' = \theta_1' \cup \theta_2'$. We have $(\theta \cup \theta') \Vdash C$ since it satisfies C_1 and C_2. Furthermore, for all x, we have $\Gamma_1(x)(\theta \cup \theta_1') \leq (t\theta \wedge \lbag p_1 \rbag//p_1)(x)$ and $\Gamma_2(x)(\theta \cup \theta_2') \leq (t\theta \setminus \lbag p_1 \rbag//p_2)(x)$. Then, $\Gamma(x)(\theta \cup \theta') = \Gamma_1(x)(\theta \cup \theta') \vee \Gamma_2(x)(\theta \cup \theta') = \Gamma_1(x)(\theta \cup \theta_1') \vee \Gamma_2(x)(\theta \cup \theta_2')$, since A_1 and A_2 are disjoint and both are disjoint from $\mathsf{var}(t)$. Finally, $\Gamma_1(x)(\theta \cup \theta_1') \vee \Gamma_2(x)(\theta \cup \theta_2') \leq (t\theta//p)(x)$. □

Theorem A.39 (Soundness of constraint generation and rewriting)**.** *Let e be an expression, t a type, and Γ a type environment. If $e \colon t \Rightarrow C$, $\Gamma \vdash C \rightsquigarrow D$, and $\theta \Vdash D$, then $\Gamma\theta \vdash_{\mathbb{S}} e \colon t\theta$.*

Proof. By structural induction on e.

Case $e = x$ We have

$$x \colon t \Rightarrow \{x \mathrel{\dot{\leq}} t\}$$
$$\Gamma \vdash \{x \mathrel{\dot{\leq}} t\} \rightsquigarrow \{t_x[{}^{\beta_1}/_{\alpha_1}, \ldots, {}^{\beta_n}/_{\alpha_n}] \mathrel{\dot{\leq}} t\} \qquad \Gamma(x) = \forall\{\alpha_1, \ldots, \alpha_n\}.\, t_x\,.$$

Let $A = \{\alpha_1, \ldots, \alpha_n\}$. By α-renaming we assume $A \sharp \theta$; then we have $(\Gamma\theta)(x) = (\forall A.\, t_x)\theta = \forall A.\, (t_x\theta)$. Consider the substitution $\theta_x = [{}^{\beta_1\theta}/_{\alpha_1}, \ldots, {}^{\beta_n\theta}/_{\alpha_n}]$. It has domain A, so we can derive $\Gamma\theta \vdash_{\mathbb{S}} x \colon t_x\theta\theta_x$.

We show $t_x\theta\theta_x = t_x[{}^{\beta_1}/_{\alpha_1}, \ldots, {}^{\beta_n}/_{\alpha_n}]\theta$ by showing $\alpha\theta\theta_x = \alpha[{}^{\beta_1}/_{\alpha_1}, \ldots, {}^{\beta_n}/_{\alpha_n}]\theta$ holds for all $\alpha \in \mathsf{var}(t_x)$. Either $\alpha \in A$ or not. In the first case, $\alpha = \alpha_i$ for some i; then $\alpha\theta\theta_x = \alpha\theta_x = \beta_i\theta$ and $\alpha[{}^{\beta_1}/_{\alpha_1}, \ldots, {}^{\beta_n}/_{\alpha_n}]\theta = \beta_i\theta$. In the latter, $\alpha \neq \alpha_i$ for all i; then $\alpha\theta\theta_x = \alpha\theta$, since $\mathsf{var}(\alpha\theta) \cap \mathsf{dom}(\theta_x) = \varnothing$ and $\alpha[{}^{\beta_1}/_{\alpha_1}, \ldots, {}^{\beta_n}/_{\alpha_n}]\theta = \alpha\theta$.

Therefore we derive $\Gamma\theta \vdash_{\mathbb{S}} x \colon t_x[{}^{\beta_1}/_{\alpha_1}, \ldots, {}^{\beta_n}/_{\alpha_n}]\theta$ by *Ts-Var*. Finally, since $\theta \Vdash t_x[{}^{\beta_1}/_{\alpha_1}, \ldots, {}^{\beta_n}/_{\alpha_n}] \mathrel{\dot{\leq}} t$, we derive $\Gamma\theta \vdash_{\mathbb{S}} x \colon t\theta$ by subsumption.

Case $e = c$ We have

$$c \colon t \Rightarrow \{c \mathrel{\dot{\leq}} t\} \qquad \Gamma \vdash \{c \mathrel{\dot{\leq}} t\} \rightsquigarrow \{c \mathrel{\dot{\leq}} t\}\,.$$

Analogously to the previous case, we first apply *Ts-Const* and then conclude by subsumption.

Case $e = \lambda x.\, e_1$ We have

$$\lambda x.\, e_1 \colon t \Rightarrow \{\mathtt{def}\ \{x \colon \alpha\}\ \mathtt{in}\ C_1, \alpha \to \beta \mathrel{\dot{\leq}} t\} \qquad e_1 \colon \beta \Rightarrow C_1$$
$$\Gamma \vdash \{\mathtt{def}\ \{x \colon \alpha\}\ \mathtt{in}\ C_1, \alpha \to \beta \mathrel{\dot{\leq}} t\} \rightsquigarrow D_1 \cup \{\alpha \to \beta \mathrel{\dot{\leq}} t\} \qquad \Gamma, \{x \colon \alpha\} \vdash C_1 \rightsquigarrow D_1\,.$$

By the induction hypothesis we derive $\Gamma\theta, \{x \colon \alpha\theta\} \vdash_{\mathbb{S}} e_1 \colon \beta\theta$. We apply *Ts-Abstr* to derive $\Gamma\theta \vdash_{\mathbb{S}} \lambda x.\, e_1 \colon (\alpha \to \beta)\theta$. Since $\theta \Vdash D$, we have $(\alpha \to \beta)\theta \leq t\theta$. Hence, we derive by subsumption $\Gamma\theta \vdash_{\mathbb{S}} \lambda x.\, e_1 \colon t\theta$.

Case $e = e_1\, e_2$ We have

$$e_1\, e_2 \colon t \Rightarrow C_1 \cup C_2 \cup \{\beta \mathrel{\dot{\leq}} t\} \qquad e_1 \colon \alpha \to \beta \Rightarrow C_1 \qquad e_2 \colon \alpha \Rightarrow C_2$$
$$\Gamma \vdash C_1 \cup C_2 \cup \{\beta \mathrel{\dot{\leq}} t\} \rightsquigarrow D_1 \cup D_2 \cup \{\beta \mathrel{\dot{\leq}} t\} \qquad \Gamma \vdash C_1 \rightsquigarrow D_1 \qquad \Gamma \vdash C_2 \rightsquigarrow D_2\,.$$

We derive $\Gamma\theta \vdash_{\mathbb{S}} e_1 \colon (\alpha\theta) \to (\beta\theta)$ and $\Gamma\theta \vdash_{\mathbb{S}} e_2 \colon \alpha\theta$ by the induction hypothesis. Then by *Ts-Appl* we derive $\Gamma\theta \vdash_{\mathbb{S}} e_1\, e_2 \colon \beta\theta$, and finally—since $\beta\theta \leq t\theta$—we conclude by subsumption.

Case $e = (e_1, e_2)$ We have

$$(e_1, e_2) \colon t \Rightarrow C_1 \cup C_2 \cup \{\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t\} \qquad e_1 \colon \alpha_1 \Rightarrow C_1 \qquad e_2 \colon \alpha_2 \Rightarrow C_2$$
$$\Gamma \vdash C_1 \cup C_2 \cup \{\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t\} \rightsquigarrow D_1 \cup D_2 \cup \{\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t\}$$
$$\Gamma \vdash C_1 \rightsquigarrow D_1 \qquad \Gamma \vdash C_2 \rightsquigarrow D_2\,.$$

We have $\Gamma\theta \vdash_{\mathbb{S}} e_i \colon \alpha_i\theta$ for both i by the induction hypothesis. Then, we derive $\Gamma\theta \vdash_{\mathbb{S}} (e_1, e_2) \colon (\alpha_1 \times \alpha_2)\theta$ by *Ts-Pair*, and finally conclude by subsumption.

Case $e = \text{`}tag(e_1)$ We have

$$\text{`}tag(e_1) \colon t \Rightarrow C \qquad C = C_1 \cup \{\text{`}tag(\alpha) \mathrel{\dot{\leq}} t\} \qquad e_1 \colon \alpha \Rightarrow C_1$$
$$\Gamma \vdash C \rightsquigarrow D \qquad D = D_1 \cup \{\text{`}tag(\alpha) \mathrel{\dot{\leq}} t\} \qquad \Gamma \vdash C_1 \rightsquigarrow D_1\,.$$

Analogous to the previous case. We apply the induction hypothesis, then *Ts-Tag*, then subsumption.

Case $e = \texttt{match}\ e_0\ \texttt{with}\ (p_i \to e_i)_{i\in I}$ We have

$\texttt{match}\ e_0\ \texttt{with}\ (p_i \to e_i)_{i\in I} : t \Rightarrow C$

$C = \{\texttt{let}\ [C'_0](\Gamma_i\ \texttt{in}\ C'_i)_{i\in I}, \beta \dot{\leq} t\}$

$C'_0 = C_0 \cup (\bigcup_{i\in I} C_i) \cup \{\alpha \dot{\leq} \bigvee_{i\in I} \lbag p_i \rbag\} \qquad e_0 : \alpha \Rightarrow C_0$

$\forall i \in I \qquad t_i = (\alpha \setminus \bigvee_{j<i} \lbag p_j \rbag) \wedge \lbag p_i \rbag \qquad t_i /\!/\!/ p_i \Rightarrow (\Gamma_i, C_i) \qquad e_i : \beta \Rightarrow C'_i$

$\Gamma \vdash C \rightsquigarrow D$

$\Gamma \vdash C'_0 \rightsquigarrow D'_0 \qquad \Gamma \vdash C_0 \rightsquigarrow D_0 \qquad D'_0 = D_0 \cup (\bigcup_{i\in I} C_i) \cup \{\alpha \dot{\leq} \bigvee_{i\in I} \lbag p_i \rbag\}$

$\theta_0 \in \mathsf{tally}(D'_0) \qquad \forall i \in I.\ \Gamma, \mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0) \vdash C'_i \rightsquigarrow D'_i$

$D = \mathsf{equiv}(\theta_0) \cup (\bigcup_{i\in I} D'_i) \cup \{\beta \dot{\leq} t\}$

and we must show $\Gamma\theta \vdash_{\mathbb{S}} \texttt{match}\ e_0\ \texttt{with}\ (p_i \to e_i)_{i\in I} : t\theta$.

We prove it by establishing, for some types $\hat{t_0}$ and $\hat{t_i}$, $\hat{t'_i}$ for each i, that

$$\Gamma\theta \vdash_{\mathbb{S}} e_0 : \hat{t_0} \qquad \hat{t_0} \leq \bigvee_{i\in I} \lbag p_i \rbag \qquad \hat{t_i} = (\hat{t_0} \setminus \bigvee_{j<i} \lbag p_j \rbag) \wedge \lbag p_i \rbag$$
$$\forall i \in I. \qquad \Gamma\theta, \mathsf{gen}_{\Gamma\theta}(\hat{t_i} /\!/ p_i) \vdash_{\mathbb{S}} e_i : \hat{t'_i} \qquad \bigvee_{i\in I} \hat{t'_i} \leq t\theta\ .$$

Since $\theta_0 \in \mathsf{tally}(D'_0)$, $\theta_0 \Vdash D'_0$ and thus $\theta_0 \Vdash D_0$. Then, from

$$e_0 : \alpha \Rightarrow C_0 \qquad \Gamma \vdash C_0 \rightsquigarrow D_0 \qquad \theta_0 \Vdash D_0$$

we derive $\Gamma\theta_0 \vdash_{\mathbb{S}} e_0 : \alpha\theta_0$ by the induction hypothesis.

Let $A = \mathsf{var}(\alpha\theta_0) \setminus \mathsf{mvar}(\Gamma\theta_0) = \{\alpha_1, \ldots, \alpha_n\}$. Let $B = \{\beta_1, \ldots, \beta_n\}$ be a set of type variables such that $B \sharp \Gamma, \theta, \theta_0$ and let $\theta^\star = [{}^{\beta_1}/_{\alpha_1}, \ldots, {}^{\beta_n}/_{\alpha_n}]$. We derive $\Gamma\theta_0 \vdash_{\mathbb{S}} e_0 : \alpha\theta_0\theta^\star$ by Corollary A.24, since $\theta^\star$ does not act on meaningful variables of $\Gamma\theta_0$. By Lemma A.23, we derive $\Gamma\theta_0\theta \vdash_{\mathbb{S}} e_0 : \alpha\theta_0\theta^\star\theta$; by Lemma A.22, $\Gamma\theta \vdash_{\mathbb{S}} e_0 : \alpha\theta_0\theta^\star\theta$ (we prove the required premises below).

We take $\hat{t_0} = \alpha\theta_0\theta^\star\theta$. We have $\alpha\theta_0\theta^\star\theta \leq \bigvee_{i\in I} \lbag p_i \rbag$ because $\theta_0 \Vdash D'_0$ implies $\alpha\theta_0 \leq \bigvee_{i\in I} \lbag p_i \rbag$ and because subtyping is preserved by substitutions (recall that the accepted types of patterns are closed). We also have $\hat{t_i} = t_i\theta_0\theta^\star\theta$ for all i.

For each branch i, from

$$e_i : \beta \Rightarrow C'_i \qquad \Gamma, \mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0) \vdash C'_i \rightsquigarrow D'_i \qquad \theta \Vdash D'_i$$

we derive $\Gamma\theta, (\mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta \vdash_{\mathbb{S}} e_i : \beta\theta$ by the induction hypothesis. We derive by Lemma A.22 $\Gamma\theta, \mathsf{gen}_{\Gamma\theta}(\hat{t_i} /\!/ p_i) \vdash_{\mathbb{S}} e_i : \beta\theta$ (we prove the premises below). Thus we have $\hat{t'_i} = \beta\theta$ for every branch; we apply *Ts-Match* to derive $\Gamma\theta \vdash_{\mathbb{S}} \texttt{match}\ e_0\ \texttt{with}\ (p_i \to e_i)_{i\in I} : \beta\theta$, then subsumption to derive $\Gamma\theta \vdash_{\mathbb{S}} \texttt{match}\ e_0\ \texttt{with}\ (p_i \to e_i)_{i\in I} : t\theta$.

Proof of $\Gamma\theta \vdash_{\mathbb{S}} e_0 : \alpha\theta_0\theta^\star\theta$ *from* $\Gamma\theta_0\theta \vdash_{\mathbb{S}} e_0 : \alpha\theta_0\theta^\star\theta$ To apply Lemma A.22, we must show

$$\Gamma\theta \sqsubseteq \Gamma\theta_0\theta \qquad \mathsf{mvar}(\Gamma\theta) \subseteq \mathsf{mvar}(\Gamma\theta_0\theta)\ .$$

To prove $\Gamma\theta \sqsubseteq \Gamma\theta_0\theta$, consider an arbitrary $(x : \forall A_x.\, t_x) \in \Gamma$. By α-renaming, we assume $A_x \sharp \theta, \theta_0$; then, we must prove $\forall A_x.\, t_x\theta \sqsubseteq \forall A_x.\, t_x\theta_0\theta$. For every γ, $\gamma\theta \simeq \gamma\theta_0\theta$ since $\theta \Vdash \mathsf{equiv}(\theta_0)$. Hence, $t_x\theta \simeq t_x\theta_0\theta$.

Since $t_x\theta \simeq t_x\theta_0\theta$ implies $\mathsf{mvar}(t_x\theta) = \mathsf{mvar}(t_x\theta_0\theta)$ by Lemma A.19, this also shows $\mathsf{mvar}(\Gamma\theta) \subseteq \mathsf{mvar}(\Gamma\theta_0\theta)$.

Proof of $\Gamma\theta, \mathsf{gen}_{\Gamma\theta}(\hat{t_i} /\!/ p_i) \vdash_{\mathbb{S}} e_i : \beta\theta$ *from* $\Gamma\theta, (\mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta \vdash_{\mathbb{S}} e_i : \beta\theta$ By Lemma A.22, we can prove the result by showing

$$\Gamma\theta, \mathsf{gen}_{\Gamma\theta}(\hat{t_i} /\!/ p_i) \sqsubseteq \Gamma\theta, (\mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta$$
$$\mathsf{mvar}(\Gamma\theta, \mathsf{gen}_{\Gamma\theta}(\hat{t_i} /\!/ p_i)) \subseteq \mathsf{mvar}(\Gamma\theta, (\mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta)\ .$$

The second condition is straightforward. For the first, we prove, for every $x \in \mathsf{capt}(p_i)$, $\mathsf{gen}_{\Gamma\theta}((\hat{t_i} /\!/ p_i)(x)) \sqsubseteq (\mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0(x)))\theta$. Let $\Gamma_i(x) = t_x$. Then, $\mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0(x)) = \forall A.\, t_x\theta_0$, where A is $\mathsf{var}(\alpha\theta_0) \setminus \mathsf{mvar}(\Gamma\theta_0)$ as defined above (not all variables in A appear in $t_x\theta_0$, but schemes are defined disregarding useless quantification). By α-renaming, we have $\mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0(x)) = \forall B.\, t_x\theta_0\theta^\star$ and, since $B \sharp \theta$, $(\mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0(x)))\theta = \forall B.\, t_x\theta_0\theta^\star\theta$.

Since $\hat{t_i} \leq \hat{t_0} = \alpha\theta_0\theta^\star\theta$ and since $\theta \circ \theta^\star \circ \theta_0 \Vdash C_i$ (because $\theta_0 \Vdash C_i$), by Lemma A.37 we have $(\hat{t_i} /\!/ p_i)(x) \leq t_x\theta_0\theta^\star\theta$. Then, $\mathsf{gen}_{\Gamma\theta}((\hat{t_i} /\!/ p_i)(x)) \sqsubseteq \forall B.\, t_x\theta_0\theta^\star\theta$ holds because all variables in B may be quantified when generalizing $(\hat{t_i} /\!/ p_i)(x)$, since no β_i appears in $\Gamma\theta$. □

Theorem A.40 (Completeness of constraint generation and rewriting)**.** *Let e be an expression, t a type, and Γ a type environment. Let θ be a type substitution such that $\Gamma\theta \vdash_{\mathbb{S}} e : t\theta$.*

Let $e\colon t \Rightarrow_A C$*, with* $A \sharp \Gamma, \mathsf{dom}(\theta)$*. There exist a type-constraint set* D*, a set of fresh type variables* A'*, and a type substitution* θ'*, with* $\mathsf{dom}(\theta') = A \cup A'$*, such that* $\Gamma \vdash C \rightsquigarrow_{A'} D$ *and* $(\theta \cup \theta') \Vdash D$*.*

Proof. By structural induction on e.

Case $e = x$ We have

$$x\colon t \Rightarrow_\varnothing \{x \dot{\leq} t\}$$
$$\Gamma\theta \vdash_{\mathbb{S}} x\colon t\theta \qquad (\Gamma\theta)(x) = \forall A_x.\, t_x \qquad \mathsf{dom}(\theta_x) \subseteq A_x \qquad t_x\theta_x\theta \leq t\theta\,.$$

Given $A_x = \{\alpha_1, \ldots, \alpha_n\}$, we pick a set $A' = \{\beta_1, \ldots, \beta_n\}$ of fresh variables. Let $\hat{\theta} = [\,{}^{\beta_i}/_{\alpha_i} \mid \alpha_i \in A_x\,]$. We have $\Gamma \vdash \{x \dot{\leq} t\} \rightsquigarrow_{A'} \{t_x\hat{\theta} \dot{\leq} t\}$.

We pick $\theta' = [\,{}^{\alpha_i\theta_x\theta}/_{\beta_i} \mid \beta_i \in A'\,]$. It remains to prove that $(\theta \cup \theta') \Vdash \{t_x\hat{\theta} \dot{\leq} t\}$, that is, that

$$t_x\hat{\theta}(\theta \cup \theta') \leq t(\theta \cup \theta') = t\theta$$

(the equality above holds because the variables in A' are fresh).

We prove $t_x\hat{\theta}(\theta \cup \theta') = t_x\theta_x\theta$ (from which we can conclude because $t_x\theta_x\theta \leq t\theta$). We prove it by showing $\gamma\hat{\theta}(\theta \cup \theta') = \gamma\theta_x\theta$ for every $\gamma \in \mathsf{var}(t_x)$. If $\gamma \in A_x$, then $\gamma = \alpha_i$ for some i. Then, $\gamma\hat{\theta} = \beta_i$ and $\gamma\hat{\theta}(\theta \cup \theta') = \alpha_i\theta_x\theta$. If $\gamma \notin A_x$, then $\gamma\hat{\theta}(\theta \cup \theta') = \gamma\theta$ (the variables θ' is defined on do not appear in t_x); likewise, $\gamma\theta_x\theta = \gamma\theta$ since θ_x is only defined on variables in A_x.

Case $e = c$ We have

$$c\colon t \Rightarrow_\varnothing \{c \dot{\leq} t\} \qquad \Gamma\theta \vdash_{\mathbb{S}} c\colon t\theta \qquad c \leq t\theta\,.$$

We have $\Gamma \vdash \{c \dot{\leq} t\} \rightsquigarrow_\varnothing \{c \dot{\leq} t\}$. Let $\theta' = [\,]$. We have $(\theta \cup \theta') \Vdash \{c \dot{\leq} t\}$ because $c\theta = c \leq t\theta$.

Case $e = \lambda x.\, e_1$ We have

$$\lambda x.\, e_1\colon t \Rightarrow_{A_1 \uplus \{\alpha,\beta\}} \{\mathtt{def}\ \{x\colon \alpha\}\ \mathtt{in}\ C_1, \alpha \to \beta \dot{\leq} t\} \qquad e_1\colon \beta \Rightarrow_{A_1} C_1 \qquad A_1, \alpha, \beta \sharp t$$
$$\Gamma\theta \vdash_{\mathbb{S}} \lambda x.\, e_1\colon t\theta \qquad \Gamma\theta, \{x\colon t_1\} \vdash_{\mathbb{S}} e_1\colon t_2 \qquad t_1 \to t_2 \leq t\theta\,.$$

Let $\theta^\star = \theta \cup [\,{}^{t_1}/_{\alpha}, {}^{t_2}/_{\beta}]$. Note that $\Gamma\theta^\star = \Gamma\theta$ and $t\theta^\star = t\theta$, because $\{\alpha_1, \alpha_2\} \sharp \Gamma, t$.

We have $(\Gamma, \{x\colon \alpha\})\theta^\star \vdash_{\mathbb{S}} e_1\colon \beta\theta^\star$, $e_1\colon \beta \Rightarrow_{A_1} C_1$, and $A_1 \sharp \mathsf{dom}(\theta^\star)$. By the induction hypothesis, therefore, $\Gamma, \{x\colon \alpha\} \vdash C_1 \rightsquigarrow_{A_1'} D_1$ and $(\theta^\star \cup \theta_1') \Vdash D_1$, for some D_1, A_1', θ_1' such that $\mathsf{dom}(\theta_1') = A_1 \cup A_1'$ and that the variables in A_1' are fresh.

$\Gamma, \{x\colon \alpha\} \vdash C_1 \rightsquigarrow_{A_1'} D_1$ implies $\Gamma \vdash \mathtt{def}\ \{x\colon \alpha\}\ \mathtt{in}\ C_1 \rightsquigarrow_{A_1'} D_1$. Hence, we have $\Gamma \vdash C \rightsquigarrow_{A_1'} D = D_1 \cup \{\alpha \to \beta \dot{\leq} t\}$. Let $\theta' = [\,{}^{t_1}/_{\alpha}, {}^{t_2}/_{\beta}] \cup \theta_1'$. It is defined on the correct domain and it solves the constraints, since it solves D_1 and since $(\alpha \to \beta)\theta' = t_1 \to t_2 \leq t\theta$.

Case $e = e_1\ e_2$ We have

$$e_1\ e_2\colon t \Rightarrow_{A_1 \uplus A_2 \uplus \{\alpha,\beta\}} C_1 \cup C_2 \cup \{\beta \dot{\leq} t\}$$
$$e_1\colon \alpha \to \beta \Rightarrow_{A_1} C_1 \qquad e_2\colon \alpha \Rightarrow_{A_2} C_2 \qquad A_1, A_2, \alpha, \beta \sharp t$$
$$\Gamma\theta \vdash_{\mathbb{S}} e_1\ e_2\colon t\theta \qquad \Gamma\theta \vdash_{\mathbb{S}} e_1\colon t_1 \to t_2 \qquad \Gamma\theta \vdash_{\mathbb{S}} e_2\colon t_1 \qquad t_2 \leq t\theta\,.$$

Let $\theta^\star = \theta \cup [\,{}^{t_1}/_{\alpha}, {}^{t_2}/_{\beta}]$. Note that $\Gamma\theta^\star = \Gamma\theta$ and $t\theta^\star = t\theta$, since $\alpha, \beta \sharp \Gamma, t$.

We have $\Gamma\theta \vdash_{\mathbb{S}} e_1\colon (\alpha \to \beta)\theta^\star$, $e_1\colon \alpha \to \beta \Rightarrow_{A_1} C_1$, and $A_1 \sharp \mathsf{dom}(\theta^\star)$. By the induction hypothesis, therefore, $\Gamma \vdash C_1 \rightsquigarrow_{A_1'} D_1$ and $(\theta^\star \cup \theta_1') \Vdash D_1$, for some D_1 and θ_1' with $\mathsf{dom}(\theta_1') = A_1 \cup A_1'$.

Likewise, by applying the induction hypothesis to the derivation for e_2, we derive $\Gamma \vdash C_2 \rightsquigarrow_{A_2'} D_2$ and $(\theta^\star \cup \theta_2') \Vdash D_2$, for some D_2 and θ_2' with $\mathsf{dom}(\theta_2') = A_2 \cup A_2'$.

We can thus conclude that $\Gamma \vdash C \rightsquigarrow_{A_1' \cup A_2'} D = D_1 \cup D_2 \cup \{\beta \dot{\leq} t\}$. Let $\theta' = [\,{}^{t_1}/_{\alpha}, {}^{t_2}/_{\beta}] \cup \theta_1' \cup \theta_2'$. It is defined on the correct domain and $\theta \cup \theta'$ solves the constraints: it solves both D_1 and D_2, and $\beta(\theta \cup \theta') = \beta\theta' = t_2 \leq t\theta = t(\theta \cup \theta')$.

Case $e = (e_1, e_2)$ We have

$$(e_1, e_2)\colon t \Rightarrow_{A_1 \uplus A_2 \uplus \{\alpha_1,\alpha_2\}} C_1 \cup C_2 \cup \{\alpha_1 \times \alpha_2 \dot{\leq} t\}$$
$$e_1\colon \alpha_1 \Rightarrow_{A_1} C_1 \qquad e_2\colon \alpha_2 \Rightarrow_{A_2} C_2 \qquad A_1, A_2, \alpha_1, \alpha_2 \sharp t$$
$$\Gamma\theta \vdash_{\mathbb{S}} (e_1, e_2)\colon t\theta \qquad \Gamma\theta \vdash_{\mathbb{S}} e_1\colon t_1 \qquad \Gamma\theta \vdash_{\mathbb{S}} e_2\colon t_2 \qquad t_1 \times t_2 \leq t\theta\,.$$

Analogous to the previous case. We define $\theta^\star = \theta \cup [\,{}^{t_1}/_{\alpha_1}, {}^{t_2}/_{\alpha_2}]$ and proceed as above.

Case $e = \text{`}tag(e_1)$ We have

$$\text{`}tag(e_1) : t \Rightarrow_{A_1 \uplus \{\alpha\}} C_1 \cup \{\text{`}tag(\alpha) \dot{\leq} t\} \qquad e_1 : \alpha \Rightarrow_{A_1} C_1 \qquad A_1, \alpha \sharp t$$
$$\Gamma\theta \vdash_{\mathbb{S}} \text{`}tag(e) : t\theta \qquad \Gamma\theta \vdash_{\mathbb{S}} e_1 : t_1 \qquad \text{`}tag(t_1) \leq t\theta \,.$$

Analogous to the two previous cases. Here we define $\theta^\star = \theta \cup [t_1/\alpha]$.

Case $e = \texttt{match}\ e_0\ \texttt{with}\ (p_i \to e_i)_{i \in I}$ We have

$$\begin{array}{l}
\texttt{match}\ e_0\ \texttt{with}\ (p_i \to e_i)_{i\in I} : t \Rightarrow_A \{\texttt{let}\ [C_0'](\Gamma_i\ \texttt{in}\ C_i'), \beta \dot{\leq} t\} \\
\quad e_0 : \alpha \Rightarrow_{A_0} C_0 \qquad t_i = (\alpha \setminus \bigvee_{j<i} \lfloor p_j \rfloor) \wedge \lfloor p_i \rfloor \\
\quad \forall i \in I \qquad t_i /\!/\!/ p_i \Rightarrow_{A_i} (\Gamma_i, C_i) \qquad e_i : \beta \Rightarrow_{A_i'} C_i' \\
\quad C_0' = C_0 \cup (\bigcup_{i\in I} C_i) \cup \{\alpha \dot{\leq} \bigvee_{i\in I} \lfloor p_i \rfloor\} \\
\quad A = A_0 \uplus (\biguplus_{i\in I} A_i) \uplus (\biguplus_{i\in I} A_i') \uplus \{\alpha, \beta\} \\
\Gamma\theta \vdash_{\mathbb{S}} \texttt{match}\ e_0\ \texttt{with}\ (p_i \to e_i)_{i\in I} : t\theta \\
\quad \Gamma\theta \vdash_{\mathbb{S}} e_0 : t_0 \qquad t_0 \leq \bigvee_{i\in I} \lfloor p_i \rfloor \qquad t_i^\star = (t_0 \setminus \bigvee_{j<i} \lfloor p_j \rfloor) \wedge \lfloor p_i \rfloor \\
\quad \forall i \in I \qquad \Gamma\theta, \mathsf{gen}_{\Gamma\theta}(t_i^\star /\!/ p_i) \vdash_{\mathbb{S}} e_i : t_i' \qquad t' = \bigvee_{i\in I} t_i' \leq t\theta \,.
\end{array}$$

Let $\theta^\star = \theta \cup [t_0/\alpha]$. Then we have

$$e_0 : \alpha \Rightarrow_{A_0} C_0 \qquad \Gamma\theta^\star \vdash_{\mathbb{S}} e_0 : \alpha\theta^\star \qquad A_0 \sharp \Gamma, \mathsf{dom}(\theta^\star)$$

and, by the induction hypothesis, we find D_0, A_0' (containing fresh variables), and θ_0' such that

$$\Gamma \vdash C_0 \rightsquigarrow_{A_0'} D_0 \qquad \theta^\star \cup \theta_0' \Vdash D_0 \qquad \mathsf{dom}(\theta_0') = A_0 \cup A_0' \,.$$

From $\Gamma \vdash C_0 \rightsquigarrow_{A_0'} D_0$ we can derive

$$\Gamma \vdash C_0' \rightsquigarrow_{A_0'} D_0' = D_0 \cup (\bigcup_{i\in I} C_i) \cup \{\alpha \dot{\leq} \bigvee_{i\in I} \lfloor p_i \rfloor\}$$

because subtyping constraints are always rewritten to themselves.

For each branch i, note that $t_i\theta^\star = t_i^\star$. By Lemma A.38, we can find $\theta_i^\star$ such that

$$\mathsf{dom}(\theta_i^\star) = A_i \qquad \theta^\star \cup \theta_i^\star \Vdash C_i \qquad \forall x \in \mathsf{capt}(p_i).\ \Gamma(x)(\theta^\star \cup \theta_i^\star) \leq (t_i^\star /\!/ p_i)(x) \,.$$

Note also that $\theta^\star \Vdash \alpha \dot{\leq} \bigvee_{i\in I} \lfloor p_i \rfloor$. We therefore have $\theta^\star \cup \theta_0' \cup (\bigcup_{i\in I} \theta_i^\star) \Vdash D_0'$. Let $\theta^{\star\star} = \theta^\star \cup \theta_0' \cup (\bigcup_{i\in I} \theta_i^\star)$.

By the properties of tallying, if $\mathsf{var}(D_0') = \{\alpha_1, \ldots, \alpha_n\}$ and given a set $B = \{\alpha_1', \ldots, \alpha_n'\}$ of fresh variables, there exist two substitutions $\theta_0 \in \mathsf{tally}(D_0')$ and θ_0'' such that

$$\mathsf{dom}(\theta_0) = \mathsf{var}(D_0') \qquad \mathsf{var}(\theta_0) = B \qquad \mathsf{dom}(\theta_0'') = B$$
$$\forall \gamma \notin \mathsf{var}(\theta_0).\ \gamma\theta_0(\theta^{\star\star} \cup \theta_0'') \simeq \gamma\theta^{\star\star} \,.$$

Let $\theta^\top = \theta^{\star\star} \cup [t'/\beta] \cup \theta_0''$. To apply the induction hypothesis for a branch i, we need

$$e_i : \beta \Rightarrow_{A_i'} C_i' \qquad (\Gamma, \mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta^\top \vdash_{\mathbb{S}} e_i : \beta\theta^\top \qquad A_i' \sharp \Gamma, \mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0), \mathsf{dom}(\theta^\top) \,.$$

We derive the typing judgment above by subsumption and by weakening (we prove the premises below). As for the freshness condition, note that the variables in $\Gamma\theta_0$ are all either in Γ or in $\mathsf{var}(\theta_0)$; in the latter case, they are fresh by our choice of B.

By applying the induction hypothesis to each branch i, we therefore find D_i, A_i'' (of fresh variables), and θ_i' such that

$$\Gamma, \mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0) \vdash C_i' \rightsquigarrow_{A_i''} D_i \qquad \theta^\top \cup \theta_i' \Vdash D_i \qquad \mathsf{dom}(\theta_i') = A_i' \cup A_i'' \,.$$

Hence, we have

$$\Gamma\theta \vdash \{\texttt{let}\ [C_0'](\Gamma_i\ \texttt{in}\ C_i'), \beta \dot{\leq} t\} \rightsquigarrow_{A'} \mathsf{equiv}(\theta_0) \cup (\bigcup_{i\in I} D_i) \cup \{\beta \dot{\leq} t\} \,,$$

where $A' = A_0' \cup (\bigcup_{i\in I} A_i'') \cup \mathsf{var}(\theta_0)$.

We take $\theta' = [t_0/\alpha, {}^{t'}/\beta] \cup \theta_0' \cup (\bigcup_{i\in I} \theta_i^\star) \cup \theta_0'' \cup (\bigcup_{i\in I} \theta_i')$. It has the correct domain; we must only show

$$\theta \cup \theta' \Vdash \mathsf{equiv}(\theta_0) \cup (\bigcup_{i\in I} D_i) \cup \{\beta \dot{\leq} t\} \,.$$

The last constraint is satisfied since $\beta(\theta \cup \theta') = t' \leq t\theta$. Constraints in $\mathsf{equiv}(\theta_0)$ are of the form $\alpha \leq \alpha\theta_0$ or $\alpha\theta_0 \leq \alpha$, for $\alpha \in \mathsf{dom}(\theta_0)$. Since these α are not in $\mathsf{var}(\theta_0)$, we have $\alpha\theta_0(\theta^{\star\star} \cup \theta_0'') \simeq \alpha\theta^{\star\star}$ and hence $\alpha\theta_0(\theta \cup \theta') \simeq \alpha(\theta \cup \theta')$. For each i, since $\theta^\top \cup \theta_i' \Vdash D_i$, we have also $\theta \cup \theta' \Vdash D_i$ (the other substitutions we add are not defined on the variables in D_i).

Proof of $(\Gamma, \mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta^\top \vdash_{\mathbb{S}} e_i : \beta\theta^\top$ *from* $\Gamma\theta, \mathsf{gen}_{\Gamma\theta}(t_i^\star /\!/ p_i) \vdash_{\mathbb{S}} e_i : t_i'$ From $\Gamma\theta, \mathsf{gen}_{\Gamma\theta}(t_i^\star /\!/ p_i) \vdash_{\mathbb{S}} e_i : t_i'$, we derive $\Gamma\theta, \mathsf{gen}_{\Gamma\theta}(t_i^\star /\!/ p_i) \vdash_{\mathbb{S}} e_i : \beta\theta^\top$ by subsumption, since $t_i' \leq t' = \beta\theta^\top$. We

then apply Lemma A.22, which requires us to show the two premises

$$(\Gamma, \mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta^\top \sqsubseteq \Gamma\theta, \mathsf{gen}_{\Gamma\theta}(t_i^\star /\!/ p_i)$$
$$\mathsf{mvar}((\Gamma, \mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta^\top) \subseteq \mathsf{mvar}(\Gamma\theta, \mathsf{gen}_{\Gamma\theta}(t_i^\star /\!/ p_i)) \,.$$

Note that $\Gamma\theta = \Gamma\theta^\top$ since the two substitutions only differ on variables introduced by constraint generation or tallying. Simplifying, we need to show

$$\mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0)\theta^\top \sqsubseteq \mathsf{gen}_{\Gamma\theta}(t_i^\star /\!/ p_i) \qquad \mathsf{mvar}((\mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta^\top) \subseteq \mathsf{mvar}(\Gamma\theta) \,.$$

To prove the former, consider $x \in \mathsf{capt}(p_i)$ and let $\Gamma_i(x) = t_x$. We must show $\mathsf{gen}_{\Gamma\theta_0}(t_x\theta_0)\theta^\top \sqsubseteq \mathsf{gen}_{\Gamma\theta}((t_i^\star /\!/ p_i)(x))$. We have

$$\mathsf{gen}_{\Gamma\theta_0}(t_x\theta_0) = \forall B_x.\, t_x\theta_0 \qquad B_x = \mathsf{var}(t_x\theta_0) \setminus \mathsf{mvar}(\Gamma\theta_0) \,.$$

Note that all variables in $\mathsf{var}(t_x\theta_0)$ are in $\mathsf{var}(\theta_0)$: this is because all variables in $\mathsf{var}(t_x)$ occur in D_0' (α occurs in the exhaustiveness constraint, variables introduced by pattern environment generation occur in C_i) and hence are in the domain of θ_0. Then, $B_x \subseteq B$: its elements are some of the α_i' in B. Consider a set $B_x' = \{\, \alpha_i'' \mid \alpha_i' \in B_x \,\}$ of fresh variables and the renaming $\tilde\theta = [\,{}^{\alpha_i''}/_{\alpha_i'} \mid \alpha_i' \in B_x\,]$: we have

$$\mathsf{gen}_{\Gamma\theta_0}(t_x\theta_0) = \forall B_x'.\, t_x\theta_0\tilde\theta \qquad B_x = \mathsf{var}(t_x\theta_0) \setminus \mathsf{mvar}(\Gamma\theta_0)$$

and, since the variables in B_x' are fresh,

$$(\mathsf{gen}_{\Gamma\theta_0}(t_x\theta_0))\theta^\top = \forall B_x'.\, t_x\theta_0\tilde\theta\theta^\top \,.$$

Consider an arbitrary instance $(t_i^\star /\!/ p_i)(x)\hat\theta$ of $\mathsf{gen}_{\Gamma\theta}((t_i^\star /\!/ p_i)(x))$; we have $\mathsf{dom}(\hat\theta) \subseteq \mathsf{var}((t_i^\star /\!/ p_i)(x)) \setminus \mathsf{mvar}(\Gamma\theta)$. We must show that there exists an instance of $(\mathsf{gen}_{\Gamma\theta_0}(t_x\theta_0))\theta^\top$ which is a subtype of it. We take the instance $t_x\theta_0\tilde\theta\theta^\top\check\theta$, with $\check\theta = [\,{}^{\beta_i\theta^\top\hat\theta}/_{\beta_i'} \mid \beta_i' \in B_x'\,]$. We have $t_x\theta_0\tilde\theta\theta^\top\check\theta = t_x\theta_0\theta^\top\hat\theta$: for each $\alpha_i' \in \mathsf{var}(t_x\theta_0)$,

$$\alpha_i'\tilde\theta\theta^\top\check\theta = \alpha_i''\theta^\top\check\theta = \alpha_i''\check\theta$$

(since all α_i'' are fresh), and $\alpha_i''\check\theta = \alpha_i'\theta^\top\hat\theta$. We have $t_x\theta_0\theta^\top\hat\theta \simeq t_x\theta^\top\hat\theta$ and $t_x\theta^\top\hat\theta \leq (t_i^\star /\!/ p_i)(x)\hat\theta$ since $t_x\theta^\top \leq (t_i^\star /\!/ p_i)(x)$.

As for the condition on variables, we have $\mathsf{mvar}((\mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta^\top) \subseteq \mathsf{var}((\mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta^\top)$. Since $\mathsf{var}(\mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0)) \subseteq \mathsf{mvar}(\Gamma\theta_0)$, $\mathsf{var}((\mathsf{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta^\top) \subseteq \mathsf{mvar}(\Gamma\theta_0\theta^\top) = \mathsf{mvar}(\Gamma\theta)$. □

A.5 Extensions

We give full definitions for the three variants of the $\mathbb{S}$ system that we have sketched in Section 6.

A.5.1 Overloaded functions

To remove the restriction on the use of intersection types for functions, we change the typing rule *Ts-Abstr*: we allow the derivation of an intersection of arrow types for a λ-abstraction if each of these types is derivable. The modified rule is the following.

$$\textit{Ts-Abstr}\ \frac{\forall j \in J.\ \Gamma, \{x\colon t_j'\} \vdash e\colon t_j}{\Gamma \vdash \lambda x.\, e\colon\ \bigwedge_{j\in J} t_j' \to t_j}$$

Furthermore, we change the typing rule for pattern matching so that redundant branches are excluded from typing. This is necessary to use intersections effectively for pattern matching: in practice, to be able to assign to a function defined by pattern matching one arrow type for each branch.

$$\textit{Ts-Match}\ \frac{\Gamma \vdash_{\mathbb{S}} e_0\colon t_0 \qquad t_0 \leq \bigvee_{i\in I} \lfloor p_i \rfloor \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lfloor p_j \rfloor) \wedge \lfloor p_i \rfloor \qquad \forall i \in I\ \begin{cases} t_i' = \mathbb{0} & \text{if } t_i \leq \mathbb{0} \\ \Gamma, \mathsf{gen}_\Gamma(t_i /\!/ p_i) \vdash_{\mathbb{S}} e_i\colon t_i' & \text{otherwise} \end{cases}}{\Gamma \vdash_{\mathbb{S}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i\in I}\colon\ \bigvee_{i\in I} t_i'}$$

Finally, we also change the rule *Ts-Var* for variables: we allow a variable to be typed with any intersection of instantiations, rather than just with a single instantiation.

$$\textit{Ts-Var}\ \frac{\forall i \in I.\, t_i \in \mathsf{inst}(\Gamma(x))}{\Gamma \vdash_{\mathbb{S}} x\colon\ \bigwedge_{i\in I} t_i}$$

This allows us to instantiate type schemes which express parametric polymorphism (for instance, $\forall\alpha.\, \alpha \to \alpha$) into types which express ad hoc polymorphism (e.g., $(\mathtt{bool} \to \mathtt{bool}) \wedge (\mathtt{int} \to \mathtt{int})$).

A.5.2 Refining the type of expressions in pattern matching

The extension we present here improves the typing of pattern matching by introducing more precise types for some variables in the matched expression when typing the branches. These refined types take into account which patterns have been selected and which have not; they are introduced for variables that appear in the matched expression, possibly below pairs or variant constructors, but not inside applications or `match` constructs.

We reuse pattern environment generation to describe the derivation of these refined types. However, we need to introduce a new production for patterns to use when translating expressions to patterns:

$$p ::= \cdots \mid \langle p, p \rangle \ .$$

Patterns of the form $\langle p, p \rangle$ should not occur in programs; they are only for internal use in the type system. Unlike normal pair patterns, these patterns may include repeated variables.

We need not define the dynamic semantics of these patterns, as it won't be used. We define their accepted type as $\wr\langle p_1, p_2 \rangle\int = \wr p_1\int \times \wr p_2\int$ and environment generation as

$$t//\langle p_1, p_2 \rangle = \pi_1(t)//p_1 \between \pi_2(t)//p_2 \ ,$$

where $\between$, defined as

$$(\Gamma \between \Gamma')(x) = \begin{cases} \Gamma(x) & \text{if } x \in \mathsf{dom}(\Gamma) \setminus \mathsf{dom}(\Gamma') \\ \Gamma'(x) & \text{if } x \in \mathsf{dom}(\Gamma') \setminus \mathsf{dom}(\Gamma) \\ \Gamma(x) \wedge \Gamma'(x) & \text{if } x \in \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') \end{cases}$$

is the pointwise intersection of type environments.

We define a translation $(\!|\cdot|\!)$ of expressions to patterns. It preserves variables and variants, converts pairs to the new form, and turns everything else into a wildcard.

$$(\!|e|\!) = \begin{cases} x & \text{if } e = x \\ \langle (\!|e_1|\!), (\!|e_2|\!) \rangle & \text{if } e = (e_1, e_2) \\ \text{`}tag((\!|e_1|\!)) & \text{if } e = \text{`}tag(e_1) \\ \_ & \text{otherwise} \end{cases}$$

We change the typing rule for pattern matching as follows.

$$\textit{Ts-Match}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathbb{S}} e_0 : t_0 \qquad t_0 \leq \bigvee_{i\in I} \wr p_i\int \wedge \wr (\!|e_0|\!)\int \qquad t_i = (t_0 \setminus \bigvee_{j<i} \wr p_j\int) \wedge \wr p_i\int \\ \forall i \in I \qquad \Gamma, \mathsf{gen}_\Gamma(t_i//(\!|e_0|\!)), \mathsf{gen}_\Gamma(t_i//p_i) \vdash_{\mathbb{S}} e_i : t'_i\end{array}}{\Gamma \vdash_{\mathbb{S}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i\in I} :\ \bigvee_{i\in I} t'_i}$$

The main difference is the addition of the type environment $\mathsf{gen}_\Gamma(t_i//(\!|e_0|\!))$ which provides the refined types for the variables in $(\!|e_0|\!)$. This environment is added before the usual one for the pattern p_i: hence, the capture variables of p_i still take precedence.

We also add the requirement $t_0 \leq \wr (\!|e_0|\!)\int$ to ensure $t_i//\wr e_0\int$ is well defined. This is not restrictive because any well-typed e can be typed with a subtype of $\wr (\!|e|\!)\int$.

A.5.3 Applicability to OCaml

We change the semantics of pattern matching to include undefined results. These occur when matching constants of different basic types or when matching different constructors (for instance, a constant and a pair). We use the following definition.

Definition A.47 (Semantics of pattern matching)**.** *We write v/p for the result of matching a value v against a pattern p. We have either $v/p = \varsigma$, where ς is a substitution defined on the variables in $\mathsf{capt}(p)$, $v/p = \Omega$, or $v/p = \mathfrak{V}$. In the first case, we say that v matches p (or that p accepts v); in the second, we say that matching fails; in the third, we say that it is undefined.*

The definition of v/p is given inductively in Figure 16.

Recall that the function $b_{(\cdot)}$ (used here for v/c) assigns a basic type b_c to each constant c.

The notions of reduction are unchanged, but the rule *R-Match* is made more restrictive by the changed definition of v/p: a `match` expression reduces only if matching succeeds for a branch and fails—but is never undefined—for all previous branches. The type system should therefore ensure that, in a well-typed expression $\mathtt{match}\ v\ \mathtt{with}\ (p_i \to e_i)_{i\in I}$, $v/p_i = \mathfrak{V}$ never happens. While this is true for $\mathbb{K}$, $\mathbb{S}$ has to be restricted to ensure this.

We first define the *compatible type* $\lceil p \rceil$ of a pattern p inductively as follows:

$$\begin{array}{rlrl} \lceil \_ \rceil &= \lceil x \rceil = \mathbb{1} & \lceil c \rceil &= b_c \\ \lceil (p_1, p_2) \rceil &= \lceil p_1 \rceil \times \lceil p_2 \rceil & \lceil \text{`}tag(p) \rceil &= \text{`}tag(\lceil p \rceil) \vee (\mathbb{1}_{\mathbb{V}} \setminus \text{`}tag(\mathbb{1})) \\ \lceil p_1 \& p_2 \rceil &= \lceil p_1 \rceil \wedge \lceil p_2 \rceil & \lceil p_1 | p_2 \rceil &= \lceil p_1 \rceil \vee \lceil p_2 \rceil \ , \end{array}$$

$$
\begin{aligned}
v/\_ &= [\,] \\
v/x &= [v/x] \\
v/c &= \begin{cases} [\,] & \text{if } v = c \\ \Omega & \text{if } v \in \mathcal{C},\ b_v = b_c, \text{ and } v \neq c \\ \mho & \text{otherwise} \end{cases} \\
v/(p_1, p_2) &= \begin{cases} \varsigma_1 \cup \varsigma_2 & \text{if } v = (v_1, v_2) \text{ and } \forall i.\ v_i/p_i = \varsigma_i \\ \Omega & \text{if } v = (v_1, v_2),\ \exists i.\ v_i/p_i = \Omega, \text{ and } \forall i.\ v_i/p_i \neq \mho \\ \mho & \text{otherwise} \end{cases} \\
v/\text{`}tag(p_1) &= \begin{cases} \varsigma_1 & \text{if } v = \text{`}tag(v_1) \text{ and } v_1/p_1 = \varsigma_1 \\ \Omega & \text{if } v = \text{`}tag(v_1) \text{ and } v_1/p_1 = \Omega \text{ or if } v = \text{`}tag_1(v_1) \text{ and } \text{`}tag_1 \neq \text{`}tag \\ \mho & \text{otherwise} \end{cases} \\
v/p_1 \& p_2 &= \begin{cases} \varsigma_1 \cup \varsigma_2 & \text{if } \forall i.\ v/p_i = \varsigma_i \\ \Omega & \text{if } \exists i.\ v/p_i = \Omega \text{ and } \forall i.\ v/p_i \neq \mho \\ \mho & \text{otherwise} \end{cases} \\
v/p_1 | p_2 &= \begin{cases} v/p_1 & \text{if } v/p_1 \neq \Omega \\ v/p_2 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 16. Semantics of pattern matching including undefined results.

where $\mathbb{1}_{\mathrm{v}}$ is the top type for variants, defined in Section 4.3, Footnote 5. For all well-typed values v, $\Gamma \vdash_{\mathbb{S}} v \colon \lceil p \rceil$ holds if and only if $v/p \neq \mho$.

We change the rule for pattern matching by requiring the type t_0 we assign to the matched expression to be a subtype of all compatible types $\lceil p_i \rceil$.

$$
\textit{Ts-Match}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathbb{S}} e_0 \colon t_0 \qquad t_0 \leq \bigvee_{i \in I} \lfloor p_i \rfloor \wedge \bigwedge_{i \in I} \lceil p_i \rceil \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lfloor p_j \rfloor) \wedge \lfloor p_i \rfloor \\ \forall i \in I \qquad \Gamma, \mathsf{gen}_\Gamma(t_i // p_i) \vdash_{\mathbb{S}} e_i \colon t_i' \end{array}}{\Gamma \vdash_{\mathbb{S}} \mathtt{match}\ e_0\ \mathtt{with}\ (p_i \to e_i)_{i \in I} \colon \bigvee_{i \in I} t_i'}
$$

Note that this condition is somewhat more restrictive than necessary: patterns which follow a catch-all (wildcard or variable) pattern—or in general that are useless because previous patterns already cover all cases—can be left out of the intersection. The precise condition would be

$$
t_0 \leq \bigvee_{i \in I} \left(\lfloor p_i \rfloor \wedge \bigwedge_{j<i} \lceil p_j \rceil \right),
$$

but we choose the simpler condition since they only differ in case there is redundancy.