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 (Garrigue 2002). 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.

ICFP'16, September 18–24, 2016, Nara, Japan
ACM. 978-1-4503-4219-3/16/09...$15.00
http://dx.doi.org/10.1145/2951913.2951928

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* (Frisch et al. 2008), 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

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

value `` `A `` (polymorphic variants values are literals prefixed by a backquote), OCaml statically deduces that the result will be of 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 ``([< `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 ]``.

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 Garrigue 2004).

```
# 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

[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.

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., [CAML-LIST 1] 2007; [CAML-LIST 2] 2000; [CAML-LIST 3] 2005; [CAML-LIST 4] 2004; Nicollet 2011; Wegrzanowski 2006).

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 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 the 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. (2015) 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. (2015) 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 (Benzaken et al. 2003; CDuce)). 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 contained in the extended version available online (Castagna et al. 2016).

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 $x, y, z, \ldots$, a set $\mathcal{C}$ of constants, ranged over by c, and a set $\mathcal{L}$ of tags, ranged over by `*tag*. Tags are used to label variant expressions.

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

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

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.

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: $\texttt{let}\ x = e_0\ \texttt{in}\ e_1 \overset{\text{def}}{=} \texttt{match}\ e_0\ \texttt{with}\ x \to e_1$.

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

Figure 1. Semantics of pattern matching.

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

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

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* $p_1|p_2$, $\mathsf{capt}(p_1) = \mathsf{capt}(p_2)$*, 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.*

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

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 $[{}^{v_1}/_{x_1}, \dots, {}^{v_n}/_{x_n}]$ 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

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

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 $\mathbb{k}$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 (Garrigue 2002, 2015). 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 $\alpha, \beta, \gamma, \dots$. 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* "•" 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 $\{`tag_1, \ldots, `tag_n\}$;*
- *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 $\{`tag_1 \colon \tau_1, \ldots, `tag_n \colon \tau_n\}$ (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 $`tag \in L$, whenever both $`tag \colon \tau_1 \in T$ and $`tag \colon \tau_2 \in T$, 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 $(\{\texttt{`A}, \texttt{`B}\}, \mathcal{L}, \{\texttt{`A: int}, \texttt{`B: bool}\})$; `[< `A of int | `B of bool ] as` β corresponds to β of kind $(\varnothing, \{\texttt{`A}, \texttt{`B}\}, \{\texttt{`A: int}, \texttt{`B: bool}\})$; finally `[< `A of int | `B of bool & unit > `A] as` $\gamma$ corresponds to $\gamma$ of kind $(\{\texttt{`A}\}, \{\texttt{`A}, \texttt{`B}\}, \{\texttt{`A: int}, \texttt{`B: bool}, \texttt{`B: unit}\})$.

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.

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

The typing of variant expressions uses the kinding environment. Rule *Tk-Tag* states that $`tag(e)$ 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 $`tag \in L$ and $`tag \colon \tau \in T$, where $\tau$ is a type for $e$. Note that $T$ may not assign more than one type to $`tag$, since $`tag \in L$.

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 $`tag(p)$

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

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 $`tag$.

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 (2004) and Maranget (2007).

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 $e \leadsto e'$.*

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 $e \leadsto e'$, 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 (2011). The use of semantic subtyping makes the definition of a typing algorithm challenging: Castagna et al. (2014, 2015) 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. (2014, 2015), 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, $\beta, \gamma, \dots$) 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:

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

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 $(\{$`` `A ``$\}, \{$`` `A ``, `` `B ``$\}, \{$`` `A: bool ``, `` `B: 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 $t_{\text{L}} = \grave{}\mathtt{A}(\mathtt{bool})$ and the upper one as $t_{\text{U}} = \grave{}\mathtt{A}(\mathtt{bool}) \vee \grave{}\mathtt{B}(\mathtt{int})$. In our system, α should be a variable with bounded quantification, which can only be instantiated by types t such that $t_{\text{L}} \leq t \leq t_{\text{U}}$.

However, we do not need to introduce bounded quantification as a feature of our language: we can use type connectives to encode it as proposed by Castagna and Xu (2011, cf. Footnote 4 therein). The possible instantiations of α (with the bounds above) and the possible instantiations of $(t_{\text{L}} \vee \beta) \wedge t_{\text{U}}$, 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 (2011), 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 (2015) for a detailed explanation.

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

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

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

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

In brief, subtyping is given a semantic definition, in the sense that $t_1 \leq t_2$ 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 (2011) that subtyping is preserved by type substitutions: $t_1 \leq t_2$ implies $t_1\theta \leq t_2\theta$ 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. (2014, 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

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

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 $\llparenthesis p \rrparenthesis$ *of a pattern p is defined inductively as:*

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

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

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

$$
\text{where:} \quad \mathsf{low}_K(L,T) = \textstyle\bigvee_{\grave{}tag \in L} \grave{}tag(\bigwedge_{\grave{}tag\colon \tau \in T} [\![\tau]\!]_K)
$$

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

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

$t_0 \leq \bigvee_{i \in I} \lfloor p_i \rfloor$ 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 $\lfloor p_j \rfloor$ for $j < i$ (i.e., not captured by any previous pattern), and in $\lfloor p_i \rfloor$ (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 $t = (\alpha \times \beta) \vee (\texttt{bool} \times \texttt{int})$, we have $\pi_1(t) = \alpha \vee \texttt{bool}$ and $\pi_2(t) = \beta \vee \texttt{int}$. Given any tag $\grave{}tag$, $\pi_{\grave{}tag}$ does likewise for variant types with that tag. See Castagna et al. (2014, Appendix C.2.1) and Petrucciani (2015) for the full details.

Definition 4.4 (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:*

$$
\begin{aligned}
t /\!/ \_ &= \varnothing & t /\!/ (p_1, p_2) &= \pi_1(t) /\!/ p_1 \cup \pi_2(t) /\!/ p_2 \\
t /\!/ x &= \{x\colon t\} & t /\!/ \grave{}tag(p) &= \pi_{\grave{}tag}(t) /\!/ p \\
t /\!/ c &= \varnothing & t /\!/ p_1 \& p_2 &= t /\!/ p_1 \cup t /\!/ p_2
\end{aligned}
$$

$$
t /\!/ p_1 | p_2 = (t \wedge \lfloor p_1 \rfloor) /\!/ p_1 \ \mathbin{\vee\!\!\vee} \ (t \setminus \lfloor p_1 \rfloor) /\!/ p_2
$$

where $(\Gamma \mathbin{\vee\!\!\vee} \Gamma')(x) = \Gamma(x) \vee \Gamma'(x)$.

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 $[\![\forall A.\, K' \triangleright \tau]\!]_K = \forall A.\, [\![\tau]\!]_{K,K'}$ 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 (2002, 2015)—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 $(\{\texttt{`A}\}, \{\texttt{`A}, \texttt{`B}\}, \{\texttt{`A: int}, \texttt{`B: unit}\})$, is translated into $(\texttt{`A(int)} \vee \alpha) \wedge (\texttt{`A(int)} \vee \texttt{`B(unit)})$.

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

$$
\textstyle\bigvee_{\grave{}tag \in \mathsf{dom}(T)} \grave{}tag(\bigwedge_{\grave{}tag\colon \tau \in T} [\![\tau]\!]_K) \ \vee \ \bigvee_{\grave{}tag \notin \mathsf{dom}(T)} \grave{}tag(\mathbb{1})
$$

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 $\mathbb{1}_{\mathsf{V}}$.[5]

In practice, a type $(t_{\mathsf{L}} \vee \alpha) \wedge t_{\mathsf{U}}$ 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 $\mathbb{1}_{\mathsf{V}}$ can itself be defined by complementation as

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

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

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

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

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

Figure 4. Constraint generation rules.

Theorem 4.4 (Preservation of typing)**.** *Let e be an expression, K a non-recursive kinding environment, Γ a $\mathbb{k}$-type environment, and τ a $\mathbb{k}$-type. If $K; \Gamma \vdash_{\mathbb{K}} e\colon \tau$, 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 judgments 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. (2015), 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. (2015) 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. (2015).

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) (Pottier and Rémy 2005): 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. (2015). 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

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

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

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.

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

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 $\pi_{\grave{}\mathit{tag}}$ 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 $\alpha \mathrel{\dot{\leq}} \bigvee_{i \in I} \lbag p_i \rbag$, 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 $\{t_1 \mathrel{\dot{\leq}} t'_1, \dots, t_n \mathrel{\dot{\leq}} t'_n\}$ (i.e., no `let`, `def`, or expression variables). 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 $\beta_1, \dots, \beta_n$, 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 non-deterministic. 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

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

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. (2015) 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 $t_1 \times t'_1 \mathrel{\dot{\leq}} t_2 \times t'_2$ would simplify to the conjunction of two constraints $t_1 \mathrel{\dot{\leq}} t_2$ and $t'_1 \mathrel{\dot{\leq}} t'_2$. With semantic subtyping—where products are seen as Cartesian products—that simplification is sound, but it is not the only possible choice: either $t_1 \mathrel{\dot{\leq}} \mathbb{0}$ or $t'_1 \mathrel{\dot{\leq}} \mathbb{0}$ is also enough to ensure $t_1 \times t'_1 \mathrel{\dot{\leq}} t_2 \times t'_2$, since both ensure $t_1 \times t'_1 \simeq \mathbb{0}$. The three possible choices can produce incomparable solutions.

Castagna et al. (2015, 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 $(\alpha \to \beta) \to \alpha\,$`list`$\,\to \beta\,$`list`). The application of `map` to the successor function `succ : int → int` has type `int list → 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

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

with the type $(\mathtt{true} \to \mathtt{false}) \wedge (\mathtt{false} \to \mathtt{true})$, which is more precise than $\mathtt{bool} \to \mathtt{bool}$. 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 (Reynolds 1997). With this rule alone, however, one has only the so-called *coherent overloading* (Pierce 1991), 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 (Frisch et al. 2002; Benzaken et al. 2003) 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 $\mathtt{true} \to \mathtt{false}$ nor as $\mathtt{false} \to \mathtt{true}$.

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 $t'_j = \mathtt{true}$, we can derive $t_j = \mathtt{false}$ (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 $\lambda x.\,\mathtt{match}\ x\ \mathtt{with}$ `` `A `` $\to$ `id2` $x \mid \_ \to x$, where `id2` has domain `` `A `` $\vee$ `` `B ``. Like OCaml, $\mathbb{S}$ requires the type of x to be a subtype of `` `A `` $\vee$ `` `B ``, but this constraint is unnecessary because `id2` x is only computed when $x =$ `` `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* (Tobin-Hochstadt and Felleisen 2010) or *flow typing* (Pearce 2013).

We first consider pattern matching on a variable. In an expression $\mathtt{match}\,x\ \mathtt{with}\ (p_i \to e_i)_{i\in I}$ 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 $t_0 \wedge$ `` `A ``, where t_0 is the type we have derived for x. As a result, the constraint $t_0 \wedge$ `` `A `` $\leq$ `` `A `` $\vee$ `` `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 $(\!|`tag(e)|\!) = `tag((\!|e|\!))$. 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

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

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, $\texttt{1}/\texttt{true} = \texttt{1}/(\texttt{true}, \texttt{true}) = \mho$ (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, $\lceil \texttt{1} \rceil = \texttt{int}$. 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 (2002, 2015). There exist several other systems with structural polymorphism: for instance, the earlier one by Garrigue (1998) and more expressive constraint-based frameworks, like the presentation of HM(X) by Pottier and Rémy (2005). 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 (2002, 2015), $\mathbb{K}$ 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 (2004) 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 (2002) 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 Garrigue 2004, Section 4.1).

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

$\mathbb{S}$ reuses the subtyping relation defined by Castagna and Xu (2011) and some of the work described in Castagna et al. (2014, 2015) (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. (2014, 2015) 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. (2014, 2015) 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 (2005) and Dunfield (2007) 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. (2014) 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 (Tobin-Hochstadt and Felleisen 2008, 2010) 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. (2015) whose Section 6 contains a thorough comparison with the type system of Typed Racket.[8] 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.

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., Nicollet 2011; Wegrzanowski 2006). The solution can be even more satisfactory if we extend the current syntax of OCaml types. For instance, Nicollet (2011) shows the OCaml function `function `A` $\to$ `` `B | x`` $\to$ `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`)` $\to \alpha$. Our reconstruction algorithm deduces instead the type $\alpha \to$ `` (`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 [CAML-LIST 1] 2007; [CAML-LIST 4] 2004). If we want to preserve the current syntax of OCaml types, this type should be approximated as `([> `B] as` $\alpha$`)` $\to \alpha$; 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 (Blume et al. 2008). 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`` $\to$ `` `B) & ((``α``\`A)`` $\to$ ``(``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. (2008) (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. (2015) 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. (2014)): the subtyping relation of Castagna and Xu (2011) 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 are very grateful to Jacques Garrigue who gave invaluable feedback on an early version of this work. This work was partially supported by the Oracle Labs External Research Project "Deep Integration of Programming Languages in Databases with Truffle".

[8] Actually, ℂDuce local type inference is more general than the one in Typed Racket, insofar as in an application it locally infers the instantiation for both the function and the argument while Typed Racket does only the former.

References

V. Benzaken, G. Castagna, and A. Frisch. ℂDuce: an XML-centric general-purpose language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 51–63, 2003.

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.

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.

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.

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.

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)*, pages 289–302, 2015.

G. Castagna, T. Petrucciani, and K. Nguyen. Set-theoretic types for polymorphic variants. Technical report, Université Paris Diderot, May 2016. Extended version. `https://hal.archives-ouvertes.fr/view/index/docid/1325644`.

CDuce. `http://www.cduce.org`.

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

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

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

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

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

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

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.

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.

J. Garrigue. Programming with polymorphic variants. In *ACM SIGPLAN Workshop on ML*, 1998. Informal proceedings.

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

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

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

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

V. Nicollet. Do variant types in OCaml suck? `http://goo.gl/F0Owal`, Mar. 2011. Blog post.

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.

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

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.

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.

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.).

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

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

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