CONSTRUCTOR UNIVERSITY

Implementation of the Set-Theoretic Types for Polymorphic Variants in OCaml

by

Roman Venediktov

Bachelor Thesis in Computer Science

Submission: May 16, 2023

Supervisor: Anton Podkopaev
Industry Advisor: Daniil Berezun
Industry Advisor: Dmitrii Kosarev

Constructor University | School of Computer Science and Engineering

Abstract

Polymorphic variants are a useful feature of the OCaml language, but their current implementation relies on kinding constraints to simulate subtyping via unification, resulting in a complex formalization and a type system that often displays unintuitive or excessively restrictive behaviors. A more expressive system, grounded in set-theoretic types with semantic subtyping, has been shown to be both more precise and intuitive across numerous examples. However, it has not yet been implemented in the OCaml compiler.

This work introduces a formalization and implementation of a simplified typing system rooted in set-theoretic types within the OCaml compiler. Our implementation serves as a proof of concept for a hybrid type inference system that marries the traditional Hindley-Milner algorithm with a system that integrates semantic subtyping and constraint solving. The system we propose presents a more intuitive and flexible approach to the typing of polymorphic variants in OCaml.

Contents

1 Introduction

1.1 Overview of the Polymorphic Variants

A polymorphic variant is a pair that consists of a tag and its associated value. In the code snippets, the interactive top-level prompt of OCaml is denoted by "#", which is terminated by a double semicolon, followed by the system response.

```
# let v = `A 12;;
val v : [> `A of int ] = `A 12
```

This is the declaration of a polymorphic variant value with the tag `A` and the associated value of type `int`. The tag or constructor name is stored at runtime (as a hash) and can be used to pattern match on it.

```
# match v with `A x -> x + 1 | `B x -> x - 1;;
- : int = 13
```

The primary distinction between normal and polymorphic variants lies in the fact that the constructors of polymorphic variants can be used without prior declaration. Moreover, the type of the value is inferred from the context.

```
# let f v = match v with `A x -> x + 1 | `B x -> int_of_string x;;
val f : [< `A of int | `B of string ] -> int = <fun>
```

The inferred type express that the function can accept values that are either `A` with an associated value of type `int`, or `B` with an associated value of type `string`. It does not accept values with any other tags or with a different type of associated value.

Another construct that can be utilized with polymorphic variants is the constrained variable in the function type.

```
# let f v = match v with `A x -> `A (x + 1) | v -> v;;
val f : ([> `A of int ] as 'a) -> 'a = <fun>
```

The variable `'a` is constrained to be a polymorphic variant with the tag `A` and an associated value of type `int`. This implies that the argument's type will be unified with the type `[> `A of int]` before it is substituted in every other occurrence of the `'a` variable.

The type of a polymorphic variant can generally be defined as `[< UB > LB]`, where `UB` represents the optional upper bound, a list of tags with the types of associated values that can be written to the value of that type. `LB` is the optional lower bound, a list of tags with the types of associated values that can be read from the value of that type. The associated value in the upper bound and the lower bound must be identical. If the upper bound equals the lower bound, the type is written as `[UB]`.

Polymorphic variants represent an effort to merge static safety with code reusability, all the while preserving a succinct syntax. They were initially introduced to incorporate union types within the context of Hindley-Milner type systems [8]. Nevertheless, their potential is restrained by the kinding constraints of the Hindley-Milner type system, which seeks to imitate subtyping without explicitly introducing it. The existing system makes extensive use of the ML type system, particularly unification for type inference. While this method maximizes reuse, it introduces considerable complexity, making polymorphic variants challenging to understand, especially for novices. Furthermore, this approach curtails expressivity and obstructs several applications that traditional union types could facilitate.

1.2 Overview of the Set-Theoretic Types

In [5], a type system for polymorphic variants, based on set-theoretic types and semantic subtyping, was proposed. The authors demonstrated that their proposed system is strictly more expressive than the existing system, meaning it types more programs while preserving type safety. This system also eliminates some pathological cases present in the current implementation, thereby resulting in a more intuitive and predictable type system. However, the full implementation of their system is not possible due to the absence of Runtime Type Information in OCaml. Thus, the authors proposed a modified system that ameliorates some of the issues and remains strictly more expressive than the current implementation.

In the set-theoretic system, the type of a polymorphic variant is the set of tags along with their associated types. There are no upper or lower bounds. The interpretation of the type depends on the context in which it is used. For instance, when the type is used as a function argument, it is the set of tags with their associated types that the function can accept. Conversely, when the type is used as a function result, it is the set of tags with their associated types that the function can return.

The superiority of the proposed system can be demonstrated through the following examples:

Example 1: loss of polymorphism. Let's examine the identity function `id` that could be applied to the limited set of tags.

```
# let id v = match v with `A | `B -> v;;
val id : ([< `A | `B ] as 'a) -> 'a = <fun>
```

The function's type appears sound as it only accepts values of certain tags and returns a value of the same type. However, upon applying this function, we obtain a value with an unexpected type.

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

As can be seen, the inferred type of the value is not the expected `[> 'A']` and contains a redundant upper bound. This can lead to unexpected type errors, as demonstrated below:

```
# [id `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
```

In contrast, the set-theoretic system is capable of inferring the expected type:

```
# let id v = match v with `A | `B -> v;;
val id : ((`A | `B) as 'a) -> 'a = <fun>
# id `A;;
- : `A = `A
# [id `A; `C'];;
- : (`A | `C) list = [`A; `C']
```

This example was not inferred by any program but rather, demonstrates the expected behavior of the system.

Example 2: Roughly-Typed Pattern Matching. Let's examine the remap function, which converts the value of a specific tag into another tag, without affecting the values of other tags.

```
# let remap v = match v with `A -> `B | v -> v;;
val remap : ([> `A | `B ] as 'a) -> 'a = <fun>
```

When this function is used to map a list, we obtain a resulting list of an unexpected type:

```
# List.map remap [`A; `B; `C];;
- : [> `A | `B | `C ] list = [`B; `B; `C]
```

While it is clear that the resulting list cannot contain the tag `A` as it is mapped to the tag `B`, the current system is incapable of inferring this property. On the other hand, the set-theoretic system can infer the expected type:

```
# let remap v = match v with `A -> `B | v -> v;;
val remap : 'a -> `B | 'a \ `A = <fun>
# List.map remap [`A; `B; `C];;
- : (`B | `C) list = [`B; `B; `C]
```

Despite these examples being somewhat artificial, they effectively highlight the issues with the current system and demonstrate the superiority of the set-theoretic system. More practical examples of problems with the current system can be found in [1][2][3][4].

The proposed system, despite being specifically designed for OCaml with a promised implementation, has only been implemented as an external type checker for a subset of OCaml in a separate repository [12]. Given that the implementation of this system necessitates rewriting a substantial portion of the OCaml type checker and introducing new basic types, it is unlikely that this implementation will gain community acceptance. As a result, we propose a simplified version of the system and undertake an experiment to implement it in the existing type checker, without requiring strong interoperability with other subsystems of the OCaml language.

2 Statement and Motivation of Research

2.1 Hindley-Milner system

The current implementation of the OCaml typechecker is deeply rooted in the application of the Hindley-Milner system [10] [11]. This system is of significant importance in the domain of functional programming languages, with OCaml being a notable instance of such languages.

The principal function of the Hindley-Milner system is unification. This process aims at finding an appropriate substitution that equates two different types. Unification operates in contexts where certain values' types are anticipated to be compatible. A typical example of such a context is an application of a function, where the argument and the left side of the arrow type of the function need to be compatible.

To illustrate the process of unification, consider the following code snippet:

```
# let f v = fun _ -> v;;
val f : 'a -> 'b -> 'a = <fun>
# let g = f 1;;
```

```
val g : '_weak1 -> int = <fun>
```

In this example, the function `f` is applied to the value `1`. During this application, the unification process is activated. The type of the argument `'a` is compared with the type of the value `int`, leading to a substitution `'a -> int` that makes the types equivalent. This identified substitution is then applied to the type of the function `f`, resulting in a new type `'_weak1 -> int` for the function. The type variable `'_weak1` is termed as 'weak' as it is not constrained by its context. This allows it to potentially unify with any other type.

2.2 Current system formalization (K-system)

In this work, the formalization of the OCaml system is primarily derived from a seminal paper that presents a detailed description of the set-theoretic system [5]. A countable set of type variables, denoted by α, is assumed to exist. Additionally, a finite set of basic types, represented by b, is considered, with a function $b(\cdot)$ that maps constants to basic types.

Definition 2.1 (Types)**.** A type τ^k is a term inductively generated by the following grammar.

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

The system only uses the types of core ML: any supplementary information is represented through the kinds of type variables.

Definition 2.2 (Kinds)**.** A kind κ^k is either the unconstrained kind "•" or a constrained kind, that is, a triple (L, U, T) where:
- L is a finite set of tags $\{tag_1, ..., tag_n\}$;
- U is either a finite set of tags or the set $\mathbb{L}$ of all tags;
- T is a finite set of pairs of a tag and a type, written $\{tag_1 : t_1^k, ..., tag_n : t_n^k\}$ (its domain $\mathrm{dom}(T)$ is the set of tags occurring in it);

and where the following conditions hold:
- $L \subseteq U, L \subseteq \mathrm{dom}(T)$
- if $U \neq \mathbb{L}$ then $U \subseteq dom(T)$
- Tags in L have a single type in T, that is, if $tag \in L$, whenever both $tag : t_1^k \in T$ and $tag : t_2^k \in T$, we have $t_1^k = t_2^k$

The correctness of type transformations is governed by the admissibility of the type substitution. A type substitution is considered admissible if the kind of the substituted variable is stronger than that of the original one. The relation of strength is defined as follows:

$$(L, U, T) \vDash (L', U', T') \Leftrightarrow L \supseteq L \cap U \subseteq U' \cap T \supseteq T'$$

Inference within the K-system operates using the standard algorithm for Hindley-Milner systems. Although there are no explicit variables in the type for direct substitution, unification equalizes the types by transforming the kinds of the variables. The lower bound of the unified variable is the intersection of the lower bounds of the variables being unified, while the upper bound of the unified variable is the union of the upper bounds of the variables being unified. The mapping from tag to the type of the associated value is the union of the mappings, with tags present in both mappings being recursively unified.

This system behavior provides an explanation for the issue of polymorphism loss. The following code snippet offers an illustration:

```
# let id v = match v with `A | `B -> v;;
val id : ([< `A | `B ] as 'a) -> 'a = <fun>
# id `A;;
- : [< `A | `B > `A ] = `A
```

In this instance, unification occurred between the left side of the arrow type `([< A | B ] as 'a)` and the type of the argument `A`. The outcome of this unification was the type `[< A | B > A]`, which was then substituted into the type of the application.

2.3 Set-theoretic system formalization (S-system)

In the S-system formalization, subtyping is introduced via a semantic definition based on set-theoretic types, as opposed to its encoding through instantiation. The system utilizes type connectives and subtyping to represent variant types as unions and to encode bounded quantification through the use of union and intersection operations.

As for previous definition, we assume a set V of type variables (denoted by α), alongside the sets C, L, and B which represent language constants, tags, and basic types (represented by c, tag, and b respectively).

Definition 2.3 (Types)**.** A type τ^s is a term inductively generated by the following grammar.

$$\tau^s ::= \alpha \mid b \mid c \mid \tau^s \to \tau^s \mid \tau^s \times \tau^s \mid tag(\tau^s) \mid \tau^s \cup \tau^s \mid \neg\tau^s \mid 0$$

The following abbreviations will be used:

$$\tau_1^s \cap \tau_2^s := \neg(\neg\tau_1^s \cup \neg\tau_2^s); \quad \tau_1^s \setminus \tau_2^s := \tau_1^s \cap (\neg\tau_2^s); \quad 1 := \neg 0$$

Variant types and bounded quantification. The S-system encodes bounds and tag-to-content type mappings using standard type constructions. For instance, to simulate a variable α of kind ({A}, {A,B}, {A : bool, B : int}) in the S-system, the common scheme $(t_l^s \cup \beta) \cap t_u^s$ can be used, where t_l^s is the lower bound and t_u^s is the upper bound. This results in the expression $(A(bool) \cup \beta) \cap (A(bool) \cup B(int))$.

Subtyping. Subtyping is semantically defined as $t_1^s \leq t_2^s$ if and only if $[\![t_1^s]\!] \subseteq [\![t_2^s]\!]$, where $[\![\cdot]\!]$ represents an interpretation function mapping types to sets of elements from a domain (intuitively, the set of values of the language).

The authors of the foundational paper for the S-system demonstrated that their system extends the K-system through a type translation from K to S, and they established that this translation preserves the typing relation.

Type inference within the S-system occurs in two main phases. The first phase, constraint generation, involves generating constraints from an expression e and a type t, which records the conditions under which e may be assigned type t. In the second phase, constraint solving, these constraints are solved (if possible) to obtain a type substitution θ. Solution of the constraint set is achieved using the "tallying" algorithm [6]. The authors of [5] proved that the type inference in the S-system is both sound and complete.

2.4 Issues with the S-system

While the S-system is indeed more expressive in comparison to the K-system, it is also correspondingly more complex, which consequently gives rise to several issues:

Firstly, the S-system introduces comprehensive set-theoretic types, not solely for polymorphic variants but for all types. As a consequence, it may be excessively expressive for commonly used types such as `string` or `int`. One can envision types like "`string \ "A"`" which express the possibility that a value can contain any string value, with the exception of the string `"A"`. For many programmers, the prospect of dealing with such types can be daunting.

Secondly, the type inference algorithm for the S-system exhibits non-deterministic behavior. This implies that the same program can be typed differently depending on the order in which constraints are processed. Although this does not pose an issue for type safety, it can create problems for programmers because the type of the program may diverge from the anticipated one. This variability also compounds the complexity of implementing a comprehensive typechecker.

Thirdly, the "tallying" algorithm employed in the S-system has a certain section where it is obliged to iterate through all potential solutions at each level of depth, which may result in an exponential increase in complexity. There is currently no clear understanding about the frequency at which this occurs in practice, how significantly it impacts performance, or whether there exist any potential avenues for optimization. Given that performance is a key consideration for a compiler, this represents a potential issue warranting further investigation.

Finally, the current implementation of the "tallying" algorithm is not without its flaws and has been observed to enter an infinite constraint-solving loop. This behavior is not acceptable for the typechecker of an industrial-grade programming language. Therefore, additional work is necessitated to rectify the algorithm.

3 Proposed system (P-system)

3.1 Formalization

The implementation of the S-system entails a significant rewrite of the typechecker, which is beyond the scope of this work and is not likely to gain acceptance from the OCaml community. Given this and the aforementioned issues with the S-system, we suggest a simplified version of the system that is more straightforward and easier to implement. The primary aim of this simplification is to restrict the usage of set-theoretic operations to the polymorphic variants subsystem only.

To realize this, we categorize the types syntactically into two groups: those belonging to the polymorphic variants subsystem and all other types. We maintain kinds but restrict their role to the mapping from the tag to the associated value's type. The types of the polymorphic variants subsystem constitute a pair comprising a static mapping from tag to type, and the type of the polymorphic variant, which includes only tags and their set-theoretic relations. We infer the type of the polymorphic variant using the same methodology as in the S-system, but we avoid gathering subtyping constraints for other types.

For the purpose of showcasing the system, we will reuse the simplistic language of polymorphic variants, as introduced in Section 2 of [5].

As for our previous definitions, we consider a set V of type variables (ranged over by α), and the sets L and B of tags and basic types (ranged over by tag and b respectively).

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

$$\tau^p ::= a \mid b \mid \tau^p \to \tau^p \mid \tau^p \times \tau^p \mid (\kappa^p, \tau_t^p)$$
$$\tau_t^p = a_t \mid tag \mid \tau_t^p \cup \tau_t^p \mid \neg\tau_t^p \mid \top$$

where:
- τ_t^p is the type of the polymorphic variant value;
- a_t is the variables of the polymorphic variant type;
- κ^p is the kind of the polymorphic variant type;

The following abbreviations will be used for polymorphic variant types:

$$\tau_1^s \cap \tau_2^s := \neg(\neg\tau_1^s \cup \neg\tau_2^s); \quad \tau_1^s \setminus \tau_2^s := \tau_1^s \cap (\neg\tau_2^s); \quad 1 := \neg 0$$

Definition 3.2 (Kinds)**.** A kind, denoted as κ^p, is defined as a mapping from tags to general types (τ^p).

To manage the correctness of the substitution, we establish a strength relation between the kinds of the polymorphic variant types, similar to what is done in the K-system:

$$\kappa_1^p \vDash \kappa_2^p \Leftrightarrow (\forall tag, t_p : (tag, t_p) \in \kappa_2^p \implies (tag, t_p) \in \kappa_1^p)$$

Here, $\kappa_1^p \vDash \kappa_2^p$ implies that κ_1^p is stronger than κ_2^p. In other words, it represents a submapping relation.

The type substitution has become a triplet: for variables of general types, for variables of polymorphic variant types, and for kinds of polymorphic variant types. And for kinds, we necessitate that the substituted kind is stronger than the original one.

The P-system showcases more expressiveness than the K-system, as any type from the K-system can be converted into a type in the P-system using the same methodology as the K-system to S-system transformation. However, the P-system is less expressive than the S-system because it doesn't allow union and negation on types other than polymorphic variants, and it imposes stringent restrictions on kind relations.

The first limitation prevents us from using types like `True` or `int`
`1`, which can assist in the typing of certain programs but are not generally useful.

The second limitation is more significant. It prevents us from typing the following function:

```
# let remap v = match v with `A 10 -> `A "A" | v -> v;;
```

This occurs because the kinds of the polymorphic variant type that are syntactically compatible are restricted to having the same type for each common tag. Consequently, the initially inferred type `int` conflicts with the type `string` in the match expression. This case is covered in the S-system. But since it's not covered in the K-system either, such a restriction is not a significant drawback.

3.2 Type inference

Due to the fundamental simplification of the system compared to the S-system, the type inference algorithm could also be simplified. However, the principal idea of the algorithm remains unchanged from the S-system, wherein we gather subtyping constraints and subsequently resolve them. The distinction arises solely in the constraint-solving algorithm.

The resultant set of constraints is treated as a directed graph, with the nodes of the graph being the variables and the edges being the constraints.

In the first phase of the algorithm, we remove cycles in the graph by merging nodes that are strongly connected. Following this, we independently solve the upper bound and the lower bound for the variable.

During the solution process, we encounter two types of nodes:

The first type includes nodes that are in the context for the current type. For instance, this could be the type of the argument of a function while we're resolving the type of the result, or the type of the first element in a tuple while we're resolving the type of the second element. Such nodes can be used in the solution as they are, and we do not need to resolve them and any transitive relation caused by them. This is because we can utilize that as a variable in the output and all the constraints are assured to be satisfied by the constrained type in the declaration of that variable.

The second type of nodes are those that were not used in the context for the current type. An example of this might be a type of a local variable in a function while we are calculating the function's type. These nodes serve as intermediate nodes that are used for transitive relations and will be satisfied if all the transitive relations are met.

The algorithm for solving the upper bound of a variable, for instance, is illustrated in the following pseudocode:

```
let solve context node =
  match node with
  | Tags tags -> Tags tags
  | Variable v -> begin
      # The node is in the context or equal to any such node
      if node `in` context
      # We could use this variable as is
      then Variable(node)
      # The solution is the intersection
      # of the solutions for the immediate supertypes
      else Intersection(map (solve context) (immediate_supertypes node))
    end
```

In short, it is merely an intersection of the reachable contextual variables and tags in the constraints graph. As the graph does not contain a cycle, we can do this in a recursive manner. To solve for the lower bound, we simply replace the intersection with the union, and the `immediate_supertypes` with the `immediate_subtypes`.

Once we receive the solution for each bound, we can construct the resulting type as a `(lb ∪ 'a) ∩ ub` where 'lb' is the lower bound, 'ub' is the upper bound, and 'a' is the fresh variable.

Using this algorithm, we can obtain a solution that satisfies the system. However, for real-world usage, it is post-processed to remove repetition of the tags in the upper and lower bounds, thereby reducing the size of the resulting type.

To check if the system is satisfiable, we must solve each variable without context. In this case, each type will appear as the upper bound and the lower bound, both consisting only of tags. If the lower bound is not a subset of the upper bound, the system is not satisfiable and a type error must be raised.

3.3 Possible improvements of the system

The system proposed herein is a formalization of the implemented system, but it is not the most expressive system that fulfills the requirements listed. In the future, several potential enhancements to the system could be undertaken.

The first improvement involves excluding the already matched tags from the type of the polymorphic variant value in the 'else' branch of matching. The current system does not rectify the issue with roughly typed pattern matching. For instance, for the following code:

```
# let remap v = match v with ‘A -> ‘B | v -> v;;
```

The current implementation will infer the type of the function as `(T as 'a) -> B | 'a` instead of the expected `(T as 'a) -> B | 'a \ ‘A`. This enhancement could be realized without significant changes to the inference algorithm and merely requires further investigation into the possibility of introducing floating types in the typechecker.

The second potential enhancement is the relaxation of the strength relation between kinds of the polymorphic variant types. As opposed to the current requirements, we could allow for the mapped type of the stronger kind to be a supertype of the mapped type of the weaker one. This would not impact the inference in any way, but merely add more expressiveness in cases of nested polymorphic variant types and permit changes to the associated type of some tags. This would be akin to the introduction of subtyping for the general types, where the non-polymorphic variant types would be in such relation only if they are equal, while for the polymorphic variant types, the relation would be more flexible.

For instance, for the following function:

```
# let remap v = match v with ‘A _ -> ‘A "A" | v -> v;;
```

The argument could have a kind without any restriction on the associated type for `‘A`, while the current formalization requires it to be `string`, as is the case for the result type.

4 Implementation of the proposed system

4.1 Current state of the typechecker

The primary challenge with modifying the OCaml typechecker is articulated in the `TODO.md` file located in the respective directory: *"There is a consensus that the current implementation of the OCaml typechecker is overly complex and fragile. A big rewriting "from scratch" might be possible or desirable at some point, or not, but incremental cleanup steps are certainly accessible and could bring the current implementation in a better shape at a relatively small cost and in a reasonably distant future."* However, given that this document

was created in 2018 and has seen only minor modifications up to 2020, improvements to the type checker codebase in the near future seem unlikely. Consequently, we had to work with what was available.

The OCaml type system has been using the Hindley-Milner [10] [11] system since its inception 27 years ago, leading the type checker to heavily rely on invariants that are specific to the Hindley-Milner system. This leads to a high cost of changes that do not preserve such invariants. The specific invariants that were broken will be described in detail in section 4.2.

The inference in the Hindley-Milner system relies on the "`unify`" function, which is utilized to find a substitution that makes two types equal.The unification between types is syntax-directed, such as between the argument and the left side of the arrow type of the applied function. Following unification, all equality constraints are reflected in the type, negating the need for further solution.

Another crucial component of the typechecker is the substitution. In OCaml, this is achieved not by traversing the type but by leveraging the physical equality of the types. Post unification, they make the unified type reference the same type description. This approach serves as a valid optimization because types that have become equal are expected to remain so indefinitely. In addition to memory optimization, this results in all variables that were unified becoming physically equal, even if they were nested deep within the type, such as the first element of a tuple. Consequently, any subsequent changes in any of them will affect all of them, as is expected in the substitution.

4.2 Issues encountered during Implementation

The system's implementation was done in a forked repository of the OCaml compiler [9], based on version 4.14 of OCaml due to its stability and long-term support promise.

4.2.1 Collection of subtyping constraints

Unlike the Hindley-Milner system where all type constraints can be easily copied from one type to another, the subtyping system interconnects all polymorphic variant types via their constraints. As a result, new constraints on one type can influence other types through the previously established constraints.

The current implementation houses these subtyping constraints in a global set, which presents some drawbacks. The primary issue is that these constraints are not retained during serialization, as there is no pre-serialization phase that would permit us to house them within the type. This does not impact the typing validity within a single file. For multiple files, the types from other files can be sourced from the interface file, which preserves all necessary information in the declaration. The only issue surfaces in instances involving polymorphic variant types, where the implementation file's type might not be compatible with the type in the interface file. This is due to the inability to retrieve the real type of the implemented function during comparison with the one from the interface.

This problem is not fundamental, and several solutions exist to rectify it. One solution would be to serialize the constraints set or house the constraints related to the type within the type itself. Both of these solutions, however, would significantly inflate the size of the serialized file. An optimal solution would be to introduce a finalization phase during which all constraints could be solved and preserved in the tree as another type with only the

resulting data. However, this has not been implemented as the current version is just a prototype.

4.2.2 Solving types at the end of constraint collection

In a system based on constant solving, the actual inference of the type is carried out at the end of constraint collection. This diverges from the behavior of the Hindley-Milner system, where the type is iteratively solved, and at each point in time, it represents the most precise type that could be inferred from the current context. Although this problem was touched upon in the previous section, it warrants separate mention due to its significance.

This issue arises when we need to generate a list of possible tags and present the type to the user (in the case of interpretation mode through the console, or for language servers that require types to show in hints). While getting the list of possible tags was handled by an encapsulated function and didn't require significant interoperability modifications, the printing of the types lacked a preprocessing phase. Furthermore, the inference process might involve transforming the type tree, which is undesirable as this type is expected to remain unchanged and may be used in further inference. The solution implemented involved adding a preprocessing phase that infers all polymorphic variant types in the tree with the appropriate context and memorizes the result. This result is then used in the printing phase instead of the original type.

4.2.3 Subtyping in unification

The unification function in the Hindley-Milner system, which originally aims to make types equal, must be adjusted due to the introduction of subtyping. Now, it needs to convert one type into a subtype of another, rather than making them identical. In most cases, subtyping is undefined, and the function will disregard its existence, behaving as it used to. The only difference arises when we need to monitor changes in the subtyping direction, such as in cases like arrow type or type arguments. For arrow types, it's done simply by swapping the expected subtyping direction between unified types. For type arguments, it requires establishing variance on each type argument, which hasn't been completed yet.

4.2.4 Inequality after unification

Following the unification process, types were made physically equal by assigning a reference from one to another. This was an acceptable approach because types that became equal were expected to remain equal indefinitely, optimizing memory usage for the compiler. However, with the new unification semantics, this optimization is invalid as types might merely be subtypes of one another. The main issue is that this technique was used not only for optimization but also for future substitutions. Thus, we cannot simply remove it as it would disrupt future unifications.

The apparent solution was to link only the type variables nested in the type. However, this doesn't work as those variables can be unified with the polymorphic variant type. In such instances, we must transform all unified variables into the polymorphic variant type and add all constraints that were collected for them when they were variables. Given these requirements, the only solution is to collect constraints for variables as well as for polymorphic variant types. If a variable is unified with a type that is not a variable, we must repeat all unifications with other variables that were collected as constraints for that variable.

The issue with this solution is that it necessitates significant rewriting of various parts of the typechecker, which hasn't been done yet. However, it doesn't affect the system's correctness in most cases, as variables are usually transformed into polymorphic variants before any other unifications with other variables.

4.2.5 Unification of kinds

Unification of kinds presents another challenge. This issue can be demonstrated in a scenario where we have three types, t_1, t_2, t_3, and we unify t_1 with t_2 and t_2 with t_3 with the same subtyping relation. In this case, the kind of t_1 and t_3 would not be directly unified. This could lead to a situation where some tags to associated type mappings are not synchronized between them. A solution is to iterate over all previously unified types and unify their kinds with the new kind. However, this could significantly impact performance and potentially result in a unification cycle in cases of backward references in associated types. Due to these reasons, the current implementation uses a disjoint-set forest [7] to physically merge kinds that have been unified. This allows us to avoid iterating over all of the unified types, but it restricts us to providing only the semantics that this implementation allows.

4.2.6 Non-local unification failure

When a new constraint is introduced, it can influence all the types that are reachable within the constraint graph, even those that are not directly associated. Consequently, to monitor typing failures, we must verify the entire constraint system's solvability after incorporating any new constraint. However, iterating through all the types could negatively impact performance. The current implementation only checks the unified types, which could theoretically lead to unnoticed typing failures. Nonetheless, these would eventually be detected in a different context, causing an assertion failure, which is acceptable for the prototype implementation. The optimal solution would involve implementing online data structures that can identify typing issues without fully solving all the types.

4.2.7 Preserving constraints when copying types

In OCaml, when typing a function application, the original function's type should remain unaltered. To ensure this, the OCaml type checker makes a copy of the function type before unifying its arguments. However, as polymorphic variant types are linked to other types through constraints, these constraints need to be preserved during the copying process. For example, the subtyping relation between the result type and the argument type must be maintained.

To achieve this, occurrences of polymorphic variant types are monitored during the copying process, and the constraints associated with each of them are copied after the complete type copy has been made. This allows us to preserve their internal relations. Moreover, the original type is resolved at the end of the copying process, and any non-internal relations that affect the resulting type are added to the copied type as independent relations.

Another issue with copying is that changes in the copied type can sometimes affect the original type, and there's no explicit flag or any other mechanism to track such cases. It depends on the internal properties of the previous representation of the polymorphic

variant type, among other conditions. Also, the sharing of types is done in a manner that relies on the previous representation of the polymorphic variant type, and it does not follow the semantics of full sharing. As a result, it's unclear how to replicate such behavior in the new representation. Due to these factors, the current implementation does not preserve the sharing of types and simply copies them as independent types. This might affect the correctness in some edge cases that have not been identified yet. To rectify this problem, we need to conduct further investigation into the expected behavior.

5 Evaluation of the Implementation

The current implementation is not entirely correct for all code due to unresolved issues. However, it functions correctly when polymorphic variant types are used solely at the top level in the internal representations. In such instances, we observe that some programs become typeable that were not typeable before. For example:

```
# let id v = match v with `A | `B -> v;;
val id : ((`A | `B) as 'a) -> 'a = <fun>
# id `A;;
- : `A = `A
# [id `A; `C];;
- : (`A | `C) list = [`A; `C]
```

This is the actual output of the modified compiler, whereas the original compiler would raise a type error on the last line. Another example of a program that is typed better than expected is the following:

```
# let f y = match y with `A -> `C | y -> y;;
val f : (T as 'a) -> `C | 'a = <fun>
# f `D;;
- : `C | `D = `D
# f `A;;
- : `C = `C
```

The original compiler would generate this output:

```
# let f y = match y with `A -> `C | y -> y;;
val f : ([> `A | `C ] as 'a) -> 'a = <fun>
# f `D;;
- : [> `A | `C | `D ] = `D
# f `A;;
- : [> `A | `C ] = `C
```

There is an issue with the modified compiler in that the expected associated type for tag A is not displayed in the type. However, this is merely a printing issue, as it is managed by the typechecker. Despite this, it is quite apparent that our system infers more precise types than the original one.

The performance of the implemented system was not a primary goal. As a result, some internally-used algorithms were chosen simply because they were easier to implement than the optimal ones. Hence, the current implementation is significantly slower than the original one.

6 Conclusions

This study has presented a streamlined version of the S-system[5], exhibiting greater expressiveness than the current system for polymorphic variants in OCaml. Despite being less expressive than the S-system, the proposed system we have introduced addresses several complications of the existing system. Additionally, it simplifies the type inference algorithm, avoids excessively precise types, and necessitates reduced implementation effort within the OCaml compiler.

Moreover, we have put forth an implementation of the proposed system, which acts as a proof of concept for the introduction of subtyping to the OCaml type system. It should be noted, however, that the existing implementation is a prototype and is only partially functional. There are several performance and correctness issues that must be resolved before this implementation could be presented to the OCaml community. These challenges, while significant, provide a direction for future research and development.

References

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

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

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

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

[5] Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. “Set-theoretic types for polymorphic variants”. Sept. 2016. DOI: 10.1145/2951913.2951928. URL: https://doi.org/10.1145%2F2951913.2951928.

[6] Giuseppe Castagna et al. “Polymorphic Functions with Set-Theoretic Types: Part 2: Local Type Inference and Type Reconstruction”. New York, NY, USA, Jan. 2015, pp. 289–302. DOI: 10.1145/2775051.2676991. URL: https://doi.org/10.1145/2775051.2676991.

[7] Bernard A. Galler and Michael J. Fisher. “An Improved Equivalence Algorithm”. New York, NY, USA, May 1964, pp. 301–303. DOI: 10.1145/364099.364331. URL: https://doi.org/10.1145/364099.364331.

[8] Jacques Garrigue. “Programming with Polymorphic Variants”. 1998. URL: https://caml.inria.fr/pub/papers/garrigue-polymorphic_variants-ml98.pdf.

[9] “Github repository with the implementation of the P-system”. URL: https://github.com/e2e4b6b7/ocaml/tree/constraints.

[10] R. Hindley. “The Principal Type-Scheme of an Object in Combinatory Logic”. 1969, pp. 29–60. URL: http://www.jstor.org/stable/1995158.

[11] Robin Milner. “A theory of type polymorphism in programming”. 1978, pp. 348–375. DOI: https://doi.org/10.1016/0022-0000(78)90014-4. URL: https://www.sciencedirect.com/science/article/pii/0022000078900144.

[12] “setvariants”. Implementations of the set-theoretic types for the subset of the OCaml language. Available online: https://www.cduce.org/ocaml/, https://www.cduce.org/ocaml/bi. URL: https://gitlab.math.univ-paris-diderot.fr/petrucciani/setvariants.

Statutory Declaration

Family Name, Given/First Name	Venediktov, Roman
Matriculation number	30007033
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

.....

Date, Signature