CONSTRUCTOR UNIVERSITY

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

by

Roman Venediktov

Bachelor Thesis in Computer Science

Submission: May 14, 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, leading to an awkward formalization and a type system with unintuitive and/or unduly restrictive behavior. While there is the system based on the set-theoretic types with semantic subtyping that is more expressive than the current one (it types more programs while preserving type safety) and it was demonstrated to be more precise and intuitive in several examples, it has yet to be implemented in the OCaml compiler.

This work presents a formalization and implementation of a simplified typing system based on set-theoretic types in the OCaml compiler. This implementation demonstrates the feasibility of a mixed-type inference system that combines the classic Hindley-Milner algorithm with a system incorporating semantic subtyping and constraint solving. The resulting system offers a more intuitive and flexible approach to polymorphic variant typing in OCaml.

Contents

1 Introduction

1.1 Overview of the Polymorphic Variants

Polymorphic variants is the pair of tag and the associated value. "#" denotes the interactive toplevel prompt of OCaml, whose input is ended by a double semicolon and followed by the system response.

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

This is the declaration of the polymorphic variant value that has 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 main difference with the normal variants is that you can use the polymorphic variant constructors without declaring it first and 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 means that function accepts the values that are either `A` with the associated value of type `int` or `B` with the associated value of type `string` and accepts neither values with any other tags nor values with another type of the associated value.

In general the type of the polymorphic variant could be defined as `[< UB > LB]`, where `UB` is the optional upper bound, the list of tags with the types of the assosiated value that could be written to the value of that type and `LB` is the optional lower bound, the list of tags with the types of the assosiated value that could be read from the value of that type. The assosiated value in the upper bound and the lower bound required to be equal. If the upper bound is equal to the lower bound, the type is written as `[UB]`.

Polymorphic variants provide a balance between static safety and code reuse while maintaining a concise syntax. They were initially proposed as a solution to incorporate union types into Hindley-Milner (HM) type systems [3]. However, their capability is limited, due to the superimposition of a system of kinding constraints on the HM type system, which is used to simulate subtyping without actually introducing it. The current system reuses the ML type system, including unification for type inference, as much as possible. However, this approach yields significant complexity, making polymorphic variants difficult to understand, especially for beginners. Furthermore, it limits expressiveness and prohibits several useful applications that general union types make possible.

1.2 Overview of the Set-Theoretic Types

In [1], a type system for the polymorphic variants based on set-theoretic types and semantic subtyping was proposed. The authors showed that it is strictly more expressive than the current system that means that it types more programs while preserving type safety. The proposed system removes some pathological cases of the current implementation, resulting in a more intuitive and predictable type system. While the full implementation of their system is not possible due to the absence of Runtime Type Information in

OCaml, the authors propose a modified system that fixes some of the issues and remains strictly more expressive than the current implementation.

The type of the polymorphic variant in the set-theoretic system is the set of the tags with the types of the associated value. There is no upper bound and the lower bound. The meaning of the type depends on the context where it is used. For example, if the type is used as the argument of the function, the type is the set of the tags with the types of the associated value that could be accepted by the function and if the type is used as the result of the function, the type is the set of the tags with the types of the associated value that could be returned by the function.

The superiority of the proposed system could be demonstrated in the following examples:

Example 1: loss of polymorphism. Let us consider 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 type of the function is looks soundness as it accepts only the values of such tags and return the value of the same type. However if we apply this function we will reuslt the value with the unexpected type.

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

As we can see the inferred type of the value is not the expected `[> ‘A’]` and contains redundant upper bound. This may lead to the unexpected type error in the following example:

```
# [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
```

Whereas the set-theoretic system is able to infer 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 just an expected behavior of the system.

Example 2: roughly-typed pattern matching. Let us consider the remap function that converts the value of the specified tag into another tag while do not affect the values of other tags.

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

And if we would like to map a list with such function we will get the unexpected type of the resulting list:

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

While it is obvious that the resulting list can not contain the tag `‘A` as it is mapped to the tag `‘B`, the current system is not able to infer such property. However, the set-theoretic system is able to 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 the fact that these examples are artificial, they demonstrate the problems with the current system and the superiority of the set-theoretic system. Another and more ”real life” examples of the problems with current system can be found in the TODO: add citation of the: [CAML-LIST 1] 2007; [CAML-LIST 2] 2000; [CAML-LIST 3] 2005; [CAML-LIST 4] 2004; Nicollet 2011; Wegrzanowski 2006)

Although the proposed system was specifically designed for OCaml and the authors promised an implementation, it was only implemented as an external type checker for a subset of OCaml in a separate repository [6]. As the implementation of the described system requires rewriting a significant part of the OCaml type checker and introducing new basic types, it is unlikely that the implementation will be accepted by the community. Therefore, we propose a simplified version of the system and conduct an experiment to implement it in the existing type checker without strong interoperability with the other subsystems of the OCaml language.

2 Statement and Motivation of Research

2.1 Hindley-Milner system

Current implementation of the OCaml typechecker highly relies on the Hindley-Milner system [4] [5]. The main function in this system is unification that is find a substitution that makes types equal. It calls on the types of the values that are expected to be compatible, for example, the argument and the left side of the arrow type of the applied function.

We can see the example of the unification in the following code:

```
# let f v = fun _ -> v;;
val f : ’a -> ’b -> ’a = <fun>
# let g = f 1;;
val g : ’_weak1 -> int = <fun>
```

When we applied the function `f` to the value `1` the type of the argument `’a` was unified with the type of the value `int` and the resulting substitution was `’a -> int`. Then the substitution was applied to the type of the function `f` and the resulting type was `’_weak1 -> int`. The type variable `’_weak1` is called weak as it is not constrained by the context and could be unified with any type.

2.2 Current system formalization (K-system)

The formalization of the OCaml system utilized in this work was derived from the paper describing the set-theoretic system [1]. 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.

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

TODO: definition of the intersection and exclusion

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 $t_1^k = t_2^k$

TODO: admissibility of the substitution

Inference for the K-system is done using a standard algorithm for Hindley-Milner systems. While ther is no explicit variables in the type for the strightforward substitution, the unifcation just makes the types equal transforming kinds of the variables. The lowerbound of the unified variable is the intersection of the lowerbounds of the unified variables and the upperbound of the unified variable is the union of the upperbounds of the unified variables. The mapping from the to the type of the assosiated value is the merge of the mappings where the tags that was presented in the both mapping are recursively unified.

Such behavior explain the loss of polymorphism issue:

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

When we unified the left side of the arrow type `([< ‘A | ‘B ] as ’a)` with the type of the argument `‘A` the resulting type was `[< ‘A | ‘B > ‘A]` and it was subsituted to the type of the application.

2.3 Set-theoretic system formalization (S-system)

In that system, subtyping is added using a semantic definition based on set-theoretic types, rather than encoding it via instantiation. They utilize type connectives and subtyp-

ing to represent variant types as unions and to encode bounded quantification by union and intersection.

As before, we consider a set V of type variables (ranged over by α) and the sets C, L, and B of language constants, tags, and basic types (ranged over by c, tag, and b respectively).

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

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

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

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 $[\![\cdot]\!]$ is an interpretation function mapping types to sets of elements from a domain (intuitively, the set of values of the language).

The authors of the paper have demonstrated that their system extends the K-system through a translation of types from K to S and have established that this translation preserves the typing relation.

Type inference in S-system proceeds in two main phases. In the first, constraint generation, they generate constraints 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, they solve (if possible) these constraints to obtain a type substitution θ. Solution of the constraint set was done using the ”tallying” algorithm [2]. In the [1] were proved that the type inference in S-system is sound and complete.

2.4 Issues with the S-system

While the S-system is more expressive than the K-system, it is also more complex. And it leads to several issues:

The S-system introduces full-flegged set-teoretic types not only for the polymorphic variants but for all the types. As a result it may be too expressive for common types like `string` or `int`. Types like ”`string \ "A"`” expressing that that value can contain any string value excluding the string `"A"` may be intimidating for programmers.

The type inference algorithm for the S-system is non-deterministic. It means that the same program may be typed differently depending on the order of the constraints. While it is not a problem for the type safety, it may be a problem for the programmer as the type of the program may be different from the expected one. In addition it makes the implementation of the whole typechecker more complex.

The ”tallying” algorithm has certain section where it must iterate through all potential solutions at each depth, which can result in exponential complexity. There is no clarity about how often this happens in practice, how much it affects performance, and is there any way to optimize it. As the performance is rather important for compier, it is a potential issue that requires additional investigation.

In addition, the current implementation of the ”tallying” algorithm has bugs and may enter an infinite constraint-solving loop. This is not acceptable for the typechecker of the industrial programming language. Therefore, there is the additional work required to fix the algorithm.

3 Proposed system (P-system)

3.1 Formalization

The implementation of the S-system requires rewriting a significant part of the typechecker, which could not be done within the scope of this work and is unlikely to be accepted by the OCaml community. Due to this and the listed issues with the S-system, we propose a simplified version of the system that is more intuitive and easier to implement. The main idea of the simplification was to limit the use of set-theoretic operations only to the polymorphic variants subsystem.

To achieve this, we syntactically divide the types into two groups: the types of the polymorphic variants subsystem and the other types. We preserve kinds but limit their function only to the mapping from the tag to the type of the associated value. The types of the polymorphic variants subsystem are the pair of static mapping from tag to type and the type of the polymorphic variant that contains only tags and their set-theoretic relations. We infer type of the polymorphic variant using the same technique as in the S-system, but we do not collect subtyping constraints for other types.

For the purpose of demonstrating the system, we will reuse the simple language of polymorphic variants introduced in Section 2 of [1].

As before, 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;

TODO: definition of the intersection and exclusion

Definition 3.2 (Kinds)**.** A kind κ^p is a map from tags to general types (τ^p)

To control the correctness of the substitution we define strength relation between kinds of the polymorphic variant types as in the K-system:

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

where $\kappa_1^p \vDash \kappa_2^p$ means that κ_1^p is stronger than κ_2^p. On the other words it is just a submapping relaion.

The type substitution becames the triplet: for variables of general types, for variables of polymorphic variant types and for kinds of polymorphic variant types. And for kinds we require that the subsituted kind is stronger than the original one.

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 main idea of the algorithm remains the same as in the S-system, where we collect subtyping constraints and solve them. The difference is only in the algorithm of the constraint solving.

The set of the resulting constraints is treated as directed graph. The nodes of the graph are the variables and the edges are the constraints.

On the first phase of the algorithm we are removing the cycles in the graph by merging the nodes that are strongly connected. After that we independently solve the upper bound and the lower bound for the variable.

During the solving there are two types of nodes:

First type is such that are in the context for the current type. For example, it can be the type of the argument of the function while we solving the type of the result or the type of the first element in tuple while we solving the type of the second element. Such nodes can be used in the solution as is, and we do not need to solve them and any transitive relation caused by them as we can use that as variable in the output and all the constraints are guaranteed to be satisfied by the constrained type in the declaration of that variable.

Second type of nodes is the nodes that was not used in the context for the current type. For example it could be a type of the local variable in function while we are calculation type of the function. Such nodes are just an intermediate nodes that are used for transitive relations and will be satisfied if all of the transitive relations will be satisfied.

Algorithm of solving, for example, the upper bound of the variable is the following (written in 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 brief, it is just an intersection of the reachable contextual variables and tags in the constraints graph. As the graph does not contain cycle we can do it in such recursive way. To solve for lower bound we just replace the intersection with the union and the `immediate_supertypes` with the `immediate_subtypes`.

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

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

To check if the system is satisfiable we have to solve each variable without context. In that case each type will looks like the upper bound and the lower bound that both consists only of tags. And if the lowerbound is not a subset of the upperbound the system is not satisfiable and type error have to be raised.

3.3 Possible improvements of the system

The proposed system is not the formalization of the implemented system but not the most expressive system satisfying the listed requirements. There are several possible improvements of the system that could be done in the future.

The first improvement is to add exclude the already matched tags from the type of the polymorphic variant value in the else branch of matching. Currently system do not fix the issue with roughly typed pattern matching. For example for followoing code:

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

Current implementation will infer the type of the function as `(T as 'a) -> `B | 'a` instead of expected `(T as 'a) -> `B | 'a \ `A`. This improvement could be done without a significant changes to the inference algorithm and just require more investigation on the possibility to introduce a floating types in the typechecker.

The second possible improvemnt is the weakening of the strength relation between kinds of the polymorphic variant types. Instead of 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. It will not affect the inference at all and just add more expressiveness in case of the nested polymorphic variant types. It will be equal to the introduction of the subtyping for the general types where the non polymorphic variant types will be in such relation only in case they are equal while for the polymorphic variant types the relation will be more flexible.

4 Implementation of the proposed system

The main issue with modifying the OCaml type checker is described in the `TODO.md` file 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, since this document was created in 2018 and has only been slightly modified until 2020, there is no chance for improvements to the type checker codebase in the near future. As a result, we had to work with what we had.

The OCaml type checker is based on the Hindley-Milner system [4] [5], and inference in this system relies on the "`unify`" function, which is used to find a substitution that makes two types equal. Unification between types is syntax directed, for example, between the argument and the left side of the arrow type of the applied function. After unification, all the equality constraints are reflected in the type, and no further solution is required.

The OCaml type system has been using the Hindley-Milner system since its inception 27 years ago, and as a result, the type checker heavily relies on the invariants that are specific to the HM system. This leads to a high cost of changes that do not preserve such invariants.

4.1 Collection of subtyping constraints

In contrast to the Hindley-Milner system, where all type constraints can be simply copied from one type to another, the subtyping system connects all polymorphic variant types via their constraints. This means that new constraints on one type can affect other types through the previously established constraints.

The current implementation stores these subtyping constraints in a global set, which has some drawbacks. The main issue is that the constraints are not preserved during serialization, as there is no pre-serialization phase that would allow us to store them in the type. This does not affect the typing validity within a single file, and in the case of multiple files, the types from other files can be taken from the interface file, which preserves all the necessary information in the declaration. The only problem arises in cases involving polymorphic variant types, where the implementation file's type may not be compatible with the type in the interface file as there is no possibility to retrieve the real type of the implemented function during the comparison with the one from the interface.

This problem is not fundamental, and there are several solutions to fix it. One solution would be to serialize the constraints set or store the constraints related to the type in the type. However, both of these solutions would significantly increase the size of the serialized file. An optimal solution would be to add a finalization phase in which all the constraints can be solved and preserved in the tree as another type with only the resulting data.

4.2 Solving types at the end of constraint collection

This problem was previously mentioned in the previous section, but it is important to state it independently due to its significance. In contrast to the HM system, which solves types during expression traversal and requires no final phase, the new system needs to post-process the type tree to solve all the polymorphic variants. This process may involve transforming the tree into another structure. Solving this problem requires significant architectural changes and a lot of work, so it has not yet been implemented. The current implementation solves polymorphic variant types before printing, which works well for now, but better architecture ideas should be investigated in the future.

4.3 Variance in unification

The main function in the HM system is unification, which is originally intended to make types equal. However, since we introduced subtyping, we need to modify the unification algorithm to make one type a subtype of the other type, rather than making them equal.

For most classes, subtyping is not defined, so the function will just ignore variance and behave as before. The only difference is that we need to control variance changes in cases like arrow type.

For polymorphic variant types, we register a subtyping constraint in the global set and unify the mappings from the tags to types. For every shared tag, we have to unify their mapped types with the same subtyping destination as for the whole type. For other tags, we need to create mappings to a new type variable and then unify them, as we do with the shared tags.

4.4 Inequality after unification

During the unification process, types were made physically equal by assigning a reference to one of them to another. This was a valid optimization technique as the types that became equal were expected to remain equal forever, and this approach optimized memory usage for the compiler. However, with the new semantics, this optimization is no longer valid. Therefore, the solution was to remove the optimization and fix all the places where the unification algorithm relied on it.

4.5 Preserving constraints when copying types

When a function application is typed in OCaml, the type of the original function should not be modified. To achieve this, the OCaml type checker creates a copy of the function type before unifying its arguments. However, as polymorphic variant types are associated with constraints linking them to other types, these constraints must be preserved during the copy. For instance, the subtyping relation between the result type and the argument type must be preserved.

To ensure this, occurrences of polymorphic variant types are tracked during the copy, and the constraints associated with each of them are copied after the full type copy is completed. Because of this, we are preserving their internal relations. Moreover, copied non-internal constraints become marked as one-directional, as they should not affect the original type, but only become the new constraints for the copied type.

4.6 Detection of the polymorphic variables in function types

Consider the following function:

```
fun f v =
match v with
| ‘A -> C
| v  -> v
```

It is expected to have a type of "⊤ as 'a -> 'a | ‘C". However, simply solving for both variables results in a type of "⊤ -> ⊤" since neither of them has upper-bound constraints. To obtain the expected type, we need to consider that the argument type is present in the context when solving for the result type. Originally, this was achieved by physically equating the types on both sides of the arrow in the context. However, this method does not work in our case since the type should be solved with consideration of the context.

Therefore, two different functions were created to solve constraints: one for type checking without context, and another for human-readable printing with context.

4.7 Non-local unification failure

When a new constraint is added, it can affect all the types that are reachable in the constraint graph, even if they are not directly related. Therefore, in order to track typing issues, we need to check the solvability of the entire constraint system after adding any new constraint. However, the current implementation simply iterates over all the types, which can negatively impact performance. A less complex algorithm to address this issue could be implemented in the future.

5 Evaluation of the system

TODO

6 Conclusions

This work has presented a simplified version of the S-system[1], that is still extends the current system for polymorphic variants in OCaml. Despite being less expressive than the S-system, our proposed system resolves several issues of the current system. Furthermore, it simplifies the type inference algorithm, avoids overly precise types, and requires less implementation effort in the OCaml compiler.

We have also provided an implementation of the proposed system, which serves as a proof of concept for introducing subtyping to the OCaml type system. While the current implementation is a working prototype, it still has some performance and architectural issues that need to be addressed before presenting it to the OCaml community. Nonetheless, we believe that our work provides a promising starting point for further research in this area, and we hope that it will inspire other researchers to explore similar ideas.

References

[1] 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.

[2] 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.

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

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

[5] 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.

[6] “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.