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

by

Roman Venediktov

Bachelor Thesis in Computer Science

Submission: April 18, 2023

Supervisor: Prof. Alexander Omelchenko
Research assistant: Daniil Berezun
Research assistant: Dmitrii Kosarev

Constructor University | School of Computer Science and Engineering

Abstract

Warning: this text is in draft state, in many places in chapters 2 and 3 todo tags have been left for future formalisation and work. Despite this, I have tried to make it coherent and with the expected final structure.

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 a set-theoretic type system[1] with semantical subtyping has been proposed to address this issue, 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 with semantical subtyping 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 semantical subtyping and constraint solving. The resulting system offers a more intuitive and flexible approach to polymorphic variant typing in OCaml, with potential applications in software development and type theory research.

Contents

1 Introduction

Polymorphic variants are a key feature of OCaml that provide a balance between static safety and code reuse while maintaining a concise syntax. Initially proposed as a solution to incorporate union types into Hindley-Milner (HM) type systems [3], polymorphic variants cover many of the applications of union types, explaining their success. However, their capability is limited to finite enumerations of tagged values, rather than forming unions of values of generic types, 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 reconstruction, 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.

In [1], a system based on set-theoretic types and subtyping was proposed, which was shown to be strictly more expressive than the current system, typing 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. The authors demonstrate the superiority of their system over the current implementation in three examples where OCaml infers imprecise types or classifies them as type errors. Their system types them better and does not require excessive constructions from the programmer to fix typing problems. 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 two of the three given issues and remains strictly more expressive than the current implementation.

Although the system proposed in [1] 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 interop with the other subsystems of the OCaml language.

2 Statement and Motivation of Research

2.1 Issues with the current system

TODO: Examples from the [1]

2.2 Current OCaml 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 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: any supplementary information is represented through the kinds of type variables.

Definition 2.2 (Kinds)**.** A kind κ 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 $\mathcal{L}$ of all tags;
- T is a finite set of pairs of a tag and a type, written $\{tag_1 : \tau_1$, ..., $tag_n : \tau_n\}$ (its domain $\mathrm{dom}(T)$ is the set of tags occurring in it);
and where the following conditions hold:
- $L \subseteq U, L \subseteq \mathrm{dom}(T)$, and, $\mathrm{if}\, U \neq \mathcal{L}\, \mathrm{then}\, U \subseteq dom(T)$;
- Tags in L have a single type in T, that is, if $tag \in L$, whenever both $tag : \tau_1 \in T$ and $tag : \tau_2 \in T$, we have $\tau_1 = \tau_2$

Definition 2.3 (Type schemes)**.** TODO

Definition 2.4 (Admissibility of a type substitution)**.** TODO

Definition 2.5 (Instances of a type scheme)**.** TODO

The authors of the paper also provide proofs for the theorems of progress, subject reduction, and type soundness for the proposed system.

2.3 Set-theoretic system formalization (S-system)

2.3.1 Formalization

In their system, subtyping is added using a semantic definition based on set-theoretic types, rather than encoding it via instantiation. They utilize type connectives and subtyping 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 2.6 (Types)**.** A type τ is a term inductively generated by the following grammar.

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

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 $(\tau_l \cup \beta) \cap \tau_u$ where τ_l is the lower bound and τ_u is the upper bound, resulting in $(A(bool) \cup \beta) \cap (A(bool) \cup B(int))$.

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

Definition 2.7 (Type schemes)**.** TODO

The authors of the paper also provide proofs for the theorems of progress, subject reduction, and type soundness for the proposed system.

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.

2.3.2 Type reconstruction

Type reconstruction for a program (a closed expression) e consists in finding a type t such that $\emptyset \vdash Se : t$ can be derived: we see it as finding a type substitution θ such that $\emptyset \vdash Se : \alpha\theta$ 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 $\Gamma\theta \vdash Se : t\theta$ holds, if any such θ exists.

Reconstruction 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].

2.3.3 Issues with the system

The precision of the type system can lead to fragility, as functions with types like ”`string \ "Hello"`” may be intimidating for programmers.

The non-deterministic nature of the type reconstruction algorithm makes it difficult to support in the compiler and comprehend for programmers.

The ”tallying” algorithm has certain section where it must iterate through all potential solutions at each depth, which can result in exponential complexity.

The current implementation of the ”tallying” algorithm has bugs and may enter a constraint-solving loop.

3 Proposed system

3.1 Formalization

The main problems for the implementation were identified as the new basic types such as ”`1`” or ”`int \ 1`”. The introduction of these types 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. As a result, we decided to limit the use of set-theoretic operations only to the polymorphic variants subsystem.

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 is a term inductively generated by the following grammar.

$$\tau ::= a \mid b \mid \tau \rightarrow \tau \mid \tau \times \tau \mid (\kappa, \tau_t)$$
$$\tau_t = a_t \mid tag \mid \tau_t \cup \tau_t \mid \tau_t \setminus \tau_t \mid \top$$

where τ_t is the tagged-type and κ is the kind of tagged-type.

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

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

Definition 3.4 (Admissibility of a type scheme)**.** TODO

Definition 3.5 (Admissibility of a type substitution)**.** TODO

Definition 3.6 (Instances of a type scheme)**.** TODO

Definition 3.7 (Compatible Tagged-Types)**.** We say that two tagged-types are compatible if for each tag in their kind mapping, the mapped types are equal. (TODO)

Subtyping. In this system, subtyping is defined as in the S-system, but it is only defined on compatible tagged-types.

3.2 Properties

Theorems of Progress, Subject Reduction, and Type Soundness.

Proof that the Proposed System is the extension of the K-system.

Equivalence of the Proposed System with the S-system for programs without content in tagged values.

3.3 Types reconstruction

Due to the fundamental simplification of the system compared to the S-system, the type reconstruction 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. In our case, we solve these constraints using just the transitive closure on the graph of constraints.

Soundness and completeness of the type reconstruction algorithm

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 to make two types equal. Unification between types is called syntactically 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 "$\top$ `as 'a -> 'a | ‘C`". However, simply solving for both variables results in a type of "$\top$ `->` $\top$" 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 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 reconstruction 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.

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