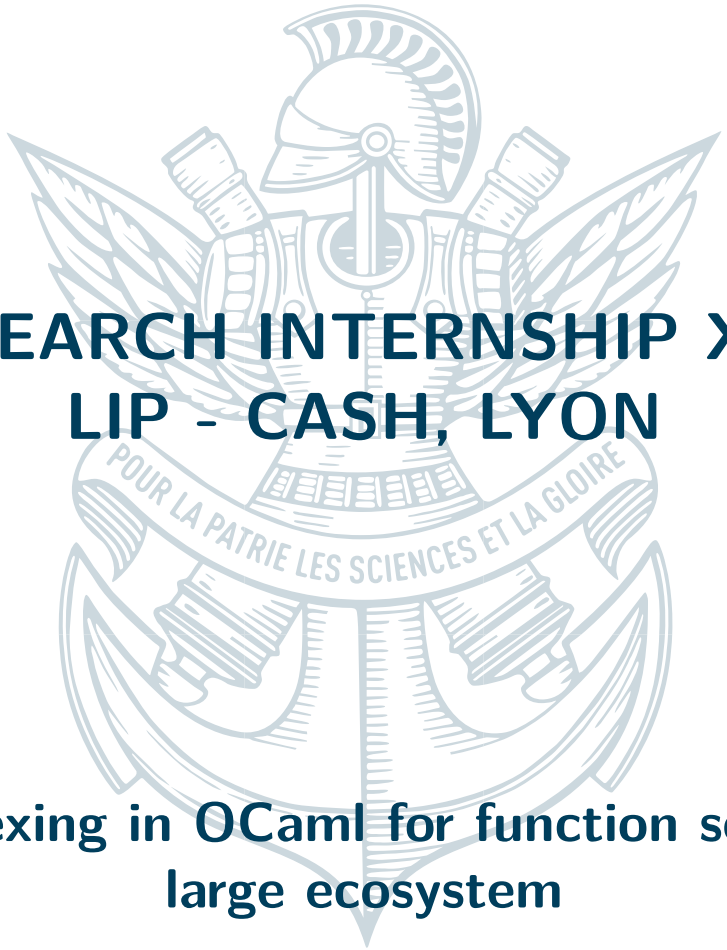

RESEARCH INTERNSHIP X-3A
LIP - CASH, LYON

Type indexing in OCaml for function search in a large ecosystem

June 20, 2022

Pauline Garelli
supervised by Gabriel Radanne and Laure Gonnord

ÉCOLE POLYTECHNIQUE

IP PARIS

ABSTRACT

When it comes to finding functions in libraries, in whichever language, programmers struggle with the need for efficient tools able to retrieve all relevant functions in a reasonable time. For the OCaml language, `dowsindex` was developed to offer a solution to this problem, via the notion of isomorphisms of types. It relies on the construction of an Index to precompute information on the libraries and thus facilitate the search algorithm. We aim to improve its performances by proposing a new data structure representing the matching relation between types, along with a new heuristic to optimize the type search.

ACKNOWLEDGEMENTS

I would like to thank my tutors Gabriel Radanne and Laure Gonnord for their support throughout this internship. Gabriel's daily presence allowed me to approach the subject with serenity and to progress efficiently. His good mood and his numerous tea-pots have always boosted me when I needed it. Laure knew how to bring a different point of view to allow us to move in the right direction. I thank her for her inspiring spirit, kindness and firmness.

Finally, I would like to thank the entire CASH team at LIP, who, through their warm welcome, their love for their work and their undisguised craziness, showed me that research in computer science is an exciting profession and something I may want to pursue in the future.

1 INTRODUCTION

Regardless of the language, programming is not only an art of ingenious creation of new algorithms, but also of the efficient use of already existing code. For this purpose, most languages have extensive libraries which offer solutions to all kinds of problems (processing of data structures, mathematical calculation, etc.). However, finding a specific function in increasingly large ecosystems can be an extremely tedious task. Often, search by name is proposed, but names of modules and the functions that compose them are quite arbitrary, and thus difficult for the programmer to find.

When we are looking for a function in a specific code, there is usually one property that we have a relatively clear idea of: the type of the function itself. Indeed, we know what we want the function to return, but also what arguments it will probably take. We could thus look for functions by doing a simple text search in the ecosystem of their types, but this would be too restrictive as it does not allow "approximative search": for instance, we might want to abstract the order of arguments or the eventual curryfication.

In functional programming and in strongly typed languages such as OCaml, the types of functions are easily accessible, which allows to develop searching algorithms based on isomorphisms of types (in which "similar types" are considered equivalent [5]) such as the work of Rittri [10]. Based on this model, Allain et al. [2] propose a tool for function search by type in the OCaml language, named `dowsindex`.

As type operations on type isomorphisms are very costly in terms of time complexity [9], C. Allain develops an heuristic on the search algorithm to improve its performances and allow `dowsindex` to scale to reasonable library sizes. In particular, it relies on the construction of an Index storing information beforehand on the types of a library.

Our work during this internship is a followup in the direction of using heuristics to improve the performances of `dowsindex`, in order to reach a scaling on even larger libraries, such as a whole Opam ecosystem. In that perspective, we develop new techniques to optimize type matching and library browsing, by extending the Index with a new data structure called Poset.

2
OUR TOOL: DOWSINDEX

The tool we have been working on during this internship is named `dowsindex`[1]. It offers support to search for functions by their types in an OCaml ecosystem. Here is a quick overview of the way one can use this tool.

First, `dowsindex` needs to load the libraries on which the developer will be working and in which it will be looking for functions. In the loading process, `dowsindex` will compute some information about the types of the functions contained in the libraries, and stock it into a data structure called the Index. We may pass as an argument the name of the libraries we want to work with, otherwise it will just load the entire Opam ecosystem.

```
$ dowsindex save <package>
```

For example, we might work mainly with the library `containers`[2] and thus want to find our functions in the ecosystem formed solely with this library. We then start with the following command when we use `dowsindex` for the first time:

```
$ dowsindex save containers
```

Our tool is now ready to be used for a type search. This is done with the following command:

```
$ dowsindex search <type>
```

A type will be written like a regular OCaml type in the form of a string, for example if we are looking for a function checking the presence of an element in a list of integers, we can search for the following type:

```
int list * int -> bool
```

Our first query will thus look like this:

```
$ dowsindex search "int list  * int -> bool"
```

The result for such a query in the containers package is the following:

```
CCList.memq : 'a -> 'a list -> bool
CCListLabels.memq : 'a -> set:'a list -> bool
```

[1] https://github.com/Drup/dowsing
[2] https://opam.ocaml.org/packages/containers/

We retrieve, as expected, the two "mem" functions of the library `containers`, from the `CCList` and `CCListLabels` modules.

As we can see, the order in which we give the arguments doesn't matter, nor the curryfication of the functions. Moreover, even if we ask for a function on lists of integers, `dowsindex` is able to find the "mem" functions, which are polymorphic, by looking for more general types than the one we provide. For this query, `dowsindex` takes $2 * 10^{-3}$ seconds to look inside the 2400 functions of the `containers` library.

The `dowsindex` tool also comes with some additional functions to measure the performances and do statistics on the search time. Indeed, we can measure the time taken by `dowsindex` to compare a query type with the ones in the library, grouped by a certain characteristic of types, for example the number of variables.

```
$ ./dowsindex stats "int -> int -> int" --measure vars
-------------------------------------------------------
measure    total time (ms)    avg. time (μs)    # unif.
-------------------------------------------------------
0          11.3714            11.3714            1000
1          0.613451           4.26008            144
2          5.2495             9.57938            548
3          1.19901            6.21247            193
4          1.85132            7.12046            260
5          57.7247            916.265            63
6          1.45769            12.6756            115
7          0.290155           13.8169            21
8          0.681877           11.9628            57
9          0.0581741          8.31059            7
10         0.00786781         7.86781            1
12         0.0140667          14.0667            1
14         0.0829697          20.7424            4
-------------------------------------------------------
total time (s): 0.0806022
total # unif.: 2414
```

The different lines correspond to the types grouped by their number of unique variables, which goes from 0 to 14 here. The `unif.` column indicates the number of comparisons between types `dowsindex` had to make. Without any optimization, the total matches with the number of types present in the library, because our tool has to compare the query type with each type of the library in order to retrieve all functions compatible with our type.

We can see here that most types in the library don't have any variable (i.e., are not polymorphic), and they are also taking quite a lot of time to be compared with the query type. Types with 5 unique variables also seem to be especially problematic. This information can be useful when trying to optimize our tool, because it gives us an overview of the most time-expensive operations for a given type search.

3
PROBLEM STATEMENT: USING TYPE ISOMORPHISMS FOR FUNCTION SEARCH

The approach chosen for function search in `dowsindex` is a search by types modulo isomorphism. This section explicits the notions of types and isomorphisms (Section 3.1), specifies the framework in which we use these concepts (Section 3.2) and exposes the main obstacle to an efficient search (Section 3.3), which motivates our work on the optimization of `dowsindex`.

3.1 The type isomorphism approach

3.1.1 • What is a type?

Types are a way of categorizing objects in a language, in order to give sense to the manipulated data and to check the validity of the actions performed on those objects. In a strongly typed language such as OCaml, any data structure must have a definite type and can only contain what its type indicates. For example, a list of integers in OCaml (type *int list*) cannot contain a boolean value (type *bool*), and only functions taking an `int list` or an `'a list` (i.e., which are parametric in their content, also called *polymorphic*) can be applied to it.

Definition 3.1.1 (Type τ)
Let $\mathcal{V}$ be a set of variables (expressing polymorphism in OCaml), and $\mathcal{F}$ a set of constructors indexed by their arity.

A type $\tau \in \mathcal{T}$ is of the following form:

$$\begin{aligned} \tau \in \mathcal{T} = \alpha \in \mathcal{A} & \qquad (Variable) \\ \mid \tau_1 \times \cdots \times \tau_n, n \in \mathbb{N} & \qquad (Tuple) \\ \mid \tau_1 \longrightarrow \tau_2 & \qquad (Arrow) \\ \mid f_i(\tau_1, ..., \tau_i), i \in \mathbb{N} & \qquad (Constructor) \end{aligned}$$

Note : This way of writing types will be used for our theoretic definitions and results throughout the report, but the OCaml syntax will be used in some examples for the sake of clarity. Example 3.1.2 exhibits the natural way of translating OCaml types into our theoretic frame.

Example 3.1.2
The Ocaml type `int * bool * 'a -> int list` is an arrow composed of a tuple and the constructor List of arity one, with arguments who are themselves composed of constructors and a variable.

Constructors are noted with their name indexed by their arity.

$$Int_0 \times Bool_0 \times \alpha \longrightarrow List_1(Int_0)$$

with $\alpha \in \mathcal{A}$.

Example 3.1.3
The empty tuple can be seen as the neutral element for tuples and will be noted *unit*, to remain consistent with the OCaml type `unit`.

The Definition 3.1.1 does not cover the whole subtlety of Ocaml's typing system, but this degree of precision will suffice to characterize most functions in the common libraries.

3.1.2 • What are type isomorphisms?

Our goal is to find types in a library which are close or equivalent to our query type, in order to draw the most relevant functions. But what does it mean for two types to be "close"?

Instinctively, we could say that two types are equivalent if there is an invertible function which transforms any function of the first type to a function of the second type, and its inverse which allows to go in the reverse way.

Example 3.1.1
Let f be a function of the following type: `int * bool -> bool list`

We can transform it into a function f' of the type `int -> bool -> bool list` through the following function:

```
let transform f x y = f (x, y)
in let f' = transform f
```

And in the other way:

```
let transform' f' (x,y) = f' x y
in let f = transform' f'
```

However, this semantic definition of equivalence is wide, and as shown later in Fig. 3, it can be very costly to compute. This is why we define a more restrictive definition of types which are equivalent for our search.

This theory is called ACIC due to the four rules it contains:

- **A**ssociativity: tuples are associative
- **C**ommutativity: tuples are commutative
- **I**dentity: `unit` is the neutral element for tuples
- **C**urryfication: arrows to the left of an arrow can be removed and the types grouped into tuples

Definition 3.1.2 (ACIC-Equivalence)
The relation of equivalence, noted $\equiv$, is the transitive closure of the reflexive and symmetric relation defined by the following rules:

$$\alpha \times \beta \equiv_{\mathrm{T}} \beta \times \alpha \qquad (\times\text{-commutativity})$$
$$\alpha \times (\beta \times \gamma) \equiv_{\mathrm{T}} (\alpha \times \beta) \times \gamma \qquad (\times\text{-associativity})$$
$$\mathit{unit} \times \alpha \equiv_{\mathrm{T}} \alpha \qquad (\times\text{-}\mathit{unit})$$
$$(\alpha \times \beta) \to \gamma \equiv_{\mathrm{T}} \alpha \to \beta \to \gamma \qquad (\text{curryfication})$$

Example 3.1.3
The following types are ACIC-equivalent:

- `int * float` $\equiv$ `float * int`
- `int * int -> int` $\equiv$ `int -> int -> int`
- `bool * (unit * int * float) -> int -> bool` $\equiv$ `float * (bool * int) * int -> bool`

The notion of equivalence allows us to equate functions which have almost exactly the same type as our query type, modulo the order and grouping of variables, the presence of unit type or curryfication.

Example 3.1.4
With only the equivalence, a query for type `int * (float * float) -> int list` could return the following results:

```
int -> float * float -> int list
float -> float -> int -> int list
float * (float * int) -> int list
unit -> float * float * int -> int list
```

3.1.3 • The matching relation

The search modulo ACIC-equivalence could be enough to return plenty of expected functions, but it might not return every function that could be useful. For example, we could be interested in a more polymorphic function, which would be able to perform the operation we are looking for not only on lists of integers but on lists of any type.

A way to find such functions is through the notion of matching. Indeed, the notion of matching includes that of equivalence, while allowing the instanciation of the variables in one of the two types.

An instanciation of a variable consists in the specification of the corresponding polymorphic type into a type without the variable in question.

Example 3.1.1
In OCaml, the instanciation of the variable `'a` as `int` will allow us to go from polymorphic type :
`'a * 'a list -> 'a list`
to the monomorphic type :
`int * int list -> int list`

Instanciation of variables can be formalized through the notion of a substitution.

Definition 3.1.2 (Substitution)
In a type theory with a set of variables $\mathcal{A}$, a substitution is a function σ from $\mathcal{A}$ to the set of all types $\mathcal{T}$.

Note : For the sake of clarity, substitutions will be noted as a set of correspondances (map) between variables and typed, ignoring fixed points.

Example 3.1.3
The following substitution σ associates the variables α and γ to other types, but all other variables are fixed points.

$$\sigma = \{\ \alpha \mapsto Int_0;\ \gamma \mapsto (List_1(Int_0) \longrightarrow Float_0)\}$$

The extension of the substitution will be the function allowing us to transform a polymorphic type into a less polymorphic type with instanciated variables.

Definition 3.1.4 (Extension of a substitution)
The extension $\hat{\sigma}$ of a subsitution σ is the only endomorphism of $\mathcal{T}$ whose restriction to $\mathcal{A}$ is σ.
In other words, $\hat{\sigma}$ can be inductively defined by:

$$\begin{aligned}
\hat{\sigma}(\alpha) &= \sigma(\alpha) && \forall \alpha \in \mathcal{A} \\
\hat{\sigma}(\tau_1 \times \cdots \times \tau_n) &= \hat{\sigma}(\tau_1) \times \cdots \times \hat{\sigma}(\tau_n) && \forall n \in \mathbb{N} \\
\hat{\sigma}(\tau_1 \longrightarrow \tau_2) &= \hat{\sigma}(\tau_1) \longrightarrow \hat{\sigma}(\tau_2) && \\
\hat{\sigma}(f_i(\tau_1, \ldots, \tau_{a_i})) &= f_i(\hat{\sigma}(\tau_1), \ldots, \hat{\sigma}(\tau_{a_i})) && \forall i \in I
\end{aligned}$$

If a type τ_1 can be obtained through the extension of a substitution σ from the type τ_2, we will write:

$$\tau_2 \succ_\sigma \tau_1$$

Note : For the sake of clarity, we allow by abuse of language to refer to the extension of a substitution as the substitution itself.

Example 3.1.5

$$(\alpha \longrightarrow Int_0 \longrightarrow \gamma) \succ_\sigma (Int_0 \longrightarrow Int_0 \longrightarrow \gamma)$$

with the substitution

$$\sigma = \{\ \alpha \mapsto Int_0;\ \gamma \mapsto \beta\}$$

This translates into the Ocaml syntax as:
$\sigma = \{$ `'a` $\mapsto$ `int` ; `'c` $\mapsto$ `'b` $\}$
`'a -> int -> 'c` $\succ_\sigma$ `int -> int -> 'b`

We are now able to define the notion of matching, noted $\preccurlyeq$, or more precisely ACIC-matching since it is dependant on the ACIC-equivalence theory.

Definition 3.1.6 (ACIC-Matching)
A type τ is said to match with a type τ' if their exists a type τ_0 such that:

$$\tau' \succ \tau_0 \equiv \tau$$

In other words, τ is equivalent to a type which can be obtained from τ' through a substitution.

We will write

$$\tau \preccurlyeq \tau'$$

or $\tau \preccurlyeq_\sigma \tau'$ if the substitution σ is known.

Example 3.1.7
The following types are matching:

- `int` $\preccurlyeq_\sigma$ `'a` with $\sigma = \{$ `'a` $\mapsto$ `int` $\}$
- `float * 'a -> 'a` $\preccurlyeq_\sigma$ `int list -> float -> int list` with $\sigma = \{$ `'a` $\mapsto$ `int list` $\}$
- `int -> 'a * 'a` $\preccurlyeq_\sigma$ `'c -> 'b * 'a` with $\sigma = \{$ `'b` $\mapsto$ `'a` ; `'c` $\mapsto$ `int` $\}$

With this new definition, we may search not only for types equivalent to our query τ, but for all types which are more general than τ modulo $\equiv$, i.e., all types τ' such that $\tau \preccurlyeq \tau'$.

Example 3.1.8
When querying for a type `int * int list -> int list`, we will now retrieve the following types via ACIC-matching:

- `int -> int list -> int list`
- `int -> 'a list -> 'a list`
- `'a list * 'a -> 'a list`

Note : The matching relation, unlike equivalence, is not symmetric, therefore we can write that `float -> int -> int` matches with `int * float -> 'a`, but `int * float -> 'a` does not match with `float -> int -> int` (the first type cannot be instantiated).

Theorem 3.1.9 (Matching is a partial order on types)
The matching relation is reflexive, transitive and antisymmetric modulo ACIC-equivalence. Thus, matching defines a partial order on types.

Proof.
Here are a few sketches of proof for each property:

- *Reflexivity*: done via an empty substitution (i.e., the identity function)
- *Transitivity*: done via a composition of substitutions
- *Antisymmetric*: if one of the substitution is empty, then we have the ACIC-equivalence by definition of the matching relation, or else we have $\tau \succ_\sigma \tau'$ and $\tau' \succ_{\sigma'} \tau$, σ must have instantiated only variables that are not in τ, or instantiated them into other variables in order for τ' to go back to τ via σ', so $\tau \equiv \tau'$ modulo possibly the name of the variables
- *Matching is not a total ordering*: `int * 'a -> bool` and `float` are not in a matching relation. It should be noticed that most pairs of types are actually uncomparable.

□

Note : We can notice that the matching relation embodies the intuitive notion of a type being "more general" than another. For example, `'a -> 'a list`, as a polymorphic type, is more general than `int -> int list`, which is translated in our theory as it being "greater" in the partial order established by ACIC-matching: `int -> int list` $\preccurlyeq$ `'a -> 'a list`

3.1.4 • Unification

Using the ACIC-matching relation as the criterion of our search tool will help us retrieve all expected functions in the majority of cases, but at times, we may want to also authorize the instantiation of our query type and not just the instantiation of types from the library.

For example, when looking for a printing function for lists, we might not know exactly which arguments it takes : only a list, or a list and a printer for the values of the list. Thus, we may ask for a function taking a type `'a * 'b list` as argument, and authorize the instantiation of our query type so that we can retrieve functions with more arguments than just the list.

In that perspective, we define the relation of unification between types, noted $\cong$.

Definition 3.1.1 (ACIC-Unification)
Two types τ and τ' are said to unify if there exists two equivalent types which can be obtained respectively from τ and τ' by substitution, ie:

$$\tau \cong \tau' \Leftrightarrow (\exists(\tau_0, \tau_1) : \tau \succ \tau_0 \wedge \tau' \succ \tau_1 \wedge \tau_0 \equiv \tau_1)$$

Example 3.1.2
The following types unify:

- `int * 'a` $\cong$ `'a * float`
- `float -> 'a -> int` $\cong$ `int * float -> int`
- `'a list -> int  -> ('a * int list)` $\cong$ `unit -> 'a * bool list -> ('a list * bool)`

Note : We can notice that the relations of matching and equivalence both imply a relation of unification, respectively where one or two substitutions are the identity.

Example 3.1.3
To summarize our notions, Fig. 1 gives an example for the relation between ACIC-unification, ACIC-matching and ACIC-equivalence.

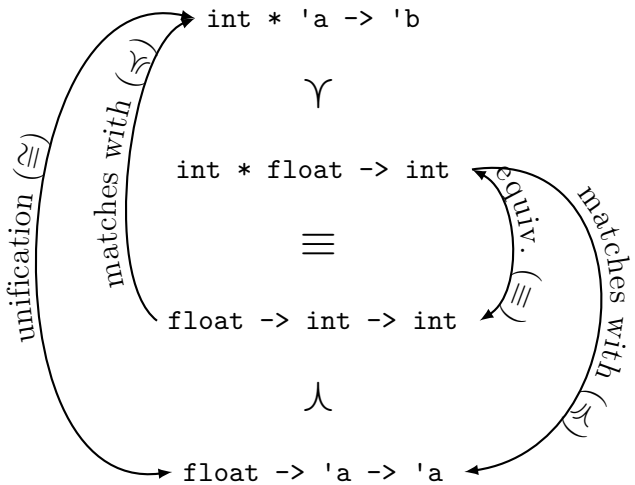

Figure 1: Equivalence, Matching and Unification

We have seen multiple ways of defining the notion of compatibility between types, namely ACIC-equivalence and the corresponding ACIC-matching and ACIC-unification. In the following Section 3.2, we will see how those notions have been integrated in `dowsindex`.

3.2 Unification: usage and complexity

In this section, we aim to explain which choices are made in `dowsindex` concerning the unification process and provide an overview of the general framework for unification.

3.2.1 • Equivalence in `dowsindex`

In order to compute ACIC-equivalence easily, `dowsindex` works on normalized types, which are defined as follows.

Definition 3.2.1
A normalized type $\hat{\tau}$ is either :

- a variable, $\alpha \in \mathcal{V}$
- a constructor $f_i(\hat{\tau_1}, \ldots, \hat{\tau_i})$
- a multiset of normalized and non-multiset types, which is not a singleton, $\{\hat{\tau_1}, \ldots, \hat{\tau_n}\}, n \neq 1$
- an arrow with arguments grouped as a non empty multiset $\{\hat{\tau_1}, \ldots, \hat{\tau_n}\} \longrightarrow \tau_0$

Example 3.2.2
In order to keep the writing of types consistant, we will write normalized types in the OCaml syntax, and interpret OCaml tuples as multisets.

The following types are normalized :

- `(bool * int) -> int`
- `( 'a ) -> 'a`
- `float list`

On the contrary, these types are not normalized :

- `(bool * (int * 'a)) -> int` (multiset in a multiset)
- `int -> int list -> int list` (arguments not grouped in a multiset)

We admit here that any type from $\mathcal{T}$ can be put in the form of a normalized type, along with the following theorem.

Theorem 3.2.3 (ACIC-equivalence on normalized types)
The ACIC-equivalence on types is the equality on normalized types.

Example 3.2.4
The types `int -> int list -> bool -> bool` and `(int list * int) * bool -> bool` are both normalized to the type `(int list * int * bool) -> bool`, and are thus ACIC-equivalent.

Theorem 3.2.3 simplifies our work greatly : once types have been put in normalized form at the beginning of the algorithm (as implemented by Allain [1]), we don't have to worry about ACIC-equivalence anymore since it corresponds exactly to the equality.

3.2.2 • Unification in `dowsindex`

The idea of using ACIC-unification to search for types in a library is proposed by Rittri [10]. The algorithm currently implemented in `dowsing` is an adaptation of the AC-unification algorithm proposed by Boudet [3] and is implemented by Gabriel Radanne.

As we saw through Section 3.1.4, instantiation of the query type can help us retrieve more relevant functions in some cases. However, matching is usually the relation adapted to function search. Thus, by default, `dowsindex` searches matching types for our query.

If we want to authorize the instantiation of some variables in the query type, we must specify it by using a different type syntax and naming explicitly the variables which are not instantiable, separated from the rest of the type by a point.

Example 3.2.1
Let's suppose we are looking for a printing function for lists. Such a function might only take a list as argument, it might also ask for a printer for the values of the list, or even more arguments. Since we don't know for sure which arguments it could take, we will allow to instantiate a variable in the arguments tuple:

```
$ dowsindex search "b . 'a -> 'b list -> unit"
```

Since we want a polymorphic printer, we cite `b` as a variable which is not instantiable, but we leave `a` free for instantiation.

3.2.3 • Equivalence theories

The ACIC-unification rules have not been selected randomly, but are linked to the definition of type isomorphism as the existence of an invertible function between functions of the two types. Further in this section, refer to that definition with the equivalence symbol $\simeq$. In a given set of types, $\simeq$-equivalence can be said to be "equivalent to" (i.e. defines the same isomorphisms as) a theory, i.e., a set of rules holding in that environment.

Definition 3.2.1
Let us introduce, as in [5] the following theory $Th^1_{\times T}$:

$$\begin{aligned}
\alpha \times \beta &\equiv_{\mathrm{T}} \beta \times \alpha && (\times\text{-commutativity})\\
\alpha \times (\beta \times \gamma) &\equiv_{\mathrm{T}} (\alpha \times \beta) \times \gamma && (\times\text{-associativity})\\
\mathit{unit} \times \alpha &\equiv_{\mathrm{T}} \alpha && (\times\text{-}\mathit{unit})\\
(\alpha \times \beta) \to \gamma &\equiv_{\mathrm{T}} \alpha \to \beta \to \gamma && (\text{curryfication})\\
\mathit{unit} \to \alpha &\equiv_{\mathrm{T}} \alpha && (\text{curry-}\mathit{unit})\\
\alpha \to (\beta \times \gamma) &\equiv_{\mathrm{T}} (\alpha \to \beta) \times (\alpha \to \gamma) && (\text{distribution})\\
\alpha \to \mathit{unit} &\equiv_{\mathrm{T}} \mathit{unit} && (\text{dist-}\mathit{unit})
\end{aligned}$$

Theorem 3.2.2 ($Th^1_{\times T}$ is sound [5])
In first-order λ-calculus with surjective pairing and unit type, the theory $Th^1_{\times T}$ is sound.

In other words, for two types τ and τ' in this λ-calculus, their equivalence in the $Th^1_{\times T}$ theory implies $\simeq$-equivalence, or:

$$\forall(\tau, \tau'), \tau \equiv_{Th^1_{\times T}} \tau' \Rightarrow \tau \simeq \tau'$$

Theorem 3.2.3 ($Th^1_{\times T}$ is complete [5])
In first-order λ-calculus with surjective pairing and unit type, the theory $Th^1_{\times T}$ is complete.

In other words, for two types τ and τ' in this λ-calculus, $\simeq$-equivalence implies equivalence in the $Th^1_{\times T}$ theory, or:

$$\forall(\tau, \tau'), \tau \simeq \tau' \Rightarrow \tau \equiv_{Th^1_{\times T}} \tau'$$

The soundness result is crucial for us: we want to retrieve only functions with types which are indeed equivalent to our query type.

However, completeness is less essential: it gives us the assurance that we will find all equivalent types, but this also implies a complex theory with many rules, some of which might not be that necessary to us. As long as we retrieve the most important functions, we may set aside the others.

This allows Rittri [10] to propose a specific theory for the research by type, among the different existent theories defined by the rules they allow and the ones they cut from $Th^1_{\times T}$. Here are a few theories less complete than $Th^1_{\times T}$ with the rules they have abandoned:

- Linear-$Th^1_{\times T}$: the two rules distribution and dist-*unit* correspond to isomorphisms which are not linear, and are therefore abandoned in the linear-$Th^1_{\times T}$ theory
- ACIC-theory: apart from the non-linear rules, it also cuts the curry-*unit* rule
- AC-theory: as its name suggests, this theory only keeps $\times$-commutativity and $\times$-associativity, it is a minimalist theory for type isomorphisms.

Then how do we choose out theory for type search? A reason for keeping ACIC-rules is their respective importance for function search:

- $\times$-commutativity and $\times$-associativity are essential if we want to forget about the order and the grouping of arguments in our functions
- $\times$-*unit* is allowing us to ignore the number of arguments by instantiating a variable as unit
- curryfication is essential to abstract the way of passing arguments to the function

The rules chosen for different theories of equivalence are summarized in Fig. 2.

3.2.4 • Complexity results on type isomorphisms

Another aspect and our main motivation in cutting rules from $Th^1_{\times T}$ is the complexity of the computation of equivalence, matching and unification in such a theory. Indeed, as shown in Fig. 3, unification is always very costly, forcing us to try to reduce the complexity of its computation as much as possible.

The main obstacle to choosing a complete theory such as $Th^1_{\times T}$ is that the unification modulo this theory is undecidable, i.e., there is no algorithm giving an answer to the problem

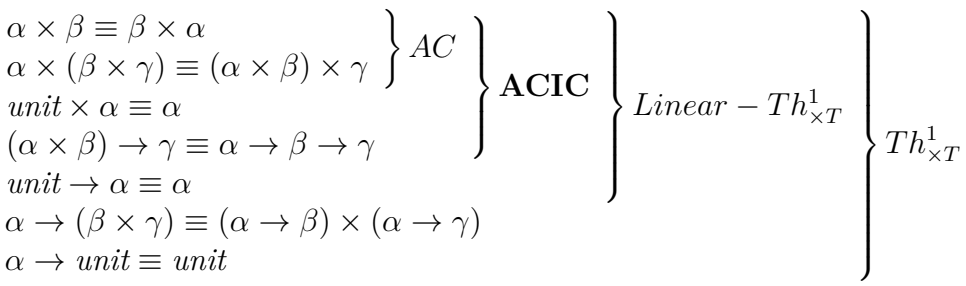

Figure 2: Rules for equivalence theories

Theories	$Th^1_{\times T}$	Linear-$Th^1_{\times T}$	**ACIC-theory**	AC-theory
Equivalence	?	Polynomial [12]	Polynomial	Polynomial
Matching	NP-complete for terms in normal form [9]	NP-complete	NP-complete	NP-complete [8]
Unification	Undecidable [9]	NP-complete [9]	NP-complete	NP-complete [8]

Figure 3: Complexity of unification and matching operations depending on the equivalence theory

of two types unifying or not in finite time. However, when we lose some rules from the complete theory, we obtain smaller theories where the unification problem is only NP-complete[3]. From there comes the desire to remove as many unnecessary rules as possible from $Th^1_{\times T}$, and thus to choose the ACIC-theory for our problem.

However, unification is still very costly in that theory, which motivates further work on our tool to obtain the result of a function search in reasonable time.

3.3 Our approach for search by types

The search for functions using type isomorphisms was first proposed by M. Rittri in [10] and implemented in the ML language.

It consists in the application of the unification algorithm on a query type successively with all other types from the library on which we are searching, and it returns all types matching with our query type.

Note : Although we are trying to retrieve only the types in a matching relationship with our query type, we are still using the unification algorithm for two reasons:

- to be able to allow punctually the instantiation of some variables in the query

[3]Note that in our case, considering the size of the entries (several thousands of functions in a library of reasonable size), this complexity still implies extensive computation times

- since matching modulo ACIC-equivalence is also NP-complete, we don't lose much in using the unification algorithm directly

The goal of `dowsindex` is first to implement this idea in the OCaml language, which is also very adapted to the problem, and then to allow it to scale on larger function ecosytems than it previously did for the ML language (*Lazy ML* library of 294 types).

3.3.1 • The wall of unification complexity

As we mentioned in Section 3.2.4, solving the unification problem comes at a high cost, even when the result is negative (*i.e* two types do not unify).

Concretely, when performing an exhaustive search with ACIC-unification in the library *containers*, by attempting to unify our query type with each type in the library, we obtain the following results:

```
$ dowsindex stats "'a * 'b -> (int -> 'b list) * 'a list -> 'a * 'b list"
-------------------------------------------------------------
measure         total time (ms)    avg. time (μs)    # unif.
-------------------------------------------------------------
variable        10518.8            53125.3            198
constructor     27.734             15.3142            1811
tuple           1857.82            4632.97            401
other           0.0331402          8.28505            4
-------------------------------------------------------------
total time (s): 12.4044
total # unif.: 2414
```

For a highly polymorphic type such as this one, the search took 12.4 seconds for only 2414 types in *containers*, which is a rather small library.

Clearly, an exhaustive function search by types modulo isomorphism does not scale to a bigger environment. Since our theory is already the simplest one we could use while keeping the necessary properties of isomorphisms for a search by types, we need to find another way of working around the unification's complexity problem.

As we have seen, the unification procedure is very costly. Our goal for the rest of this report is thus to attempt to pre-process our collection of functions in order to avoid as many unification as possible:

- We first present in Section 4 the original technique developed in `dowsindex` by C. Allain, which uses the shape of types to index and classify them.
- We then present in Section 5.2 a new technique which uses the relation between types to avoid unifications.

4
INDEXING SHAPE PROPERTIES ON TYPES

The first strategy used in `dowsindex` to decrease the number of unifications is to pre-compute simple "shape" properties on types allowing us to deduce the unification relation between types without having to actually run the unification algorithm. Those properties are stored in the `Index` and then retrieved when needed during the search. This approach was implemented by C. Allain last year in `dowsindex`.

In this section, we first motivate this approach with measures from practical examples. We then present the notion of Feature and their Index.

4.1 Exploring the performance of unification

The optimization strategy proposed by Allain et al. [2] for `dowsindex` is divided in two steps. The first step is to identify patterns of the most complex types for unification, namely the ones taking the most time to unify with a given query type. Then, we attempt to find properties of those types, which could help us avoid the use of our unification algorithm on them and thus gain in computation time on our search.

For this purpose, C. Allain developed *measures* on types to understand which properties of types could make them especially difficult to unify with.

Example 4.1.1
A measure used by C. Allain was the type of the head.

Definition 4.1.2 (Measure on types: Head)
The Head measure is a function from the set of all types $\mathcal{T}$ to itself, which associates, to each type of our theory, the type of its head ν.

The head ν can be seen as the outermost right element from a type, or the return type of a function. It is defined by:

$$\begin{aligned} \nu(\tau \longrightarrow \tau_0) &= \tau_0 \\ \nu(\tau) &= \tau \qquad\qquad \text{for all other types} \end{aligned}$$

Example 4.1.3
Here are the heads of a few types:

- $\nu($ `int * bool -> 'a list -> 'a` $) =$ `'a`
- $\nu($ `float` $) =$ `float`
- $\nu($ `int list -> (int * int)` $) =$ `int * int`

The procedure was to look at a type search in the library and to time the sum of all unifications, grouped by the values of types for the measure.

They obtained the following result:

```
$ ./dowsindex stats "int -> int -> int" --measure head
-------------------------------------------------------------
measure        total time (ms)    avg. time (µs)    # unif.
-------------------------------------------------------------
variable       506.137            311.086            1627
constructor    62.705             2.45229            25570
tuple          11.2598            2.73429            4118
other          0.815153           3.09944            263
-------------------------------------------------------------
total time (s): 0.580917
total # unif.: 31578
```

The conclusion of the experiment for this measure was that types with a variable as head were way more difficult than others to unify with. This gave an indication on the most important types to avoid a unification with, and thus the kind of properties we might be looking for on types.

4.2 Features: a type property

The main principle of Features is to find easily computed properties which could, in some cases, ensure us that two types are NOT unifiable, and therefore save us from applying the unification algorithm on them.

A Feature designates the association of an encoding function and an unification criterion. Namely, the encoding function is a function of types, which retrieves an interesting property with respect to ACIC-unification. The corresponding criterion is a theorem used to decide whether two types may be unified or cannot be unified at all, based on their encoding value.

4.2.1 • Encoding function on types

We illustrate here the notion of encoding function with two examples of Features used in `dowsindex`, Head and Arity.

Definition 4.2.1 (Encoding function: Head)
The Head encoding function associates to a type in $\mathcal{T}$ the type of its head denoted ν, as introduced in Definition 4.1.2.

Note : The encoding function of the Head feature is exactly the same function as the Head measure, but the two notions should not be confused: while the Head measure is a way of sorting types according to their head when doing statistics on the performances of our tool, the Head feature is part of an optimization of `dowsindex` in the sense that it gives a theoretic frame allowing to avoid the computation of the unification for some types.

Definition 4.2.2 (Base of types)
The base is a multiset defined on normalized types (see Section 3.2.1) by :

- $B(\alpha) = \{\alpha\}$
- $B(f_i(\hat{\tau_1}, \ldots, \hat{\tau_i})) = (\bigcup_{k=1}^{i} B(\hat{\tau_k})) \bigcup \{f_i\}$
- $B(\{\hat{\tau_1}, \ldots, \hat{\tau_n}\}) = \bigcup_{k=1}^{n} B(\hat{\tau_k})$
- $B(\{\hat{\tau_1}, \ldots, \hat{\tau_n}\} \longrightarrow \tau_0) = \bigcup_{k=0}^{n} B(\hat{\tau_k})$

where $\bigcup$ is the union on multisets.

It can be seen as the multiset of all constructors and variables present in a type.

Definition 4.2.3 (Tail of types)
The tail is a multiset defined on normalized types by :

- $B(\{\hat{\tau_1}, \ldots, \hat{\tau_n}\} \longrightarrow \tau_0) = \{\hat{\tau_1}, \ldots, \hat{\tau_n}\}$
- $B(\hat{\tau}) = \emptyset$ for all other types

It can be seen as the multiset of arguments of an arrow type, which is empty if the type is not an arrow.

Definition 4.2.4 (Encoding function: Arity)
The Arity encoding function associates to a type τ a pair $\mathcal{A}(\tau) = (\texttt{is\_var}, \texttt{nb\_const}) \in \mathcal{B} \times \mathbb{N}$ defined by:

- Boolean `is_var`: this value indicates whether τ contains any variable in its base
- Int `nb_const`: this value indicates the number of constructors in the tail of τ

Note : This function encodes, as its name indicates, the arity of a function with type τ. Indeed, we count the number of constructors in the tail, which corresponds to the arguments of the function. The presence of a variable in the base indicates that there could be more constructors by instantiation.

Therefore, a condensed way of denoting the Arity encoding function is as follows:

- $(\texttt{false}, i)$ is noted $(= i)$ forall $i \in \mathbb{N}$
- $(\texttt{true}, i)$ is noted $(\geq i)$ forall $i \in \mathbb{N}$

From now on, we use this notation for Arity encoding.

Example 4.2.5
Here are the values of a few types with respect to the Arity encoding :

- $\mathcal{A}$(`(int * bool) -> int`) = (= 2)
  (no variable and two constructors on the left of the arrow)
- $\mathcal{A}$(`(int * int list * 'a) -> 'a`) = ($\geq$ 3)
  (a variable and three constructors on the left of the arrow)
- $\mathcal{A}$(`int`) = (= 0)
  (no variable and no constructor on the left of the arrow, since it is not an arrow)

4.2.2 • Unification criterion

The second part of a Feature is a theorem (or criterion) giving the incompatibility for unification of some types based on their values of the corresponding measure. Let's take again the example of the Head Feature.

Theorem 4.2.1 (Criterion for unification: Head)
If two types are ACIC-unifiable, and that none of their heads is a variable, then their heads are equal, or:

$$\forall(\tau,\tau') \in \mathcal{T}^2, \nu(\tau) \notin \mathcal{A} \wedge \nu(\tau') \notin \mathcal{A} \wedge (\tau \cong \tau') \Rightarrow \nu(\tau) = \nu(\tau')$$

Theorem 4.2.2 (Corollary)
The reciprocal of this property gives us the following property:

$$\forall(\tau,\tau') \in \mathcal{T}^2, \nu(\tau) \notin \mathcal{A} \wedge \nu(\tau') \notin \mathcal{A} \wedge (\nu(\tau) = \nu(\tau')) \Rightarrow (\tau \ncong \tau')$$

In other words, if two types have different heads, none of which is a variable, then they are not unifiable.

Example 4.2.3
The following pairs of types are not unifiable:

- `'a * int -> int` and `'a -> bool` because their heads are not variables and are not equal (`int` $\neq$ `bool`)
- `('a -> int) * bool -> int list` and `float` because their heads are not variables and are not equal (`int list` $\neq$ `float`)

The following pairs of types may be unifiable according to the Head Feature:

- `unit -> int -> 'a` and `float -> float` because one of the heads is a variable
- `bool * int -> int` and `int -> int` because their heads are equal
- `'a list -> int -> 'a` and `int list -> int -> int` because one of the heads is a variable

Note : If we compute the values of the head for all types in the library and for our query, a quick comparison between them can allow us to decide whether they are not unifiable, or if they may be unifiable (notice that in that case, we gain no information on the unification between the types in question, we only know that they may unify).

We can do the same for the Arity Feature.

Theorem 4.2.4 (Criterion for unification: Arity)
If two types τ and τ' unify, then the arities of the ACIC-equivalent types they instantiate to are equal.

The contraposition of this theorem translates naturally with the Arity encoding as the following corollary.

Theorem 4.2.5 (Corollary)
Two types τ and τ' cannot unify if their Arity encoding are not compatible, i.e. in the following cases:

- $\mathcal{A}(\tau) = (= i)$ and $\mathcal{A}(\tau') = (= j)$ with $i \neq j$
- $\mathcal{A}(\tau) = (= i)$ and $\mathcal{A}(\tau') = (\geq j)$ with $i < j$
- symmetrically, $\mathcal{A}(\tau) = (\geq i)$ and $\mathcal{A}(\tau') = (= j)$ with $i > j$

Example 4.2.6
The following types cannot unify since their Arity encoding are not compatible :
- `int -> int -> int` of arity $(= 2)$ and `float -> float` of arity $(= 1)$
- `'a list * bool * int -> 'a` of arity (≥ 3) and `int list -> int list` of arity $(= 2)$

The following types may, however, be unifiable according to the Arity Feature :
- `int -> int -> int` of arity $(= 2)$ and `float * float -> float` of arity $(= 2)$
- `'a list * bool * int -> 'a` of arity (≥ 3) and `'a -> 'a` of arity (≥ 0)

4.3 Storage: structure of Trie

In order to store the values of all types for each Feature and to access them in an efficient way, we use a specific data structure called a Trie.

Our Trie of Features is a rooted tree as shown in Fig. 4 where the leaves are sets of types from the library. Each level of depth in the tree corresponds to a Feature, and the nodes are possible values of the corresponding feature.

Example 4.3.1
The type τ : `int * 'a list * 'a -> bool` takes the following values for the Features' measures:

- *Head*: $\nu(\tau) =$ `bool`
- *Arity*: $\mathcal{A}(\tau) = (= 2)$

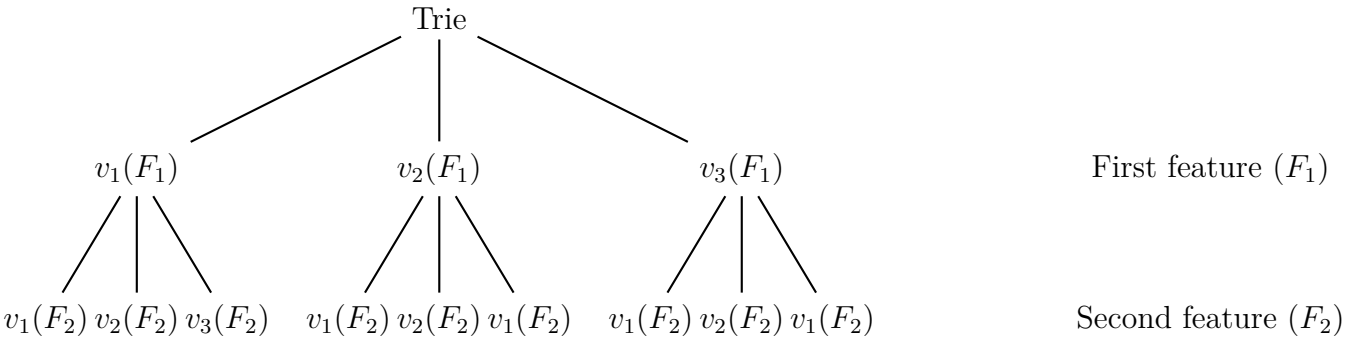

Figure 4: Structure of the Trie ($v_3(F_2)$ is the third value of the second feature)

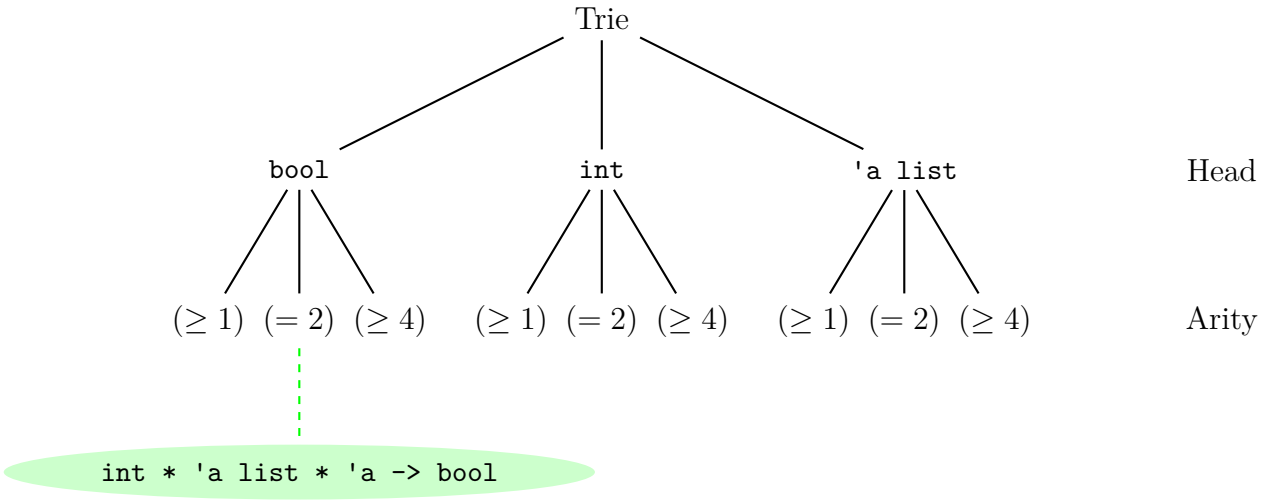

Figure 5: The place of type `int * 'a list * 'a -> bool` in a simple Trie

Therefore, in a two-level Trie with the Features *Head* and *Arity*, this type would be in the subtree of types with *Head* value `bool`, and in that subtree, it would be placed in the subsubtree for types with *Arity* value (= 2), as shown in Figure 5.

This structure of Trie allows us to easily retrieve the set of all types with specific values for different Features.

4.4 Querying the Trie

Once the Trie has been statically computed, it contains all the types of the library, sorted by their feature values. Therefore, given a query type, we can browse the Trie to retrieve all possibly compatible types of the library according to all Features. This process is done by visiting the different branches of the Trie and eliminating, at each level, the subtrees corresponding

to uncompatible values with the query type.

Example 4.4.1
For the query type τ : `int * 'a list * 'a -> bool`, the *uncompatible* values for the different Features are as follow:

- *Head:* types τ' such that $(\nu(\tau') \neq \alpha \in \mathcal{A}) \wedge (\nu(\tau') \neq \texttt{bool})$
- *Arity:* types τ' such that $(\mathcal{A}(\tau') = (= i), i \neq j) \vee (\mathcal{A}(\tau') = (\geq i), i \geq 3)$

4.5 Results on the use of the Index

With this approach, on a Trie built with only two Features, C. Allain was able to decrease the search time of most types by at least 70% and up to 99%. Table 1 is an extract of those results, presented in the research report [1].

type	total time (s)		nb. unifications	
	without criterion	with criterions	without criterion	with criterions
$int \to int \to int$	0.638	0.001 (0.23%)	31578	121 (0.38%)
$int \to int \to int \to int$	1.622	0.001 (0.08%)	31578	107 (0.34%)
$int \to (int \to int) \to list(\alpha)$	11.193	0.011 (0.93%)	31578	141 (0.45%)
$int \to float \to bool \to unit$	0.636	0.004 (0.60%)	31578	126 (0.40%)
$\alpha \to int \to unit$	1.180	0.225 (19.06%)	31578	2443 (7.74%)
$int \to int \to \alpha$	5.097	1.566 (30.71%)	31578	3677 (11.64%)

Table 1: Comparison of unification time with and without the use of Features

Yet for some types, the time of search can go up to several seconds for especially polymorphic types, even though our search ecosystem is still of a reasonable size (around 30000 functions to browse in here).

Though we could have kept looking for improvements through new features, we decided to try another approach and look for a different kind of properties on types with respect to unification. Our contribution is the definition of a new Index data structure based on type relations, as described in the next section.

5
CONTRIBUTION: A MATCHING POSET TO LEVERAGE RELATIONS BETWEEN TYPES

Although this previous work on `dowsindex` improved greatly the performances of the tool, our type search still takes too much time for some queries, as we can see for `int -> int -> 'a` which took 5 seconds to return its result. There is therefore a need for more research on how to improve our tool's performance. One particular source of difficulty are queries with high polymorphism, which are harder to classify by simply using the shape as was done in the last section.

For this purpose, we investigate a new lead: using relation between types, and notably the *matching* relation presented in Section 3.1.3, to statically build a new structure called the Matching Poset to shortcut unification. We first present the theoretical result underlying this idea in Section 5.1, before presenting the notion of Matching Poset (Section 5.2), how to build it (Section 5.3) and how to query it in combination with the Feature Index (Section 5.4).

5.1 Matching: a relation between types

The principle of Features is to store information on the types and use them to deduce the result of some unifications without having to perform our unification algorithm. In our new approach, the principle is essentially the same, except for the kind of information we will be storing: instead of looking for properties of the types themselves, we will be looking at properties of the relation between the types, the matching relation in this case.

We saw in Section 3.1.3 that the matching relation defines a partial order on types. This property has a great advantage: it makes it easy to store the matching relation between all types in the library, in order to use those relations for well-chosen properties concerning the unification of types. In the following section, we expose two theorems which serve as a basis for our improvement of the `dowsindex` tool.

5.1.1 • Theorems on matching

The two following theorems are propagating unification properties based on the matching relation. Namely, they are allowing us to deduce new unifications from the ones we know between our query type and the library's types, or on the contrary to deduce types that or not-unifiable based on types already known as not-unifiable.

Those theorems both confirm some rather intuitive ideas on types, linked to the interpretation of the matching relation as a measure of types being more general than others.

Theorem 5.1.1 (Matching theorem 1)
If two types τ_q and τ_{lib} unify, then τ_q also unifies with all types which are more general than τ_{lib}.

$$\forall(\tau_q, \tau_{lib}), \tau_q \cong \tau_{lib} \implies (\forall \tau'_{lib}, \tau_{lib} \preccurlyeq \tau'_{lib} \Rightarrow \tau_q \cong \tau'_{lib})$$

Proof.
The proof is done by working on normalized types for the ACIC-equivalence, and composing the substitutions from the unification on τ_{lib} and the matching on τ'_{lib}. □

Example 5.1.2
Since the types `int -> float -> 'a list` and `(float * unit * int) -> bool list` unify, we can deduce the following unifications between `int -> float -> 'a list` and types which are more general than `(float * unit * int) -> bool list`:

- `int -> float -> 'a list` $\cong$ `('a * unit * int) -> bool list`
- `int -> float -> 'a list` $\cong$ `float -> 'a -> int -> 'b list`

Theorem 5.1.3 (Matching theorem 2)
If two types τ_q and τ_{lib} don't unify, then τ_q does not unify with any type which is less general than τ_{lib} either.

$$\forall(\tau_q, \tau_{lib}), \tau_q \ncong \tau_{lib} \Rightarrow (\forall \tau'_{lib}, \tau'_{lib} \preccurlyeq \tau_{lib} \Rightarrow \tau_q \ncong \tau'_{lib})$$

Example 5.1.4
Since the types `'a -> bool` and `'a -> 'b -> 'a list` don't unify, we can deduce that `'a -> bool` does not unify with the following types which are less general (i.e., more specific) than `'a -> 'b -> 'a list`:

- `'a -> bool` $\ncong$ `bool -> 'b -> bool list`
- `'a -> bool` $\ncong$ `(int * int) -> int list`

As we just saw, Theorem 5.1.1 and 5.1.3 give us the result of the unification of types under certain conditions. This lead to the main idea of our contribution: during our search, whenever we do the computation of the unification between the query type and a type in the library, we can propagate the result to other pairs of types by following the matching relation. Thus, we can avoid the computation of the unification for those types.

Note : Actually, we will only use Theorem 5.1.3 on the propagation of non-unification.

The main argument for this choice is linked to the interest of the unification algorithm, which computes more than just the relation of unification between two types. Indeed, when two types unify, the algorithm also returns the unifiers, which are the substitutions leading to the two ACIC-equivalent types justifying the unification relation. Those unifiers could be useful to sort the list of the retrieved functions, in the sense that the size of unifiers is a good measure

of the proximity between two types, and thus smaller unifiers may correspond to functions which are better suited for our query.

Yet, if we were to use Theorem 5.1.1, we would deduce the unification without running the algorithm, and thus we wouldn't have access to the unifiers.

5.2 The Matching Partial Order Set

To leverage these two theorems, we introduce a new structure called the Matching Partially Ordered Set, or *Poset* for short, which contains the whole information on the matching relation between types of the libraries.

5.2.1 • Structure of the Poset

The natural representation of partially ordered sets are Directed Acyclic Graphs (DAG). The vertices of such a **graph** represent the elements of the set, that is types of library functions in our case, and the edges represent the binary relation between them, that is the partial order "matching" here.

Note : Since ACIC-equivalence is already resolved by manipulating normalized types (see Section 3.2.1), we can consider types in the quotient set of $\mathcal{T}$ by $\equiv$. This allows us to refer here to the matching relation as just a matter of substitution, ignoring the equivalence.

Since the matching relation is not symmetric, the edges are **directed**. An edge between types τ_1 and τ_2 means therefore:

$$\tau_1 \preccurlyeq \tau_2$$

, namely the type τ_1 can be obtained from an instanciation of the variables of τ_2.

Finally, the graph can be proven to be **acyclic** in the case of a partially ordered set. Supposing the existence of a cycle $\tau_0 \leq \tau_1 \leq ... \leq \tau_n \leq \tau_0$ in our graph, by successive compositions of the substitutions leading from τ_0 to τ_n, from τ_n to τ_{n-1} and so on, there would then be a non-empty substitution (i.e., different than the identity) going from τ_0 to itself, which is absurd.

The main interest of the graph is to be able to retrieve all types which are in a matching relation with a given type τ: the types τ' such that $\tau' \preccurlyeq \tau$ are actually the whole offspring of τ in the graph, while the types τ'' such that $\tau \preccurlyeq \tau''$ are its ancestry.

Since the relation is transitive, we do not need to keep track of all relations between types but only the corresponding the transitive reduction of the matching relation (i.e., the smallest relation of which matching is the transitive closure), therefore gaining space by only representing necessary edges.

5.2.2 • Implementation

For the implementation of this Directed Acyclic Graph, we used the `OcamlGraph`[4] library. Thanks to the GraphViz[5] extension of this package, we can obtain graphical representations of our graphs, as illustrated in Example 5.2.1.

Example 5.2.1
A Poset on four types (`'a -> 'b`, `int -> 'b`, `'a -> int`, `int -> int`) is depicted in Fig. 6.

The type `'a -> 'b` can be instanciated either to `int -> 'b` through $\sigma_1 = \{$ `'a` $\mapsto$ `int` $\}$ or to `'a -> int` with $\sigma_2 = \{$ `'b` $\mapsto$ `int` $\}$, which can both be instanciated to `int -> int` via the respective substitutions σ_2 and σ_1.

This leads to the following matching relations :

- `int -> int` $\preccurlyeq$ `int -> 'b`
- `int -> int` $\preccurlyeq$ `'a -> int`
- `int -> 'b` $\preccurlyeq$ `'a -> 'b`
- `'a -> int` $\preccurlyeq$ `'a -> 'b`

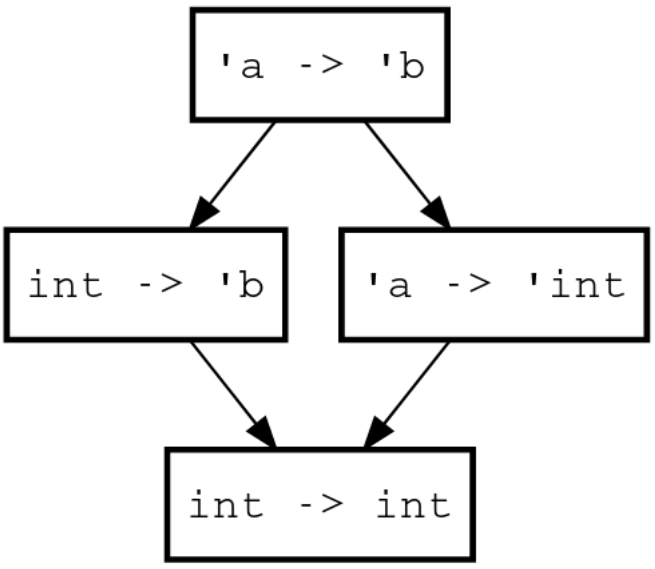

Figure 6: A simple Poset

[4] `http://ocamlgraph.lri.fr/`
[5] `https://graphviz.org/`

5.3 Building the Poset from the library: insertion algorithm

While done at pre-computation time, the Matching Poset can still be large for big libraries. Nevertheless, it should be very shallow: most types are not in relation with each others. We now describe an iterative algorithm to build the Poset and take advantage of its sparseness.

5.3.1 • Principle of the insertion algorithm

The construction of the Poset is done by inserting all the library types one by one in the graph. Let's consider τ_0, a type to be inserted in a current poset.

Already inserted types in the Poset can be divided in four groups depending on their matching relation to τ_0, namely:

- Bigger : the types τ which are more general than τ_0, i.e., such that $\tau_0 \preccurlyeq \tau$;
- Smaller : the types τ which are more specific than τ_0, i.e., such that $\tau \preccurlyeq \tau_0$;
- Equal : the types that are τ equal to τ_0 is the sense of ACIC-equivalence, i.e., such that $\tau \equiv \tau_0$ (which is equivalent to τ being both Bigger and Smaller than τ_0)
- Uncomparable : since the matching relation is not a total order, τ can be neither Bigger nor Smaller than τ_0

If we stored all the matching relations in the Poset, we would have to link τ_0 to all the types in the Bigger and the Smaller groups. However, we are only interested in the transitive reduction of the matching relation. Therefore, we will need to determine three sets:

- the set of types which will be parents (immediate ancestors) of τ_0
- the set of types which will be children (immediate sucessors) of τ_0
- the set of edges which should be removed in order to keep the transitive reduction of the matching relation

Note : An interesting property of the Poset is that when comparing τ_0 with the other types, the resulting groups are spatially delimited in the Poset, as shown in Proposition 5.3.1.

Proposition 5.3.1
If we exclude the possible Equal zone (singleton τ_0), the three remaining groups of types can be identified as three delimited zones :

- Bigger zone : types Bigger than τ_0 are grouped in the upper part of the Poset and are delimited by a "lower upper bound" (LUB)
- Smaller zone : types Smaller than τ_0 are grouped in the lower part of the Poset and are delimited by an "upper lower bound" (ULB)
- Uncomparable zone : the zone formed by the rest of the types which are Uncomparable with τ_0 constitutes the central part of the Poset

Proof.
The delimitation of those zones comes from transitivity of the matching relation. If a type

matches with our query type τ_0 (i.e., is Smaller), then all its descendants in the Poset will also match with it (i.e., be Smaller). Symmetrically, if τ_0 matches with a type (i.e., that type is Bigger), then it also matches with its ascendants (i.e., they are also Bigger). □

An example of a Poset with delimited zones is given in Example 5.3.2.

Example 5.3.2 (Zones in a Poset)
Let us consider $\tau_0 =$ `int * 'a -> int` to insert in the Poset depicted in Fig. 7.

The Bigger zone is marked red, the Smaller zone is marked green, and the Uncomparable zone is marked orange.

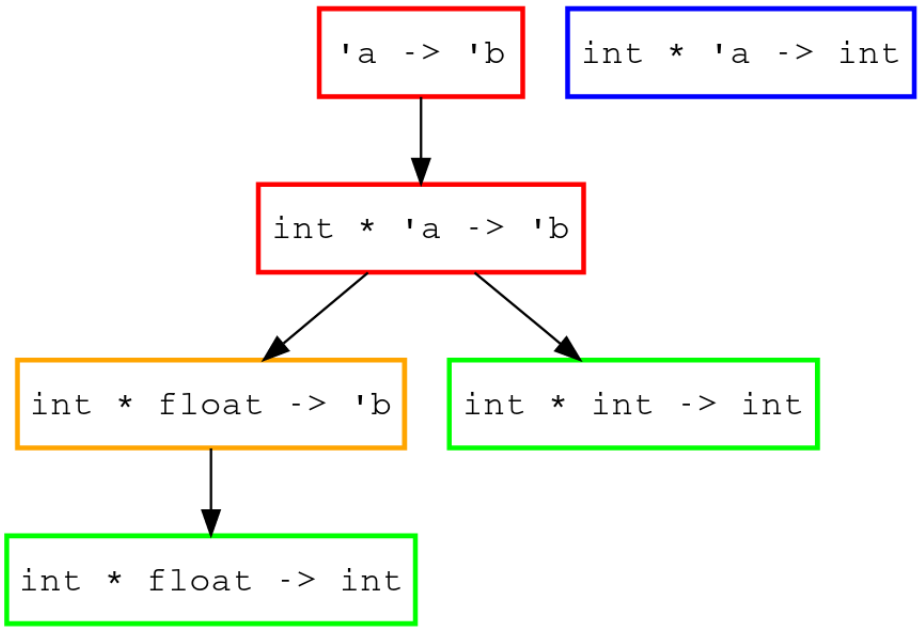

Figure 7: Zones for the insertion of type `int * 'a -> int` in a small Poset

From this results the following definition of the LUB and ULB zones :

- a type in the "lower upper bound" is Bigger than τ_0 and so are its ancestors, while its descendants are all either Uncomparable or Smaller than τ_0
- symmetrically, a type in the "upper lower bound" is Smaller than τ_0 and so are its descendants, while its ancestors are all either Uncomparable or Bigger than τ_0

The types contained in those bounds are actually the ones which should be linked to τ_0. When inserted in the Poset, the type τ_0 will be pointed to by all types of the LUB and point to all types of the ULB. Moreover, the edges between a type from LUB and a type from ULB should be removed, since they will now be obtained through τ_0 by transitivity of the matching relation.

Example 5.3.3
In the previous example, members of the LUB and ULB zones for the insertion of `int * 'a -> int` would be :

- *LUB-zone*: `int * 'a -> 'b`
- *ULB-zone*: `int * float -> int` and `int * int -> int`

The edges to be added are shown in blue and those to be removed in grey in Fig. 8.

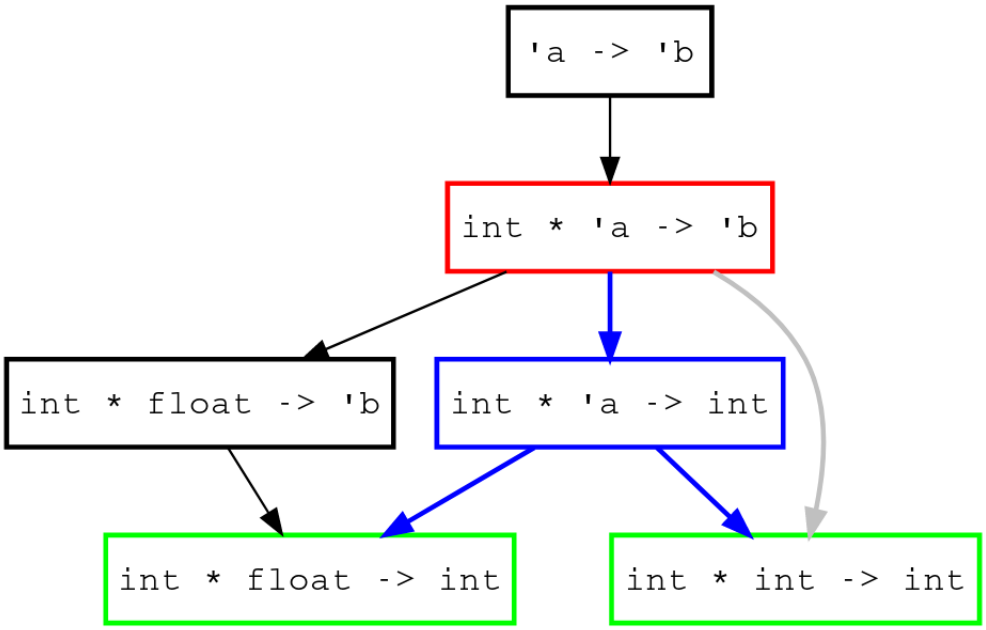

Figure 8: Insertion of type `int * a -> int` in a small Poset

We are now able to define a high-level version of our algorithm.

Algorithm 1 Insertion of τ_0 in the Poset Structure, high-level version

```
1: procedure INSERT_IN_POSET(type τ0, Poset P)
2:     (LUB, ULB) = IDENTIFY_LUB_ULB(P, τ0)
3:     for all τ in LUB do
4:         Add edge (τ, τ0)
5:     for all τ in ULB do
6:         Add edge (τ0, τ)
7:     for all τ, τ' in P∩ (LUB × ULB) do
8:         Remove edge (τ, τ')
```

5.3.2 • Finding LUB and ULB (procedure Identify_LUB_ULB)

Since the majority of types in the Poset are likely to be Uncomparable with τ_0, we identify the LUB and ULB sets by browsing our Poset down from the top and then up from the bottom. This allows us to stop as the frontier of the Bigger zone when we go down, and to stop as the frontier of the Smaller zone when we go up, in order not to visit the whole Poset.

While browsing our Poset down from the tops (i.e., the set of nodes which don't have any parents), we fill two sets:

- the LUB-set, which will correspond at the end of the algorithm to the LUB-zone
- the Removal set, which will correspond to the edges which should be removed at the end.

The procedure is basically a worklist algorithm that keep a queue of nodes to be visited. When a new node is visited, the algorithm performs actions depending on the result of the comparison to τ_0:

- Equal: τ_0 was already present, end of the algorithm
- Bigger: add the type to the LUB-set and remove its parents from it, add the children to the list of nodes to visit, keep browsing
- Uncomparable: do nothing and keep going
- Smaller: add the edge between the previous node and this one to the set of edges to be removed, keep going

We proceed then to the visit of the Poset up from the bottoms (i.e., the set of nodes which don't have any children) and fill the last set, the ULB-set which will correspond to the ULB-zone at the end of the algorithm. This computation is done symmetrically to that of LUB.

5.3.3 • Final Code for Insertion in a Poset

The pseudo-code in Ocaml Style for our insertion algorithm is divided in three main functions : the visit of the Poset down to identify the LUB-zone (Figure 9) and the Remove-set of edges to be discarded, the visit of the Poset up to identify the ULB-zone (Figure 10, and the global adding function which applies the changes (Figure 11).

```
let visit_down poset tau_0 =
 let lub_set = Set.empty in
 let edges_to_remove = Set.empty in
 let to_visit = Queue.create () in
 let rec visit (prev, current) =
   try let comp = compare current tau_0 in
   match comp with
   | Equal -> raise (Type_already_present current)
   | Bigger ->
       let l = G.succ poset current in
       List.iter (fun next -> Queue.push (current, next) to_visit) l;
       (* adding the children to the queue of nodes to visit *)
       Set.remove lub_set prev;
       (* previous node is not in the LUB-zone, remove it *)
       Set.add lub_set current;
       (* current node is in the LUB-zone of the Poset formed by already visited
nodes, add it *)
       visit (Queue.pop to_visit)
   | Uncomparable ->
       visit (Queue.pop to_visit)
   | Smaller ->
       Set.add edges_to_remove (prev, current);
       (* since we are visiting this node, it is a child of a Bigger node, so the
edge between them must be removed *)
       visit (Queue.pop to_visit)
   with Queue.Empty -> lub_set, edges_to_remove
   (* queue is empty, return the sets *)
 in
 (* beginning with the tops of the poset *)
 Set.iter (fun v -> Queue.push (None, v) to_visit) poset.tops;
 visit (Queue.pop to_visit)
```

Figure 9: Pseudo ocaml code for identifying LUB and edges to be removed (browse down)

```
let visit_up poset tau_0 =
  let ulb_set = Set.empty in
  let to_visit = Queue.create () in
  let rec visit (prev, current) =
    try let comp = compare current tau_0 in
    match comp with
    | Equal -> raise (Type_already_present current)
    | Bigger -> visit (Queue.pop to_visit)
    | Uncomparable -> visit (Queue.pop to_visit)
    | Smaller ->
        let l = G.pred poset current in
        List.iter (fun next -> Queue.push (current, next) to_visit) l;
        (* adding the children to the queue of nodes to visit *)
        Set.remove ulb_set prev;
        (* previous node is not in the ULB-zone, remove it *)
        Set.add ulb_set current;
        (* current node is in the ULB-zone of the Poset formed by already visited
nodes, add it *)
        visit (Queue.pop to_visit)
    with Queue.Empty -> ulb_set
    (* queue is empty, return the set *)
  in
  (* beginning with the bottoms of the poset *)
  Set.iter (fun v -> Queue.push (None, v) to_visit) poset.bottoms;
  visit (Queue.pop to_visit)
```

Figure 10: Pseudo ocaml code for identifying ULB (browse up)

```
(** adds tau_0 in current poset G **)
let add poset tau_0 =
    let lub_zone, edges_to_remove = visit_down poset tau_0 in
    let ulb_zone = visit_up poset tau_0 in
    Set.iter (fun v -> G.add_edge poset (v, tau_0)) lub_zone;
    Set.iter (fun v -> G.add_edge poset (tau_0, v)) ulb_zone;
    Set.iter (fun e -> G.remove_edge poset e) edges_to_remove;;
```

Figure 11: Pseudo ocaml code for propagating changes in the poset

5.3.4 • Correctness of the algorithm

The proof of our algorithm relies on two parts: the correctness of the LUB and ULB zones delimited after browsing the Poset, and the correctness of the insertion if those zones have been correctly identified.

In those proofs, we will use the following definitions:

- A directed edge from a type to another in a poset $\mathcal{P}$ will be noted $\dashrightarrow$.
- The binary relation infered from a poset $\mathcal{P}$ is the relation of descendance, meaning that to proove the correctness of our algorithm, the relation infered from our final Poset must be exactly the matching relation:

$$\tau' \preccurlyeq \tau \Leftrightarrow (\exists(\tau_1, \dots, \tau_n), \tau \dashrightarrow \tau_1 \dashrightarrow \cdots \dashrightarrow \tau_n \dashrightarrow \tau')$$

- A graph is the transitive reduction of the relation it represents, if for any non trivial path between from a type τ to another type τ', there is no edge linking directly τ to τ' :

$$(\exists(\tau_1, \dots, \tau_n), n \in \mathbb{N}^*, \tau \dashrightarrow \tau_1 \dashrightarrow \cdots \dashrightarrow \tau_n \dashrightarrow \tau') \Rightarrow \neg(\tau \dashrightarrow \tau')$$

Finding LUB and ULB zones :
Let us give a few clues towards the proof that our insertion algorithm finds the correct LUB and ULB zones.

Definition 5.3.1 ($(\mathcal{P}, \tau_0) - LUB$ zone)
Let τ_0 be the type to insert in the poset $\mathcal{P}$. The LUB zone (Lower Upper Bound zone) corresponding to this insertion is the set of all types τ such that:

- $\tau_0 \preccurlyeq \tau$ (τ is Bigger than τ_0)
- $\forall \tau', \tau \dashrightarrow \tau' \Rightarrow \tau_0 \not\preccurlyeq \tau'$ (children are not Bigger than τ_0)

Definition 5.3.2 ($(\mathcal{P}, \tau_0) - ULB$ zone)
Let τ_0 be the type to insert in the poset $\mathcal{P}$. The ULB zone (Upper Lower Bound zone) corresponding to this insertion is the set of all types τ such that:

- $\tau \preccurlyeq \tau_0$ (τ is Smaller than τ_0)
- $\forall \tau', \tau' \dashrightarrow \tau \Rightarrow \tau' \not\preccurlyeq \tau_0$ (parents are not Smaller than τ_0)

During the visit of the Poset for insertion of a type τ_0, the following invariants are verified:

- the ULB-set corresponds to the ULB-zone in the Poset formed by the already visited nodes
- the LUB-set corresponds to the LUB-zone in the Poset formed by the already visited nodes

Correctness of the Poset after insertion :
We will also give a few indications for the proof of the correctness of the insertion, once admitted the correctness of the LUB and ULB zones.

The following invariant holds during the creation of the Poset: After an insertion, the graph is the transitive reduction of the matching relation on all types present in the Poset.

Theorem 5.3.3 (All relations are correct in the Poset)

$$\forall(\tau, \tau') \in \mathcal{P}^2, (\exists(\tau_1, \dots, \tau_n), \tau \dashrightarrow \tau_1 \dashrightarrow \cdots \dashrightarrow \tau_n \dashrightarrow \tau') \Rightarrow \tau' \preccurlyeq \tau$$

Proof.
After inserting a type τ_0 in the Poset, all matching relations indicated by the Poset are either:

- a relation given by an edge already present in the Poset: by correctness of the Poset before insertion, it is still true
- a relation given by an edge added in the insertion process: it is either an edge between a type of LUB and τ_0, and by definition of LUB as a subset of the types Bigger than τ_0, it is correct, or it is between τ_0 and a type of ULB, and it is correct with the same reasoning on Smaller types.
- a relation given by transitivity of the edges in the graph, which is correct because all edges have just been proven to be correct, and the matching relation is transitive

□

Theorem 5.3.4 (All correct relations are in the Poset)

$$\forall(\tau,\tau') \in \mathcal{P}^2, \tau' \preccurlyeq \tau \Rightarrow (\exists(\tau_1,\dots,\tau_n), \tau \dashrightarrow \tau_1 \dashrightarrow \cdots \dashrightarrow \tau_n \dashrightarrow \tau')$$

Proof.
All relations between types of the Poset after an insertion of type τ_0 are of the following form:

- a relation between types already present in the Poset before insertion: if it has not been deleted then we are done, and if it has been deleted then it was a relation between someone from LUB and ULB. By construction of our algorithm, there exists now an edge between the type from LUB and τ_0 and one between τ_0 and the types from ULB, so by transitivity the relation between the two types is present
- a relation between types previously in the Poset and τ_0: by definition of LUB, if a type is Bigger than τ_0, either it belongs to LUB and is therefore linked to τ_0, or it has a descendant in LUB, which allows to deduce its matching relation with τ_0. Same goes for Smaller types and ULB.

□

Theorem 5.3.5 (The relations are minimal in the Poset)

$$\forall(\tau,\tau') \in \mathcal{P}^2, (\exists(\tau_1,\dots,\tau_n), n \in \mathbb{N}^*, \tau \dashrightarrow \tau_1 \dashrightarrow \cdots \dashrightarrow \tau_n \dashrightarrow \tau') \Rightarrow \neg(\tau \dashrightarrow \tau')$$

Proof.
By our invariant, we assume the Poset verified the property before the insertion (i.e., was the transitive reduction of the matching relation on types present before insertion).

Let (τ,τ') be two types linked by the following path $\tau \dashrightarrow \tau_1 \dashrightarrow \cdots \dashrightarrow \tau_n \dashrightarrow \tau', n \in \mathbb{N}^*$.
Let us reason on the following cases :

- if none of the types is equal to τ_0, since we only added edges containing τ_0 in the insertion, the path existed before, and by the property on the Poset before insertion, we had no edge between τ and τ'. Since they are both different from τ_0, the edge hasn't been added in the insertion, so we have $\neg(\tau \dashrightarrow \tau')$
- else if $\tau = \tau_0$, then we have $\tau_0 \dashrightarrow \tau_1$, which means that τ_1 belonged to the ULB-zone, and thus none of its descendants did. Since edges from τ_0 were only added towards types of the ULB-zone, in particular, we have $\neg(\tau \dashrightarrow \tau')$
- else if $\tau' = \tau_0$, then with the symmetrical reasoning on the LUB zone, we also have $\neg(\tau \dashrightarrow \tau')$
- else if $n = 1$, we thus have $\tau \dashrightarrow \tau_0 \dashrightarrow \tau'$, meaning that τ was part of the LUB zone and τ' of the ULB zone. If there was an edge between them, it was removed during the insertion, so we have $\neg(\tau \dashrightarrow \tau')$
- else if $n > 1$, there exists $i \in [\![1\,;n]\!]$ such that $\tau_i = \tau_0$. Since $n > 1$, there exists also $i' \in [\![1\,;n]\!]$ such that $\tau' \preccurlyeq \tau_i \preccurlyeq \tau$, so by correction of the algorithm prooved in Theorem 5.3.4, there was a path $\tau \dashrightarrow \hat{\tau_1} \dashrightarrow \cdots \dashrightarrow \hat{\tau_n} \dashrightarrow \tau', n \in \mathbb{N}^*$ before insertion, and by our invariant, there was no edge between τ and τ', yet they are both different from τ_0 so no edge was added between them, so we have $\neg(\tau \dashrightarrow \tau')$

□

The conjonction of Theorems 5.3.3, 5.3.4, 5.3.5 prooves that our invariant is correct, and thus after insertion of all types, our Poset represents exactly the matching relation on our library.

5.4 Querying the Poset

5.4.1 • A new flow in the search algorithm

In order to use the information contained in the Poset efficiently while keeping the efficient Trie filtering, the general flow of our search algorithm must change. Previously, the search algorithm computed the list of all functions compatible with our query type according to the flow in Fig. 12.

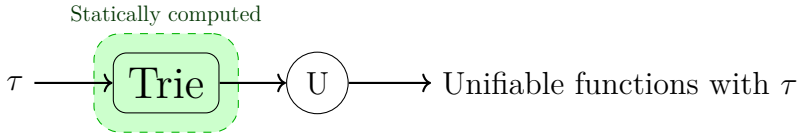

Figure 12: Search algorithm without Poset

The Trie, computed statically, contains the whole library sorted according to the Features. A query type τ is passed and an algorithm on the Trie eliminates uncompatible types to pass a list of possibles types, which is treated linearly with the unification algorithm to return all functions with an unifiable type for τ.

With the addition of the Poset to the Index, the flow of the algorithm is changed as shown in Figure 13.

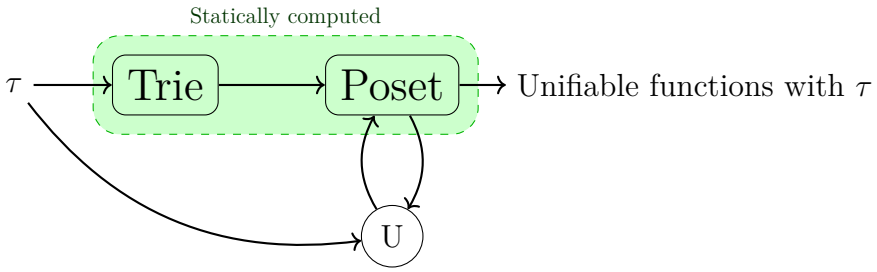

Figure 13: Search algorithm with Poset

The Trie and the Poset are still computed statically and contain the whole library information. Once a query type τ is given, an algorithm on the Trie eliminates incompatibles types and provides an other algorithm on Poset with a list of possible types. With that list and some calls to the unification algorithm, the algorithm on Poset selects the types and returns all functions with an type which unifies with τ.

5.4.2 • Link between Trie and Poset

As said in the previous section, before the implementation of the Poset, the type search strategy was to try unifying on all types compatible with our query type according to the Features. In our new algorithm, the idea is to iterate on the Poset and not on the compatible elements of the Trie. However, we have to keep the resulting of filtering using the Features. More precisely, we need to represent the set of all types selected by the Trie in a way that is compact and fast to query.

For this purpose, we attribute a unique integer identifier to each type when building the Trie: these identifiers are generated in a linear way, following the Depth-First search of the Trie, as shown in Fig. 14.

With this numbering scheme, sets of types are thus simply unions of interval of integers. We use discrete interval encoding trees (`DIET` [6]) to represent and manipulate such sets. When querying the Trie, we directly return the types as such `DIET` of integers, as shown in Fig. 15.

5.4.3 • The search algorithm

In our new algorithm flow (Fig. 13), the type is first used to query the Trie, which returns a compact representation of the set of candidate types as a DIET. Then, a second algorithm run on the Poset and returns the list of functions whose types unify with the query.

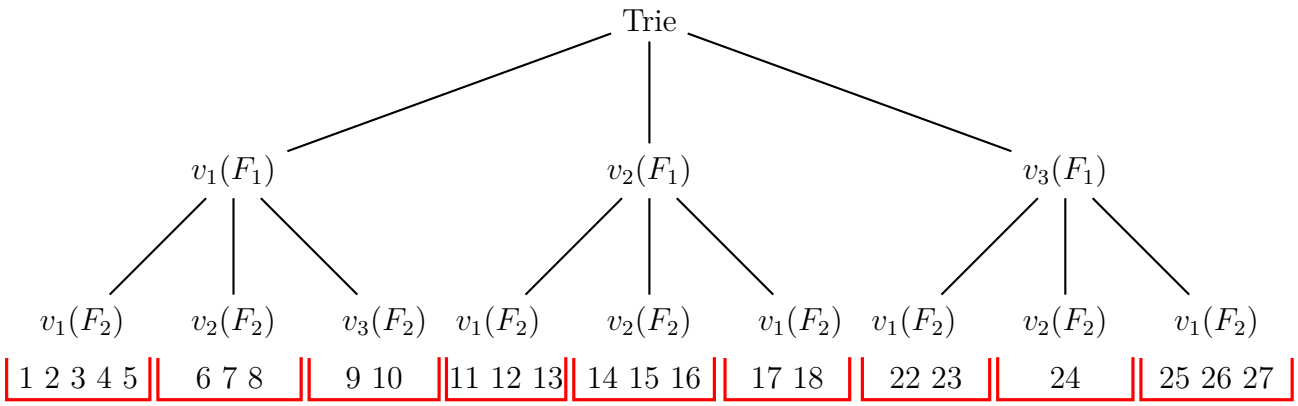

Figure 14: Allocation of identifiers to types in the Trie

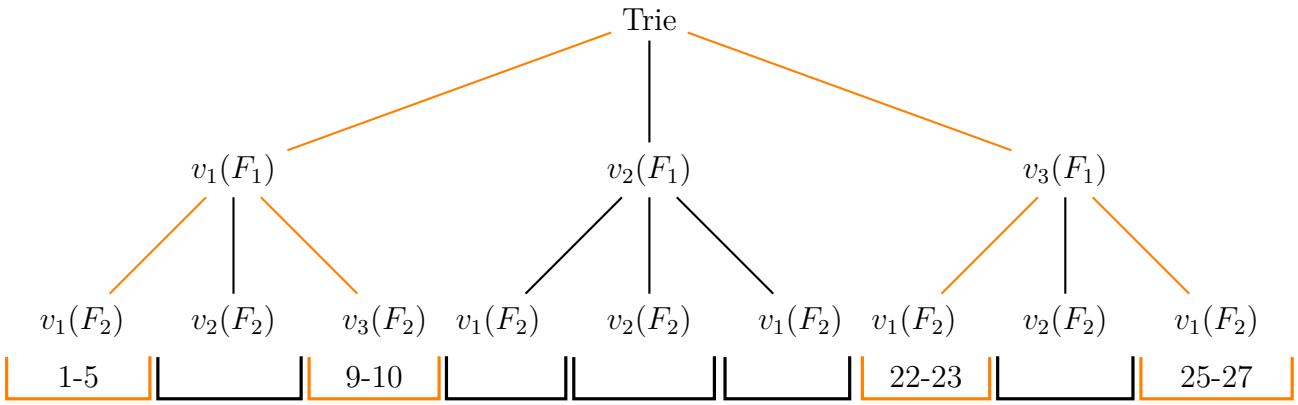

Figure 15: Range of types with certain feature values

Search algorithm with Poset

Let τ_0 be our query type.

We begin with an empty set of unifiable types, noted $\mathcal{S}_{\cong}$. The Poset is browsed in a Breadth-First Search fashion, and visited types τ are treated as follows:

- if τ is in $\mathcal{S}_{\cong}$ (i.e., we already saw that it is a unifiable type), keep visiting
- else if τ is not in the DIET (i.e., we know that it is not unifiable according to the Features, or it has already been handled by our algorithm), remove all its descendants in the Poset from the DIET (by Theorem 5.1.3, we know that those are not unifiable either), keep visiting
- else, call the unification algorithm on τ and τ_0. If they are unifiable, add τ to $\mathcal{S}_{\cong}$ and keep visiting. Else, remove it from DIET along with its descendants (Theorem 5.1.3 again). Keep visiting.

Pseudo-code

```
let check poset ~query:tau_0 ~diet =
  let unifs = ref Set.empty in
  let diet = ref diet in
  let to_visit = Queue.create () in
  let rec visit_down tau =
    try
    if Set.mem tau !unifs then visit (Queue.pop to_visit)
    else if not (Diet.mem tau !diet) then (
      iter_succ poset tau (fun ty -> diet:= Diet.remove diet ty.id);
      visit_down (Queue.pop to_visit))
      (* iterate on all descendants of tau, remove them from diet *)
    else
      match Unification.unify tau_0 tau with
      | Yes ->
          unifs:= Set.add tau !unifs;
          let l = G.succ poset tau in
          List.iter (fun next -> Queue.push next to_visit) l;
          (* add children to the list of nodes to visit *)
          visit_down (Queue.pop to_visit)
      | No ->
          iter_succ poset tau (fun ty -> diet:= Diet.remove diet ty.id);
          (* iterate on all descendants of tau, remove them from diet *)
          visit_down (Queue.pop to_visit)
    with Queue.Empty -> unifs
  in
  Set.iter (fun v -> Queue.push v to_visit) poset.tops;
  (* begin with the tops of the poset *)
  visit_down (Queue.pop to_visit);;
```

Figure 16: Pseudo code for a query

5.5 Optimisation: matching computation

While the matching relation is computed statically (when pre-processing the libraries), it is still important to reduce the computation time as much as possible.

In this section, we take interest in a way to optimize our algorithm with respect to the matching.

5.5.1 • Analysis: how is matching computed ?

In `dowsindex`, the matching relation between a type τ and a type τ' is computed with the following steps:

1. We freeze the variables from τ (i.e., replacing them by uninstantiable variables) and try

to unify $Freeze(\tau)$ with τ'

2. We freeze the variables from τ' (i.e., replacing them by uninstantiable variables) and try to unify $Freeze(\tau')$ with τ

3. If only the first unification is positive, then we have $\tau \preccurlyeq \tau'$, if only the second, we have $\tau' \preccurlyeq \tau$, if both are positive then τ and τ' are equal modulo ACIC-equivalence, and if none is positive then they are uncomparable.

Thus, computing the matching relation between two types costs two iterations of the unification algorithm, which motivates a new optimization specific to the matching.

5.5.2 • Proposition: using Features to optimise the matching (Const)

The notion of Feature was introduced to cut unifications in the search algorithm, however is also applies to matching as a double-unification process. This is why we used the Features for the computation of the matching relation while building the Poset as well.

But we also found a specific way of avoiding unification computations in the matching: a new Feature specifically for the matching, called Const.

The Const Matching Feature is based on the observation of constructors in both types. Indeed, when trying to prove that a type τ matches with a type τ', since none of the variables from τ can be instantiated, and all constructors are left unchanged by ACIC-equivalence, we have the intuition that all constructors present in τ' must already be present in τ.

As done for unification, a Matching Feature will be the combination of an encoding function and a criterion.

Definition 5.5.1 (Encoding function: Const)
The encoding function for the Feature associates, to each type τ, the multiset of the constructors of τ. The multiset is defined as a set where each element can be present more than once, and is thus stored along with its multiplicity.

Example 5.5.2
Here is the encoding of a few types, where multisets are noted as a set with elements indexed by their multiplicity.

- Const(`int -> int -> int`) = { `int`^{3}}
- Const(`'a -> 'a list -> bool`) = { `list`^{1}, `bool`^{1}}
- Const(`(float * int * float list) -> float list`) = { `int`^{1}, `list`^{2}, `float`^{3}}

Theorem 5.5.3 (Matching criterion: Const)
The matching criterion for Const is the following theorem:

$$\forall(\tau, \tau') \in \mathcal{T}^2, \tau \preccurlyeq \tau' \Rightarrow \text{Const}(\tau') \subseteq \text{Const}(\tau)$$

As for unification features, we will use the criterion with its contraposition.

Theorem 5.5.4 (Matching criterion: contraposition of Const)

$$\forall(\tau,\tau') \in \mathcal{T}^2, \text{Const}(\tau') \nsubseteq \text{Const}(\tau) \Rightarrow \tau \not\leq \tau'$$

Example 5.5.5
The following types do not match because their values for Const are not compatible:

- `(float * int * float list) -> float list` $\not\leq$ `int -> int -> int`
  because { $\texttt{int}^3$} $\nsubseteq$ { $\texttt{int}^2$, $\texttt{list}^1$, $\texttt{bool}^1$}
- `'a -> 'a list -> bool` $\not\leq$ `(float * int * float list) -> float list`
  because { $\texttt{int}^1$, $\texttt{list}^2$, $\texttt{float}^3$} $\nsubseteq$ { $\texttt{list}^1$, $\texttt{bool}^1$}

This new Feature has been integrated to prevent the computation of some unification: before doing a matching comparison when building the Poset, we first compute the values of the two types for the Const encoding, and see if there is any inclusion of the multisets. Any non-present inclusion avoids doing a unification.

Example 5.5.6
If we want to indentify the matching relation between types `int -> int -> int` and `'a -> int`, we have the following:

- Since { $\texttt{int}^3$} $\nsubseteq$ { $\texttt{int}^1$}, we deduce `'a -> int` $\not\leq$ `int -> int -> int`
- Since { $\texttt{int}^1$} $\subseteq$ { $\texttt{int}^3$}, we will have to compute the unification between $Freeze($ `int -> int -> int` $)$ and `'a -> int`

6 STATE OF THE ART

In this section, we give an overview of related work to our tool `dowsindex`.

6.1 Function search

When it comes to function search, there exist interfaces in almost every language to allow for textual search in the different library. Namely, a textual query is matched to functions by their name or part of their documentation. Some of those tools include efficient handling of software databases [7]. In OCaml, the `Merlin`[6] IDE offers a search based on polarity in the local typing environment.

`Hoggle`[7] is a research tool for Haskell which proposes textual search and *search by types*. However, the search remains very syntaxic (type signatures are parsed textually and compared by approximative shape) and therefore cannot be proven to be sound nor complete.

Our approach with `dowsindex` has been motivated by the algorithm proposed by M. Rittri [10] of a search by types in the ML language, using type isomorphisms. No such tool existed for OCaml to our knowledge.

6.2 Theoretical results on type isomorphisms

Research on the complexity of type isomorphism problems has been particularly active for several decades. This research includes results on the decidability and complexity of the equivalence, the matching and the unification for many theories such as first-order [8] and second-order λ-calculus [5].

Many results are still to be proven in this field.

6.3 Algorithms related to `dowsindex`

Regarding unification algorithms, the algorithm used in `dowsindex` is inspired from the one proposed by A. Boudet [4] for AC-unification.

The structure of our Index and in particular of the Trie is inspired by the work of S. Schulz [11] on the indexing of clause subsumption.

[6] `https://github.com/ocaml/merlin`
[7] `https://hoogle.haskell.org/`

7 FUTURE WORK

7.1 Benchmark of dowsindex

In the remaining few weeks of this internship, our main objective is to establish the gain in performance of our work. Namely, we will measure the gain in time as well as in the number of unification for a search in a library with our new algorithm including the Poset.

We will also measure the construction time of the Poset and the optimization with the addition of the Const Feature for matching.

7.2 New features

A direction for future work can be to find new features, either for unification or for matching specifically, by following the same procedure as Allain [1].

7.3 Sorting the results

An interesting work could also be done on the sorting of the results at the end of the algorithm. Indeed, we could implement the idea of Rittri [10] to sort the functions by size of the unifiers between the types of the functions and our query, as suggested in Section 5.1.1.

8 CONCLUSION

During this internship, we worked on improving the `dowsindex` tool through the completion of the previously existing Index. We created a new structure, the Poset, which gathers the information of the matching relation between all the types of a library.

This allowed us to use a theorem concerning matching and unification in order to limit the use of the very expensive unification algorithm in the search. Moreover, we have developed a new type of Feature specific to matching in order to lighten the computation of the latter too.

In the next few weeks, we will be able to measure the improvement in performance of our contribution to `dowsindex`. There will remain, afterwards, many possible tracks of optimization, and the need for an interface in order to, we hope, democratize the use of `dowsindex` for all the programmers in OCaml.

REFERENCES

[1] C. Allain. Recherche de fonctions par unification modulo isomorphismes de types (internship report). 2021.

[2] C. Allain, G. Radanne, and L. Gonnord. Isomorphisms are back! In *ML 2021 - ML Workshop*, pages 1–3, Virtual, France, Aug. 2021. URL https://hal.archives-ouvertes.fr/hal-03355381.

[3] A. Boudet. Competing for the ac-unification race. *J. Autom. Reason.*, 11(2):185–212, 1993. doi: 10.1007/BF00881905. URL https://doi.org/10.1007/BF00881905.

[4] A. Boudet. Competing for the ac-unification race. *Journal of Automated Reasoning*, 11: 185–212, 2004. doi: 10.1007/BF00881905. URL https://doi.org/10.1007/BF00881905.

[5] R. D. Cosmo. *Isomorphisms of Types: From Lambda-Calculus to Information Retrieval and Language Design*. Birkhauser Verlag, CHE, 1995. ISBN 376433763X.

[6] M. Erwig. Diets for fat sets. *Journal of Functional Programming*, 8(6):627–632, 1998. doi: 10.1017/S0956796898003116.

[7] W. B. Frakes and B. A. Nejmeh. Software reuse through information retrieval. *SIGIR Forum*, 21(1–2):30–36, sep 1986. ISSN 0163-5840. doi: 10.1145/24634.24636. URL https://doi.org/10.1145/24634.24636.

[8] P. Narendran and D. Kapur. Complexity of unification problems with associative-commutative operators. *Journal of Automated Reasoning*, 9:261–288, 1992. ISSN 1573-0670. URL https://doi.org/10.1007/BF00245463.

[9] P. Narendran, F. Pfenning, and R. Statman. On the unification problem for cartesian closed categories. *The Journal of Symbolic Logic*, 62(2):636–647, 1997. ISSN 00224812. URL http://www.jstor.org/stable/2275552.

[10] M. Rittri. Retrieving library functions by unifying types modulo linear isomorphism. *RAIRO-Theor. Inf. Appl.*, 27(6):523–540, 1993. doi: 10.1051/ita/1993270605231. URL https://doi.org/10.1051/ita/1993270605231.

[11] S. Schulz. *Simple and Efficient Clause Subsumption with Feature Vector Indexing*, pages 45–67. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-36675-8. doi: 10.1007/978-3-642-36675-8_3. URL https://doi.org/10.1007/978-3-642-36675-8_3.

[12] S. V. Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22:1387–1400, 1983. ISSN 1573-8795. URL https://doi.org/10.1007/BF01084396.