Isomorphisms are back!

Clément Allain, Gabriel Radanne, Laure Gonnord

Isomorphisms are back!

Smart indexing for function retrieval by unification modulo type isomorphisms

Clément Allain
EnsL, UCBL, CNRS, LIP
clement.allain@inria.fr

Gabriel Radanne
Inria, EnsL, UCBL, CNRS, LIP
gabriel.radanne@inria.fr

Laure Gonnord
Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP
laure.gonnord@ens-lyon.fr

1 Introduction

Sometimes, we need a function so deeply that we have to go out and search for it. How do we find it? Sometimes, we have clues: a function which manipulates `list` is probably in the `List` module ... but not always! Sometimes, all we have is its functionality: doing the sum of a list of integers. Unfortunately, search by functionality is difficult.

Rittri [1] proposed an approximation: use the *type* of the function as a key to search through libraries: in our case, **`int list`** `->` **`int`**. To avoid stumbling over details such as the order of arguments, he proposed to use matching *modulo type isomorphism* – a notion broader than syntactic equality.

Unfortunately, algorithms for unification modulo type isomorphisms are extremely costly. Doing an exhaustive search over the whole ML ecosystem was possible at the time (with a standard library of 294 functions), but is certainly not possible anymore for the OCaml ecosystem, with 3259 opam packages, each containing several hundreds or thousands of functions.

We present *Dowsing*, a tool to search functions in OCaml libraries by using their types as key. Here is an example of the tool in action for the example above:

```
> search db "int list -> int"
int list -> int:
  BatList.sum
  Thread.wait_signal
'a list -> int:
  List.length
'a -> int:
  Hashtbl.hash
'a list -> 'a:
  List.hd
  BatList.max
  BatList.min
  BatList.last
  BatList.first
```

Dowsing is a work in progress, but is already capable of executing queries over a full opam switch containing big libraries such as Core or Batteries, in a few milliseconds. Such a feat is achieved through novel indexing techniques that allow to index types in way that is compatible with unification modulo type isomorphisms.

In this article, we give a quick introduction to our tool and hint at some details of our indexing techniques. The talk will present both practical and formal aspects of this work in greater details, along with future plans.

2 Unification modulo type isomorphism

Type isomorphisms and their use in function search have been widely studied in the 90's [1, 2, 3]. The first requirement is that the order of function parameters do not matter. This notion of *equivalence modulo isomorphism* is much more flexible than syntactic equality.

Function retrieval systems using type isomorphisms have been implemented in Lazy ML [1] and Coq [4]. Following [1], we consider linear isomorphisms expressing associativity and commutativity of `*`, neutrality of `unit` and curryfication:

$$
\begin{aligned}
\texttt{('a * 'b) * 'c} &\sim \texttt{'a * ('b * 'c)} \\
\texttt{'a * 'b} &\sim \texttt{'b * 'a} \\
\texttt{unit * 'a} &\sim \texttt{'a} \\
\texttt{('a * 'b) -> 'c} &\sim \texttt{'a -> ('b -> 'c)}
\end{aligned}
$$

An additional requirement is the ability to retrieve more general types that admit an instance equivalent to the query : in our example `'a list -> 'a` is such an instance.

Such an optimized unification algorithm has already been proposed [6]. Still, we can easily trigger its exponential behavior on highly polymorphic functions from Batteries and Core.

3 Smart indexing and smart queries

While the fundamental complexity of unification seems like a difficult obstacle to overcome, our aim is not to unify quickly, but to *search* quickly. We can thus benefit from indexing and pre-computing databases. In particular, there has been important progress regarding term indexing [7, 5] that allow to speed up such search procedures in numerous context and varied theories.

We propose a two steps approach:

- We once pre-process the ecosystem into a database indexed by a set of *features*. The function identifiers are stored in a *trie* according to their *feature vectors* [7] that encode structural information on their types.
- The query phase itself uses these *feature vectors* to reduce the number of actual unification calls.

Features are designed such that if the feature coming from the request is not compatible with a candidate feature in the database, then we know for sure that they cannot be unified. The challenge is then to design features that are both sufficiently discriminating and compatible with our notion of unification. The development of new features can be informed by metrics regarding the actual time spent in the search procedure. If a lot of time is spent doing unification over highly polymorphic functions, features must discriminate those the most.

We have so far identified three such features. One example is the feature *by head*, which classify the head of a function into categories. For instance, the head of `'a list -> 'a` is "variable", while the head of `'a array -> 'a array` is the "constructor" `array`. based on this classification, we can quickly decide if two types might unify.

4 Conclusion

We have presented a function retrieval system for OCaml. By combining unification modulo isomorphism and smart indexing techniques, it can efficiently manage a large database. In particular, features allow to overcome in practice the efficiency limitations imposed by rich unification, as informed by metrics collected on concrete queries.

References

[1] Rittri, M. "Retrieving library functions by unifying types modulo linear isomorphism". In: *RAIRO-Theor. Inf. Appl.* 27.6 (1993), pp. 523–540. DOI: 10.1051/ita/1993270605231. URL: https://doi.org/10.1051/ita/1993270605231.

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

[3] Paliath Narendran, Frank Pfenning, and Richard Statman. "On the Unification Problem for Cartesian Closed Categories". In: *J. Symb. Log.* 62.2 (1997), pp. 636–647. DOI: 10.2307/2275552. URL: https://doi.org/10.2307/2275552.

[4] David Delahaye. "Information Retrieval in a Coq Proof Library Using Type Isomorphisms". In: *Types for Proofs and Programs*. Ed. by Thierry Coquand et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 131–147. ISBN: 978-3-540-44557-9.

[5] I. V. Ramakrishnan, R. C. Sekar, and Andrei Voronkov. "Term Indexing". In: *Handbook of Automated Reasoning (in 2 volumes)*. Ed. by John Alan Robinson and Andrei Voronkov. Elsevier and MIT Press, 2001, pp. 1853–1964. DOI: 10.1016/b978-044450813-3/50028-x. URL: https://doi.org/10.1016/b978-044450813-3/50028-x.

[6] A. Boudet. "Competing for the AC-Unification Race". In: *Journal of Automated Reasoning* 11 (2004), pp. 185–212.

[7] Stephan Schulz. "Simple and Efficient Clause Subsumption with Feature Vector Indexing". In: *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*. Ed. by Maria Paola Bonacina and Mark E. Stickel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 45–67. 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.