Back to the 90s!

Fast Indexing for Search by Types

Clément Allain Gabriel Radanne Laure Gonnord

Problem

Every programmer has encountered this problem once:

I'm looking for a function that does X, where to find it?

Often there is an "intuitive" approach: I want a function on time, I look in the `Time` module. This does not always work (auxiliary modules . . .).

⇒ We can search functions using a very familiar abstraction: their types!

Our tool: Dowsing!

- Finds types “up to” order of arguments, instantiation, ...
- Knows about packages/libraries
- Scales to modern ecosystems (for instance, opam)

```
$ search "'a list * 'a -> bool"
...
List.mem : 'a -> 'a list -> bool
...

$ search "'a list -> ('a * 'b -> 'b) -> 'b -> 'b"
...
List.fold_left :
  ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
List.fold_right :
  ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
...
```

How does it all work ?

Our tool: Dowsing!

- Finds types “up to” order of arguments, instantiation, . . .
- Knows about packages/libraries
- Scales to modern ecosystems (for instance, opam)

```
$ search "'a list * 'a -> bool"
...
List.mem : 'a -> 'a list -> bool
...

$ search "'a list -> ('a * 'b -> 'b) -> 'b -> 'b"
...
List.fold_left :
  ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
List.fold_right :
  ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
...
```

How does it all work ?

Existing approaches

Search by type modulo isomorphism:

- Sound and complete ✔
- Computationally intensive ✘

Hoogle (`https://hoogle.haskell.org/`):

- Neither sound nor complete (more like “text search in types”) ✘
- Scales well ✔

Search by type modulo isomorphism

Given a type τ, finds all functions in the libraries with types which are equivalent/match/unify to τ up to some set of “simplifications”.

We consider the following simplifications (i.e., isomorphisms):

$$a \times b \equiv_T b \times a \tag{$\times$-comm}$$

$$a \times (b \times c) \equiv_T (a \times b) \times c \tag{$\times$-assoc}$$

$$\texttt{unit} \times a \equiv_T a \tag{$\times$-$\texttt{unit}$}$$

$$(a \times b) \rightarrow c \equiv_T a \rightarrow (b \rightarrow c) \tag{curry}$$

Problem: Unification/Matching modulo isomorphism is *expensive*.

Smart indexing

Some remarks:

- When searching, many types do not match
- Even when failing, unification is expensive
- Performance of unification highly depends on the types (more than $*1000$ variance)

Battle plan:

1. Experimentally measure to identify types taking lots of time
2. Introduces “shortcuts”, to skip unification for these expensive types
3. Pre-process the database of types to compute shortcuts in advance
4. Rinse and repeat

Smart indexing

Some remarks:

- When searching, many types do not match
- Even when failing, unification is expensive
- Performance of unification highly depends on the types (more than $*1000$ variance)

Battle plan:

1. Experimentally measure to identify types taking lots of time
2. Introduces "shortcuts", to skip unification for these expensive types
3. Pre-process the database of types to compute shortcuts in advance
4. Rinse and repeat

First metric: Number of unique variables

Exemples:

- $\mathit{vars}(\alpha \to \alpha) = 1$
- $\mathit{vars}((\alpha \to \beta \to \alpha) \to \alpha \to \mathit{list}(\beta) \to \alpha) = 2$

```
stats "int -> int -> int" -measure unique-vars
```

vars	total time (ms)	avg. time (μs)	nb. unif.
0	53.8554	1.28044	42060
1	76.8702	8.60038	8938
2	68.7156	18.6575	3683
3	10.241	9.16014	1118
4	3.55721	12.3514	288

Observations:

- 75% unifs/ 25% time without any variables
- Unification on polymorphic type is harder

Second of metric: The head

Exemples:

- $\mathit{head}(\mathit{unit} \to \alpha) = \mathsf{var}$
- $\mathit{head}(\mathit{int} \to \mathit{int} \to \mathit{list}(\alpha)) = \mathsf{cons}_{\mathit{list}}$

```
stats "int -> int -> int" -measure head
```

vars	total time (ms)	avg. time (μs)	nb. unif.
variable	88.8178	39.7218	2236
constructor	80.8148	1.77701	45478
tuple	16.0539	1.91574	8380
other	0.442982	1.91767	231

Observations:

- 95% with a simple constructor at the head
- Case with variable head are pathological

Metrics

We have many other metrics. It's very easy to implement new ones.

Preliminary conclusions:

- Cheap cases (no variables, simple head, . . .)
  ⇒ Still many of them, Easy to skip
- Expansive cases (Too many variables, lot's of sharing)
  ⇒ Hard to skip, but skips are very worthwhile

Unification criterion

We are looking for criterions that are necessary (but not sufficient!).

A criterion is composed of:

- A domain of values D
- $encode : Types \to D$
- $compat : D \times D \to Bool$
- $\tau_1 \equiv_T \tau_2 \implies compat(encode(\tau_1), encode(\tau_2))$

We thus get a filter!

$\neg(compat(encode(\tau_1), encode(\tau_2))) \implies \tau_1 \not\equiv_T \tau_2$

Unification criterion

We are looking for criterions that are necessary (but not sufficient!).

A criterion is composed of:

- A domain of values D
- $encode : Types \to D$
- $compat : D \times D \to Bool$
- $\tau_1 \equiv_T \tau_2 \implies compat(encode(\tau_1), encode(\tau_2))$

We thus get a filter!
$\neg(compat(encode(\tau_1), encode(\tau_2))) \implies \tau_1 \not\equiv_T \tau_2$

Unification criterion – Head matching

If two types have incompatible heads, they can never unify:

- $\dots \to \mathit{int} \not\equiv_T \dots \to \mathit{float}$
- $\dots \to \mathit{list}(\alpha) \not\equiv_T \dots \to \mathit{int} \times \mathit{int}$
- $\dots \to \mathit{int} \overset{?}{\equiv}_T \dots \to \alpha$

We precompute the heads for all types in the database and store them compactly

Unification criterion – Head matching – benchmarking

Searching in a local install of opam:
$\sim$ 250 packages, 31578 functions

Type	Nb unif. w shortcut	Gain
$int \to int \to int$	2714	91.4%
$int \to int \to int \to int$	2714	91.4%
$int \to (int \to int) \to list(\alpha)$	2945	90.7%
$int \to float \to bool \to \text{unit}$	5745	81.8%
$\alpha \to int \to \text{unit}$	5745	81.8%
$int \to int \to \alpha$	31578	0%

- We correctly avoid many unifications
- We need to work more for polymorphic queries

Unification criterion – Multiplicity

If multiplicities are incompatible, types can never unify:

$$\begin{aligned} & \mathit{int} \to \mathit{int} \to \mathit{int} \not\equiv_T & \qquad & \mathit{multiplicity} = 2 \\ & \mathit{int} \to \mathit{float} & & \mathit{multiplicity} = 1 \end{aligned}$$

$$\begin{aligned} & \mathit{int} \to \mathit{int} \not\equiv_T & \qquad & \mathit{multiplicity} = 1 \\ & \mathit{int} \to \mathit{int} \to \alpha \to \alpha & & \mathit{multiplicity} \geq 2 \end{aligned}$$

$$\begin{aligned} & \mathit{int} \to (\mathit{unit} \to \mathit{unit}) \to \mathit{unit} \not\equiv_T & \qquad & \mathit{multiplicity} = 2 \\ & \mathit{int} \to (\mathit{int} \to \mathit{float}) \to \mathit{int} \to \alpha & & \mathit{multiplicity} \geq 3 \end{aligned}$$

$$\begin{aligned} & \mathit{int} \to \mathit{list}(\beta) \to \mathit{int} \overset{?}{\equiv}_T & \qquad & \mathit{multiplicity} = 2 \\ & \mathit{int} \to \alpha & & \mathit{multiplicity} \geq 1 \end{aligned}$$

Unification criterion – Multiplicity – benchmarking

Searching in a local install of opam:
$\sim$ 250 packages, 31578 functions

Type	Nb unif. w shortcut	Gain
$int \to int \to int$	121	99.62%
$int \to int \to int \to int$	107	99.66%
$int \to (int \to int) \to list(\alpha)$	141	99.55%
$int \to float \to bool \to \text{unit}$	126	99.6%
$\alpha \to int \to \text{unit}$	2443	92.26%
$int \to int \to \alpha$	3677	88.36%

- Combined with the previous criterion!
- Now decent at polymorphic queries

More shortcuts and combinations

We introduced more shortcuts and plan to investigate even more (for instance, the relative positions of variables)

How to combine them?
We use a trie-like structure of “features”.

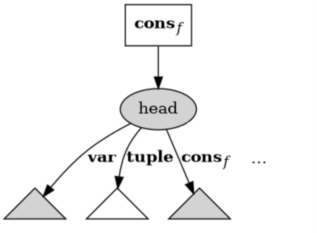
Unification criterions – combination

Updated criterion:

- A domain of values D
- $compat : D \times D \to Bool$
- ...
- An order on D such that $compat$ is monotonous

A trie of criterions:

- Given some criterions, we associate types to words in $\overline{D_i}$
- The database is stored as a trie of words
- Given a query (here, a constructor f), we recursively select all sub-tries with potentially valid types.



Dowsing – In the trenches

Some practical details on the implementation:

- Types are obtained by crawling the cmis
- Types are turned into a normal form
    - Application is n-ary
    - Arguments are sorted and tuples are merged
    - Hash-consing *everywhere*
- Memory representation/storage of the database not optimized so far (marshal)
- Full modified AC-unification implemented for the "SML" fragment, using the Boudet Algorithm
- A fairly creative stacked-functor design for the trie with heterogeneous words

Final benchmark

Benchmark on database containing containers, batteries and base:
$\sim$ 10000 functions

Type	Nb unif. w shortcuts	Time (ms)
$int \to int \to int$	50	0.368ms
$int \to int \to int \to int$	45	0.649ms
$int \to (int \to int) \to list(\alpha)$	67	0.415ms
$int \to float \to bool \to \mathrm{unit}$	62	1.26ms
$\alpha \to int \to \mathrm{unit}$	62	0.592ms
$int \to int \to \alpha$	29	0.393ms
$list(\alpha) \to \_ \to \alpha$	642	391ms

⇒ Instant in practice for many queries. Still work to do on very polymorphic queries

Ongoing and Future work – Indexing

We believe we can push indexing much further!

- Add new measures and appropriate criterions
- Find other ways to shortcut unification
- Proper serialization format for the database

Example of shortcut we do not exploit yet:

- Given τ_1 more general than τ_2, if another τ unifies with τ_2, it also unifies with τ_1. We can compute the “more general” semi-lattice in advance, and use it to avoid unifications.

Ongoing and Future work – Type system and Unification

Still many holes to fill on the type system and unification aspect

- We only cover the SML fragment. We are still missing polymorphic variants, objects, first class modules, (modular explicits), . . .
  Make a Bet: Is everything still decidable?
- Type aliases are unfolded eagerly right now. Can we do better?
- The unification procedure is reasonably efficient, but not highly tuned.
  ⇒ In particular, we really want early exit

Additional ideas:

- Search *type declarations* modulo isomorphism
- Consider isomorphic algebraic data types
- Look at modules . . .

Ongoing and Future work – Software Development

There is also quite a lot of dev to do:

- Reuse the odoc infrastructure
- Plug this into opam CI
- Develop a web-based frontend

Conclusion

We presented *Dowsing*, a new approach to search in libraries by types:

- Sound and complete
- Scales well to medium ecosystem (and beyond?)
- Good methodological approach to improve it further

Our technique is formalized and implemented:
`https://github.com/Drup/dowsing`
Still lot's of work to do to make this practical!