Back to the 90s!

Fast Indexing for Search by Types

Clément Allain    Gabriel Radanne    Laure Gonnord
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)

```sh
$ 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 ✓
Given a type \( \tau \), finds all functions in the libraries with types which are equivalent/match/unify to \( \tau \) up to some set of “simplifications”.

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

- \( a \times b \equiv_\tau b \times a \) (\( \times \)-comm)
- \( a \times (b \times c) \equiv_\tau (a \times b) \times c \) (\( \times \)-assoc)
- \( \text{unit} \times a \equiv_\tau a \) (\( \times \)-unit)
- \( (a \times b) \rightarrow c \equiv_\tau a \rightarrow (b \rightarrow c) \) (curry)

**Problem:** Unification/Matching modulo isomorphism is *expensive.*
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
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 \( \times 1000 \) 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:

- \( \text{vars}(\alpha \rightarrow \alpha) = 1 \)
- \( \text{vars}((\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \text{list}(\beta) \rightarrow \alpha) = 2 \)

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

<table>
<thead>
<tr>
<th>vars</th>
<th>total time (ms)</th>
<th>avg. time ((\mu)s)</th>
<th>nb. unif.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>53.8554</td>
<td>1.28044</td>
<td>42060</td>
</tr>
<tr>
<td>1</td>
<td>76.8702</td>
<td>8.60038</td>
<td>8938</td>
</tr>
<tr>
<td>2</td>
<td>68.7156</td>
<td>18.6575</td>
<td>3683</td>
</tr>
<tr>
<td>3</td>
<td>10.241</td>
<td>9.16014</td>
<td>1118</td>
</tr>
<tr>
<td>4</td>
<td>3.55721</td>
<td>12.3514</td>
<td>288</td>
</tr>
</tbody>
</table>

Observations:

- 75% unifs/ 25% time without any variables
- Unification on polymorphic type is harder
Second of metric: The head

Exemples:

- $\text{head}(\text{unit} \rightarrow \alpha) = \text{var}$
- $\text{head}(\text{int} \rightarrow \text{int} \rightarrow \text{list}(\alpha)) = \text{cons}_{\text{list}}$

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

<table>
<thead>
<tr>
<th>vars</th>
<th>total time (ms)</th>
<th>avg. time (µs)</th>
<th>nb. unif.</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>88.8178</td>
<td>39.7218</td>
<td>2236</td>
</tr>
<tr>
<td>constructor</td>
<td>80.8148</td>
<td>1.77701</td>
<td>45478</td>
</tr>
<tr>
<td>tuple</td>
<td>16.0539</td>
<td>1.91574</td>
<td>8380</td>
</tr>
<tr>
<td>other</td>
<td>0.442982</td>
<td>1.91767</td>
<td>231</td>
</tr>
</tbody>
</table>

Observations:

- 95% with a simple constructor at the head
- Case with variable head are pathological
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
We are looking for criterions that are necessary (but not sufficient!).

A criterion is composed of:

- A domain of values \( D \)
- \( encode : Types \rightarrow D \)
- \( compat : D \times D \rightarrow \text{Bool} \)
- \( \tau_1 \equiv_T \tau_2 \Rightarrow \text{compat}(encode(\tau_1), encode(\tau_2)) \)

We thus get a filter!

\[ \neg (\text{compat}(encode(\tau_1), encode(\tau_2))) \Rightarrow \tau_1 \not\equiv_T \tau_2 \]
We are looking for criterions that are necessary (but not sufficient!).

A criterion is composed of:

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

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:

- ··· → \textit{int} \not\equiv_T ··· → \textit{float}
- ··· → \textit{list}(\alpha) \not\equiv_T ··· → \textit{int} \times \textit{int}
- ··· → \textit{int} \equiv_T ··· → \alpha

We precompute the heads for all types in the database and store them compactly.
Searching in a local install of opam:
~ 250 packages, 31578 functions

<table>
<thead>
<tr>
<th>Type</th>
<th>Nb unif. w shortcut</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{int} \rightarrow \text{int} \rightarrow \text{int} )</td>
<td>2714</td>
<td>91.4%</td>
</tr>
<tr>
<td>( \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int} )</td>
<td>2714</td>
<td>91.4%</td>
</tr>
<tr>
<td>( \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{list}(\alpha) )</td>
<td>2945</td>
<td>90.7%</td>
</tr>
<tr>
<td>( \text{int} \rightarrow \text{float} \rightarrow \text{bool} \rightarrow \text{unit} )</td>
<td>5745</td>
<td>81.8%</td>
</tr>
<tr>
<td>( \alpha \rightarrow \text{int} \rightarrow \text{unit} )</td>
<td>5745</td>
<td>81.8%</td>
</tr>
<tr>
<td>( \text{int} \rightarrow \text{int} \rightarrow \alpha )</td>
<td>31578</td>
<td>0%</td>
</tr>
</tbody>
</table>

- We correctly avoid many unifications
- We need to work more for polymorphic queries
If multiplicities are incompatible, types can never unify:

\[
\begin{align*}
\text{int} & \rightarrow \text{int} \rightarrow \text{int} \not\equiv_T \\
\text{int} & \rightarrow \text{float} \\
\text{int} & \rightarrow \text{int} \not\equiv_T \\
\text{int} & \rightarrow \text{int} \rightarrow \alpha \rightarrow \alpha \\
\text{int} & \rightarrow (\text{unit} \rightarrow \text{unit}) \rightarrow \text{unit} \not\equiv_T \\
\text{int} & \rightarrow (\text{int} \rightarrow \text{float}) \rightarrow \text{int} \rightarrow \alpha \\
\text{int} & \rightarrow \text{list}(\beta) \rightarrow \text{int} \equiv_T \\
\text{int} & \rightarrow \alpha \\
\end{align*}
\]
Searching in a local install of opam:
～250 packages, 31578 functions

<table>
<thead>
<tr>
<th>Type</th>
<th>Nb unif. w shortcut</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>$int \rightarrow int \rightarrow int$</td>
<td>121</td>
<td>99.62%</td>
</tr>
<tr>
<td>$int \rightarrow int \rightarrow int \rightarrow int$</td>
<td>107</td>
<td>99.66%</td>
</tr>
<tr>
<td>$int \rightarrow (int \rightarrow int) \rightarrow \text{list}(\alpha)$</td>
<td>141</td>
<td>99.55%</td>
</tr>
<tr>
<td>$int \rightarrow float \rightarrow bool \rightarrow unit$</td>
<td>126</td>
<td>99.6%</td>
</tr>
<tr>
<td>$\alpha \rightarrow int \rightarrow unit$</td>
<td>2443</td>
<td>92.26%</td>
</tr>
<tr>
<td>$int \rightarrow int \rightarrow \alpha$</td>
<td>3677</td>
<td>88.36%</td>
</tr>
</tbody>
</table>

- 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 $D$
- $compat : D \times D \rightarrow \text{Bool}$
- ...  
- An order on $D$ such that $compat$ is monotonous

A trie of criterions:

- Given some criterions, we associate types to words in $D_i$
- 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.
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
Benchmark on database containing containers, batteries and base:
~ 10000 functions

<table>
<thead>
<tr>
<th>Type</th>
<th>Nb unif. w shortcuts</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int $\rightarrow$ int $\rightarrow$ int</td>
<td>50</td>
<td>0.368ms</td>
</tr>
<tr>
<td>int $\rightarrow$ int $\rightarrow$ int $\rightarrow$ int</td>
<td>45</td>
<td>0.649ms</td>
</tr>
<tr>
<td>int $\rightarrow$ (int $\rightarrow$ int) $\rightarrow$ list($\alpha$)</td>
<td>67</td>
<td>0.415ms</td>
</tr>
<tr>
<td>int $\rightarrow$ float $\rightarrow$ bool $\rightarrow$ unit</td>
<td>62</td>
<td>1.26ms</td>
</tr>
<tr>
<td>$\alpha$ $\rightarrow$ int $\rightarrow$ unit</td>
<td>62</td>
<td>0.592ms</td>
</tr>
<tr>
<td>int $\rightarrow$ int $\rightarrow$ $\alpha$</td>
<td>29</td>
<td>0.393ms</td>
</tr>
<tr>
<td>list($\alpha$) $\rightarrow$ _ $\rightarrow$ $\alpha$</td>
<td>642</td>
<td>391ms</td>
</tr>
</tbody>
</table>

⇒ Instant in practice for many queries. Still work to do on very polymorphic queries.
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 $\tau_1$ more general than $\tau_2$, if another $\tau$ unifies with $\tau_2$, it also unifies with $\tau_1$. We can compute the “more general” semi-lattice in advance, and use it to avoid unifications.
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
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!