# Towards practical out-of-order unification

Nils Anders Danielsson* and Víctor López Juan

University of Gothenburg and Chalmers University of Technology

Some implementations of type theories have a mechanism for implicit arguments, that allows users to omit some code from their programs and proofs. This kind of feature can be useful. However, at least in the case of Agda [8] it can also be slow, and sometimes one might wish that more implicit arguments could be instantiated. We are chipping away at these problems, and this note describes some preliminary results.

We have constructed an example that is too demanding for the current implementation of Agda. This example involves the definition of a dependently typed object language inside Agda, and then a definition of "setoid" inside this language. The definition of the object language only allows well-typed terms to be constructed, following McBride [6]. The definition of "setoid" in this language involves a large number of implicit arguments that the type-checker has to infer.

Our prototype, a variant of Tog [5], improves upon Agda by managing to type-check the example, and doing so in reasonable time and space.[1] The main techniques used to achieve this result are heterogeneous unification in the style of Gundry and McBride [4, 3], and hash consing inspired by Shao et al. [9] and Filliâtre and Conchon [2]. Note, however, that the prototype excludes several features of Agda, including termination checking, and that this should be kept in mind when benchmark data is analysed.

**Homogeneous vs. heterogeneous unification** Dependent type-checking can be reduced to solving a set of *general constraints* of the form $\Gamma \vdash t : A \approx u : B$ [5]. Such a constraint is well-formed if in context $\Gamma$ term $t$ has type $A$ and term $u$ has type $B$. Implicit arguments in the program syntax are represented by typed metavariables in the constraint syntax. When a metavariable $\alpha$ is instantiated by a term $t$ (which we require to be closed), any occurrence of $\alpha$ becomes definitionally equal to $t$. A constraint is solved by instantiating each metavariable in such a way that the two sides of the constraint become definitionally equal.

In a **homogeneous** unification approach, the two sides of a constraint have the same type. Mazzoli and Abel [5] split each general constraint $\Gamma \vdash t : A \approx u : B$ into two homogeneous *unifier constraints*, $C_1 \equiv \Gamma \Vdash A \approx B : \mathrm{Set}$ and $C_2 \equiv \Gamma \Vdash t \approx u : A$, where $C_2$ is only meaningful once $C_1$ has been solved. If the user program is ill-typed and $C_2$ is solved before $C_1$, then one can end up in a situation where the type-checker treats ill-typed terms as being well-typed [8]. Tog in homogeneous mode (and, to a lesser degree, Agda) will not tackle $C_2$ until $C_1$ is solved. This approach is too strict for our example, because it prevents unification under binders ($\lambda$ and $\Pi$) until the types of the bound variables have been unified.

By contrast, in the **heterogeneous** approach due to Gundry and McBride [4], unifier constraints can have two different types: $C_1 \equiv \Gamma \Vdash A : \mathrm{Set} \approx B : \mathrm{Set}$, $C_2 \equiv \Gamma \Vdash t : A \approx u : B$. The constraint $C_2$ can now be tackled before $C_1$. In the case where both sides of a constraint are headed by a binder (e.g. $\Gamma \Vdash \lambda y.t : \Pi(x : A_1)B_1 \approx \lambda y.u : \Pi(x : A_2)B_2$), the constraint is simplified by introducing a twin variable $\hat{x}$ of dual type $A_1 \ddagger A_2$, defined so that $\acute{x} : A_1$ and $\grave{x} : A_2$. The new constraint becomes $\Gamma, \hat{x} : A_1 \ddagger A_2 \Vdash t[\acute{x}/y] : B_1 \approx u[\grave{x}/y] : B_2$.

If a constraint reaches an irreducible form, say $\Gamma \Vdash \alpha\, u_1 \ldots u_n : A \approx t : B$, and the equation is in the pattern fragment [7, 1], then there will be a unique solution, here

*Supported by a grant from the Swedish Research Council (621-2013-4879).

[1] However, note that we have not proved that Tog is correctly implemented, so we give no guarantee that programs that are accepted by Tog are actually well-typed.

$\alpha := \lambda u_1 \ldots u_n.t$. However, before instantiating $\alpha$, the unifier must perform a compatibility check: $\Gamma \Vdash A : \mathrm{Set} \approx B : \mathrm{Set}$ must hold, and ditto for the two types of any twin variable $\hat{x}$ whose two projections both occur in the constraint. Thus it may seem as if the heterogeneous approach is no more powerful than the homogeneous approach. However, as the unifier works on $C_2$, intermediate constraints may be generated which, when solved, lead to the instantiation of metavariables. This additional information could be just what is needed to solve $C_1$.

**Our implementation** We have extended the prototype Tog [5] with a heterogeneous unifier based on Gundry and McBride's algorithm [4]. If a naive implementation of heterogeneous unification is used, then there is a risk that a significant amount of redundant work is performed when the types of left-hand and right-hand sides are similar. We use hash consing, and assign a unique identifier to each term. We can then memoize term-traversing operations (including normalization, substitution, computation of metavariables and free variables in terms, pruning of redundant metavariable arguments, $\eta$-expansion, and twin variable removal).

**Preliminary results** We have compared our heterogeneous unifier, an updated version of Tog's homogeneous unifier, and Agda.[2] Tog with the heterogeneous unifier type-checks the setoid example mentioned above in 131 s with hash consing and memoization (HC&M) disabled, and in 22 s with HC&M enabled. Tog with the homogeneous unifier deems the example unsolvable in 8 s with HC&M enabled, and Agda runs out of memory after more than 5 h. However, in other cases Tog (with the heterogeneous unifier and HC&M enabled) runs slower than Agda. It is an open question whether the use of HC&M can sometimes lead to excessive memory usage due to large memo tables.

We note that this case study provides one data point in favour of Gundry's belief that heterogeneous unification with twin variables [4] "handles a sufficiently broad class of problems to be useful for elaboration of a dependently typed language" [3, §4.3.4].

# References

[1] Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. In *TLCA 2011*. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-21691-6_5.

[2] Jean-Christophe Filliâtre and Sylvain Conchon. Type safe modular hash-consing. In *ML'06*, 2006. doi:10.1145/1159876.1159880.

[3] Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[4] Adam Gundry and Conor McBride. A tutorial implementation of dynamic pattern unification. Unpublished, 2012. URL `http://adam.gundry.co.uk/pub/pattern-unify/`.

[5] Francesco Mazzoli and Andreas Abel. Type checking through unification. Preprint arXiv:1609.09709v1 [cs.PL], 2016.

[6] Conor McBride. Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In *WGP'10*, 2010. doi:10.1145/1863495.1863497.

[7] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 1992. doi:10.1016/0747-7171(92)90011-R.

[8] Ulf Norell and Catarina Coquand. Type checking in the presence of meta-variables. Unpublished, 2007. URL `http://www.cse.chalmers.se/~ulfn/papers/meta-variables.html`.

[9] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *ICFP '98*, 1998. doi:10.1145/289423.289460.

[2] Tog: `https://lopezjuan.com/project/togt`, commit `1c7c2ce`. Agda: `https://github.com/agda/agda`, tag `v2.5.2`. Environment: Fedora 25 x64, Intel 2630QM, 16 GB RAM, GHC 8.0.1, up to 2 GB runtime heap.