Omnidirectional type inference for ML

Alistair O'Brien
ajo41@cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Didier Rémy
didier.remy@inria.fr
Inria
France

Gabriel Scherer
gabriel.scherer@inria.fr
Inria & Irif, Université Paris Cité
France

At the heart of the Damas-Hindley-Milner (ML) type system lies the ability to *guess* types. For instance, in $\lambda x.\, e$, the parameter type for x is *guessed* and subsequently constrained by the body e. The elegance of the ML discipline is that such guesses are never arbitrary: there always exists a "most general guess"—a *principal type*. Every well-typed expression e admits a principal type σ from which all other valid types for e are instances of σ. This guarantees both predictability and efficiency of inference: local typing decisions are always optimal without resorting to backtracking.

Principality, however, is *fragile*. Many extensions of ML—GADTs [4, 10], higher-rank polymorphism [7, 11], and static overloading [2]—break it. Consider the following example in OCaml using overloaded record field names:

```
type point    = {x : int; y : int}
type gs_point = {x : int; y : int; color : int}

let getx p  = r.x
                ^^^
Error (warning 41 [ambiguous-name]):
x belongs to several types: gs_point point.
```

When typing `getx`, one could *guess* the type of `r` to be either `point` or `gs_point`—neither of which is strictly more general than the other, violating principality!

Principality can always be recovered through explicit type annotations—for instance, annotating the projection `r.x` as `(r : point).x`. Each *fragile* construct (*e.g.* an overloaded field projection $e.\ell$) therefore admits a *robust* counterpart (*e.g.* a qualified projection $e.T.\ell$), whose *natural* typing rules preserve principality. However, robust forms are cumbersome to use, as they always require explicit annotations. Inference algorithms thus rely on a minimal amount of *known* type information to disambiguate fragile terms and elaborate them into their robust counterparts [9].

In practice, specifying what counts as *known type information* declaratively is difficult. Existing approaches typically rely on one of two frameworks: *bidirectional* type inference [8] and *π-directional* inference [3]. Both impose some *static* ordering of inference using it to propagate inferred types and user-provided annotations as known information. For instance, in a bidirectional system, a type τ is *known* when it is: **(1)** part of an annotation; **(2)** supplied as input to a checking conclusion ($\Gamma \vdash e \Leftarrow \tau$); or **(3)** an output of a synthesizing premise ($\Gamma \vdash e \Rightarrow \tau$). While effective in many settings, the rigidity of a static ordering causes even simple examples—whose types could easily be guessed—to be rejected; for instance, `fun r → (r.x, r.color)` is ill-typed in OCaml.

We propose *omnidirectional* type inference, which relies on a *dynamic* order of inference: constraints may be solved in any order, suspending whenever progress requires *known* type information. Other constraints may continue to be solved; once the missing information becomes available (typically via unification), the suspended typing constraints are resumed.

This approach comes naturally for simply typed systems (when inference is purely unification-based), but becomes challenging in the presence of ML-style *let-generalization*. In existing ML inference algorithms, typing a let-binding let $x = e_1$ in e_2 follows a static order: first type e_1, then generalize its type, and finally type the body e_2 under the extended environment. Consider `let f r = r.x in f {color; x; y}`. Here, we cannot generalize the type of `f` first, since the type of `r.x` is still unknown. Instead, we must typecheck the body `f {color; x; y}`, thereby discovering that `r` has the record type `gs_point`. We solve this by introducing *incremental instantiation i.e.*, the ability to instantiate type schemes that are not yet fully determined and consequently revisit their instances when they are being refined, *incrementally*.

Thus far, we have developed:

(1) The OmniML calculus, an extension of ML featuring advanced extensions of OCaml such as static overloading of record labels, together with an *omnidirectional recipe* to systematically derive its typing rules.

(2) A sound, complete, and *principal* constraint-based inference algorithm for OmniML, along with an efficient prototype implementation available at https://github.com/johnyob/omniml.

(3) A novel constraint language for omnidirectional inference, equipped with a declarative semantics for suspended constraints using a new characterization of *known* type information.

Suspending unsolved constraints is not new—it appears in dependently typed systems [1, 5, 6] and in OutsideIn [10] in the form of *delayed implication* constraints—but combining this technique with ML-style *local let-generalization*, and giving it a declarative semantics, is novel. Our ultimate aim is to apply this to OCaml, where *local let-generalization* is indispensible (unlike in Haskell, which only generalizes at the top level), yet this makes omnidirectional inference especially challenging to specify and implement.

References

[1] Guillaume Allais, Malin Altenmüller, Conor McBride, Georgi Nakov, Fredrik Nordvall Forsberg, and Craig Roy. 2022. TypOS: An operating system for typechecking actors. In *28th International Conference on Types for Proofs and Programs, TYPES*. 20–25.

[2] Arthur Charguéraud, Martin Bodin, Jana Dunfield, and Louis Riboulet. 2025. Typechecking of Overloading. In *Journées Francophones des Langages Applicatifs*.

[3] Jacques Garrigue and Didier Rémy. 1999. Extending ML with Semi-Explicit Higher-Order Polymorphism. *Information and Computation* 155, 1/2 (1999), 134–169. http://www.springerlink.com/content/m303472288241339/ A preliminary version appeared in TACS'97.

[4] Jacques Garrigue and Didier Rémy. 2013. Ambivalent Types for Principal Type Inference with GADTs. In *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8301)*, Chung-chieh Shan (Ed.). Springer, 257–272. https://doi.org/10.1007/978-3-319-03542-0_19

[5] Gérard Huet. 1975. A unification algorithm for typed λ-calculus. *Theoretical Computer Science* 1, 1 (1975).

[6] Dale Miller. 1991. Unification of Simply Typed Lamda-Terms as Logic Programming. In *Logic Programming, Proceedings of the Eigth International Conference, Paris, France, June 24-28, 1991*, Koichi Furukawa (Ed.). MIT Press, 255–269.

[7] Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 54–67. https://doi.org/10.1145/237721.237729

[8] Benjamin C. Pierce and David N. Turner. 1998. Local Type Inference. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, David B. MacQueen and Luca Cardelli (Eds.). ACM, 252–265. https://doi.org/10.1145/268946.268967

[9] FranCcois Pottier and Yann Régis-Gianas. 2006. Stratified type inference for generalized algebraic data types. In *Principles of Programming Languages (POPL)*. 232–244. http://cambium.inria.fr/~fpottier/publis/pottier-regis-gianas-popl06.pdf

[10] Tom Schrijvers, Simon L. Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. 2009. Complete and decidable type inference for GADTs. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, Graham Hutton and Andrew P. Tolmach (Eds.). ACM, 341–352. https://doi.org/10.1145/1596550.1596599

[11] Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A quick look at impredicativity. *Proc. ACM Program. Lang.* 4, ICFP (2020), 89:1–89:29. https://doi.org/10.1145/3408971