Frozen inference constraints for type-directed disambiguation
Olivier Martinot, Gabriel Scherer

To cite this version:
Olivier Martinot, Gabriel Scherer. Frozen inference constraints for type-directed disambiguation. ML Family Workshop, Aug 2021, online, South Korea. hal-03510890

HAL Id: hal-03510890
https://inria.hal.science/hal-03510890
Submitted on 4 Jan 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Frozen inference constraints for type-directed disambiguation

Olivier Martinot, Gabriel Scherer

Abstract
We present work-in-progress on type inference in presence of overlapping variant constructor or record field names; abstract engine, Inferno [Pottier 2014a,b].

Disambiguation within a constraint-based type inference is robust and predictable, but it relies on annotations in subterms, a simple form of bidirectional type inference. This is translated away during the constraint-generation phase, in terms of these more general constraints. Correspondingly, we propose a new constraint construct, frozen constraints, that is less specific than disambiguation of variant constructors.

1 Introduction
OCaml allows to use datatypes (sums/variants and records) with overlapping variant constructor or record field names; in this case it uses type-based disambiguation to tell which constructor or field is meant, or warns. (This feature was added in OCaml 4.01 in 2013; it is also found in other languages, Agda for example.)

\begin{verbatim}
let x : t = A
let y : u = A
let z = A
\end{verbatim}

In constraint-based inference, typability of the user program is translated into a typing constraint whose unknowns are type (meta)variables, and passed to a constraint solver. Type-inference designers and implementors are encouraged to pick a relatively generic constraint language, with a few expressive constructs/combinators, to keep the solver simple.

1.3 Constructor disambiguation using frozen constraints
Constraint generation for a term \( t \) is typically described as a function \( \| t \|_\alpha \) that generates a constraint \( C \), where the inference variable \( \alpha \) represents the expected type for the term \( t \) in its context. For example, the constraint for an application can be defined as

\[
\| t \|_\alpha \overset{\text{def}}{=} \exists \beta_1. \exists \gamma_u. ((\beta_1 = y_u \to \alpha) \land \| t \|_{\beta_1} \land \| u \|_{\gamma_u})
\]

To support constructor disambiguation, we generate for a constructor application \( K t \) the constraint

\[
\| K t \|_\alpha \overset{\text{def}}{=} \exists \beta_1. \exists \gamma_u. ((\alpha) f_{K,\beta_1} \land \| t \|_{\beta_1})
\]

where \( f_{K,\beta_1} \) is a callback that is called once \( \alpha \) has been (partially) inferred to some non-variable type, typically a datatype \( D \). At this point there is no ambiguity anymore: several constructors \( K \) may exist in scope, but at most one of them has type \( D. f_{K,\beta_1} \) will look for \( D \) in the type declaration environment, and a unification constraint between \( \beta_1 \) and the argument type of its constructor \( K \). Type-inference fails with
an error if $\alpha$ has been unified with a type that is not a sum/variant type, or if it has no constructor $K$.

This use of frozen constraints does not try to disambiguate constructors in all possible cases. For example, if may be that there are two constructors $K$ declared in the environment, but the type of $t$ is only compatible with one of them; our approach makes no attempt to enumerate the declarations of $K$ and try them with backtracking. This differs from previous proposals for disambiguation using disjunctive constraints – generating a disjunction of constraints, one for each constructor $K$ in the environment. We don’t guess, we wait for the context to propagate disambiguation information on the type of $K$. This is much closer in spirit and behavior to the bidirectional propagation of type annotations, but it uses general unification rather than bidirectional propagation, so it would apply in more situations.

For example, the following uses of the constant constructor $K$ at type $D$ would be disambiguated, illustrating the absence of an arbitrary direction of propagation to type applications: $\langle x : D \rangle K,$ and $(x : D \rightarrow t) K,$ and $(\lambda x . \text{match } x \text{ with } K \rightarrow u) (t : D)$.

**Remark.** One may think of this approach to typed-directed disambiguation as a limited form of qualified types (say, type-class inference). Inference for qualified types collects usage constraints from terms (as our frozen constraints); at generalization time, unsolved usage constraints are turned into qualified constraint abstractions, to be instantiated separately by each caller. We use frozen constraints for a language feature for which the qualified-type approach is not desirable: we do not want ad-hoc polymorphism on constructor names, their meaning should be exactly the same for all callers of the surrounding definition. We discuss in the next section the issues that arise at generalization-time.

Another interesting distinction is that frozen constraints may in general produce arbitrary constraints once unfrozen, whereas a qualified type constraint or type-class typically can express only simpler constraints on its qualified names. It is not obvious, for example, how one could allow for disambiguation of GADT constructors using qualified types – how to express the still-unknown existential types and equality constraints in the qualified constraint, and how to type-check its application under this constraint.

### 3 Implementation challenges

When encountering a frozen constraint $\langle \alpha \rangle C,$ our solver waits for $\alpha$ to be unified with some non-variable type structure. At the very end of the solving process, if some constraints remain frozen, we don’t make any guess but fail with a type error – the user needs to add annotations to disambiguate the program.

The difficulty is the interaction between frozen constraints and let-generalization. Suppose a definition let $x = t$ in the middle of the program, where we want let to be typed using Hindley-Milner generalization. If a frozen constraint $\langle \alpha \rangle f$ is generated within the constraint for $t$, and remains frozen once the solver finishes working on $t$, what should we do? Should we fail now, or delay the frozen constraint, and how does this interact with generalization?

We reason on whether $\alpha$ itself is generalizable. If $\alpha$ is generalizable, then this means that no type in the rest of the program mentions $\alpha$. No inference work in the rest of the program will ever deduce a non-variable structure for $\alpha$, so we can as well fail now with an error.

If $\alpha$ is not generalizable, it is tempting to postpone the resolution of $\langle \alpha \rangle f$ and go on with generalization. But this is unsound: the constraints generated by $f$ once $\alpha$ will be known may mention other inference variables currently in scope, some of which may be in generalizable position. If we generalize without looking at $f$, they will be out of scope by the time $\langle \alpha \rangle f$ gets unfrozen.

Our solution is to reason on $f$ as a closure: we reason on the set of inference variables that are “captured” by $f$, mentioned in any possible expansion of $f$ – in our implementation we ask the programmer of the constraint generator to list these captured variables.

If none of the captured variables of the callback are generalizable, then the frozen constraint can safely be delayed to later – failing now would be incomplete. If some of the captured variables, say $\beta$, are generalizable, we have a problem again: it is unsound in general to generalize now without unfreezing $\langle \alpha \rangle f$, but is incomplete to fail now, because unfreezing $\alpha$ would result in a constraint $f(\alpha)$ that may unify $\beta$ with a non-generalizable type.

Consider for example the following program:

```
fun old ->
  let g x = 1 + old (K x) in
  (g 0, (old : D -> int))
```

When inferring `let g x = ...` before generalization, $K x$ produces a frozen constraint whose needed variable is its return type $\alpha$, and the use of the `old` function makes this $\alpha$ non-generalizable. The callback of the frozen constraint mentions the inference variable of $x$, which is generalizable. Generalizing now would give the polymorphic type `$a \rightarrow \text{int}$, which is unsound. But failing is incomplete, as the program contains an annotation `(old : D -> \text{int})` that would unfreeze the frozen constraint, whose solving would in turn correctly infer $x$ depending on the declaration of $K$.

The more complete approach in this case (which we hope is actually complete) is to `suspend disbelief`: if generalization encounters a young variable $\beta$ that is needed by a yet-unsolved callback function frozen on the non-generalizable $\alpha$, we stop mid-way through generalization, producing a partially-frozen scheme $\forall a_1...a_n . \langle \gamma \rangle \beta ... \gamma$ remains in an uncertain state until $\alpha$ is unfrozen, and we continue inferring the rest of the program, hoping to be able to un-freeze $\alpha$. When our solver encounters an instantiation constraint

Olivier Martinot, Gabriel Scherer
We need to move from a linear array/stack of regions to a tree, so they may be uniquely indexed by their rank.

For more intuition on the connection to region-based memory allocation, see Kiselyov.

4 Theoretical challenges

We can give a simple semantics to frozen constraints, in the usual style of a judgment $\phi \vdash C$ stating that $\phi$ is a solution to the constraint $C$, where $\phi$ is a mapping from inference variables to fully-inferred types, which do not mention inference (meta)variables anymore.

\[
\phi \vdash f(\phi(\alpha)) \\
\phi \vdash \langle\alpha\rangle f
\]

However, our solver is not complete with respect to this semantics. Consider for example the complete constraint (on a free existential variable $\alpha$) $C \defeq \langle\alpha\rangle(\lambda_.\alpha = \text{int})$: this constraint contains no information on how to guess $\alpha$ outside the callback, but once we unfreeze it it tells us that $\alpha$ must be int. Our solver will not unfreeze the constraint and fail with a typing error, but the assignment $\alpha \mapsto \text{int}$ is a solution: $\{\alpha \mapsto \text{int}\} \vdash \langle\alpha\rangle(\lambda_.\alpha = \text{int})$ holds.

We believe that the solver is right and the semantics is wrong, or rather, that the semantics fails to match our intuition of the semantics should be. Our intent is that the value of a needed variable $\alpha$ should not be guessed, but deduced – without knowing the constraint. A semantics that encourages guessing types out of thin air is simpler, but it does not capture our intuition and we believe that it would be complex and costly to implement in practice.

We are still looking for a stricter semantics that forbids this kind of “out of thin air” solution, by imposing that the value of $\alpha$ is determined outside the frozen constraint. (This may be similar to some works in type inference that some part of their derivations to be principal, to forbid over-specialized solutions.) The question is how to do this while preserving a declarative specification, as simple as possible.

Acknowledgments

We are very grateful to François Pottier and Didier Rémy for fruitful discussions on this work.

References


François Pottier. the Inferno library: https://gitlab.inria.fr/fpottier/inferno, 2014b.