Frozen inference constraints for type-directed disambiguation

Olivier Martinot, Gabriel Scherer

Abstract

We present work-in-progress on type inference in presence of type-directed name disambiguation (where an ambiguous name is resolved using type information), following the approach of constraint-based type inference.

Our specific goal is to implement inference for OCaml sum/variant constructor disambiguation, and our approach is to introduce "frozen constraints", a more general constraint mechanism. We have a prototype implementation of frozen constraints in the Inferno library, and discuss the implementation and meta-theoretical challenges.

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.)

```
type t = A
type u = A

let x : t = A
let y : u = A

let z = A
> ^
> Warning 41 [ambiguous-name]:
> A belongs to several types: u t
> The first one was selected.
> Please disambiguate if this is wrong.
```

OCaml performs this disambiguation by propagating type annotations in subterms, a simple form of bidirectional type inference. This is robust and predictable, but it relies on arbitrary choices (in applications `t u`, do we want to propagate information from `t` to `u` or the other way around?; in `let p = e in ...`, do we want to propagate information from the pattern `p` to `e` or conversely?) that can frustrate users.

In this work, we decided to look at what a unification-based propagation of disambiguation information would look like. We implemented a prototype of type-based constructor disambiguation within a constraint-based type inference engine, Inferno [Pottier 2014a,b].

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. The specifics of their type-system 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.

2 Frozen constraints

Frozen constraints represent situations where more type information is needed to decide *how* we want to type-check something.

Let us write $\alpha \in \mathcal{V}$ for inference (meta)variables, $\tau \in \mathcal{T}$ for partially-inferred types containing inference variables, and $C \in \mathcal{C}$ for inference constraints.

A *frozen constraint* $\langle\alpha\rangle f$ is built by pairing an inference variable α, the *needed variable* of the constraint, and a function $f : \mathcal{T} \to \mathcal{C}$ from partially-inferred types to constraints, the *callback*. The intent is to wait until α gets unified to a non-variable type τ, and have the frozen constraint then behave as the constraint $f(\tau)$. (This is an informal description of the expected solver behavior.)

2.1 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 α represents the expected type for the term t in its context. For example, the constraint for an application can be defined as

$$[\![t\ u]\!]_\alpha \stackrel{\text{def}}{=} \exists\beta_t.\exists\gamma_u.\ ((\beta_t = \gamma_u \to \alpha)\ \wedge\ [\![t]\!]_{\beta_t}\ \wedge\ [\![u]\!]_{\gamma_u})$$

To support constructor disambiguation, we generate for a constructor application $\mathsf{K}\ t$ the constraint

$$[\![\mathsf{K}\ t]\!]_\alpha \stackrel{\text{def}}{=} \exists\beta_t.\ (\langle\alpha\rangle f_{\mathsf{K},\beta_t}\ \wedge\ [\![t]\!]_{\beta_t})$$

where f_{K,β_t} is a callback that is called once α 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,β_t} will look for D in the type declaration environment, and a unification constraint between β_t and the argument type of its constructor K. Type-inference fails with

an error if α 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 progagation to type applications: $(\lambda(x : D).\ t)\ K$, and $(x : D \to \tau)\ K$, and $(\lambda x.\ \mathsf{match}\ x\ \mathsf{with}\ \mathsf{K} \to u)\ (t : D)$.

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

If α 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 α 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 β, are generalizable, we have a problem again: it is unsound in general to generalize now without unfreezing $\langle\alpha\rangle f$, but is is incomplete to fail now, because unfreezing α would result in a constraint $f(\alpha)$ that may unify β 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 α, and the use of the `old` function makes this α 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 -> int`, which is unsound. But failing is incomplete, as the program contains an annotation `(old : D -> 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 β that is needed by a yet-unsolved callback function frozen on the non-generalizable α, we stop mid-way through generalization, producing a *partially-frozen* scheme $\forall\alpha_1..\alpha_n.\ldots\langle\alpha\rangle\beta\ldots.\ \beta$ remains in an uncertain state until α is unfrozen, and we continue inferring the rest of the program, hoping to be able to un-freeze α. When our solver encounters an instantiation constraint

$x \leqslant \tau$, for a use of a let-bound variable x whose generalization is suspended on the needed variable α, it must generate a fresh variable γ for the suspended part $\langle\alpha\rangle\beta$. Once α gets unfrozen, the solver must be able to come back and unify this γ with the correct generalization of β.

To summarize, when encountering a still-frozen constraint $\langle\alpha\rangle f$ at generalization time:

1. If α is generalizable, fail.
2. If α and all the variables mentioned/captured by f are non-generalizable, proceed with generalization and keep the frozen constraint for later.
3. If α is non-generalizable, but some variables captured in f are generalizable, suspend generalization of those variables and try type-checking the rest of the program.

3.1 Generalization tree

The standard approach to efficiently implement generalization is to track the *rank* of each inference variable, where the rank measures the number of nested let-bindings between the place where the inference variable is currently scoped and the root of the term. As inference progresses, unifications widen the scope of variables, so they become bound in earlier let-binding, their rank decreases. The generalizable variables are exactly those whose rank remained exactly the rank of the let-binding we are about to exit.

In particular, Inferno implements its inference state as a dynamic array of of regions[1] (sets of variables bound in the same let-binding), indexed by their rank. Unfortunately, this implementation is not flexible enough to implement "generalization suspension": if the generalization of the most recent region gets partially suspended due to frozen constraints, the region must be kept alive to store the suspended variables. Inference continue in the rest of the program, but we may then enter a new let-binding that creates a new region, that is neither a child nor an ancestor of the suspended region. We need to move from a linear array/stack of regions to a *tree* of generalization regions.

With a tree of generalization regions, the notion of "rank" does not uniquely identify a variable region anymore. When unifying two variables, it is not enough to compute their minimum rank and assign it to both. They move to the nearest common ancestor in the tree, which may be older than the region of both variables.

However, rank optimizations are still partially applicable. For example, when we generalize a leaf region, all the variables we traverse/inspect were created in this, and some were moved to one of its ancestors. All regions of interest lie in a path from the current region to the root of the region tree, so they may be uniquely indexed by their rank.

[1] 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 ϕ is a solution to the constraint C, where ϕ is a mapping from inference variables to fully-inferred types, which do not mention inference (meta)variables anymore.

$$\frac{\phi \Vdash f(\phi(\alpha))}{\phi \Vdash \langle\alpha\rangle f}$$

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

Oleg Kiselyov. How OCaml type checker works – or what polymorphism and garbage collection have in common. URL http://okmij.org/ftp/ML/generalization.html.

François Pottier. Hindley-Milner elaboration in applicative style. In *ICFP*, September 2014a. URL http://gallium.inria.fr/~fpottier/publis/fpottier-elaboration.pdf.

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