Frozen inference constraints for type-directed disambiguation

Olivier Martinot, Gabriel Scherer

Partout, Inria Saclay, France

August 21, 2021
Type-directed disambiguation

Many language support type-directed disambiguation of names. How to combine this with type inference?
Type-directed disambiguation

Many language support type-directed disambiguation of names. How to combine this with type inference?

*Type classes* (qualified types):
  - nice inference through *constraint abstraction*
  - excellent approach for operator overloading.
Type-directed disambiguation outside qualified types

A feature where type classes are not enough: data constructor disambiguation.

\[
f (K \ t) \\
mismatch t \text{ with } K \ x \rightarrow u
\]
Type-directed disambiguation outside qualified types

A feature where type classes are not enough:  
*data constructor* disambiguation.

\[
f (K \ t) \\
\text{match } t \text{ with } K \ x \rightarrow u
\]

1. We do not want to *abstract* over \( K \).
2. The type of \( K \) may not be expressible as a class argument  
   (existentials, etc.; data constructors are not functions.)
3. Different constructors \( K \) may have vastly different typing rules.
Constructor disambiguation and type inference

\[ \text{f (K t)} \]
\[ \text{match t with K x -> u} \]

Need program types to disambiguate \( K \).
Need the type of \( K \) to infer program types.

HM type inference:
propagation by unification (within generalization boundaries).

Bidirectional type inference (commonly used for disambiguation):
leafward propagation from annotations (robust)
+ some lateral propagation (fragile): \( t u \)

This Work In Progress explores unification-based type disambiguation
\textit{frozen constraints}. 
Constraint-based type inference: a primer

implicitly-typed $t \overset{\text{generate}}{\Rightarrow}$ constraint $C \overset{\text{solve}}{\Rightarrow}$ explicitly-typed $t'$

Constraint for application $t \ u$ with return type variable $\alpha$:

$$
[t \ u]_\alpha \overset{\text{def}}{=} \exists \beta_t. \exists \gamma_u. ( (\beta_t = \gamma_u \rightarrow \alpha) \land [t]_{\beta_t} \land [u]_{\gamma_u} )
$$
Frozen constraints

\[ \langle \alpha \rangle f \]

\( \alpha \): type inference variable
\( f \): function from partial types to constraints

waits on a type unification variable \( \alpha \):
when \( \alpha \) becomes (partly) defined as \( \tau \),
the constraint \( f(\tau) \) must be solved.

Constructor constraint (non-GADT case):

\[ \llbracket K \ t \rrbracket_\alpha \overset{\text{def}}{=} \exists \beta_t. (\llbracket t \rrbracket_{\beta_t} \land \langle \alpha \rangle(\lambda \tau. \beta_t = \text{arg\_type}(\tau, K))) \]

Principled (and principal) inference with type-disambiguation.
(Maybe too restrictive?)

Difficult to combine with generalization!
Practical difficulty: generalization (1/2)

If $\langle \alpha \rangle f$ remains unsolved “at the end”, type inference fails.

But when is the end?
Practical difficulty: generalization (1/2)

If $\langle \alpha \rangle f$ remains unsolved “at the end”, type inference fails.

But when is the end?

How does $\langle \alpha \rangle f$ interact with let-generalization?
Practical difficulty: generalization (2/2)

Generalization: which inference variables $\alpha$ are *local* and and can be generalized into polymorphic variables?
Practical difficulty: generalization (2/2)

Generalization: which inference variables $\alpha$ are local and can be generalized into polymorphic variables?

Frozen generalization of $\tau$:
if a variable $\beta$ of $\tau$ is “blocked” by a frozen constraint, it must be tracked during instantiation and possibly generalized later.

Partially-frozen schemas:
- On generalization: store $\beta$ as a blocked schema variable.
- On instantiation: track the instance of the partially-frozen schema.
- When $\beta$ gets unblocked: continue generalization, update tracked instances.

Delicate to implement. Difficult to implement efficiently.
Theoretical difficulty: semantics (1/3)

Constraints are given meaning by a solution relation $V \models C$.

A good constraint generator has correct solutions.

A good constraint solver (big-step function or small-step rewrites) preserves solutions.

$$
\frac{\tau[V] =_{ty} \tau'[V]}{V \models \tau = \tau'}
$$

$$
\frac{V \models C[T/\alpha]}{(T, V) \models \exists\alpha. C}
$$

How to specify frozen constraints?
Theoretical difficulty: semantics (2/3)

Natural approach:

\[ V \models f(\alpha[V]) \]
\[ \frac{V \models f(\alpha[V])} {V \models \langle \alpha \rangle f} \]

This specification allows “out of thin air” behaviors.

\[ [\alpha \mapsto \text{int}] \models \langle \alpha \rangle (\lambda \tau. \alpha = \text{int}) \]

Our solver does not: the specification is not precise enough.
We want to express that $\alpha[V]$ is determined “without looking inside $f$”. How can we do this?

Morally:

\[
\frac{C[T \text{ determines } \alpha]}{V \vdash C[f(\alpha[V])]}
\]

\[
V \vdash C[\langle \alpha \rangle f]
\]
Summary

Frozen constraints: interesting but difficult constraint combinator.

Work in progress.

Thanks! Questions?