A Two-Pass Typechecking Algorithm for Resolving Overloaded Symbols

Arthur Charguéraud

Inria

September 20th, 2022

Objectives

Goal: static resolution of overloading for OCaml, Coq, etc.
Without introducing semantic indirections.

$1 + (0 + x)$ where $x \in \mathbb{Z}$ should be *syntactic sugar* for $1_\mathbb{Z} +_\mathbb{Z} +(0_\mathbb{Z} +_\mathbb{Z} x)$
It should not be an expression that *evaluates* or *reduces* to it.

$1 + (0 + x)$ where $x \in \mathbb{R}$ should resolve to $1_\mathbb{R} +_\mathbb{R} +(0_\mathbb{R} +_\mathbb{R} x)$.

$\varnothing \cup E$ where E is a set should resolve to $\varnothing_{\mathsf{set}} \cup_{\mathsf{set}} E$.
$\varnothing \cup M$ where M is a map should resolve to $\varnothing_{\mathsf{map}} \cup_{\mathsf{map}} M$.

Instances at the syntax level

1. **A piece of notation resolves to the application of an overloaded token applied to a number of arguments.**
   $0 + x$ resolves at parsing to `add zero x`

2. **An instance is a typed value registered for a token.**
   ```
   let add = __instance (int_add : int -> int -> int)
   let add = __instance (float_add : float -> float -> float)
   ```

3. The typechecker searches for the unique matching instance, based on the type of the arguments and the expected return type.
   ```
   let r = 0 + (x:float)    →  let r = float_add (0:float)x
   let r : int = 0 + 0      →  let r = int_add (0:int)(0:int)
   ```

A two-pass algorithm

1. **Propagate type annotations and expected type for function arguments downwards.**
   Try to resolve instances based on return type, if possible.
   Else, type-check function arguments without an expected type.

2. **Compute type of subexpressions and return this info upwards.**
   Try to resolve remaining instances based on arguments and expected type. Else, return "type unknown".

3. **Re-typecheck arguments downwards when an instance is resolved.**
   E.g., one argument allows resolving the function token, then the type of the function propagates to the other arguments.

About two-pass typechecking algorithms

- **Static resolution based on arguments, as in C++ templates.**
  But resolution does not depend on the expected return type.
- **Extensive bibliography on bidirectional type-checking.**
  Challenges: intuitive/predictable; avoid quadratic/exponential.
- **Similar two-pass algorithms implemented in ADA and PVS.**
  But without support for proper polymorphism.
- **This work: generalize the ideas to ML, then Coq.**
  Summer internship 2022: prototype on core-ML. See demo.

Future work

Roadmap for the "défi" → with Martin Bodin.

1. Demonstrate overloading in a large fragment of Caml.
2. Integrate Coq extensible notation system for ML-style code.
3. Check that it works for all pieces of standard mathematical notation.
4. Understand the interactions with coercions.
5. Understand the interactions with dependent types.

Simplifying assumptions

The types of free variables is always known.

- Type of function arguments must be provided as annotations.
- A let-bound variable may have its type infer from its definition.
- "Implicit Types" may reduce the number of required annotations.
- Specific support for quantifiers/iterators/big-ops: in $\sum_{x \in E} f(x)$, if E has type set A then the bound variable x has type A.