Tabled Typeclass Resolution

Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura

Presenters: Dor Alter (s1027021), Orpheas van Rooij (s1080509)

Date: December 17, 2021

Type Classes 101

- A way to provide ad-hoc polymorphism
  - Parametric Polymorphism: $\lambda\ (a\ :\ \star)\ (x\ :\ a).\ x$
  - Adhoc Polymorphism: $\{a\ :\ \mathit{Type}\ \_\}\ [\mathit{show}\ a]\ :\ a\ \rightarrow\ \mathit{string}$
- Originated in the Haskell Language [1]
- Idea:
  - Provide extra properties to a group of types
  - Declare polymorphic functions that work on types with instances of a certain class(es)
  - Functions behave differently for each type
- These properties are passed implicitly to functions declared with type class constraints

[1] How to make ad-hoc polymorphism less ad hoc; Wadler et al.; 1989

Type Classes 101

```haskell
class Inhabited a where
    something :: a

instance Inhabited Integer where
    something = 1

instance (Inhabited a, Inhabited b) =>
         Inhabited (a, b) where
    something = (something, something)

foo :: Inhabited a => (a,a)
foo = something

```

Listing 1: Type classes in Haskell

```
class inhabited (a : Type _) :=
  (something : a)

open inhabited

instance : inhabited ℤ := ⟨1⟩

instance {a b : Type _}
         [inhabited a] [inhabited b] :
         inhabited (a × b) :=
  ⟨(something, something)⟩

def foo {a : Type _} [inhabited a] :
        (a × a) :=
        something

```

Listing 2: Type classes in Lean

Type Classes 101

```
Class inhabited (a : Type) :=
   something : a
.
Instance inhaNat : inhabited nat :=
  { something := 1 }
.
Instance inhaCompo {A B : Type} `{inhabited A, inhabited B} :
         inhabited (A * B) := { something := pair something something }
.

Compute (something : (nat * nat)). (* (1,1) *)

```

Listing 3: Type classes in Coq

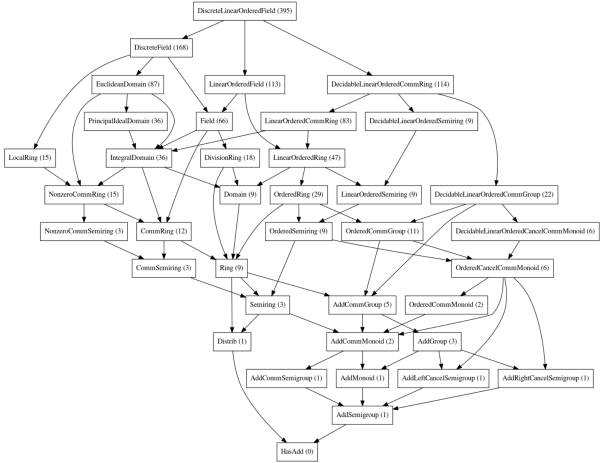
Type Classes á la Lean

- Heavily used to organize `mathlib` (library of formal mathematics)
- Additionally store **proofs** in classes (e.g. $+$ is commutative in $\mathbb{N}$)
- Instances can overlap so backtracking search needed [2]
  - Instances $\leftrightarrow$ Horn Clauses
  - Search problem $\leftrightarrow$ Resolution proof
- **Lean3** uses vanilla selective linear definite clause (SLD) resolution
- **Lean3community** uses an enhanced SLD version [3] (performs some form of caching)
- **Lean4** uses *Tabled Typeclass Resolution*
  - Memorize unresolved and solved subgoals during class resolution

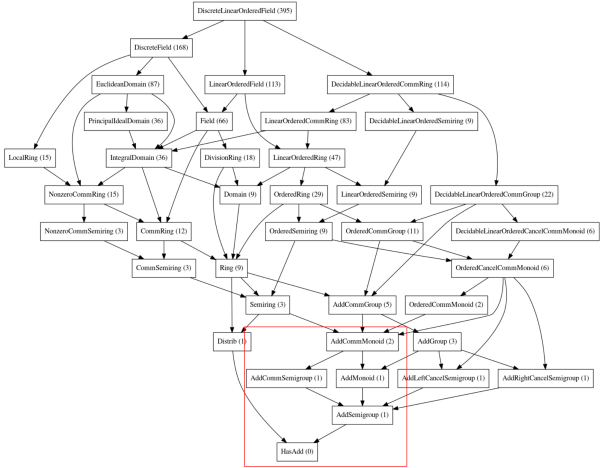
[2] Haskell doesn't allow overlapping instances. *OverlappingInstances* pragma can be used but it is **more** restrictive

[3] Not verified

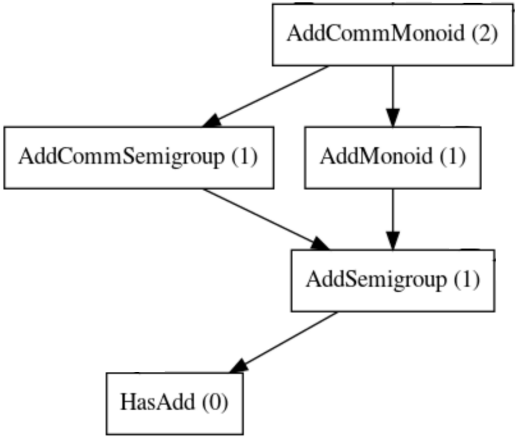
Subgraph of `mathlib`'s inheritance graph



Subgraph of `mathlib`'s inheritance graph



Subgraph of `mathlib`'s inheritance graph



Overlapping Instances

```
1  class inhabited (a : Type _) :=
2    (something : a)
3
4  open inhabited
5  open list
6
7  instance : inhabited ℤ := ⟨ 1 ⟩
8
9  instance {a b : Type _} [inhabited a] [inhabited b] : inhabited (a × b) :=
10   ⟨ (something, something) ⟩
11
12 instance nilInhabitant {a : Type _} : inhabited (list a) := ⟨ nil ⟩
13
14 instance consInhabitant {a : Type _} [inhabited a] : inhabited (list a) :=
15   ⟨ cons something nil ⟩
16
17 #eval (something : (list ℤ × list bool)) -- yields ([1], [])
18
```

Type Class Resolution

$$\textit{inhabited}\ \mathbb{Z} \tag{1}$$

$$\forall \alpha, \beta, \neg \textit{inhabited}\ \alpha \vee \neg \textit{inhabited} \beta \vee \textit{inhabited} (\alpha \times \beta) \tag{2}$$

$$\textit{inhabited}\ [\alpha] \tag{3}$$

$$\forall \alpha, \neg \textit{inhabited}\ \alpha\ \vee\ \textit{inhabited}\ [\alpha] \tag{4}$$

$$\text{Goal: } \neg \textit{inhabited}\ ([\mathbb{Z}] \times [\textit{bool}]) \tag{5}$$

$$\neg \textit{inhabited}\ [\mathbb{Z}]\ \vee \neg \textit{inhabited} [\textit{bool}]\ \ (2,5)\{\alpha := [\mathbb{Z}], \beta := [\textit{bool}]\} \tag{6}$$

$$\neg \textit{inhabited}\ \mathbb{Z} \vee \neg \textit{inhabited} [\textit{bool}]\ \ (4,6)\{\alpha := \mathbb{Z}\} \tag{7}$$

$$\neg \textit{inhabited}\ [\textit{bool}]\ \ (1,7)\{\} \tag{8}$$

$$\text{Backtrack: } \neg \textit{inhabited}\ \textit{bool}\ \ \ (4,8)\{\alpha := \textit{bool}\} \tag{9}$$

$$\text{Empty Clause: } \bot\ \ \ (3,9)\{\alpha := \textit{bool}\} \tag{10}$$

Problems with Type Class Resolution in Proof Assistants

Two problems can occur in resolution:

- Diamond problem
  - Different ways of deciding that an object is an instance of another
    i.e. multiple paths to solve a constraint
  - Exponential running time when downstream goals in diamond tower fail
- Cycle problem
  - Search space contains cycles
  - Resolution might not terminate without care
  - Coq can detect cycles thus avoid applying the same instances

Diamond Problem

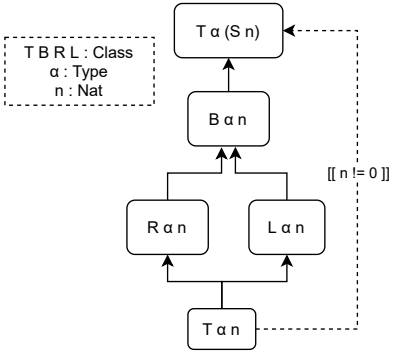


Figure: Class inheritance relationship

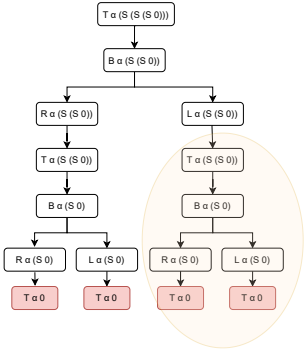


Figure: Class resolution using vanilla SLD

Cycle Problem Example

```
1  class coercible (a : Type _) (b : Type _) :=
2    (coerce : a → b)
3
4  open coercible
5
6  instance : coercible bool ℕ := ⟨(λ (x : bool), if x then 1 else 0)⟩
7
8  instance : coercible ℕ ℤ := ⟨(has_coe.coe : ℕ → ℤ) ⟩
9
10 instance : coercible ℤ bool := ⟨((λ (x : ℤ), if x = 0 then ff else tt)⟩
11
12 instance coerceTrans {a b c: Type _} [coercible a b] [coercible b c] :
13          coercible a c := ⟨(λ (x : a), coerce (coerce x : b))⟩
14
15 #eval (coerce tt : bool) -- maximum class-instance resolution depth reached
16
```

Cycle Problem Workaround in Lean3

```
class coercible (a : Type _) (b : Type _) := (coerce : a → b)
class coercibleT (a : Type _) (b : Type _) := (coerce_t : a → b)

instance : coercible bool ℕ := ...
instance : coercible ℕ ℤ := ...
instance : coercible ℤ bool := ...

instance coeTrans {a b c: Type _} [coercible a b] [coercibleT b c] :
         coercibleT a c := ⟨ λ (x : a), coerce_t (coerce x : b) ⟩
instance coeBase {a b: Type _} [coercible a b] :
         coercibleT a b := ⟨ λ (x : a), coerce x ⟩

-- instances that cause non termination need to be in coercibleT
instance coeopt {a : Type _} : coercibleT a (option a) :=
  ⟨ λ (x : a), some x ⟩

#eval (coerce_t tt : bool)  -- tt

```

Tabled Typeclass Resolution

- Uses a search forest rather than search tree (like SLD does)
- Tracks all subgoals using a *table*
- Two type of nodes:
  1. Generator node which stores a list of instances to try
  2. Consumer node which maintains a list of subgoals that remain to be solved to finish the search tree
- Two stacks:
  1. The generator stack for generator nodes
  2. The resume stack for (solution, consumer node) pairs that have yet to be tried

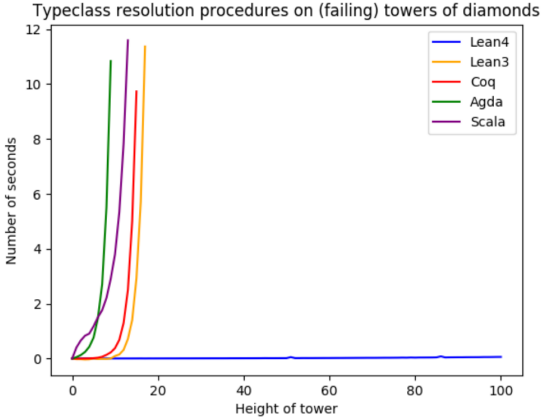
Example

We are going to see how the algorithm works on the following example:

```
instance I1: R A B
instance I2: R A C
instance I3: R C D
instance I4: {X Y Z : Type} : R X Y -> R Y Z -> R X Z
#synth R A D

```

Comparison to other Implementations for the Diamond Problem



Thank you for your attention.

Questions ?