Tabled Typeclass Resolution

Daniel Selsam[1], Sebastian Ullrich[2], and Leonardo de Moura[1]

[1] Microsoft Research, USA
[2] Karlsruhe Institute of Technology, Germany

Abstract. Typeclasses provide an elegant and effective way of managing ad-hoc polymorphism in both programming languages and interactive proof assistants. However, the increasingly sophisticated uses of typeclasses within proof assistants has exposed two critical problems with existing typeclass resolution procedures: the *diamond problem*, which causes exponential running times in both theory and practice, and the *cycle problem*, which causes loops in the presence of cycles and so thwarts many desired uses of typeclasses. We present a new typeclass resolution procedure, called *tabled typeclass resolution*, that solves these problems. We have implemented our procedure for the upcoming version (v4) of the Lean Theorem Prover, and we confirm empirically that our implementation is exponentially faster than existing systems in the presence of diamonds. Our procedure is sufficiently lightweight that it could easily be implemented in other systems. We hope our new procedure facilitates even more sophisticated uses of typeclasses in both software development and interactive theorem proving.

1 Introduction

Typeclasses were introduced in [49] as a principled way of enabling *ad-hoc* polymorphism in functional programming languages, with the motivating example of overloading equality and arithmetic operators. They were first implemented in the Haskell programming language [14], and since then they have been extended in various ways [2,30,21,19,22,10], have found many diverse uses within Haskell [18,11,15,33,25,28,29,31,37,24,4], and have emerged as an organizing principle for the language and its libraries [20].

Typeclasses have also spread to interactive proof assistants such as Isabelle/HOL [35], Coq [5,41], and Lean [34], and have had a major impact on the organization of libraries of formal mathematics [36,42,43,27,16]. However, the increasingly sophisticated uses of typeclasses within these contexts have exposed two critical problems with the existing typeclass resolution procedures: the *diamond problem* and the *cycle problem*. The diamond problem is that hierarchies of mathematical abstractions contains towers of *diamonds*—*i.e.* multiple paths between two nodes in the typeclass search tree—and the running time of the existing typeclass resolution procedures on failing subqueries is exponential in the height of these towers. The cycle problem is that many desired uses of typeclasses involve cycles in the instance graph, *e.g.* coercing in both directions between two types, yet such cycles may cause the existing resolution procedures to loop. Despite the myriad extensions to typeclass mechanisms proposed over the years, to the best of our knowledge all existing typeclass resolution procedures for interactive proof assistants are still based on (naïve) tree search and are thus susceptible to both exponential blowup in the presence of diamonds and non-termination in the presence of cycles.

Similar issues plagued early versions of logic programming systems such as Prolog and Datalog. Eventually, more sophisticated search procedures were introduced into logic programming systems to address these issues [46,48,7,3,39,8,9,12,40,45,50]. Although there have been many variants proposed in the literature, most fall under the umbrella of *tabled resolution*. The main idea of tabled resolution is that during the search, a table is maintained that maps subgoals to the set of solutions that have already been found for that subgoal, and solutions are reused from the table whenever possible instead of being recomputed. Note that naïve caching does not suffice, since multiple subgoals may mutually depend on each other.

In this work, we take inspiration from this line of work and propose a new typeclass resolution procedure that we call *tabled typeclass resolution* that solves both problems. Specifically, it eliminates the exponential blowup associated with the diamond problem and guarantees termination under the *bounded term-size assumption* [47]. We have implemented our procedure for the upcoming version (v4) of the Lean Theorem Prover[3], and we confirm empirically that our implementation is exponentially faster than existing systems in the presence of diamonds. Our procedure is sufficiently lightweight that it could easily be implemented in other systems. We hope our new procedure facilitates even more sophisticated uses of typeclasses in both software development and interactive theorem proving.

2 Preliminaries

Ad-hoc polymorphism. There are two distinct kinds of polymorphism one may want in a programming language: parametric polymorphism and ad-hoc polymorphism [44]. In parametric polymorphism, a function may be defined over a range of types as long as it behaves the same on every type in the range. For example, one may wish to write a single function `length` that returns the length of an arbitrary list, no matter what type of element the list contains.

In ad-hoc polymorphism, a function may be defined over a range of types but behave differently on different types in the range. For example, one may wish to use the same operator `+` to represent addition on many different types, such as natural numbers, rational numbers, lists and sets, even though the actual implementations of addition are arbitrarily different for each such type. One may also want to write additional functions like `double` in terms of `+` that are agnostic as to how addition is implemented for the type of its argument.

[3] `http://github.com/leanprover/lean4/blob/IJCAR20/src/Init/Lean/Meta/SynthInstance.lean`

Typeclasses. Typeclasses were introduced in [49] as a principled way of enabling ad-hoc polymorphism in functional programming languages. We first observe that it would be easy to implement an ad-hoc polymorphic function (such as addition) if the function simply took the type-specific implementation of addition as an argument and then called that implementation on the remaining arguments. For example, suppose we declare a structure in Lean to hold implementations of addition:

```
structure Add (α : Type) := (add : α → α → α)
```

Note that this statement is the Lean analogue of the Haskell statement `data Add a = Add { add : a -> a -> a }`. In the above Lean code, the field `add` has type

```
add : {α : Type} → Add α → α → α → α
```

where the curly braces around the type α mean that it is an *implicit* argument. We could implement `double` by

```
def double {α : Type} (addα : Add α) (x : α) : α := add addα x x
```

and we could double a natural number `n` by `double { add := natAdd } n`.[4] Of course, it would be highly cumbersome for users to manually pass the implementations around in this way. Indeed, it would defeat most of the potential benefits of ad-hoc polymorphism.

The main idea behind typeclasses is to make arguments such as `Add α` implicit, and to use a database of user-defined *instances* to synthesize the desired instances automatically through a process known as *typeclass resolution*. In Lean, by changing `structure` to `class` in the example above, the type of `add` becomes

```
add : {α : Type} → [Add α] → α → α → α
```

where the square brackets indicate that the argument of type `Add α` is *instance-implicit*, *i.e.* that it should be synthesized using typeclass resolution. This version of `add` is the Lean analogue of the Haskell term `add :: Add a => a -> a -> a`. Similarly, we can register an instance by

```
instance natAddInst : Add Nat := { add := natAdd }
```

Then for `n, m : Nat`, the term `add n m` triggers typeclass resolution with the goal of `Add Nat`, and typeclass resolution will synthesize the instance `natAddInst`. In general, instances may depend on other instances in complicated ways. For example, we can declare an (anonymous) instance stating that if α has addition, then `Vec α n` has addition:

```
instance {α} {n : Nat} [Add α] : Add (Vec α n) := { add := addVecAdd }
```

The set of instances in a given development can be seen as forming a logic program [38]. For example, the two instances above induce the following two Horn clauses:

1. `Add Nat`

[4] here we assume `natAdd` has already been defined.

2. `∀ α n, Add α → Add (Vec α n)`

Given a type `T`, terms of type `T` constructed using the instances in a development are in one-to-one correspondence with (resolution) proofs of the corresponding theorem in the induced logic program. This phenomenon is an instance of the Curry-Howard Isomorphism [6,17].

Typeclass resolution. Existing typeclass resolution procedures can all be seen as implementing variants of selective linear definite clause (SLD) resolution [26], which also formed the basis of early Prolog systems. We defer discussion of the salient differences among existing systems to later sections. SLD resolution performs a depth-first search of a tree in which every node has an ordered list of remaining subgoals, and every edge corresponds to resolving the conclusion of a rule against a node's first subgoal. It maintains a stack of the nodes yet to be expanded, and the main loop is as follows:

1. Peek at the top node on the stack, with remaining goals $\overrightarrow{G}$.
2. If $\overrightarrow{G}$ is empty, the query has been resolved.
3. If all instances have been tried at this node, pop the node and continue.
4. Otherwise, let I be the next instance not yet tried for this node, and resolve it with the first subgoal $\text{head}(\overrightarrow{G})$ to produce new goals $\overrightarrow{H}$, and push a node with subgoals $\overrightarrow{H} + \text{tail}(\overrightarrow{G})$ onto the stack.

Typeclasses in interactive proof assistants. For our present purposes, the main feature that distinguishes interactive proof assistants from traditional functional programming languages is the ability to manipulate theorems and proofs in addition to programs. In particular, in systems such as Lean and Coq, one may define a class that stores not just implementations of functions (*e.g.* `+`) but also proofs about implementations (*e.g.* that `+` is commutative). There are many valuable uses of typeclasses in interactive theorem proving, such as inferring that types are finite, that predicates are decidable, and that terms can be coerced into terms of other types.

In this work we focus on the most critical use of typeclasses in formal mathematics: organizing the complex web of relationships between abstract mathematical objects. For example, informally, a group is a monoid for which the binary operator satisfies additional properties; a ring has two binary operators, one of which induces a group while the other induces a monoid; and a field is a ring with both an additional operator and additional properties. In practice, the web of algebraic relationships is vastly more sophisticated than these informal examples might suggest. For example, it is critical to distinguish *e.g.* monoids from commutative monoids, groups from Abelian groups, rings from semirings, and fields from division rings. It is also critical to be able to reason about abstract objects together with various orderings, *e.g.* partially-ordered commutative monoids and linearly-ordered fields. Decidability must be tracked as well, *e.g.* to distinguish rings with undecidable linear orders from rings with decidable ones. And of course, the complex web of relationships between all these abstract objects must be maintained as well, so that *e.g.* a theorem proved about monoids can be used to prove a theorem about groups.

One of the main challenges in building libraries of formal mathematics is organizing these relationships in such a way that appropriately abstract theorems can be stated, proved, and used conveniently in all appropriate contexts. Typeclasses have proven to be an elegant and effective way of addressing all of these challenges and form the basis for prominent libraries of formal mathematics in both Lean and Coq. However, existing typeclass mechanisms suffer two critical limitations in such regimes. First, the web of relationships are littered with *diamonds*, *i.e.* multiple ways of showing that one kind of object is an instance of another, and such diamonds cause exponential blowup in the standard typeclass resolution procedure. Second, they are also littered with *cycles*, which cause SLD to loop and so must be carefully preempted. We now describe these problems in more detail.

3 The Diamond Problem

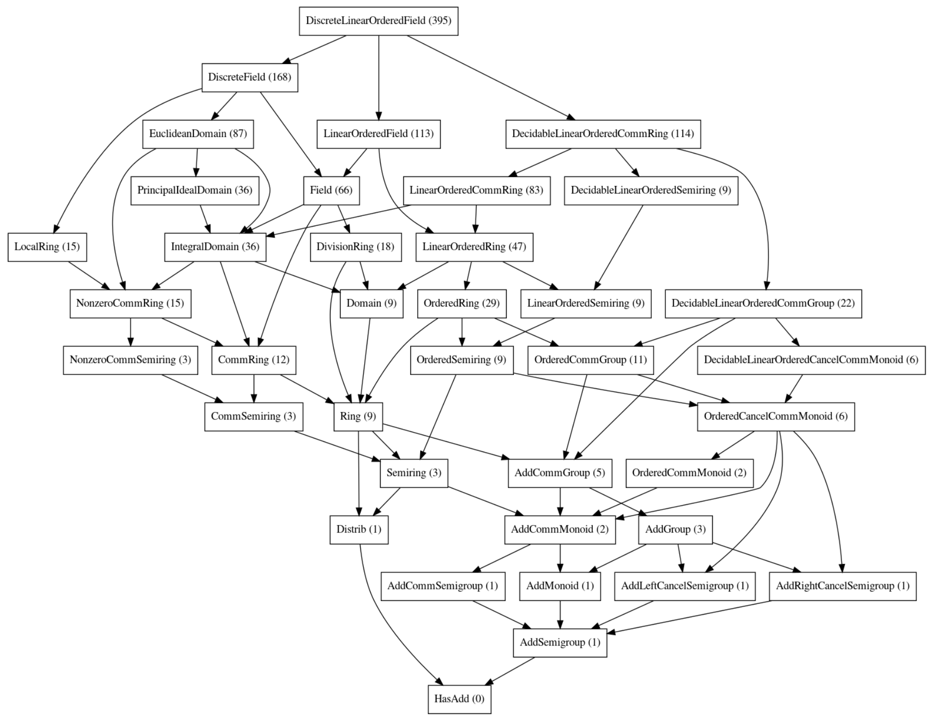


Fig. 1: A small subgraph of Mathlib's class inheritance graph. The rapidly-growing numbers in parentheses indicates the number of distinct paths from a node to the sink, `HasAdd`.

Diamonds are ubiquitous in formal mathematics. As mathematician Thomas Hales writes: “for mathematicians, diamonds are extremely natural and they occur in great abundance under many names (pullbacks, fibered products, Cartesian squares, etc.).” [13]. Figure 1 shows a small subgraph of Mathlib’s class inheritance graph. Each node represents a class, and a directed edge from one node to another signifies that the former either inherits directly from the latter, or that it includes as a parameter an instance of the latter. The number in the parentheses of a node’s label indicates how many distinct paths there are in this subgraph from that node to the sink `HasAdd`, which is used to resolve the `+` notation. Crucially, the number of paths to the sink grows exponentially in the height of this graph due to the presence of diamonds. For example, since there are two paths from `AddCommMonoid` and three from `AddGroup`, and since `AddCommGroup` inherits from both of them, there are five paths from `AddCommGroup`. At the top of the graph, we see that the number paths nearly triples in a single step, producing almost four hundred paths from `DiscreteLinearOrder`. We stress that this is only (a small subset of) the class inheritance graph, and that there are thousands of other instances in Mathlib, many of which induce additional layers of potential diamonds in the instance graph. It is hard to even count these diamonds statically, since certain diamonds may or may not be activated on different queries depending on the set of the instances in scope.

The *diamond problem* refers to the exponential blowup that occurs whenever typeclass resolution traverses all paths within a tower of diamonds of nontrivial depth. This can happen for two related reasons in SLD resolution: the entire tower may fail, or the tower may succeed but downstream goals may fail thus causing all solutions to the tower to be enumerated in sequence.

To the best of our knowledge, all existing typeclass resolution procedures take exponential time on the first case, including the ones in Lean3, Coq, and GHC. Lean3’s procedure is effectively vanilla SLD, and it takes exponential time in both cases accordingly. Coq’s procedure takes exponential time in the first case as well, but Coq extends SLD in such a way that it avoids exponential work in (the common version of) the second case. Specifically, Coq detects when a subgoal does not appear in any downstream goals and does not contain unification variables in its type, in which case it commits to the first solution found for that subgoal rather than naïvely enumerating alternative solutions that are considered unlikely to affect failing downstream goals.[5] Unlike Lean and Coq, Haskell distinguishes between class inheritance and declared instances. Class inheritance may induce diamonds, and to avoid needing to search at call-sites, GHC eagerly computes the closure of the class inheritance graph. This process takes exponential time in the height of the diamond tower (even if the results are never needed). In contrast,

[5] Haskell requires that there can only be one unique choice of instance for any given type, whereas systems based on more expressive logics such as Lean and Coq do not generally enforce this rule. However, it is still common to assume that instances are “morally canonical” when convenient, even though in some logics, different choices of instances may even affect whether downstream goals succeed or fail.

Haskell prohibits diamonds for declared instances, and so at each call-site, GHC can synthesize the desired instances without any backtracking.

§5 introduces a new typeclass resolution procedure that definitively solves the diamond problem, and that scales in the number of edges in the graph rather than the number of paths.

4 The Cycle Problem

A second problem of the typeclass resolution procedures based on SLD resolution is that it may not (and without care, does not) terminate, even when the number of distinct subgoals encountered is finite. This shortcoming imposes severe limitations on the use of typeclasses. For example, the original typeclass paper [49] suggested a `Coe` typeclass that represents coercions from type α to β:

```
class Coe (α β : Type) : Type := (coe : α → β)
```

The idea is that if a term `(x : α)` of type α is used in a context expecting a term of type β, then typeclass resolution would try to synthesize a term of type `Coe α β` and replace `(x : α)` with `(coe x : β)`. This is indeed how coercions are managed in Lean. However, coercing back and forth between two types would cause SLD resolution to loop. Thus even though *e.g.* finite sets and finite multisets may usefully coerce into each other, one direction must be chosen arbitrarily for the `Coe` instance and the other must be sacrificed.

Cycles can be very convenient in many areas of formal mathematics, and Lean3's failure to handle them has been a frequent source of frustration for Mathlib users. One desirable instance allows restricting the scalars in a module:

```
class Module (A M : Type) [Ring A] [AddCommGroup M] : Type
class Algebra (R A : Type) [CommRing R] [Ring A] : Type
instance {k A M : Type} {c : CommRing k} {r : Ring A}
         {g : AddCommGroup M} [Algebra k A] [Module A M] : Module k M
```

This instance would immediately loop if the `Module A M` subgoal were tried first, but can also induce a loop even if `Algebra k A` were tried first; for example, due to an existing instance that every commutative ring is an algebra over itself.

The *cycle problem* refers to the infinite loops that occur when typeclass resolution blindly tries to solve a particular goal as a subgoal of itself. There are two cases worth distinguishing: when there is no solution to the query, and when there is a solution that may be missed due to the loop. Lean3's procedure is effectively vanilla SLD, and it loops in both cases. Coq's procedure loops in the first case, but it can be made to succeed in the second case by toggling the iterative-deepening flag. In contrast, Haskell is not expressive enough to encode the kinds of cycles we are interested in.

The new typeclass resolution procedure we introduce in §5 solves the cycle problem definitely, and guarantees convergence under the bounded term-size assumption.

5 Tabled Typeclass Resolution

We now describe our new typeclass resolution procedure, *tabled typeclass resolution*, that avoids the exponential blowup resulting from towers of diamonds and that guarantees termination under the *bounded term-size assumption* [47]. Our procedure is based on the tabled resolution procedure introduced for Prolog in [39]. We have implemented our procedure for the upcoming version (v4) of the Lean Theorem Prover, and although our actual implementation supports some advanced features that may not be feasible in all relevant languages, these features are orthogonal to the new procedure itself and so we focus our presentation on the universally-applicable core.

5.1 High-level description

Recall from §2 that SLD resolution performs a depth-first search of a tree in which every node has an ordered list of remaining subgoals, and every edge corresponds to resolving the conclusion of a rule against a node's first subgoal. Most of the problems with SLD resolution arise from the fact that SLD will try to solve the same subgoals over and over again in different parts of the search tree. Whereas SLD resolution maintains a single search tree for the entire resolution problem, tabled typeclass resolution maintains a search *forest*, with a distinct search tree for each distinct subgoal (up to α-equivalence) encountered during resolution. Whenever a subgoal is encountered, rather than searching for solutions to it from scratch as in SLD, tabled typeclass resolution looks for the search tree corresponding to that subgoal (in the so-called "table"). If the search tree does not already exist, the current branch of the search forest is *suspended*, and control jumps to the new search tree. On the other hand, if the search tree does already exist, and if there are already solutions for it, those solutions are used and the search continues.

There are several other cases that need to be considered as well. For example, the search tree may already exist, but there may not be any solutions to it yet, in which case control does not jump to the search tree, but the fact that the current branch of the search forest depends on it is still recorded. Whenever a new solution is found to any search tree, all other branches of the search forest that depend on it are *resumed* with the new solution. Thus the search may be highly *nonlinear*. Indeed, although the algorithm we present is relatively simple, it can nonetheless induce sophisticated and counterintuitive control flow.

Our tabled typeclass resolution procedure distinguishes between two types of nodes: *generator nodes* and *consumer nodes*. Generator nodes are in one-to-one correspondence with the search trees, and form the roots of these trees. They each behave like the root node of the SLD search tree, in the sense that they are expanded by resolving the subgoal with the available instances in sequence. All other nodes are *consumer nodes*, and like the internal nodes in SLD, consumer nodes maintain a list of subgoals that remain to be solved to establish the subgoal corresponding to its search tree (which we refer to as its *ancestor goal*). However, in contrast to SLD nodes, consumer nodes are not expanded by resolving their first subgoal against the available instances, but rather by resolving it against the *solutions* that have been found for the search tree corresponding to that subgoal.

Our tabled typeclass resolution procedure maintains two distinct stacks, the *generator stack* for generator nodes, and the *resume stack* for (solution, consumer node) pairs that have yet to be tried. It also maintains a *table*, which maps each distinct subgoal to a *table entry* that includes the set of solutions already discovered for it along with every consumer node that is known to depend on it.

```
 1: procedure tabledTypeclassResolutionMainLoop
 2:     while true do
 3:         if resume stack is not empty then
 4:             pop (cnode, solution) from resume stack
 5:             if first subgoal of cnode does not resolve with solution then continue
 6:             if cnode has no remaining subgoals then
 7:                 extract new solution s to cnode's ancestor goal g
 8:                 if g is original query then return s
 9:                 if s already in table then continue
10:                 add s to cnode's table entry
11:                 for every cnode c dependent on g do
12:                     push (c, s) onto resume stack
13:             else
14:                 newConsumerNode(remaining subgoals)
15:         else if generator stack is not empty then
16:             peek at gnode on top of generator stack
17:             if no remaining instances for gnode to try then pop generator stack
18:             if next instance resolves with gnode's goal then
19:                 newConsumerNode(new subgoals)
20:         else
21:             fail
22: procedure newConsumerNode(subgoals)
23:     if first subgoal g of subgoals is not in table then
24:         insert new table entry for g into table
25:         push new generator node for g onto generator stack
26:     for each solution to g, push new cnode onto resume stack with it
27:     add new cnode to g's dependents
```

Fig. 2: High-level pseudocode for the main loop of tabled typeclass resolution.

Figure 2 provides high-level pseudocode for the body of the main loop of tabled typeclass resolution. Before entering the main loop, it creates a table entry for the query, and pushes a generator node for it onto the generator stack. Then in the loop, it firsts checks to see if the resume stack is nonempty (3). If it is, it pops a pair from it (4), and tries resolving the first subgoal of the popped consumer node with the popped solution (5). If the unification succeeds and if the consumer node has no remaining subgoals (6), then a new solution to the consumer node's ancestor goal has been discovered (7). If the ancestor goal happens to be the original query, it returns the solution (8). Otherwise, if the solution is new, it must be added to the corresponding table entry (10), and all other consumer nodes that depend on the ancestor goal must be pushed onto the resume stack along with the newly discovered solution (12). If the unification succeeds but the consumer node still has subgoals remaining, then its creates a new consumer node with them (14) by calling NewConsumerNode on the list of subgoals. This subroutine first checks to see if the first subgoal has been visited yet (23), and if not creates a table entry for it (24) and pushes a new generator node for it onto the generator stack (25). For each existing solution to the first subgoal, it pushes the consumer node along with that solution onto the resume stack (26), and finally registers the fact that the new consumer node depends on its first subgoal (27).

On the other hand, if the resume stack is empty but the generator stack is not (15), then it instead peeks at the generator node on the top of the generator stack (16). If there are no remaining instances to be tried, it pops the generator node and continues (17). If there are still instances to be tried then it tries the next instance (18), and if it succeeds, creates a new consumer node for the remaining subgoals (19). If both stacks are empty, the procedure fails (21).

5.2 Example

Before discussing implementation details, we first provide more intuition for our procedure by walking through the following small 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
```

In this example, a transitive relation `R` satisfies the three ground facts `R A B`, `R A C` and `R C D`, and the goal is to synthesize a term of type `R A D`. Figure 3 shows a visualization of tabled typeclass resolution running on this example.

Subgoal	Solutions	Dependents
`R A D`	`I4 I2 I3 : R A D`	
`R A ?X`	`I1 : R A B`, `I2 : R A C`	N2
`R B D`		N5
`R B ?X`		N7, N9
`R C D`	`I3 : R C D`	N11

Table 1: The state of the table when tabled resolution finishes on the example problem.

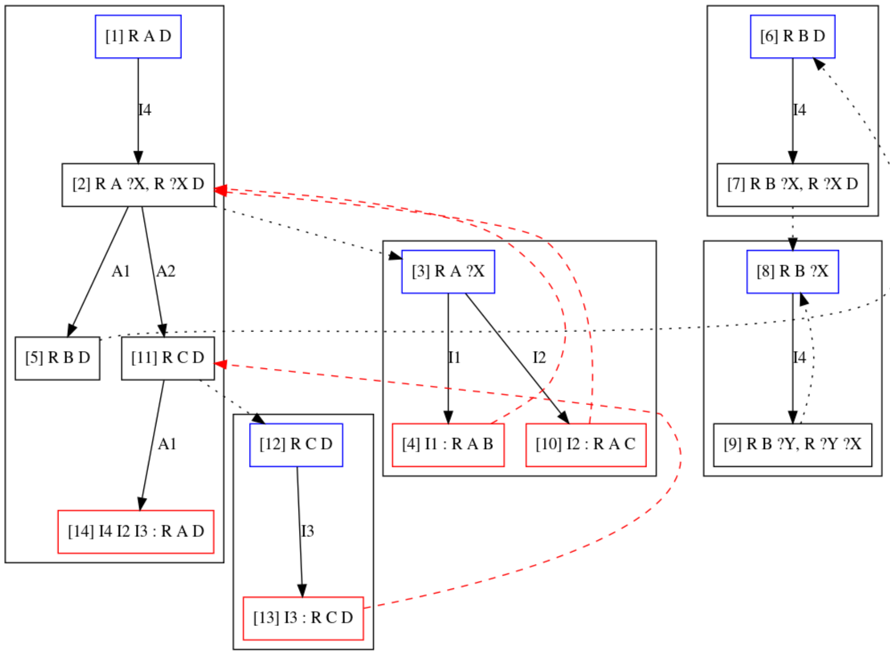


Fig. 3: Illustration of tabled typeclass resolution running on the example problem. Blue nodes indicate generator nodes, black nodes indicate consumer nodes, and red nodes indicate solutions. The nodes are numbered in the sequence that they are created. Each distinct search tree corresponding to a distinct subgoal is contained in its own rectangle. The edges within trees are solid and represent the resolution of the first subgoal of a node with either an instance or a solution from the table. There are two types of edges between trees: black (dotted) edges that represent a dependence of a consumer node on a subgoal, and red (dotted) edges indicating a solution being used to resume a consumer node.

We first create a generator node (N1) for the original goal, `R A D`, which is the blue node numbered 1 in Figure 3. We also create a new table entry for `R A D`, since it is the first time this subgoal has been encountered. The final table entries at the end of the procedure are shown in Table 1. Next, we try to resolve `R A D` with the instances in sequence. The instance `I4` is the only one that succeeds, and it produces the two subgoals `R A ?X` and `R ?X D`. We create a new consumer node (N2) with these two subgoals, and with N1 as parent. Since we have not encountered any α-variant of N2's first subgoal `R A ?X` yet, we create a new generator node (N3) and a new table entry for it. We add N2 to N3's dependency list to indicate that N2 will eventually consume the solutions found for the subgoal of N3. This dependency is represented by the dotted line leaving N2. Next, we expand the generator node N3 by trying to resolve its subgoal with the instances in sequence. `I1` succeeds and produces the solution `I1 : R A B`. We add this solution (N4) to the table entry for `R A ?X`, and then resume N2 with it. Resolving N2's first subgoal `R A ?X` with `R A B` yields a new consumer node (N5) with `R B C` as its only subgoal. We have not encountered `R B C` yet, so we create a new generator node (N6) for it as well as a new table entry, and add N5 to its dependency list.

The procedure proceeds as we have just described until creating consumer node N9, whose first subgoal `R B ?Y` is an α-variant of the subgoal `R B ?X` that already has a table entry. N9 is added to the dependency list, and since there no solutions in the table yet for `R B ?X`, control backtracks to the most recent generator node with instances that have not been tried yet (in this case N3), and continues the search with the next instance. Eventually, node N14 constitutes a solution to the original query, which the resolution procedure returns.

5.3 Suspending and resuming branches

Tabled resolution procedures for Prolog were heavily complicated by the need to save and restore the environments (*i.e.* the current assignment of unification variables) whenever suspending and resuming branches of the search forest. To resume a branch, the entire sequence of variable assignments from the root to the current node needed to be replayed. We did not even mention this challenge in the pseudocode of §5.1 because our procedure does nothing special to support saving or resuming environments: it simply stores the environment for each node using *persistent data structures* which enable compact storage of overlapping environments as well as constant-time copies (see [23] for an overview). The usual downside of using persistent datastructures is that querying, inserting, and deleting are generally slower than in their imperative counterparts. For workloads with many queries between backtracking steps (as might arise inside an SMT solver [1], the performance overhead of persistent datastructures may be devastating. However, typeclass resolution generally has the opposite profile: frequent, non-linear context jumps with relatively few queries at each step. Since saving and restoring environments using persistent datastructures are both constant time, persistent datastructures may even provide better performance characteristics for typeclass resolution than their imperative counterparts. Moreover, they *dramatically* simplify the implementation.

5.4 Finding equivalent subgoals

The table of distinct subgoals forms a key part of our tabled typeclass resolution procedure. However, a simple map datastructure does not suffice, since the operations need to be performed modulo α-equivalence. The standard approach for implementing the table in tabled resolution is to use a discrimination tree, as in [39]. Although this approach would work for us as well, our implementation simply α-normalizes the subgoal and then uses a regular hash map on the normalized result. Specifically, before performing any map operation, we traverse the type of the subgoal and replace all unassigned unification variables with constants with a reserved prefix (say α) and ascending integer suffixes. For example, `f ?X (g ?Y ?X)` and `f ?Y (g ?Z ?Y)` would both be normalized to `f` θ_0 `(g` θ_1 θ_0`)`, and so would map to the same subgoal in our table.

There are pros and cons to the two approaches. The discrimination tree approach has an advantage if subgoals tend to have many variables in them, since looking up an existing subgoal in the table modulo α-equivalence can be done without any allocations. On the other hand, the approach we take has an advantage if subgoals tend to have few variables in them, because each subterm can store a single bit indicating the presence of a unification variable, and α-normalization can short-circuit on all ground subterms. In contrast, inserting into a discrimination tree will always require a linear traversal over the subgoal. Notably, our approach also avoids the quadratic blowup usually associated with tabled resolution. The classic example from Prolog is the ternary predicate `Append` that computes the concatenation of its first two arguments and stores it in its third argument. Our procedure can resolve such queries in (quasi-)linear time for two reasons. In the `Append` example, the third argument is always a variable, while the first two are large but variable-free, and so our short-circuiting α-normalization takes constant time.

Lastly, we note that while in traditional logic programming, the only relevant form of equivalence is α-equivalence, there are many other forms of equivalence in more expressive logics such as intensional type theory (ITT), which forms the basis of both Lean and Coq. Although our table only knows about α-equivalence, it could be extended in principle to support other forms of equivalence as well.

5.5 Additional considerations

Indexing the instances. As mentioned in §5.4, whereas in traditional logic programming the only relevant form of equivalence is α-equivalence, there are many other forms of equivalence (also called *definitional equality*) in more expressive logics such as intensional type theory (ITT). Efficient indexing of terms modulo definitional equality is critical to achieving good performance in our typeclass resolution procedure. Our approach is similar to the one used in Coq: store the instances in a discrimination tree, and expose user-facing options that affect which terms are treated as rigid (at the expense of returning under-approximations) and which are treated as wildcards (at the expense of returning over-approximations).

Nested typeclass resolution. Many typeclass resolution queries in Mathlib require solving nested typeclass resolution problems during unification. For example, consider the following toy Lean snippet:

```
class Bottom (α : Type) : Type := (u:Unit:=())
class Left (α : Type) : Type := (u:Unit:=())
class Right (α : Type) : Type := (u:Unit:=())
class Foo (α : Type) [Bottom α] : Type := (u:Unit:=())
instance LeftToBot {α : Type} [Left α] : Bottom α := {u:=()}.
instance RightToBot {α : Type} [Right α] : Bottom α := {u:=()}.
instance RightToLeft {α : Type} [Right α] : Left α := {u:=()}.
instance LeftToFoo {α : Type} [Left α] : Foo α := {u:=()}.
#synth (α : Type) [right : Right α], Foo α
```

When showing implicit arguments, the goal `Foo α` is `@Foo α (@RightToBot α right)`, and the conclusion of `LeftToFoo` is `@Foo ?α (@LeftToBot ?α ?left)`. Resolving the former with the latter produces the unification subproblem `@RightToBot α right =?= @LeftToBot ?α ?left`, which requires synthesizing a term of type `?left : Left α` in order to solve. Triggering typeclass resolution inside the unifier to solve goals of this form can be seen as the analogue of unification hints for Canonical Structures [32]. Lean3 supported this feature as well, and Mathlib relies on it extensively. To the best of our knowledge, Coq does not yet support this feature.

Scheduling strategies. The pseudocode we presented in §5.1 uses a very simple scheduling strategy, and we have found this simple approach to work well for us empirically. However, many other scheduling policies have been proposed in the tabled resolution literature (see *e.g.* [45]) all of which could be applied in our regime as well.

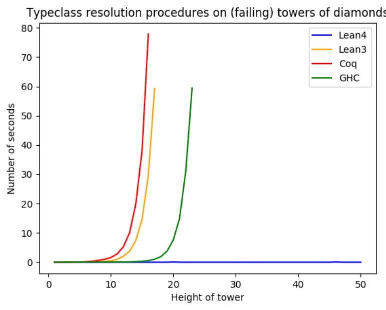
Incremental garbage collection. The nonlinear control flow of tabled resolution makes incremental garbage collection trickier than in SLD resolution. In particular, when a generator node is popped from the generator stack, it does not mean that all solutions to its subgoal have been discovered, since there may still be consumer nodes from the same search tree that are suspended on other not-yet-exhausted subgoals. An efficient way of detecting exhausted subgoals is presented in [39], and the same approach would work in our setting as well. However, we have found our memory usage to be negligible even on sophisticated queries, and so do not consider incremental garbage collection to be worth the cost in our regime.

6 Experiments

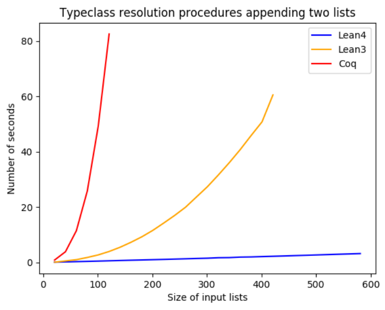
We first evaluate several typeclass resolution procedures on the quintessential (failing) tower of diamonds:

```
instance BtL (α : Type) (n : ℕ) [B α n] : L α n
instance BtR (α : Type) (n : ℕ) [B α n] : R α n
instance LtT (α : Type) (n : ℕ) [L α n] : T α n
instance RtT (α : Type) (n : ℕ) [R α n] : T α n
instance TtB (α : Type) (n : ℕ) [T α n] : B α (n+1)
#synth T Unit n -- (for some n)
```

where `B`, `L`, `R`, `T` stand for “bottom”, “left”, “right”, and “top” respectively. Figure 4a shows the performance of the various systems on this problem as a function of the `n` in the query `T Unit n`. We see that our new tabled resolution procedure (Lean4) is indeed exponentially faster than existing procedures.



(a)



(b)

Fig. 4: Comparing the performance of typeclass resolution procedures on failing towers of diamonds (4a) and appending two lists (4b). In (4a), we see that tabled typeclass resolution—in particular, our implementation of it in Lean4—scales linearly, while Coq, Lean3, and GHC all scale exponentially. In (4b), we see that our indexing approach described in §5.4 indeed avoids the quadratic blowup that tabled resolution procedures have traditionally suffered on this example.

Lastly, we confirm empirically that our approach to subgoal indexing discussed in §5.4 indeed avoids the quadratic blowup associated with tabling on the classic `Append` example. Figure 4b shows the results. Moreover, we find that even though neither Lean3 nor Coq uses tabling, neither scales linearly on this example; indeed they both seem to scale exponentially.[6]

7 Conclusion

The increasingly sophisticated uses of typeclasses within proof assistants has exposed two critical problems with existing typeclass resolution procedures: the

[6] We suspect that the exponential running times on this example for both Lean3 and Coq are not due to their typeclass resolution algorithms, but rather are due to other parts of the respective systems that are used by typeclass resolution.

diamond problem, which causes exponential running times in both theory and practice, and the *cycle problem*, which causes loops in the presence of cycles and so thwarts many desired uses of typeclasses. We have presented a new typeclass resolution procedure, called *tabled typeclass resolution*, that solves these problems. We have implemented our procedure for the upcoming version (v4) of the Lean Theorem Prover, and we have confirmed empirically that our implementation is exponentially faster than existing systems in the presence of diamonds. Our procedure is sufficiently lightweight that it could easily be implemented in other systems. We hope our new procedure facilitates even more sophisticated uses of typeclasses in both software development and interactive theorem proving.

Acknowledgments. We thank the Mathlib community for extensively stress-testing Lean3 and in so doing, exposing the problems that we have addressed in this paper. We give special thanks to Kevin Buzzard, Floris van Doorn, Johan Commelin, Patrick Massot, Mario Carneiro, Sébastien Gouëzel, Chris Hughes, Yury Kudryashov, Scott Morrison, Johannes Hölzl, Oliver Nash, Jason Rute, Reid Barton, and Jeremy Avigad for helping us to understand the problematic idioms and to isolate representative examples. We also thank Johan Commelin and Marc Huisinga for helpful feedback on early drafts.

References

1. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer (2018)
2. Chen, K., Hudak, P., Odersky, M.: Parametric type classes. In: ACM SIGPLAN Lisp Pointers. pp. 170–181. No. 1, ACM (1992)
3. Chen, W., Warren, D.S.: Tabled evaluation with delaying for general logic programs. Journal of the ACM (JACM) **43**(1), 20–74 (1996)
4. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. Acm sigplan notices **46**(4), 53–64 (2011)
5. Coq Development Team: The Coq reference manual: Release 8.9.1. INRIA (2019)
6. Curry, H.B.: Functionality in combinatory logic. Proceedings of the National Academy of Sciences of the United States of America **20**(11), 584 (1934)
7. De La Clergerie, E.V., Lang, B.: Lpda: Another look at tabulation in logic programming. In: ICLP. pp. 470–486 (1994)
8. Demoen, B., Sagonas, K.: Cat: The copying approach to tabling. In: Principles of Declarative Programming, pp. 21–35. Springer (1998)
9. Demoen, B., Sagonas, K.: Chat: The copy-hybrid approach to tabling. In: International Symposium on Practical Aspects of Declarative Languages. pp. 106–121. Springer (1999)
10. Duggan, D., Ophel, J.: Type-checking multi-parameter type classes. Journal of functional programming **12**(2), 133–158 (2002)
11. Gill, A.: Debugging haskell by observing intermediate data structures. Electr. Notes Theor. Comput. Sci. **41**(1), 1 (2000)
12. Guo, H.F., Gupta, G.: A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In: International Conference on Logic Programming. pp. 181–196. Springer (2001)

13. Hales, T.: A review of the Lean Theorem Prover. `https://jiggerwit.wordpress.com/2018/09/18/a-review-of-the-lean-theorem-prover/`, accessed: 2019-10-04
14. Hall, C.V., Hammond, K., Peyton Jones, S.L., Wadler, P.L.: Type classes in haskell. ACM Transactions on Programming Languages and Systems (TOPLAS) **18**(2), 109–138 (1996)
15. Hinze, R., Jones, S.P.: Derivable type classes. Electronic notes in theoretical computer science **41**(1), 5–35 (2001)
16. Hölzl, J., Immler, F., Huffman, B.: Type classes and filters for mathematical analysis in isabelle/hol. In: International Conference on Interactive Theorem Proving. pp. 279–294. Springer (2013)
17. Howard, W.A.: The formulae-as-types notion of construction. To HB Curry: essays on combinatory logic, lambda calculus and formalism **44**, 479–490 (1980)
18. Jones, M.P.: Computing with lattices: An application of type classes. Journal of Functional Programming **2**(4), 475–503 (1992)
19. Jones, M.P.: Type classes with functional dependencies. In: European Symposium on Programming. pp. 230–244. Springer (2000)
20. Jones, S.P.: Haskell 98 language and libraries: the revised report. Cambridge University Press (2003)
21. Jones, S.P., Jones, M., Meijer, E.: Type classes: an exploration of the design space. In: Haskell workshop. pp. 1–16 (1997)
22. Kahl, W., Scheffczyk, J.: Named instances for haskell type classes. In: Proceedings of the 2001 Haskell Workshop, number UU-CS-2001-23 in Tech. Rep. pp. 71–99. Citeseer (2001)
23. Kaplan, H.: Persistent data structures. In: Handbook of Data Structures and Applications, pp. 511–527. Chapman and Hall/CRC (2018)
24. Keller, G., Chakravarty, M.M., Leshchinskiy, R., Peyton Jones, S., Lippmeier, B.: Regular, shape-polymorphic, parallel arrays in haskell. In: ACM Sigplan Notices. vol. 45, pp. 261–272. ACM (2010)
25. Kiselyov, O., Lämmel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell. pp. 96–107. ACM (2004)
26. Kowalski, R.: Predicate logic as programming language. In: IFIP congress. vol. 74, pp. 569–544 (1974)
27. Krebbers, R., Spitters, B.: Type classes for efficient exact real arithmetic in coq. arXiv preprint arXiv:1106.3448 (2011)
28. Lämmel, R., Jones, S.P.: Scrap your boilerplate with class: extensible generic functions. ACM SIGPLAN Notices **40**(9), 204–215 (2005)
29. Lämmel, R., Ostermann, K.: Software extension and integration with type classes. In: Proceedings of the 5th international conference on Generative programming and component engineering. pp. 161–170. ACM (2006)
30. Läufer, K.: Type classes with existential types. Journal of Functional Programming **6**(3), 485–518 (1996)
31. Li, P., Zdancewic, S.: Encoding information flow in haskell. In: 19th IEEE Computer Security Foundations Workshop (CSFW'06). pp. 12–pp. IEEE (2006)
32. Mahboubi, A., Tassi, E.: Canonical structures for the working coq user. In: International Conference on Interactive Theorem Proving. pp. 19–34. Springer (2013)
33. McBride, C.: Faking it simulating dependent types in haskell. Journal of functional programming **12**(4-5), 375–392 (2002)
34. de Moura, L., Kong, S., Avigad, J., Van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: International Conference on Automated Deduction. pp. 378–388. Springer (2015)

35. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
36. Paulson, L.C.: Organizing numerical theories using axiomatic type classes. Journal of Automated Reasoning **33**(1), 29–49 (2004)
37. Pucella, R., Tov, J.A.: Haskell session types with (almost) no class. In: Haskell. vol. 8, pp. 25–36 (2008)
38. Robinson, J.A., et al.: A machine-oriented logic based on the resolution principle. Journal of the ACM **12**(1), 23–41 (1965)
39. Sagonas, K., Swift, T.: An abstract machine for tabled execution of fixed-order stratified logic programs. ACM Transactions on Programming Languages and Systems (TOPLAS) **20**(3), 586–634 (1998)
40. Shen, Y.D., Yuan, L.Y., You, J.H., Zhou, N.F.: Linear tabulated resolution based on prolog control strategy. Theory and Practice of Logic Programming **1**(1), 71–103 (2001)
41. Sozeau, M., Oury, N.: First-class type classes. In: International Conference on Theorem Proving in Higher Order Logics. pp. 278–293. Springer (2008)
42. Spitters, B., van der Weegen, E.: Developing the algebraic hierarchy with type classes in coq. In: International Conference on Interactive Theorem Proving. pp. 490–493. Springer (2010)
43. Spitters, B., Van der Weegen, E.: Type classes for mathematics in type theory. Mathematical Structures in Computer Science **21**(4), 795–825 (2011)
44. Strachey, C.: Fundamental concepts in programming languages. Higher-order and symbolic computation **13**(1-2), 11–49 (2000)
45. Swift, T., Warren, D.S.: Xsb: Extending prolog with tabled logic programming. Theory and Practice of Logic Programming **12**(1-2), 157–187 (2012)
46. Tamaki, H., Sato, T.: Old resolution with tabulation. In: International Conference on Logic Programming. pp. 84–98. Springer (1986)
47. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM (JACM) **38**(3), 619–649 (1991)
48. Vielle, L.: Recursive query processing: The power of logic. Theoretical computer science **69**(1), 1–53 (1989)
49. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 60–76. ACM (1989)
50. Zhou, N.F., Sato, T., Shen, Y.D.: Linear tabling strategies and optimizations. Theory and Practice of Logic programming **8**(1), 81–109 (2008)