Type-Level Computations for Ruby Libraries

Milod Kazerounian
University of Maryland
College Park, Maryland, USA
milod@cs.umd.edu

Sankha Narayan Guria
University of Maryland
College Park, Maryland, USA
sankha@cs.umd.edu

Niki Vazou
IMDEA Software Institute
Madrid, Spain
niki.vazou@imdea.org

Jeffrey S. Foster
Tufts University
Medford, Massachusetts, USA
jfoster@cs.tufts.edu

David Van Horn
University of Maryland
College Park, Maryland, USA
dvanhorn@cs.umd.edu

Abstract

Many researchers have explored ways to bring static typing to dynamic languages. However, to date, such systems are not precise enough when types depend on values, which often arises when using certain Ruby libraries. For example, the type safety of a database query in Ruby on Rails depends on the table and column names used in the query. To address this issue, we introduce CompRDL, a type system for Ruby that allows library method type signatures to include *type-level computations* (or *comp types* for short). Combined with singleton types for table and column names, comp types let us give database query methods type signatures that compute a table's schema to yield very precise type information. Comp types for hash, array, and string libraries can also increase precision and thereby reduce the need for type casts. We formalize CompRDL and prove its type system sound. Rather than type check the bodies of library methods with comp types—those methods may include native code or be complex—CompRDL inserts run-time checks to ensure library methods abide by their computed types. We evaluated CompRDL by writing annotations with type-level computations for several Ruby core libraries and database query APIs. We then used those annotations to type check two popular Ruby libraries and four Ruby on Rails web apps. We found the annotations were relatively compact and could successfully type check 132 methods across our subject programs. Moreover, the use of type-level computations allowed us to check more expressive properties, with fewer manually inserted casts, than was possible without type-level computations. In the process, we found two type errors and a documentation error that were confirmed by the developers. Thus, we believe CompRDL is an important step forward in bringing precise static type checking to dynamic languages.

CCS Concepts • **Theory of computation** → *Program analysis.*

Keywords type-level computations, dynamic languages, types, Ruby, libraries, database queries

ACM Reference Format:
Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S. Foster, and David Van Horn. 2019. Type-Level Computations for Ruby Libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3314221.3314630

1 Introduction

There is a large body of research on adding static typing to dynamic languages [2–4, 21, 29, 37, 38, 43–45]. However, existing systems have limited support for the case when *types* depend on *values*. Yet this case occurs surprisingly often, especially in Ruby libraries. For example, consider the following database query, written for a hypothetical Ruby on Rails (a web framework, called Rails henceforth) app:

```
Person.joins(:apartments).where({name: 'Alice', age: 30,
    apartments: {bedrooms: 2}})
```

This query uses the *ActiveRecord* DSL to join two database tables, people[1] and apartments, and then filter on the values of various columns (name, age, bedrooms) in the result.

We would like to type check such code, e.g., to ensure the columns exist and the values being matched are of the right types. But we face an important problem: what type signature do we give joins? Its return type—which should describe the joined table—depends on the value of its argument. Moreover, for n tables, there are n^2 ways to join two of them, n^3 ways to join three of them, etc. Enumerating all these combinations is impractical.

[1]Rails knows the plural of person is people.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314630

To address this problem, in this paper we introduce CompRDL, which extends RDL [18], a Ruby type system, to include method types with *type-level computations*, henceforth referred to as *comp types*. More specifically, in CompRDL we can annotate library methods with type signatures in which Ruby expressions can appear as types. During type checking, those expressions are evaluated to produce the actual type signature, and then typing proceeds as usual. For example, for the call to Person.joins, by using a singleton type for :apartments, a type-level computation can look up the database schemas for the receiver and argument and then construct an appropriate return type.[2]

Moreover, the same type signature can work for any combination of tables. And, because CompRDL allows arbitrary computation in types, CompRDL type signatures have access to the full, highly dynamic Ruby environment. This allows us to provide very precise types for the large set of Rails database query methods. It also lets us give precise types to methods of *finite hash types* (heterogeneous hashes), *tuple types* (heterogeneous arrays), and *const string types* (immutable strings), which can help eliminate type casts that would otherwise be required.

Note that in all these cases, we apply comp types to library methods whose bodies we do not type check, in part to avoid complex, potentially undecidable reasoning about whether a method body matches a comp type, but more practically because those library methods are either implemented in native code (hashes, arrays, strings) or are complex (database queries). This design choice makes CompRDL a particularly practical system which we can apply to real-world programs. To maintain soundness, we insert dynamic checks to ensure that these methods abide by their computed types at runtime. (§ 2 gives an overview of typing in CompRDL.)

We introduce λ^C, a core, object-oriented language that formalizes CompRDL type checking. In λ^C, library methods can be declared with signatures of the form $(a{<:}e_1/A_1) \to e_2/A_2$, where A_1 and A_2 are the conventional (likely overapproximate) argument and return types of the method. The precise argument and return types are determined by evaluating e_1 and e_2, respectively, and that evaluation may refer to the type of the receiver and the type a of the argument. λ^C also performs type checking on e_1 and e_2, to ensure they do not go wrong. To avoid potential infinite recursion, λ^C does not use type-level computations during this type checking process, instead using the conventional types for library methods. Finally, λ^C includes a rewriting step to insert dynamic checks to ensure library methods abide by their computed types. We prove λ^C's type system is sound. (See § 3 for our formalism.)

We implemented CompRDL on top of RDL, an existing Ruby type checker. Since CompRDL can include type-level computation that relies on mutable values, CompRDL inserts additional runtime checks to ensure such computations evaluate to the same result at method call time as they did at type checking time. Additionally, CompRDL uses a lightweight analysis to check that type-level computations (and thus type checking) terminate. The termination analysis uses purity effects to check that calls that invoke iterator methods—the main source of looping in Ruby, in our experience—do not mutate the receiver, which could introduce non-termination. Finally, we found that several kinds of comp types we developed needed to include weak type updates to handle mutation in Ruby programs. (§ 4 describes our implementation in more detail.)

We evaluated CompRDL by first using it to write type annotations for 482 Ruby core library methods and 104 Rails database query methods. We found that by using helper methods, we could write very precise type annotations for all 586 methods with just a few lines of code on average. Then, we used those annotations to type check 132 methods across two Ruby APIs and four Ruby on Rails web apps. We were able to successfully type check all these methods in approximately 15 seconds total. In doing so, we also found two type errors and a documentation error, which we confirmed with the developers. We also found that, with comp types, type checking these benchmarks required 4.75× fewer type cast annotations compared to standard types, demonstrating comp types' increased precision. (§ 5 contains the results of our evaluation.)

Our results suggest that using type-level computations provides a powerful, practical, and precise way to statically type check code written in dynamic languages.

2 Overview

The starting point for our work is RDL [18], a system for adding type checking and contracts to Ruby programs. RDL's type system is notable because type checking statically analyzes source code, but it does so *at runtime*. For example, line 6 in Figure 1a gives a type signature for the method defined on the subsequent line. This "annotation" is actually a call to the method type,[3] which stores the type signature in a global table. The type annotation includes a label :model. (In Ruby, strings prefixed by colon are *symbols*, which are interned strings.) When the program subsequently calls RDL.do_typecheck :model (not shown), RDL will type check the source code of all methods whose type annotations are labeled :model.

This design enables RDL to support the metaprogramming that is common in Ruby and ubiquitous in Rails. For example, the programmer can perform type checking after metaprogramming code has run, when corresponding type definitions are available. See Ren and Foster [37] for more details. We note that while CompRDL benefits from this runtime

[2] The use of type-level computations and singleton types could be considered dependent typing, but as our type system is much more restricted we introduce new terminology to avoid confusion (see § 2.4 for discussion).

[3] In Ruby, parentheses in a method call are optional.

type checking approach—we use RDL's representation of types in our CompRDL signatures, and our subject programs include Rails apps—there is nothing specific in the design of comp types that relies on it, and one could implement comp types in a fully static system.

2.1 Typing Ruby Database Queries

While RDL's type system is powerful enough to type check Rails apps in general, it is actually very imprecise when reasoning about database (DB) queries. For example, consider the code in Figure 1a, which has been extracted and simplified from the *Discourse* app used in our experiments (§ 5). Among others, this app uses two tables, users and emails, whose schemas are shown on lines 2 and 3. Each user has an id, a username, and a flag indicating whether the account was *staged*. Such staged accounts were created automatically by *Discourse* and can be claimed by the email address owner. An email has an id, the email address, and the user_id of the user who owns the email address.

Next in the figure, we show code for the class User, which is a *model*, i.e., instances of the class correspond to rows in the users table. This class has one method, available?, which returns a boolean indicating whether the username and email address passed as arguments are available. The method first checks whether the username was already reserved (line 8, note the postfix if). If not, it uses the database query method exists? to see if the username was already taken (line 9). (Note that in Ruby, {a: b} is a hash that maps the symbol :a, which is suffixed with a colon when used as a key, to the value b.) Otherwise, line 11 uses a more complex query to check whether an account was staged. More specifically, this code joins the users and emails table and then looks for a match across the joined tables.

We would like to type check the exists? calls in this code to ensure they are type correct, meaning that the columns they refer to exist and the values being matched are of the right type. The call on line 9 is easy to check, as RDL can type the receiver User as having an exists? method that takes a particular *finite hash type* {c1: t1, ..., cn: tn} as an argument, where the ci are *singleton types* for symbols naming the columns, and the ti are the corresponding column types.

Unfortunately, the exists? call on line 11 is another story. Notice that this query calls exists? on the result of User.joins(:emails). Thus, to give exists? a type with the right column information, we need to have that information reflected in the return type of joins. Unfortunately, there is no reasonable way to do this in RDL, because the set of columns in the table returned by joins depends on both the receiver and the *value* of the argument. We could in theory overload joins with different return types depending on the argument type—e.g., we could say that User.joins returns a certain type when the argument has singleton type :emails. However, we would need to generate such signatures for every possible way of joining two tables together, three tables together, etc.,

```
1  # Table Schema
2  # users: { id: Integer, username: String, staged: bool }
3  # emails: { id: Integer, email: String, user_id: Integer }
4
5  class User < ActiveRecord::Base
6    type "(String, String) → %bool", typecheck: :model
7    def self.available?(name, email)
8      return false if reserved?(name)
9      return true if !User.exists?({username: name})
10     # staged user accounts can be claimed
11     return User.joins(:emails).exists?({staged: true,
           username: name, emails: {email: email}})
12   end
13 end
```

(a) *Discourse* code (uses *ActiveRecord*).

```
1  type Table, :exists?, "(«schema_type(tself)») → Bool"
2  type Table, :joins, "(t<:Symbol) →
3      «if t.is_a?(Singleton)
4      then Generic.new(Table, schema_type(tself).merge(
           {t.val ⇒schema_type(t)}))
5      else Nominal.new(Table)
6      end »"
7
8  def schema_type(t)
9    if t.is_a?(Generic) ∧ (t.base == Table) #Table<T>
10     return t.param # return T
11   elsif t.is_a?(Singleton) #Type of class or :symbol
12     table_name = t.val # get the class/symbol vale
13     table_type = RDL.db_schema[table_name]
14     return table_type.param
15   else # only reached for the nominal type Table
16     return ... # returns Hash<Symbol, Object>
17   end
18 end
```

(b) Comp type annotations for query methods.

Figure 1. Type Checking Database Queries in *Discourse*.

which quickly blows up. Thus, currently, RDL types this particular exists? call as taking a Hash<Symbol, Object>, which would allow type-incorrect arguments.

Comp types for DB Queries. To address this problem, CompRDL allows method type signatures to include computations that can, on-the-fly, determine the method's type. Figure 1b gives comp type signatures for exists? and joins. It also shows the definition of a helper method, schema_type, that is called from the comp types. The comp types also make use of a new generic type Table<T> to type a DB table whose columns are described by T, which should be a finite hash type.

Line 1 gives the type of exists?. Its argument is a comp type, which is a Ruby expression, delimited by «·», that evaluates to a standard type. When type checking a call to

exists? (including those in the body of available?), CompRDL runs the comp type code to yield a standard type, and then proceeds with type checking as usual with that type.

In this case, to compute the argument type for exists?, we call the helper method schema_type with `tself`, which is a reserved variable naming the type of the receiver. The schema_type method has a few different behaviors depending on its argument. When given a type Table<T>, it returns T, i.e., the finite hash type describing the columns. When given a singleton type representing a class or a symbol, it uses another helper method RDL.db_schema (not shown) to look up the corresponding table's schema and return an appropriate finite hash type. Given any other type, schema_type falls back to returning the type Hash<Symbol, Object>.

This type signature already allows us to type check the exists? call on line 9. On this line, the receiver has the singleton type for the User class, so schema_type will use the second arm of the conditional and look up the schema for User in the DB.

Line 2 shows the comp type signature for joins. The signature's input type binds t to the actual argument type, and requires it to be a subtype of Symbol. For example, for the call on line 11, t will be bound to the singleton type for :emails. The return comp type can then refer to t. Here, if t is a singleton type, joins returns a new Table type that merges the schemas of the receiver and the argument tables using schema_type. Otherwise, it falls back to producing a Table with no schema information. Thus, the joins call on line 11 returns type

```
Table<{staged:%bool, username:String, id: Integer,
       emails: {email:String, user_id: Integer }}>
```

That is, the type reflects the schemas of both the users and emails tables. Given this type, we can now type check the exists? call on line 11 precisely. On this line, the receiver has the table type given above, so when called by exists? the helper schema_type will use the first arm of the conditional and return the Table column types, ensuring the query is type checked precisely.

Though we have only shown types for two query methods in the figure, we note that comp types are easily extensible to other kinds of queries. Indeed, we have applied them to 104 methods across two DB query frameworks (§ 5). Furthermore, we can also use comp types to encode sophisticated invariants. For example, in Rails, database tables can only be joined if the corresponding classes have a declared *association*. We can write a comp type for joins that enforces this. (We omitted this in Figure 1 for brevity.)

Finally, we note that while we include a "fallback" case that allows comp types to default to less precise types when necessary, in practice this is rarely necessary for DB queries. That is, parameters that are important for type checking, such as the name of tables being queried or joined, or the names of columns be queried, are almost always provided statically in the code.

```
1  type Hash, :[], "(k) → v"
2  type Array, :first, "() → a"
3  type :page, "() →{ info: Array<String>, title: String }"
4
5  type "() → String"
6  def image_url()
7      page[:info].first    # can't type check
8  # Fix: RDL.type_cast( page[:info], "Array< String >") .first
9   end
```

Figure 2. Type Casts in a Method.

2.2 Avoiding Casts using Comp Types

In addition to letting us find type errors in code we could not previously type check precisely enough, the increased precision of comp types can also help eliminate type casts.

For example, consider the code in Figure 2. The first line gives the type signature for a method of Hash, which is parameterized by a key type k and a value type v (declarations of the parameters not shown). The specific method is Hash#[],[4] which, given a key, returns the corresponding value. Notably, the form x[k] is desugared to x.[] (k), and thus hash lookup, array index, and so forth are methods rather than built-in language constructs.

The second line similarly gives a type for Array#first, which returns the first element of the array. Here type variable a is the array's contents type (declaration also not shown). The third line gives a type for a method page of the current class, which takes no arguments and returns a hash in which :info is mapped to an Array<String> and :title is mapped to a String.

Now consider type checking the image_url method defined at the bottom of the figure. This code is extracted and simplified from a *Wikipedia* client library used in our experiments (§ 5). Here, since page is a no-argument method, it can be invoked without any parentheses. We then invoke Hash#[] on the result.

Unfortunately, at this point type checking loses precision. The problem is that whenever a method is invoked on a finite hash type {c1: t1, ..., cn: tn}, RDL (retroactively) gives up tracking the type precisely and promotes it to Hash<Symbol, t1 or...or tn> [18]. In this case, page's return type is promoted to Hash<Symbol, Array<String> or String>.

Now the type checker gets stuck. It reasons that first could be invoked on an array or a string, but first is defined only for the former and not the latter. The only currently available fix is to insert a type cast, as shown in the comment on line 8.

[4]Here we use the Ruby idiom that A#m refers to the instance method m of class A.

One possible solution would be to add special-case support for [] on finite hash types. However, this is only one of 54 methods of Hash, which is a lot of behavior to special-case. Moreover, Ruby programs can *monkey patch* any class, including Hash, to change library methods' behaviors. This makes building special support for those methods inelegant and potentially brittle since the programmer would have no way to adjust the typing of those methods.

In CompRDL, we can solve this problem with a comp type annotation. More specifically, we can give Hash#[] the following type:

```
type Hash, :[], "(t<:Object) →
  «if tself.is_a? (FiniteHash) ∧ t.is_a? (Singleton)
      then tself.elts[t.val]
      else tself.value_type end»"
```

This comp type specifies that if the receiver has a finite hash type and the key has a singleton type, then Hash#[] returns the type of the value mapped to by the key, otherwise it returns a value type covering all possible values (computed by value_type, definition not shown).

Notice that this signature allows image_url to type check without any additional casts. The same idea can be applied to many other Hash methods to give them more precise types.

Tuple Types. In addition to finite hash types, RDL has a special *tuple type* to model heterogeneous Arrays. As with finite hash types, RDL does not special-case the Array methods for tuples, since there are 124 of them. This leads to a loss of precision when invoking methods on values with tuple types. However, analogously to finite hash tables, comp types can be used to recover precision. As examples, the Array#first method can be given a comp type which returns the type of the first element of a tuple, and the comp type for Array#[] has essentially the same logic as Hash#[].

Const String Types. As another example, Ruby strings are mutable, hence RDL does not give them singleton types. (In contrast, Ruby symbols are immutable.) This is problematic, because types might depend on string values. In particular, in the next section we explore reasoning about string values during type checking raw SQL queries.

In CompRDL, we assign singleton types to strings wherever possible. We introduce a new *const string* type representing strings that are never mutated. CompRDL treats const strings as singletons, and String methods are given comp types that compute precise types using const strings and fall back to the String type as needed. We discuss handling mutation for const strings, finite hashes, and tuples in Section 4.

2.3 SQL Type Checking

As we saw in Figure 1, *ActiveRecord* uses a DSL that makes it easier to construct queries inside of Ruby. However, sometimes programmers need to include raw SQL in their queries, either to access a feature not supported by the DSL or to improve performance compared to the DSL-generated query.

```
1  # Table Schema
2  # posts table { id: Integer, topic_id: Integer, ... }
3  # topics table { id: Integer, title: String, ... }
4  # topic_allowed_groups table { group_id: Integer,
       topic_id: Integer }
5
6  # Query with SQL strings
7  Post.joins(:topic).where('topics.title IN (SELECT
       topic_id FROM topic_allowed_groups WHERE
       group_id = ?)', self.id)
8
9  type Table, :where, "(t <: «if t.is_a? (ConstString)
10   then sql_typecheck(tself, t)
11   else schema_type(tself)
12   end») → «tself»"
```

Figure 3. Type Checking SQL Strings in *Discourse*.

Figure 3 gives one such example, extracted and simplified from *Discourse*. There are three relevant tables: posts, which stores posted messages; topics, which stores the topics of posts; and topic_allowed_groups, which is used to limit the topics allowed by certain user groups.

Line 7 shows a query that includes raw SQL. First, we join the posts and topics tables. Then where filters the joined table based on some conditions. Here, the conditions involve a nested SQL query, which can only be expressed using raw SQL that will be inserted into the final generated query. This example also shows another feature: any ?'s that appear in raw SQL are replaced by additional arguments to where. In this case, the ? will be replaced by self.id.

We would like to extend type checking to the raw SQL strings in queries, since they may have errors. In this particular example, we have injected a bug. The inner SELECT returns a set of integers, but topics.title is a string, and it is a type error to search for a string in an integer set.

To find this bug, we developed a simple type checker for a subset of SQL, and we wrote a comp type for where that invokes it as shown on line 9. In particular, if the type of the argument to where, here referred to by t, is a const string, then we type check that string as raw SQL, and otherwise we compute the valid parameters of where using the schema_type method from Figure 1. The result of where has the same type as the receiver.

The sql_typecheck method (not shown) takes the receiver type, which will be a Table<T> type, and the SQL string. One challenge that arises in type checking the SQL string is that it is actually only a fragment of a query, which therefore cannot be directly parsed using a standard SQL parser. We solve this problem by creating a complete, but artificial, SQL query into which we inject the fragment. This query is never

run, but it is syntactically correct so it can be parsed. Then, we replace any ?'s with placeholder AST nodes that store the types of the corresponding arguments.

For example, the raw SQL in Figure 3 gets translated to the following SQL query:

```
SELECT * FROM posts INNER JOIN topics
  ON a.id = b.a_id
  WHERE topics.title IN (SELECT topic_id FROM
      topic_allowed_groups WHERE group_id = [Integer])
```

Notice the table names (posts, topics) occur on the first line and the ? has been replaced by a placeholder indicating the type Integer of the argument. Also note that the column names to join on (which are arbitrary here) are ignored by our type checker, which currently only looks for errors in the where clause.

Once we have a query that can be parsed, we can type check it using the DB schema. In this case, the type mismatch between topics.title and the inner query will be reported.

In § 2.1, comp types were evaluated to produce a normal type signature. However, we use comp types in a slightly different way for checking SQL strings. The sql_typecheck method will itself perform type checking and provide a detailed message when an error is found. If no error is found, sql_typecheck will simply return the type String, allowing type checking to proceed.

2.4 Discussion

Now that we have seen CompRDL in some detail, we can discuss several parts of its design.

Dynamic Checks. In type systems with type-level computations, or more generally dependent type systems, comparing two types for equality is often undecidable, since it requires checking if computations are equivalent.

To avoid this problem, CompRDL only uses comp types for methods which themselves are not type checked. For example, Hash#[] is implemented in native code, and we have not attempted to type check *ActiveRecord*'s joins method, which is part of a very complex system.

As a result, type checking in CompRDL is decidable. Comp types are only used to type check method calls, meaning we will always have access to the types of the receiver and arguments in a method call. Additionally, in all cases we have encountered in practice, the types of the receiver and arguments are ground types (meaning they do not contain type variables). Thus, comp types can be fully evaluated to non-comp types before proceeding to type checking.

For soundness, since we do not type check the bodies of comp type-annotated methods, CompRDL inserts dynamic checks at calls to such methods to ensure they match their computed types. For example, in Figure 2, CompRDL inserts a check that page[:info] returns an Array<String>. This follows the approach of gradual [40] and hybrid [17] typing, in which dynamic checks guard statically unchecked code.

We should also note that although our focus is on applying comp types to libraries, they can be applied to any method at the cost of dynamic checks for that method rather than static checks. For example, they could be applied to a user-defined library wrapper.

Termination. A second issue for the decidability of comp types is that type-level computations could potentially not terminate. To avoid this possibility, we implement a termination checker for comp types. At a high level, CompRDL ensures termination by checking that iterators used by type-level code do not mutate their receivers and by forbidding type-level code from using looping constructs. We also assume there are no recursive method calls in type-level code. We discuss termination checking in more detail in § 4.

Value Dependency. We note that, unlike dependent types (e.g., Coq [35], Agda [33], F* [42]) where types depend directly on terms, in CompRDL types depend on the *types* of terms. For instance, in a comp type (t<:Object) → tres the result type tres can depend on the type t of the argument. Yet, since singleton types lift expressions into types, we could still use CompRDL to express some value dependencies in types in the style of dependent typing.

Constant Folding. Finally, in RDL, integers and floats have singleton types. Thus, we can use comp types to lift some arithmetic computations to the type level. For example, CompRDL can assign the expression 1+1 the type Singleton(2) instead of Integer. This effectively incorporates constant folding into the type checker.

While we did write such comp types for Integer and Float (see Table 1), we found that this precision was not useful, at least in our subject programs. The reason is that RDL only assigns singleton types to constants, and typically arithmetic methods are not applied to constant values. Thus, though we have written comp types for the Integer and Float libraries, we have yet to find a useful application for them in practice. We leave further exploration of this topic to future work.

3 Soundness of Comp Types

In this section we formalize CompRDL as λ^C, a core object-oriented calculus that includes comp types for library methods. We first define the syntax and semantics of λ^C (§ 3.1), and then we formalize type checking (§ 3.2). The type checking process includes a rewriting step to insert dynamic checks that ensure library methods satisfy their type signatures. Finally, we prove type soundness (§ 3.3). For brevity, we leave the full formalism and proofs to a companion technical report [25]. Here we provide only the key details.

3.1 Syntax and Semantics

Figure 4 gives the syntax of λ^C. *Values* v include `nil`, `true`, and `false`. To support comp types, class IDs A, which are the base types in λ^C, are also values. We assume the set of

Values	v	::=	`nil` | `true` | `false` | A
Expressions	e	::=	v | x | a | `self` | `tself`
		|	A.**new** | e;e | e == e
		|	**if** e **then** e **else** e | $e.m(e)$
		|	$\lceil A \rceil e.m(e)$
Meth. Types	σ	::=	$A \to A$
Lib. Meth. Types	δ	::=	σ | $(a{<:}e/A) \to e/A$
Programs	P	::=	`def` $A.m(x) : \sigma$ = e
		|	`lib` $A.m(x) : \delta$ | P; P
Type Env.	Γ	::=	$\emptyset$ | $x{:}A$
Dyn. Env.	E	::=	$\emptyset$ | $x{:}v$
Class Table	CT	::=	$\emptyset$ | $A.m{:}\delta, CT$
Method Sets	$\mathcal{U}$	:	user-defined methods
	$\mathcal{L}$	:	library methods

$x, a \in$ var IDs, $m \in$ method IDs, $A \in$ class IDs, $\mathcal{U} \cap \mathcal{L} = \emptyset$

Figure 4. Syntax and Relations of λ^C.

class IDs includes several built-in classes: *Nil*, the class of `nil`; *Obj*, which is the root superclass; *True* and *False*, which are the classes of `true` and `false`, respectively, as well as their superclass *Bool*; and *Type*, the class of base types A.

Expressions e include values v and variables x and a. By convention, we use the former in regular program expressions and the latter in comp types. The special variable `self` names the receiver of a method call, and the special variable `tself` names the *type* of the receiver in a comp type. New object instances are created with A.**new**. Expressions also include sequences e;e, conditionals **if** e **then** e **else** e, and method calls $e.m(e)$, where, to simplify the formalism, methods take one argument. Finally, our type system translates calls to library methods into *checked method calls* $\lceil A \rceil e.m(e)$, which checks at run-time that the value returned from the call has type A. We assume this form does not appear in the surface syntax.

We assume the classes form a lattice with *Nil* as the bottom and *Obj* as the top. We write the least upper bound of A_1 and A_2 as $A_1 \sqcup A_2$. For simplicity, we assume the lattice correctly models the program's classes, i.e., if $A \leq A'$, then A is a subclass of A' by the usual definition. Lastly, three of the built-in classes, *Nil*, *True*, and *False*, are *singleton types*, i.e., they contain only the values `nil`, `true`, and `false`, respectively. Extending λ^C with support for more kinds of singleton types is straightforward.

Method Types σ are of the form $A' \to A$ where A' and A are the domain and range types, respectively. *Library Method Types* δ are either method types or have the form $(a{<:}e'/A') \to e/A$, where e' and e are expressions that evaluate to types and that can refer to the variables a and `tself`. The base types A' and A provide an upper bound on the respective expression types, i.e., for any a, expressions e' and e should evaluate to subtypes of A' and A, respectively. These upper bounds are used for type checking comp types (§ 3.2).

Finally, *programs* are sequences of method definitions and library method declarations.

Dynamic Semantics. The dynamic semantics of λ^C are the small-step semantics of Ren and Foster [37], modified to throw blame (§ 3.3) when a checked method call fails. They use *dynamic environments* E, defined in Figure 4, which map variables to values. We define the relation $\langle E, e \rangle \Downarrow e'$, meaning the expression e evaluates to e' under dynamic environment E. The full evaluation rules use a stack as well [25], but we omit the stack here for simplicity.

Example. As an example comp type in the formalism, consider type checking the expression `true`. $\wedge$ (`true`), where the $\wedge$ method returns the logical conjunction of the receiver and argument. Standard type checking would assign this expression the type *Bool*. However, with comp types we can do better.

Recall that `true` and `false` are members of the singleton types *True* and *False*. Thus, we can write a comp type for the $\wedge$ method that yields a singleton return type when the arguments are singletons, and *Bool* in the fallback case:

```
lib Bool. ∧ (x) : (a<:Bool/Bool) → (
    if (tself == True). ∧ (a == True) then True
    else if (tself == False). ∨ (a == False) then False
    else Bool)/Bool
```

The first two lines of the condition handle the singleton cases, and the last line is the fallback case.

3.2 Type Checking and Rewriting

Figure 5 gives a subset of the rules for type checking λ^C expressions and rewriting them to insert dynamic checks at library calls. The remaining rules are straightforward, and can be found in the technical report [25]. The rules use two additional definitions from Figure 4. *Type environments* Γ map variables to base types, and the *class table* CT maps methods to their type signatures. We omit the construction of class tables, which is standard. We also use disjoint sets $\mathcal{U}$ and $\mathcal{L}$ to refer to the user-defined and library methods, respectively.

The rules in Figure 5 prove judgments of the form $\Gamma \vdash_{CT} e \hookrightarrow e' : A$, meaning under type environment Γ and class table CT, source expression e is rewritten to target expression e', which has type A.

Rule (C-Type) is straightforward: any class ID A that is used as a value is rewritten to itself, and it has type *Type*. We include this rule to emphasize that types are values in λ^C.

Rule (C-AppUD) finds the receiver type A, then looks up $A.m$ in the class table. This rule only applies when $A.m$ is user-defined and thus has a (standard) method type $A_1 \to A_2$. Then, as is standard, the rule checks that the argument type A_x is a subtype of A_1. The type of the whole call is A_2. This rule rewrites the subexpressions e and e_x, but it does not itself

Type Checking and Rewriting Rules $\boxed{\Gamma \vdash_{CT} e \hookrightarrow e : A}$

$$\frac{}{\Gamma \vdash_{CT} A \hookrightarrow A : \mathit{Type}}\ \textsc{C-Type} \qquad \frac{\Gamma \vdash_{CT} e \hookrightarrow e' : A \quad CT(A.m) = A_1 \to A_2 \quad A.m \in \mathcal{U} \quad \Gamma \vdash_{CT} e_x \hookrightarrow e'_x : A_x \quad A_x \leq A_1}{\Gamma \vdash_{CT} e.m(e_x) \hookrightarrow e'.m(e'_x) : A_2}\ \textsc{C-AppUD}$$

$$\frac{\Gamma \vdash_{CT} e \hookrightarrow e' : A \quad CT(A.m) = A_1 \to A_2 \quad A.m \in \mathcal{L} \quad A_x \leq A_1 \quad \Gamma \vdash_{CT} e_x \hookrightarrow e'_x : A_x}{\Gamma \vdash_{CT} e.m(e_x) \hookrightarrow \lceil A_2 \rceil e'.m(e'_x) : A_2}\ \textsc{C-AppLib}$$

$$\frac{\Gamma \vdash_{CT} e \hookrightarrow e' : A \quad CT(A.m) = (a{<:}e_{t1}/A_{t1}) \to e_{t2}/A_{t2} \quad A.m \in \mathcal{L} \quad \Gamma \vdash_{CT} e_x \hookrightarrow e'_x : A_x \quad a{:}\mathit{Type}, \mathtt{tself}{:}\mathit{Type} \vdash_{\lfloor\!\lfloor CT \rfloor\!\rfloor} e_{t1} \hookrightarrow e'_{t1} : \mathit{Type} \quad \langle [a \mapsto A_x][\mathtt{tself} \mapsto A], e'_{t1} \rangle \Downarrow A_1 \quad A_x \leq A_1 \quad a{:}\mathit{Type}, \mathtt{tself}{:}\mathit{Type} \vdash_{\lfloor\!\lfloor CT \rfloor\!\rfloor} e_{t2} \hookrightarrow e'_{t2} : \mathit{Type} \quad \langle [a \mapsto A_x][\mathtt{tself} \mapsto A], e'_{t2} \rangle \Downarrow A_2}{\Gamma \vdash_{CT} e.m(e_x) \hookrightarrow \lceil A_2 \rceil e'.m(e'_x) : A_2}\ \textsc{C-App-Comp}$$

Figure 5. A subset of the type checking and rewriting rules for λ^C.

insert any checks, since user-defined methods are statically checked against their type signatures (rule not shown).

Rule (C-AppLib) is similar to Rule (C-AppUD), except it applies when the callee is a library method. In this case, the rule inserts a check to ensure that, at run-time, the library method abides by its specified type.

Rule (C-App-Comp) is the crux of λ^C's type checking system. It applies at a call to a library method $A.m$ that uses a type-level computation, i.e., with a type signature $(a{<:}e_{t1}/A_{t1}) \to e_{t2}/A_{t2}$. The rule first type checks and rewrites e_{t1} and e_{t2} to ensure they will evaluate to a type (i.e., have type *Type*). These expressions may refer to a and `tself`, which themselves have type *Type*. The rule then *evaluates* the rewritten e_{t1} and e_{t2} using the dynamic semantics mentioned above to yield types A_1 and A_2, respectively. Finally, the rule ensures that the argument e_x has a subtype of A_1; sets the return type of the whole call to A_2; and inserts a dynamic check that ensures the call returns an A_2 at run-time. For instance, the earlier example of the use of logical conjunction would be rewritten to $\lceil \mathit{True} \rceil \mathtt{true}.\wedge(\mathtt{true})$.

There is one additional subtlety in Rule (C-App-Comp). Recall the example above that gives a type to *Bool.*$\wedge$. Notice that the type-level computation itself uses *Bool.*$\wedge$. This could potentially lead to infinite recursion, where calling *Bool.*$\wedge$ requires checking that *Bool.*$\wedge$ produces a type, which requires recursively checking that *Bool.*$\wedge$ produces a type, etc.

To avoid this problem, we introduce a function $\lfloor\!\lfloor CT \rfloor\!\rfloor$ that rewrites class table CT to drop all annotations with type-level expressions. More precisely, any comp type $(a{<:}e_1/A_1) \to e_2/A_2$ is rewritten to $A_1 \to A_2$. Then type checking type-level computations, in the fifth and eighth premise of (C-App-Comp), is done under the rewritten class table.

Note that, while this prevents the type checking rules from infinitely recursing, it does not prevent type-level expressions from themselves diverging. In λ^C, we assume this does not happen, but in our implementation, we include a simple termination checker that is effective in practice (§ 4).

3.3 Properties of λ^C.

Finally, we prove type soundness for λ^C. For brevity, we provide only the high-level description of the proof. The details can be found in the technical report [25].

Blame. The type system of λ^C does not prevent null-pointer errors, i.e., `nil` has no methods yet we allow it to appear wherever any other type of object is expected. We encode such errors as *blame*. We also reduce to *blame* when a dynamic check of the form $\lceil A' \rceil A.m(v)$ fails.

Program Checking and CT. In the technical report [25] we provide type checking rules not just for λ^C expressions but also for programs P. These rules are where we actually check user-defined methods against their types. We also define a notion of *validity* for a class table CT with respect to P, which enforces that CT's types for methods and fields match the declared types in P, and that appropriate subtyping relationships hold among subclasses. Given a well typed program P, it is straightforward to construct a valid CT.

Type Checking Rules. In addition to the type checking and rewriting rules of Figure 5, we define a separate judgment $\Gamma \vdash_{CT} e : A$ that is identical to $\Gamma \vdash_{CT} e \hookrightarrow e : A$ except it omits the rewriting step, i.e., only performs type checking.

We can then prove soundness of the judgment $\Gamma \vdash_{CT} e : A$ using preservation and progress, and finally prove soundness of the type checking and rewriting rules as a corollary:

Theorem 3.1 (Soundness). *For any expressions e and e', type A, class table CT, and program P such that CT is valid with respect to P, if* $\emptyset \vdash_{CT} e \hookrightarrow e' : A$ *then e' either reduces to a value, reduces to blame, or does not terminate.*

4 Implementation

We implemented CompRDL as an extension to RDL, a type checking system for Ruby [18, 37, 38, 41]. In total, CompRDL comprises approximately 1,170 lines of code added to RDL.

RDL's design made it straightforward to add comp types. We extended RDL so that, when type checking method calls, type-level computations are first type checked to ensure they produce a value of type *Type* and then are executed to produce concrete types, which are used in subsequent type checking. Finally, these call sites are rewritten to include dynamic checks that enforce the correctness of comp types.

Heap Mutation. For simplicity, λ^C does not include a heap. By contrast, CompRDL allows arbitrary Ruby code to appear in comp types. This allows great flexibility, but it means such code might depend on mutable state that could change between type checking and the execution of a method call. For example, in Figure 1, type-level code uses the global table RDL.db_schema. If, after type checking the method available?, the program (pathologically) changed the schema of User to drop the username column, then available? would fail at runtime even though it had type checked. The dynamic checks discussed in § 2 and § 3 are insufficient to catch this issue, because they only check a method call against the initial result of evaluating a comp type; they do not consider that the same comp type might yield a new result at runtime.

To address this issue, CompRDL extends dynamic checks to ensure types remain the same between type checking and execution. If a method call is type checked using a comp type, then prior to that call at runtime, CompRDL will reevaluate that same comp type on the same inputs. If it evaluates to a different type, CompRDL will raise an exception to signal a potential type error. An alternative approach would be to re-check the method under the new type.

Of course, the evaluation of a comp type may itself alter mutable state. Currently, CompRDL assumes that comp type specifications are correct, including any mutable computations they may perform. If a comp type does have any erroneous effects, program execution could fail in an unpredictable manner. Other researchers have proposed safeguards for this issue of effectful contracts by using guarded locations [15] or region based effect systems [39]. We leave incorporating such safeguards for comp types as future work. We note, however, that this issue did not arise in any comp types we used in our experiments.

Termination of Comp Types. A standard property of type checkers is that they terminate. However, because comp types allow arbitrary Ruby code, CompRDL could potentially lose this property. To address this issue, CompRDL includes a lightweight termination checker for comp types.

Figure 6 illustrates the ideas behind termination checking. In CompRDL, methods can be annotated with *termination effects* :+, for methods that always terminate (e.g., m1 and m2) and :− for methods that might diverge (e.g., m3). CompRDL allows terminating methods to call other terminating methods (Line 9) but not potentially non-terminating methods (Line 10). Additionally, terminating methods may not

```
1  type :m1, ..., terminates: :+
2  type :m2, ..., terminates: :+
3  type :m3, ..., terminates: :−
4
5  type Array, :map, ..., terminates: :blockdep
6  type Array, :push, ..., pure: :−
7
8  def m1()
9    m2() # allowed: m2 terminates
10   m3() # not allowed: m3 may not terminate
11   while ... end # not allowed: looping
12
13   array = [1,2,3] # create new array
14   array.map { |val| val+1 } # allowed
15   array.map { |val| array.push(4) }
16   # not allowed: iterator calls impure method push
17 end
```

Figure 6. Termination Checking with CompRDL.

use loops (Line 11). We assume that type-level code is not recursive, and leave checking for recursion to future work.

We believe it is reasonable to forbid the use of built-in loop constructs and to assume no recursion, because in practice most iteration in Ruby occurs via methods that iterate over a structure. For instance, array.map {block} returns a new array in which the block, a *code block* or lambda, has been applied to each element of array. Since arrays are by definition finite, this call terminates as long as block terminates and does not mutate the array. A similar argument holds other iterators of Array, Hash, etc.

Thus, CompRDL checks termination of iterators as follows. Iterator methods can be annotated with the special termination effect :blockdep (Line 5), indicating the method terminates if its block terminates and is pure. CompRDL also includes *purity effect* annotations indicating whether methods are pure (:+) or impure (:−). A pure method may not write to any instance variable, class variable, or global variable, or call an impure method. CompRDL determines that a :blockdep method terminates as long as its block argument is pure, and otherwise it may diverge. Using this approach, CompRDL will allow Line 14 but reject reject Line 15.

Type Mutations and Weak Updates Finally, to handle aliasing, our type annotations for Array, Hash, and String need to perform weak updates to type information when tuple, finite hash, and const string types, respectively, are mutated. For example, consider the following code:

```
a = [1, 'foo']; if...then b = a else...end ; a[0]='one'
```

Here (ignoring singleton types for simplicity), a initially has the type t = [Integer, String], where t is a Ruby object, specifically an instance of RDL's TupleType class. At the join point after the conditional, the type of b will be a union of t and its previous type.

Table 1. Library methods with comp type definitions.

Library	Comp Type Definitions	Ruby LoC	Helper Methods
Ruby Core Library			
Array	114	215	15
Hash	48	247	15
String	114	178	12
Float*	98	12	1
Integer*	108	12	1
Database DSL			
ActiveRecord	77	375	18
Sequel	27	408	22
Total	586	1447	83

*Helper methods for Float and Integer are shared.

We could potentially forbid the assignment to a[0] because the right-hand side does not have the type Integer. However, this is likely too restrictive in practice. Instead, we would like to mutate t after the write. However, b shares this type. Thus we perform a *weak update*: after the assignment we mutate t to be [Integer∪String, String], to handle the cases when a may or may not have been assigned to b.

For soundness, we need to retroactively assume t was always this type. Fortunately, for all tuple, finite hash, and const string types τ, RDL already records all asserted constraints $\tau' \leq \tau$ and $\tau \leq \tau'$ to support promotion of tuples, finite hashes, and const strings to types Array, Hash, and String, respectively [18]. We use this same mechanism to replay previous constraints on these types whenever they are mutated. For example, if previously we had a constraint $\alpha \leq$ [Integer, String], and subsequently we mutated the latter type to [Integer∪String, String], we would "replay" the original constraint as $\alpha \leq$ [Integer ∪ String, String].

5 Experiments

We evaluated CompRDL by writing comp type annotations for a number of Ruby core and third party libraries (§ 5.1) and using these types to type check real-world Ruby applications (§ 5.2). We discuss the results of type checking these benchmarks, including the type errors we found in the process (§ 5.3). In all, we wrote 586 comp type annotations for Ruby library methods, used them to type check 132 methods across six Ruby apps, found three bugs in the process, and used significantly fewer manually inserted type casts than are needed using RDL.

5.1 Library Types

Table 1 details the library type annotations we wrote. We chose to define comp types for these libraries due to their popularity and because, as discussed in § 2, they are amenable to precise typing with comp types. We wrote types based on the libraries' documentation as well as manual testing to ensure comp types matched associated method semantics.

- *Ruby core libraries:* These are libraries that are written in C and automatically loaded in all Ruby programs. We annotate the methods from the Array, Hash, String, Integer, and Float classes.
- *ActiveRecord:* ActiveRecord is the most used object-relational model (ORM) DSL of the Ruby on Rails web framework. We wrote comp types for ActiveRecord database query methods.
- *Sequel:* Sequel is an alternative database ORM DSL. It offers some more expressive queries than are available in ActiveRecord.

Table 1 lists the number of methods for which we defined comp types in each library and the number of Ruby lines of code (LoC) implementing the type computation logic. The LoC count was calculated with sloccount [47] and does not include the line of the type annotation itself.

In developing comp types for these libraries, we discovered that many methods have the same type checking logic. This helped us write comp types for entire libraries using a few common helper methods, e.g., schema_type in § 2.1. In total, we wrote comp type annotations for 586 methods across these libraries, comprising 1447 lines of type-level code and using 83 helper methods. These comp types can be used to type check as many of the libraries' clients as we would like, making the effort of writing them very rewarding.

5.2 Benchmarks

We evaluated CompRDL by type checking methods from two popular Ruby libraries and four Rails web apps:

- *Wikipedia Client* [14] is a Ruby wrapper library for the Wikipedia API.
- *Twitter Gem* [32] is a Ruby wrapper library for the Twitter API.
- *Discourse* [23] is an open-source discussion platform built on Rails. It uses ActiveRecord.
- *Huginn* [22] is a Rails app for setting up agents that monitor the web for events and perform automated tasks in response. It uses ActiveRecord.
- *Code.org* [12] is a Ruby app that powers code.org, a site that encourages people, particularly students, to learn programming. It uses a combination of ActiveRecord and Sequel.
- *Journey* [6] is a Rails app that provides a graphical interface to create surveys and collect responses from participants. It uses a combination of ActiveRecord and Sequel.

We selected these benchmarks because they are popular, well-maintained, and make extensive use of the libraries noted in § 5.1. More specifically, the APIs often work with hashes representing JSON objects received over HTTP, and the Rails apps rely heavily on database queries.

Since CompRDL performs type checking, we must provide a type annotation for any method we wish to type check. Our

Table 2. Type checking results.

Program	Meths	LoC	Extra Annots.	Casts	Casts (RDL)	Time (s) Median ± SIQR	Test Time No Chk (s)	Test Time w/Chk. (s)	Errs
API client libraries									
Wikipedia	16	47	3	1	13	0.06 ± 0.00	6.3 ± 0.13	6.32 ± 0.11	0
Twitter	3	29	11	3	8	0.02 ± 0.00	0.07 ± 0.00	0.08 ± 0.00	0
Rails Applications									
Discourse	36	261	32	13	22	7.77 ± 0.39	80.24 ± 0.63	81.04 ± 0.34	0
Huginn	7	54	6	3	6	2.46 ± 0.29	4.30 ± 0.21	4.59 ± 0.48	0
Code.org	49	530	53	3	68	0.49 ± 0.01	2.49 ± 0.13	2.74 ± 0.02	1
Journey	21	419	78	14	59	4.12 ± 0.08	4.52 ± 0.22	4.76 ± 0.24	2
Total	132	1340	183	37	176	14.93 ± 0.77	97.93 ± 1.31	99.53 ± 1.20	3

subject programs are very large, and hence annotating all of the programs' methods is infeasible. Instead, we focused on methods for which comp types would be most useful.

In *Wikipedia*, we annotated the entire Page API. To simplify type checking slightly, we changed the code to replace string hash keys with symbols, since RDL's finite hash types do not currently support string keys. In *Twitter*, we annotated all the methods of stream API bindings that made use of methods with comp types.

In *Discourse* and *Huginn*, we chose several larger Rails model classes, such as a User class that represents database rows storing user information. In *Code.org* and *Journey*, we type checked all methods that used Sequel to query the database. Within the selected classes for these four Rails apps, we annotated a subset of the methods that query the database using features that CompRDL supports. The features CompRDL does not currently support include the use of Rails *scopes*, which are essentially macros for queries, and the use of SQL strings for methods other than where.

Finally, because CompRDL performs type checking at runtime (see § 2.1), we must first load each benchmark before type checking it. We ran the type checker immediately after loading a program and its associated type annotations.

5.3 Results

Table 2 summarizes our type checking results. In the first group of columns, we list the number of type checked methods and the total lines of code (computed with sloccount) of these methods. The third column lists the number of additional annotations we wrote for any global and instance variables referenced in the method, as well as any methods called that were not themselves selected for type checking. The last column in this group lists the number of type casts we added. Many of these type casts were to the result of JSON.parse, which returns a nested Hash/Array data structure depending on its string input. Most of the remaining casts are to refine types after a conditional test; it may be possible to remove these casts by adding support for occurrence typing [27].

Increased Type Checking Precision. Recall from § 2.2 that comp types can potentially reduce the need for programmer-inserted type casts. The next column reports how many casts were needed using normal RDL (i.e., no comp types). As shown, approximately 4.75× fewer casts were needed when using comp types. This reflects the significantly increased precision afforded by comp types, which greatly reduces the programmer's annotation burden.

Performance. The next group of columns report performance. First we give the type checking time as the median and semi-interquartile range (SIQR) of 11 runs on a 2017 MacBook Pro with a 2.3GHz i5 processor and 8GB RAM. In total, we type checked 132 methods in approximately 15 seconds, which we believe to be reasonable. *Discourse* took most of the total time (8 out of 15 seconds). The reason turned out to be a quirk of *Discourse*'s design: it creates a large number of methods on-the-fly when certain constants are accessed. Type checking accessed those constants, hence the method creation was included in the type checking time.

The next two columns show the performance overhead of the dynamic checks inserted by CompRDL. We selected a subset of each app's test suite that directly tested the type checked methods, and ran these tests without ("No Chk") and with ("w/Chk") the dynamic checks. In aggregate (last row), checks add about 1.6% overhead, which is minimal.

Errors Found. Finally, the last column lists the number of errors found in each program. We were somewhat surprised to find any errors in large, well-tested applications. We found three errors. In *Code.org*, the current_user method was documented as returning a User. We wrote a matching type annotation, and CompRDL found that the returned expression—whose typing involved a comp type—has a hash type instead. We notified the *Code.org* developers, and they acknowledged that this was an error in the method documentation and made a fix.

In *Journey*, CompRDL found two errors. First, it found a method that referenced an undefined constant Field. We notified the developers, who fixed the bug by changing the constant to Question::Field. This bug had arisen due to

namespace changes. Second, it found a method that included a call with an argument { :action ⇒ prompt, ... } which is a hash mapping key :action to prompt. The value prompt is supposed to be a string or symbol, but as it has neither quotes nor begins with a colon, it is actually a call to the prompt method, which returns an array. The developers confirmed this bug.

When type checking the aforementioned methods in RDL (i.e., without comp types), two out of three of the bugs are hidden by other type errors which are actually false positives. These errors can be removed by adding four type casts, which would then allow us to catch the true errors. With CompRDL, however, we do not need any casts to find the errors.

6 Related Work

Types For Dynamic Languages. There is a large body of research on adding static type systems to dynamic languages, including Ruby [21, 37, 38], Racket [44, 45], JavaScript [3, 29, 43], and Python [2, 4]. To the best of our knowledge, this research does not use type-level computations.

Dependent typing systems for dynamic languages have been explored as well. Ou et al. [34] formally model type-level computation along with effects for a dynamic language. Other projects have sought to bring dependent types to existing dynamic languages, primarily in the form of refinement types [20], which are base types that are refined with expressive logical predicates. Refinement types have been applied to Ruby [26], Racket [27], and JavaScript [11, 46]. In contrast to CompRDL, these systems focus on type checking methods which themselves have dependent types. On the other hand, CompRDL uses type-level computations only for non-type checked library methods, allowing us to avoid checking comp types for equality or subtyping (§ 2.4). While sacrificing some expressiveness, this makes CompRDL especially practical for real-world programs.

Turnstile [8] is a metalanguage, hosted in Racket, for creating typed embedded languages. It lets an embedded DSL author write their DSL's type system using the host language macro system. There is some similarity to CompRDL, where comp types manipulate standard RDL types. However, CompRDL types are not executed as macros (which do not exist in Ruby), but rather in standard Ruby so they have full access to the environment, e.g., so the joins type signature can look up the DB schema.

Types For Database Queries. There have been a number of prior efforts to check the type safety of database queries. All of these target statically typed languages, an important distinction from CompRDL.

Chlipala [10] presents Ur, a web-specific functional programming language. Ur uses type-level computations over record types [36] to type check programs that construct and run SQL queries. Indeed, CompRDL similarly uses type-level computations over finite hash types (analogous to record types) to type check queries. To the best of our knowledge, Ur focuses on computations over records. In contrast, CompRDL supports arbitrary type-level computations targeting unchecked library methods, making comp types more easily extensible to checking new properties and new libraries. As discussed in § 2, for example, comp types can not only compute the schema of a joined table, but also check properties like two joined tables having a declared Rails association. Further, comp types can be usefully applied to many libraries beyond database queries (§ 5).

Similar to Ur, Baltopoulos et al. [5] makes use of record types over embedded SQL tables. Using SMT-checked refinement types, they can statically verify expressive data integrity constraints, such as the uniqueness of primary keys in a table and the validation of data inserted into a table. In addition to the contrast we draw with Ur regarding extensibility of types, to the best of our knowledge, this work does not include more intricate queries like joins, which are supported in CompRDL.

New Languages for Database Queries. Domain-specific languages have long been used to write programs with correct-by-construction, type safe queries. Leijen and Meijer [28] implement Haskell/DB, an embedded DSL that dynamically generates SQL in Haskell. Karakoidas et al. [24] introduce J%, a Java extension for embedding DSLs into Java in an extensible, type-safe, and syntax-checked way. Fowler and Brady [19] use dependent types in the language Idris to enforce safety protocols associated with common web program features including database queries written in a DSL.

Language-integrated query is featured in languages like LINQ [31] and Links [9, 13]. This approach allows programmers to write database queries directly within a statically-typed, general purpose language.

In contrast to new DSLs and language-integrated query, our focus in on bringing type safety to an existing language and framework rather than developing a new one.

Dependent Types. Traditional dependent type systems are exemplified by languages such as Coq [35], Agda [33], and F* [42]. These languages provide powerful type systems that allow programmers to prove expressive properties. However, such expressive types may be too heavyweight for a dynamic language like Ruby. As discussed in § 2.4, our work has focused on applying a limited form of dependent types, where types depend on argument types and not arbitrary program values, resulting in a system that is practical for real-world Ruby programs.

Haskell allows for light dependent typing using the combination of singleton types [16] and type families [7]. CompRDL's singleton types are similar to Haskell's, i.e., both lifting expressions to types, and comp types are analogous to anonymous type families. However, unlike Haskell, CompRDL supports runtime evaluation during type checking, and thus does not require user-provided proofs.

Scala supports *path dependent types*, a limited form of type/term dependency in which types can depend on variables, but, as of Scala version 2, does not allow dependency on general terms [1]. This allows for reasoning about database queries. For example, the Scala library *Slick* [30], much like our approach, allows users to write database queries in a domain specific language (a lifted embedding) and uses the query's AST to type check the query using Scala's path dependent types. Unlike CompRDL, Scala's path dependent types do not allow the execution of the full host language during type computations.

7 Conclusion

We presented CompRDL, a system for adding type signatures with type-level computations, which we refer to as comp types, to Ruby library methods. CompRDL makes it possible to write comp types for database queries, enabling us to type check such queries precisely. Comp type signatures can also be used for libraries over heterogeneous hashes and arrays, and to treat strings as immutable when possible. The increased precision of comp types can reduce the need for manually inserted type casts, thereby reducing the programmer's burden when type checking. Since comp type-annotated method bodies are not themselves type checked, CompRDL inserts run-time checks to ensure those methods return their computed types. We formalized CompRDL as a core language λ^C and proved its type system sound.

We implemented CompRDL on top of RDL, an existing type system for Ruby. In addition to the features of λ^C, our implementation includes run-time checks to ensure comp types that depend on mutable state yield consistent types. Our implementation also includes a termination checker for type-level code, and the type signatures we developed perform weak updates to type certain mutable methods.

Finally, we used CompRDL to write comp types for several Ruby libraries and two database query DSLs. Using these type signatures, we were able to type check six popular Ruby apps and APIs, in the process discovering three errors in our subject programs. We also found that type checking with comp types required 4.75× fewer type casts, due to the increased precision. Thus, we believe that CompRDL represents a practical approach to precisely type checking programs written in dynamic languages.

Acknowledgments

Thanks to the anonymous reviewers for their helpful comments. This research was supported in part by NSF CCF-1518844, CCF-1846350, DGE-1322106, and Comunidad de Madrid as part of the program S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union.

References

[1] Nada Amin, Karl Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. *The Essence of Dependent Object Types.* Springer International Publishing, Cham, 249–272.

[2] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. 2007. RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 Symposium on Dynamic Languages (DLS)*. ACM, New York, NY, USA, 53–64.

[3] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards Type Inference for Javascript. In *ECOOP 2005 - Object-Oriented Programming (ECOOP)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 428–452.

[4] John Aycock. 2000. Aggressive Type Inference. (2000).

[5] Ioannis G. Baltopoulos, Johannes Borgström, and Andrew D. Gordon. 2011. Maintaining Database Integrity with Refinement Types. In *ECOOP 2011 – Object-Oriented Programming*, Mira Mezini (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 484–509.

[6] Nat Budin. 2018. Journey: An online questionnaire application. (2018). https://github.com/nbudin/journey/.

[7] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. 2005. Associated Types with Class. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/1040305.1040306

[8] Stephen Chang, Alex Knauth, and Ben Greenman. 2017. Type Systems As Macros. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 694–705. https://doi.org/10.1145/3009837.3009886

[9] James Cheney, Sam Lindley, Gabriel Radanne, and Philip Wadler. 2014. Effective Quotation: Relating Approaches to Language-integrated Query. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM)*. ACM, New York, NY, USA, 15–26.

[10] Adam Chlipala. 2010. Ur: Statically-typed Metaprogramming with Type-level Record Computation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 122–133.

[11] Ravi Chugh, David Herman, and Ranjit Jhala. 2012. Dependent Types for JavaScript. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, New York, NY, USA, 587–606.

[12] Code.org. 2018. The code powering code.org and studio.code.org. (2018). https://github.com/code-dot-org/code-dot-org.

[13] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects (FMCO)*. Springer-Verlag, Berlin, Heidelberg, 266–296.

[14] David Cyril and Ken Pratt. 2018. Ruby client for the Wikipedia API. (2018). https://github.com/kenpratt/wikipedia-client.

[15] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *Proceedings of the 21st European Conference on Programming Languages and Systems (ESOP'12)*. Springer-Verlag, Berlin, Heidelberg, 214–233. https://doi.org/10.1007/978-3-642-28869-2_11

[16] Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently Typed Programming with Singletons. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 117–130. https://doi.org/10.1145/2364506.2364522

[17] Cormac Flanagan. 2006. Hybrid Type Checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 245–256. https://doi.org/10.1145/1111037.1111059

[18] Jeffrey Foster, Brianna Ren, Stephen Strickland, Alexander Yu, and Milod Kazerounian. 2018. RDL: Types, type checking, and contracts for Ruby. (2018). https://github.com/plum-umd/rdl.

[19] Simon Fowler and Edwin Brady. 2014. Dependent Types for Safe and Secure Web Programming. In *Implementation and Application of*

Functional Languages (IFL '13). ACM, New York, NY, USA, 49:49–49:60.

[20] Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. *SIGPLAN Not.* 26, 6 (May 1991), 268–277. https://doi.org/10.1145/113446.113468

[21] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. 2009. Static Type Inference for Ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC '09)*. ACM, New York, NY, USA, 1859–1866.

[22] Huginn. 2018. Huginn: Create agents that monitor and act on your behalf. (2018). https://github.com/huginn/huginn.

[23] Civilized Discourse Construction Kit Inc. 2018. Discourse: A platform for community discussion. (2018). https://github.com/discourse/discourse.

[24] Vassilios Karakoidas, Dimitris Mitropoulos, Panagiotis Louridas, and Diomidis Spinellis. 2015. A Type-safe Embedding of SQL into Java Using the Extensible Compiler Framework J%. *Computer Languages, Systems, and Structures* 41, C (2015), 1–20.

[25] Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S. Foster, and David Van Horn. 2019. Type-Level Computations for Ruby Libraries (Technical Report). (2019). arXiv:arXiv:1904.03521

[26] Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. 2018. Refinement Types for Ruby. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer International Publishing, Cham, 269–290.

[27] Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. 2016. Occurrence Typing Modulo Theories. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 296–309.

[28] Daan Leijen and Erik Meijer. 1999. Domain Specific Embedded Compilers. *SIGPLAN Not.* 35, 1 (Dec. 1999), 109–122. https://doi.org/10.1145/331963.331977

[29] Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. 2013. TeJaS: Retrofitting Type Systems for JavaScript. In *Proceedings of the 9th Symposium on Dynamic Languages (DLS)*. ACM, New York, NY, USA, 1–16.

[30] Lightbend, Inc. 2019. Slick. (2019). http://slick.lightbend.com/.

[31] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, New York, NY, USA, 706–706.

[32] Erik Michaels-Ober, John Nunemaker, Wynn Netherland, Steve Richert, and Steve Agalloco. 2018. A Ruby interface to the Twitter API. (2018). https://github.com/sferik/twitter.

[33] Ulf Norell. 2009. *Dependently Typed Programming in Agda*. Springer Berlin Heidelberg, Berlin, Heidelberg, 230–266.

[34] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types. In *Exploring New Frontiers of Theoretical Informatics*, Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell (Eds.). Springer US, Boston, MA, 437–450.

[35] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2017. *Software Foundations*. Electronic textbook, http://www.cis.upenn.edu/~bcpierce/sf.

[36] D. Rémy. 1989. Type Checking Records and Variants in a Natural Extension of ML. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, New York, NY, USA, 77–88. https://doi.org/10.1145/75277.75284

[37] Brianna M. Ren and Jeffrey S. Foster. 2016. Just-in-time Static Type Checking for Dynamic Languages. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 462–476.

[38] Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. 2013. The Ruby Type Checker. In *Object-Oriented Program Languages and Systems (OOPS) Track at ACM Symposium on Applied Computing*. ACM, Coimbra, Portugal, 1565–1572.

[39] Taro Sekiyama and Atsushi Igarashi. 2017. Stateful Manifest Contracts. *SIGPLAN Not.* 52, 1 (Jan. 2017), 530–544. https://doi.org/10.1145/3093333.3009875

[40] Jeremy Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Seventh Workshop on Scheme and Functional Programming*. ACM, Portland, OR, USA, 81–92.

[41] T. Stephen Strickland, Brianna Ren, and Jeffrey S. Foster. 2014. Contracts for Domain-Specific Languages in Ruby. In *Dynamic Languages Symposium (DLS)*. ACM, Portland, OR, 23–34.

[42] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 256–270.

[43] Peter Thiemann. 2005. Towards a Type System for Analyzing JavaScript Programs. In *Programming Languages and Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 408–422.

[44] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, New York, NY, USA, 964–974.

[45] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 395–406.

[46] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement Types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 310–325.

[47] D. A. Wheeler. 2018. SLOCCount. (2018). https://www.dwheeler.com/sloccount/.