Contracts for Domain-Specific Languages in Ruby

T. Stephen Strickland Brianna M. Ren Jeffrey S. Foster

University of Maryland, College Park

{sstrickl,bren,jfoster}@cs.umd.edu

Abstract

This paper concerns *object-oriented* embedded DSLs, which are popular in the Ruby community but have received little attention in the research literature. Ruby DSLs implement language keywords as implicit method calls to self; language structure is enforced by adjusting which object is bound to self in different scopes. While Ruby DSLs are powerful and elegant, they suffer from a lack of specification. In this paper, we introduce *contracts* for Ruby DSLs, which allow us to attribute *blame* appropriately when there are inconsistencies between an implementation and client. We formalize Ruby DSL contract checking in λ_{DSL}, a core calculus that uses premethods with instance evaluation to enforce contracts. We then describe RDL, an implementation of Ruby DSL contracts. Finally, we present two tools that automatically infer RDL contracts: TypeInfer infers simple, type-like contracts based on observed method calls, and DSLInfer infers DSL keyword scopes and nesting by generating and testing candidate DSL usages based on initial examples. The type contracts generated by TypeInfer work well enough, though they are limited in precision by the small number of tests, while DSLInfer finds almost all DSL structure. Our goal is to help users understand a DSL from example programs.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Programming by contract

Keywords domain-specific languages, software contracts

1. Introduction

Embedded domain-specific languages (EDSLs) [6, 8, 12, 21, 26, 33] let programmers express their ideas with domain-specific concepts within the syntax of a host language. EDSLs are both powerful and convenient, as they leverage host language implementations to integrate with existing libraries, interpreters, and compilers. This paper studies a kind of EDSL that, to our knowledge, has not been previously explored in the research literature: those developed and used extensively by the Ruby community. We refer to these as *object-oriented* embedded DSLs (or just *Ruby DSLs*), because they implement DSL keywords as methods bound to self. Language scoping is then controlled by rebinding self appropriately. (Examples of Ruby DSLs can be found in Section 2 and elsewhere [25].)

DLS '14, October 21, 2014, Portland, Oregon, USA.

ACM 978-1-4503-3211-8/14/10. . . $15.00.

http://dx.doi.org/10.1145/2661088.2661092

In this paper, we propose RDL, a new *contract* system for Ruby DSLs. RDL helps users understand a DSL because the DSL documentation is often insufficient. The key feature of RDL is that it integrates both standard contract checking [15, 30] and *structural* checks of correct DSL keyword availability and use, all in a Ruby-friendly way. The latter lets programmers enforce the definitions of DSLs and attribute *blame* properly when a contract is violated. For example, without contracts a client could invoke an inherited method that is not intended to be a DSL keyword, or a provider could fail to implement a keyword. RDL also supports *higher-order DSL* contracts, meaning one DSL keyword might require that one of its arguments be an object implementing a specified DSL. Finally, RDL also allows programmers to implement DSLs entirely in RDL, providing a much cleaner approach to Ruby DSL design. (Section 2 describes RDL contracts.)

To illustrate the design behind our proposed system, we formalize DSL contract checking using λ_{DSL}, a core calculus. λ_{DSL} includes four important features: standard contracts; DSL contracts, which are structural checks against both the implementation and use of a DSL written using a simple specification language; premethods, used to η-expand methods with contract checks; and *instance evaluation*, which invokes a premethod under a specific self binding. Premethods and instance evaluation are the key to making DSLs lightweight, as they permit scopes to be introduced and invoked without much fuss (i.e., without naming a method or placing one in a class). We were able to implement all of this using Findler–Felleisen style higher-order contracts [15] as the basic contracting mechanism. We prove that λ_{DSL} satisfies a contract erasure theorem—approximately speaking, programs that do not violate contracts can have their contracts erased without affecting the final result. (Section 3 presents λ_{DSL} and the erasure theorem.)

Another result of our work is a Ruby library that implements RDL. The implementation works by creating proxies that ensure that only methods in the DSL are used by clients of the DSL, and not other utility methods that happen to live in the same class. To do this, it wraps block arguments that are run under DSLs to interpose appropriate checks during block evaluation. RDL also includes specifications of basic type contracts and arbitrary pre- and postconditions on language keywords. In addition to contract checking, RDL contains a *definition mode* that allows developers to give both a contract for and an implementation of a DSL simultaneously. (Section 4 describes RDL in detail.)

To test the applicability of RDL, we developed two dynamic contract inference systems. The first, TypeInfer, uses observed values at run time to generate type contracts on method arguments and returns. The second, DSLInfer, discovers structural information about keyword scopes and nesting. There are several potential uses of TypeInfer and DSLInfer, including giving developers a running head start in developing specifications, helping developers to ensure that they have full coverage of their DSL in the test suites, as well as helping users discern the structure of DSLs they use.

```
1  sox = Team.new "Boston Red Sox" do
2    player "David Ortiz" { hits 52; at_bats 198 }
3    player "Dustin Pedroia" { hits 62; at_bats 227 }
4  end
5
6  sox.players.map { |p| p.stats.batting_average }
7  # ==> [0.309, 0.301]
```

Figure 1: Baseball DSL example

TypeInfer works by simply recording type values at runtime. DSLInfer is more complex. It starts from a small set of examples and generates *candidate examples* that modify the initial examples by copying a keyword usage from one scope to another. DSLInfer then runs the candidate examples to see which are valid and which are rejected. From these results, DSLInfer discovers which keywords are available in which scopes in the DSL, and from this it generates a contract.

We ran TypeInfer and DSLInfer on eight Ruby DSLs. We found that TypeInfer generally discovers useful types. However, in our experiments TypeInfer observed relatively few calls to the DSL methods, and hence the discovered types were sometimes imprecise, e.g., missing that a block argument may be optional because all calls in the example programs supplied a block. We found that DSLInfer generally worked quite well. We ran it on programs containing a total of 112 initial keyword nesting pairs from the same 8 DSLs. DSLInfer inferred an additional 78 pairs with 8 false negatives. Additionally, TypeInfer runs at essentially the same speed as the test cases, and DSLInfer's running time ranged from seconds on most DSLs to minutes on two DSLs. (Section 5.2 describes contract inference and our experiments.)

In summary, this paper presents what we believe is the first formal account of Ruby DSLs; RDL, a library for both defining and contracting DSLs in Ruby; TypeInfer, a tool for inferring type contracts; and DSLInfer, an inference tool for discovering the structure of existing DSLs and autogenerating RDL templates.

2. Ruby DSLs

In this section, we introduce Ruby DSLs and RDL by example. First, we present an small illustrative Ruby DSL and some example uses. Next, we show how RDL can specify the behavior of this DSL using our contract system. Lastly, we build the example DSL using RDL's support for declaratively implementing DSLs.

An example Ruby DSL. Figure 1 illustrates the use of a small example DSL that creates objects representing baseball teams. A team instance is constructed via Team.new, passing the name of the team as an argument and supplying a block that describes the team's players. Each player is defined using the player keyword, which itself takes the player's name as an argument and a block giving the player's hits and at_bats. The stored information can be used to compute statistics such as the player's batting average.

While this code might not look it, in fact this is ordinary Ruby. Here do ... end and { ... } delimit *code blocks*, which are *premethods*, i.e., lambdas that may also refer to self (which is implicitly bound). Note that both ways of delimiting a code block are completely equivalent. One code block can be passed as an argument to a method by placing it last in the list of arguments. Method argument lists need not be parenthesized. Thus, player, hits, and at_bats are just method invocations with the implicit self receiver, bound appropriately during the execution of the code blocks.

As we have just seen, a few key features enable Ruby DSLs to have a very declarative syntax while still maintaining full language access. However, despite the common use of DSLs in the Ruby community, there is no standard way of documenting DSLs, much less specifying their behavior in a machine-checkable way. Instead, users must cobble together an understanding of a DSL from a combination of examples and, if available, documentation of the classes used in implementing the DSL. Because of this, it is easy for there to be misunderstandings between the DSL implementer and user. For example, suppose the user tries to call

```
8  player_spec = RDL::Dsl.new do
9    spec :player do
10     arg 0, String
11     dsl do
12       spec :hits do
13         arg 0, Integer
14         pre_cond("More hits than at-bats!") { |h|
15           @at_bats ? h < @at_bats : true }
16       end
17       spec :at_bats do
18         arg 0, Integer
19         pre_cond("Fewer at-bats than hits!") { |a|
20           @hits ? a > @hits : true }
21       end
22     end
23   end
24 end
25
26 class Team
27   spec : initialize { dsl_from player_spec }
28 end
```

Figure 2: RDL Contract for the Baseball DSL

```
player "Mike Napoli" { hits 40; runs_scored 17 }
```

If no runs_scored method exists, Ruby throws a NoMethodError exception. The same error is reported whether the mistake is in the client (calling a keyword not intended to be there) or the DSL (failing to provide a keyword). Even worse, the user could successfully call a method not intended to be part of the language, e.g.,

```
player "Daniel Nava" { hits 10; dump_stats "nava.csv" }
```

Here, the utility function dump_stats is not part of the DSL for describing players, but rather a method meant to be used on fully specified player objects. Note that because we are invoking a method on self, even if dump_stats were made private, it would still be accessible within the block.

Specifying Ruby DSLs. To address these problems, we developed RDL to enforce contracts on DSLs. Figure 2 contains a short RDL specification for the baseball DSL. Recall this DSL has two sublanguages—inside of Team.new, player is in scope, and inside of player, hits and at_bats are in scope.

Lines 8–24 create a DSL contract—an instance of RDL::Dsl—that includes the player keyword. Inside a DSL contract definition,[1] the spec keyword defines contracts for individual methods, e.g., line 9 introduces a contract for the player method. Here we pass the method name as a *symbol* (i.e., an interned string) :player because in Ruby the bare method name player would be treated as a method invocation. The contract for player calls arg 0, Integer (line 10) to specify that the first argument given, the name of the player, is a member of the String class. (RDL converts class values like String into contracts that check objects for membership in that class.)

[1] This is indeed a DSL for defining DSL contracts. It is not implemented in terms of itself, however, since that would lead to infinite recursion.

The contract for player also uses dsl (line 11) to contract the sub-DSL in the block argument to player. Here, that nested DSL includes two keywords, hits and at_bats. When player is executed, we add checks that ensure that player runs the block argument with a self object that contains those two methods. In addition, we check that the block argument only calls those particular methods on self.

The hits and at_bats keywords do not take block arguments, so they do not use dsl. Instead, we add a contract on the first argument of each that checks that only integers are provided. We also use pre_cond to specify a precondition for each method. If a client uses the player DSL and any of the added checks fail, then RDL raises an exception with a suitable error message. Lines 14–15 describe a pre-condition that specifies if the field @at_bats has been set, then the argument to hits has to be smaller than @at_bats.

As an aside, notice that we duplicated the relationship between the values provided for hits and at_bats in the two preconditions. This is necessary because RDL cannot describe field invariants, as Ruby does has no mechanism for intercepting field writes. We plan to investigate adding such a mechanism to Ruby as future work.

The contract player_spec is a first-class object, unattached to any particular DSL implementation. To actually attach a contract to a method, we call spec at the class level. For example, line 26 reopens the Team class[2] and adds a contract on initialize (the constructor method). That contract uses the form dsl_from to add the player_spec contract to the block argument to initialize. In other words, dsl_from is just like dsl, but instead of taking a block defining the DSL-sublanguage, it takes a instance of RDL::Dsl.

Contracts in RDL can also be applied to method arguments in a higher-order fashion, and contracts can be combined. For example, Figure 3 sketches how we might extend the baseball DSL to print player records as the team is defined, where the printing mechanism is abstract, e.g., so we can print baseball cards, create webpages with players' information, etc. On lines 29–33, we define a contract for a formatting DSL that includes keywords such as header and table that abstract over the precise method of printing. Then on lines 35–39, we create a contract specifying that print's first argument is an object that conforms to print_spec—meaning, if that argument is bound to self during the execution of a block, it provides the DSL keywords specified by print_spec.

As before, we reopen the Team class to add the desired contract. Here, we combine player_spec and printable_spec to create a contract that makes both methods available within the block argument to Team.new and checks them appropriately. Finally, we create a baseball team, and inside its definition we pass a new instance of some printer object.

***Implementing Ruby DSLs with* RDL.** Finally, RDL provides the ability to define new DSLs completely from scratch, adding implementations as well as contracts for keyword methods. This allows the user to implement DSLs in a more declarative fashion than is otherwise possible. Without RDL, writing a DSL in Ruby requires understanding of Ruby metaprogramming, including various forms of run-time code evaluation.

The DSLs we have examined implement their keywords as variations on this skeleton:

```
def keyword(args, &b)
  ... work to do before calling block ...
  o = create DSL object
  ret = o.instance_exec(args to block, &b)
  ... work to do after calling block ...
  ret
end
```

[2] In Ruby, calling class C either creates a new class C if it did not previously exist, or extends C with new definitions if it did previously exist.

```
29 print_spec = Spec.new do
30   spec :header do ... end
31   spec :table do ... end
32   ...
33 end
34
35 printable_spec = Spec.new do
36   spec :print do
37     arg 0, print_spec
38   end
39 end
40
41 class Team
42   spec : initialize do
43     dsl_from (Spec.merge player_spec, printable_spec)
44   end
45 end
46
47 class BaseballCardPrinter ... end
48
49 sox = Team.new "Boston Red Sox" do
50   ...
51   player "A.J. Pierzynski" { at_bats 166; hits 47; }
52   print BaseballCardPrinter.new
53 end
```

Figure 3: RDL Higher-order Contracts

That is, the code does some pre-processing, instantiates a DSL object (either from an existing class or by creating a new class on the fly), executes the block using instance_exec to bind the target object as self (possibly with some additional arguments to the block), and then does post-processing. For most examined keyword definitions, this code contains the only uses of metaprogramming features, and so abstracting it out allows the programmer to focus on what is different about their particular DSL, and frees them from needing to be versed in Ruby metaprogramming.

Figure 4 contains an RDL implementation of the baseball DSL. We begin by creating a class Stats to store the information associated with a player. In Stats, we mix in the RDL module to gain access to its methods. Then on lines 57–62 we use keyword to create a method hits, in contrast to spec which would only contract an existing method. In the block argument to keyword, we use action to provide an implementation of the method being created—in this case, assigning the argument to the instance variable @hits—and we can still use contracting keywords like arg and pre_cond as before. The full implementation defines at_bats (not shown) similarly. Since batting_average is not part of the DSL, we define it normally. Thus, it is clear which methods are part of the DSL, and this is enforced when a Stats object is used as a DSL.

To round out this DSL example, we need to implement Player and Team. We omit the definition of Player, as it is a straightforward class that implements an appropriate data structure for storing information about players. The definition of Team is on lines 72–94. Of particular note here are the semantics of dsl forms within keyword definitions and post_task, a new form. For keyword definitions, using dsl instead of action specifies a default implementation that creates an object for the requested DSL, runs the block argument under that object, and then returns the object. This allows the DSL implementer to return information from nested languages. The post_task form takes a code block that is given the return value and original arguments to the method as arguments and is executed *after* the main implementation. The return value for the code block is ignored, as it is only executed for effect. Our RDL library rewrites player so that the new player implementation first

```
54 class Stats
55   extend RDL
56
57   keyword : hits do
58     action { |x| @hits = x }
59     arg 0, Integer
60     pre_cond("More hits than at-bats!") { |h|
61       @at_bats ? h < @at_bats : true }
62   end
63   # similarly :at_bats
64
65   def batting_average()
66     (@hits / Float(@at_bats)).round(3)
67   end
68 end
69
70 class Player ... end
71
72 class Team
73   extend RDL
74   keyword : initialize do
75     pre_task { @players = [] }
76     dsl do
77       keyword : player do
78         arg 0, String
79         dsl Stats
80         post_task { |r, n|
81           p = Player.new n, r
82           @players.push p }
83       end
84       keyword : print do
85         action { |p| @printer = p }
86       end
87     end
88
89     post_task { |r, n|
90       @name = n;
91       @printer.instance_exec {
92         header "Players of #{n}"; table { ... }
93     } }
94   end
95 end
```

Figure 4: RDL Implementation of Baseball DSL

calls the original method and then calls the post_task block. RDL automatically transfers the arguments when a method is wrapped.

For player, on line 80 the post_task receives the Stats object, the return value of the DSL block, as the first argument r and the name of the player as the second argument n. It uses these two arguments to create an instance of Player. It then adds that Player object to the accumulator @players. For initialize, the post_task sets the team's name on line 90. It then runs a code block that contains printing instructions using the previously set @printer object. It does this using instance_exec, which takes a code block and runs it using the target of instance_exec as self.

3. Formal Model of Ruby DSL Contracts

In this section, we present λ_{DSL}, a formal model of Ruby DSL contracts layered on a simple object-oriented calculus extended with Ruby-like blocks. We then use the model to prove an erasure property: removing contracts from a well-behaved program does not cause spurious changes in behavior. The next section will discuss RDL's actual implementation.

Syntax. Figure 5 gives the syntax of λ_{DSL}. A value v is either a number n; a premethod $\lambda\overrightarrow{x}.e$, a function whose body can refer to its arguments or **self**; or an object o that maps method names to premethods. Other than values, expressions e also include variable names, **self**, sequencing, local bindings via **let**, and method calls.

$$
\begin{array}{lcl}
v & ::= & n \mid \lambda\overrightarrow{x}.e \mid o \\
\\
o & ::= & [m = \lambda\overrightarrow{x}.e, \ldots] \\
\\
e & ::= & x \mid v \mid \textbf{self} \mid e; e \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid e.m(\overrightarrow{e}) \\
 & \mid & e.\textbf{iexec}(e, \overrightarrow{e}) \mid \textbf{check}^{s}_{s}\ e\ \sigma \mid \textbf{blame}\ s \\
\\
\sigma & ::= & \varphi \mid \delta \mid \textbf{delay}(\delta) \\
\varphi & ::= & \textbf{any} \mid \textbf{has}(m) \mid (\varphi\ \textbf{and}\ \varphi) \\
\delta & ::= & \textbf{dsl}\ \{m : (\overrightarrow{\sigma}), \ldots\} \\
\\
p & ::= & \langle e, o, E\rangle \\
\\
E & ::= & \oslash \mid (C, o) \cdot E \\
C & ::= & \square; e \mid \square.m(\overrightarrow{e}) \mid v.m(\overrightarrow{v}, \square, \overrightarrow{e}) \mid \textbf{let}\ x = \square\ \textbf{in}\ e \\
 & \mid & \textbf{check}^{s}_{s}\ \square\ \sigma \mid \square.\textbf{iexec}(e, \overrightarrow{e}) \mid v.\textbf{iexec}(\square, \overrightarrow{e})
\end{array}
$$

Figure 5: Syntax of λ_{DSL}

Premethods in λ_{DSL} are invoked with **iexec**, whose name comes from Ruby's instance_exec, which, as seen earlier, is used to execute blocks of code with a particular self. In a call $e_0.\textbf{iexec}(e_1, \overrightarrow{e})$, e_1 is the premethod being invoked, $\overrightarrow{e}$ are the arguments passed to the premethod, and e_0 is bound to **self** within the call.

Contract checking in λ_{DSL} is expressed as $\textbf{check}^{s_p}_{s_n}\ e\ \sigma$, where e is the value to contract, σ is the contract, s_p names the provider of the value, and s_n names the client that receives the value. These last two strings are used to appropriately *blame* the party that fails to satisfy a contract [15]. Blame is represented as an expression **blame** s, where s is the name of the party at fault.

First order contracts (φ) check immediate properties of values. The simplest λ_{DSL} contract is **any**, which is satisfied by any value. The contract $\textbf{has}(m)$ checks that an object contains a method m (this is essentially a type contract in our formalism), and the intersection contract (φ_1 **and** φ_2) checks that the value satisfies both φ_1 and φ_2. It is straightforward to extend λ_{DSL} to include other standard contracts on values [15, 30]—e.g., to support preconditions as in Section 2—but we do not do so to keep λ_{DSL} simpler.

The last two contract forms are used for DSLs. A *DSL contract* (δ) maps method names to a sequence of contracts for the method arguments. Such a contract is satisfied by an object that implements the specified DSL, i.e., that has methods of the same names whose arguments also satisfy the given contracts. For example, in Figure 3, we invoked arg 0 print_spec to require the first argument of print satisfy the print_spec DSL. In λ_{DSL}, printable_spec corresponds to $\textbf{dsl}\ \{\text{print} : (\textbf{dsl}\ \{\text{header} : (\ldots), \text{table} : (\ldots), \ldots\})\}$.

The last contract form is $\textbf{delay}(\delta)$, which checks a premethod such that, when called, its **self** object must conform to δ. We call it **delay** because the δ contract is delayed until premethod invocation time. For example, in Figure 2, we created player_spec, which contains a method player that runs its block argument under the specified DSL. In λ_{DSL}, this contract corresponds (roughly) to $\textbf{dsl}\ \{\text{player} : (\textbf{delay}(\textbf{dsl}\ \{\text{hits} : (\ldots), \text{at\_bats} : (\ldots), \ldots\}))\}$.

Semantics. We give semantics to λ_{DSL} using a standard CK abstract machine [13] extended with a slot for **self**. Thus, program states for the machine (p) are of the form $\langle e, o, E\rangle$, where e is the current expression to reduce, o is the current **self** object, and E is the *context* under which the current expression is reduced. A context is either the empty context ($\oslash$) or a sequence of contexts beginning with a pair of the most recent context frame and the object bound to **self** in that frame.

$$\langle \textbf{blame}\ s, o, E\rangle \longrightarrow \langle \textbf{blame}\ s, o, \oslash\rangle \quad \text{(BLAME)}$$
$$\langle \textbf{self}, o, E\rangle \longrightarrow \langle o, o, E\rangle \quad \text{(SELF)}$$
$$\langle v; e, o, E\rangle \longrightarrow \langle e, o, E\rangle \quad \text{(SEQ)}$$
$$\langle \textbf{let}\ x = v\ \textbf{in}\ e, o, E\rangle \longrightarrow \langle e\,[x \mapsto v], o, E\rangle \quad \text{(LET)}$$
$$\langle o_t.m(\overrightarrow{e}), o_s, E\rangle \longrightarrow \langle o_t.\textbf{iexec}(\lambda\overrightarrow{x_m}.e_m, \overrightarrow{e}), o_s, E\rangle \quad \text{(METH)}$$
where $o_t = [\ldots, m = \lambda\overrightarrow{x_m}.e_m, \ldots]$
$$\langle o_t.\textbf{iexec}(\lambda\overrightarrow{x}.e, \overrightarrow{v}), o_s, E\rangle \longrightarrow \langle e\,[\overrightarrow{x \mapsto v}], o_t, E\rangle \quad \text{(IEXEC)}$$

$$\langle \textbf{check}^{s_p}_{s_n}\ v\ \textbf{any}, o, E\rangle \longrightarrow \langle v, o, E\rangle \quad \text{(ANY)}$$
$$\langle \textbf{check}^{s_p}_{s_n}\ n\ \textbf{has}(m_s), o_s, E\rangle \longrightarrow \langle \textbf{blame}\ s_p, o_s, E\rangle \quad \text{(HAS-ERR-NUM)}$$
$$\langle \textbf{check}^{s_p}_{s_n}\ (\lambda\overrightarrow{x}.e)\ \textbf{has}(m_s), o_s, E\rangle \longrightarrow \langle \textbf{blame}\ s_p, o_s, E\rangle \quad \text{(HAS-ERR-PRE)}$$
$$\langle \textbf{check}^{s_p}_{s_n}\ o\ \textbf{has}(m_s), o_s, E\rangle \longrightarrow \langle \textbf{blame}\ s_p, o_s, E\rangle \quad \text{(HAS-ERR-METH)}$$
where $o = [\ldots, m_o = \lambda\overrightarrow{x}.e, \ldots]$ and $m_s \notin \{..., m_o, \ldots\}$
$$\langle \textbf{check}^{s_p}_{s_n}\ o\ \textbf{has}(m), o_s, E\rangle \longrightarrow \langle o, o_s, E\rangle \quad \text{(HAS)}$$
where $o = [\ldots, m = \lambda\overrightarrow{x}.e, \ldots]$
$$\langle \textbf{check}^{s_p}_{s_n}\ v\ (\varphi_1\ \textbf{and}\ \varphi_2), o, E\rangle \longrightarrow \langle \textbf{check}^{s_p}_{s_n}\ (\textbf{check}^{s_p}_{s_n}\ v\ \varphi_1)\ \varphi_2, o, E\rangle \quad \text{(AND)}$$

$$\langle \textbf{check}^{s_p}_{s_n}\ n\ \delta, o, E\rangle \longrightarrow \langle \textbf{blame}\ s_p, o, E\rangle \quad \text{(DSL-ERR-NUM)}$$
$$\langle \textbf{check}^{s_p}_{s_n}\ (\lambda\overrightarrow{x}.e)\ \delta, o, E\rangle \longrightarrow \langle \textbf{blame}\ s_p, o, E\rangle \quad \text{(DSL-ERR-PRE)}$$
$$\langle \textbf{check}^{s_p}_{s_n}\ o\ \delta, o_s, E\rangle \longrightarrow \langle \textbf{blame}\ s_p, o_s, E\rangle \quad \text{(DSL-ERR-METH)}$$
where $\delta = \textbf{dsl}\ \{\ldots, m_s : (\overrightarrow{\sigma_s}), \ldots\}$ and $o = [m_o = \lambda\overrightarrow{x_o}.e_o, \ldots]$ and $m_s \notin \{..., m_o, \ldots\}$
$$\langle \textbf{check}^{s_p}_{s_n}\ o\ \delta, o_s, E\rangle \longrightarrow \langle \textbf{blame}\ s_p, o_s, E\rangle \quad \text{(DSL-ERR-ARGS)}$$
where $\delta = \textbf{dsl}\ \{\ldots, m : (\overrightarrow{\sigma}), \ldots\}$ and $o = [\ldots, m = \lambda\overrightarrow{x}.e, \ldots]$ and $|\overrightarrow{\sigma}| \neq |\overrightarrow{x}|$
$$\langle \textbf{check}^{s_p}_{s_n}\ o\ \delta, o_s, E\rangle \longrightarrow \langle \textbf{let}\ x_o = o\ \textbf{in}\ o', o_s, E\rangle \quad \text{(DSL-OBJ)}$$
where x_o fresh
and $\delta = \textbf{dsl}\ \{m_s : (\sigma_1, \ldots, \sigma_n), \ldots\}$
and $o = [m_s = (\lambda x_1, \ldots, x_n.e_s), \ldots, m_i = \lambda\overrightarrow{x_i}.e_i, \ldots]$
and $e'_s = \textbf{let}\ x_1 = \textbf{check}^{s_n}_{s_p}\ x_1\ \sigma_1\ \textbf{in}\ \ldots\ \textbf{let}\ x_n = \textbf{check}^{s_n}_{s_p}\ x_n\ \sigma_n\ \textbf{in}\ x_o.m_s(x_1, \ldots, x_n)$
and $o' = [m_s = (\lambda x_1, \ldots, x_n.e'_s), \ldots]$

$$\langle \textbf{check}^{s_p}_{s_n}\ n\ \textbf{delay}(\delta), o, E\rangle \longrightarrow \langle \textbf{blame}\ s_p, o, E\rangle \quad \text{(DELAY-ERR-NUM)}$$
$$\langle \textbf{check}^{s_p}_{s_n}\ o\ \textbf{delay}(\delta), o_s, E\rangle \longrightarrow \langle \textbf{blame}\ s_p, o_s, E\rangle \quad \text{(DELAY-ERR-OBJ)}$$
$$\langle \textbf{check}^{s_p}_{s_n}\ (\lambda\overrightarrow{x}.e)\ \textbf{delay}(\delta), o, E\rangle \longrightarrow \langle \lambda\overrightarrow{x}.(\textbf{check}^{s_n}_{s_p}\ \textbf{self}\ \delta).\textbf{iexec}(\lambda\overrightarrow{x}.e, \overrightarrow{x}), o, E\rangle \quad \text{(DELAY-PRE)}$$

Figure 6: Core rules

Figure 6 gives the core reduction rules for the abstract machine. The first group of rules, which form our object calculus base, are straightforward. Rule (BLAME) escapes from the current context, propagating the blame error to the top level. Rule (SELF) fetches the current **self** object out of the program state. Rule (SEQ) evaluates the first expression, then discards the result to evaluate the second. Rule (LET) binds a value to a local variable by substituting the value for the name in the body. Rule (METH) looks up the corresponding premethod for the given method name. Instead of evaluating the premethod directly, it delegates to Rule (IEXEC), which takes a premethod and evaluates its body, setting the target object o_t as the new **self**. This rule does not need to store the current **self**, since that is restored if necessary by the context popping rules.

The rest of the core rules describe how a **check** expression checks a contract against a given value. Rule (ANY) checks **any** contracts, which always succeed and produce the original value. Rules (HAS-ERR-NUM) and (HAS-ERR-PRE) catch the cases where a non-object is provided, Rule (HAS-ERR-METH) errors if the object does not contain the requested method, and Rule (HAS) encodes a successful contract check. Rule (AND) applies each contract to the desired value in turn, thus composing the contract checks appropriately.

The next set of rules check DSL contracts. The first two rules trigger an error when checking a non-object value. Rule (DSL-ERR-METH) triggers an error if object o does not have some method m_s listed in the contract. Rule (DSL-ERR-ARGS) triggers an error if object o has a method m listed in the contract such that the arity of m in the object and in the contract differ. Since all four error cases are due to the provider of the value, the blame string corresponding to the provider are used in the resulting blame error. For example, in $\textbf{check}^{p}_{c}\ []\ (\textbf{dsl}\ \{\mathsf{add} : (\ldots)\})$, p is blamed for the failure since it provided an object that does not contain an add method. Here, p stands for the blame information for the provider of the empty object, and c stands for the client that would receive it in the absence of the contract failure.

Finally, Rule (DSL-OBJ) applies a DSL contract to a suitable object o. Just as in higher-order contracts for functions, DSL contracts must be deferred until the methods of o are actually invoked. The rule achieves this by creating a proxy object o' containing all the methods m_s in the DSL contract, but *not* the methods m_i that appear in o but not in the contract. The body e'_s of each method m_s checks that each argument x_i matches the corresponding contract δ_i for that argument. If all checks pass, it delegates the call m_s to the underlying object, which is bound to x_o. Note that the arguments are checked using swapped blame, since the arguments are values coming from the client that invoked the method. That is, in the following expression:

$$(\textbf{check}^{p}_{c}\ [\mathsf{hits} = \lambda x.e_b]\ (\textbf{dsl}\ \{\mathsf{hits} : (\delta_h), ...\})).\mathsf{hits}(147)$$

the client can be seen as sending the value 147 through the contract check as an argument to the premethod implementation $\lambda x.e_b$ of the provider. If the client fails to provide a value that passes the corresponding contract δ_h, then the client should be blamed. Thus, the check in the new proxy object is $\textbf{check}^{c}_{p}\ 147\ \delta_h$, with the client as the provider of the value 147 and the server as its consumer.

The last three rules check higher-order DSL contracts (**delay**). The first two fail and blame the value provider if a non-premethod value is provided. Rule (DELAY-PRE) takes the premethod and

$$\langle e_1; e_2, o, E\rangle \longrightarrow \langle e_1, o, (\square; e_2, o) \cdot E\rangle \quad \text{(PUSH-SEQ)}$$
where e_1 is not a value
$$\langle v, o', (\square; e, o) \cdot E\rangle \longrightarrow \langle v; e, o, E\rangle \quad \text{(POP-SEQ)}$$

Figure 7: Example context rules

creates a new premethod that delays the DSL contract check. When executed, the new premethod first checks the DSL contract on **self** and then executes the original premethod on the resulting object. Since the client provides the **self** object, that is, the client receiving the premethod chooses on which object the premethod should be evaluated, the rule swaps the blame strings for the internal **check**. This meshes with the blame swapping for arguments in Rule (DSL-OBJ), as self is an implicit argument of the premethod.

Context Handling. The remaining abstract machine rules manage context frames. There are two types of context rules: those that drill down to find a reducible expression, pushing frames onto the context; and those that place values back into their proper context and restore the corresponding **self** object. Figure 7 gives an example of each type of rule. A corresponding rule for each expression type is needed. Rule (PUSH-SEQ) takes a state whose expression is a sequence that begins with a non-value. The resulting state is the head of the sequence, and the new context is the old context with an initial frame pushed that stores both the rest of the sequence and the current **self** object. Rule (POP-SEQ) is the inverse operation: if the state contains a value as its expression and the top frame contains the remainder of a sequence, then the new state has the sequence with the value as the new head as the expression to evaluate and the old **self** object restored.

3.1 Contract Erasure

Once we have a system of contracts for our language, programmers can use those contracts to describe their expectations about program behavior separately from the code that implements the behavior. However, we want to make sure that contracts do not alter the behavior of already correct programs in unexpected ways. We now prove this formally similarly to Findler and Felleisen [15].

First, we define a metafunction ERASE$[\![e]\!]$ that takes an expression e and removes all contract checks. We elide the definition of ERASE$[\![\cdot]\!]$ here for space reasons; it simply walks the expression and replaces every use of **check** with its first argument.

Second, we define a metafunction EVAL$[\![e]\!]$ that evaluates an expression and then either returns the result, for a number, or abstracts it to a result summarizing the kind of value or error:

$$\text{EVAL}[\![e]\!] = \begin{cases} n & \text{if } \langle e, [], \oslash\rangle \longrightarrow^* \langle n, [], \oslash\rangle \\ \text{block} & \text{if } \langle e, [], \oslash\rangle \longrightarrow^* \langle \lambda\overrightarrow{x}.e, [], \oslash\rangle \\ \text{obj} & \text{if } \langle e, [], \oslash\rangle \longrightarrow^* \\ & \quad \langle [m = \lambda\overrightarrow{x}.e, \ldots], [], \oslash\rangle \\ \text{blame: } s & \text{if } \langle e, [], \oslash\rangle \longrightarrow^* \langle \textbf{blame } s, [], \oslash\rangle \\ \text{error} & \text{if } \langle e, [], \oslash\rangle \longrightarrow^* p \text{ and} \\ & \quad \nexists p'.p \longrightarrow p' \end{cases}$$

We need this abstraction step because programs with contracts *do* actually produce syntactically different blocks and objects than non-contracted programs—recall rules (DELAY-PRE) and (DSL-OBJ), which η-expand or proxy calls and then add contract checks. Note that the evaluator is a partial function since the expression may not terminate.

Now we can state the following contract erasure theorem:

THEOREM 3.1. *For all e, if* EVAL$[\![e]\!]$ *is a number, "block," or "obj," then* EVAL$[\![e]\!]$ = EVAL$[\![$ERASE$[\![e]\!]]\!]$.

Proof sketch. We look at the trace of reductions for both the unerased and erased programs. We set up an approximation relation that relates values in the trace for the unerased program to values in the trace for the erased program. For most steps, this just relates values placed into corresponding contexts in the two programs. In the case of **check** reductions, though, the approximation maps the result of the **check** expression to the corresponding unaltered value in the erased program. Program states in the trace for the unerased program are then approximately equal to program states in the trace for the erased program if the values in the unerased state map to the corresponding value in the erased state.

Since operations on checked (proxied) values eventually reduce to the same operation on unchecked values, with possibly proxied arguments, we show our reduction relation respects this approximation, modulo the extra reduction steps introduced by contracted values in the unerased program. That is, the two programs reduce similarly except for the reduction of contracted values, which starts and ends with states approximately equal to the same state in the checked program. We use this fact as the basis for a stuttering equivalence between the two traces. We also show approximately equal values are equal under the EVAL metafunction. ■

$$\begin{array}{lcl}
\langle \text{CSpec}\rangle & ::= & \mathit{Class} \mid \mathit{Range} \mid \textsf{flat } \mathit{Proc} \\
& \mid & \textsf{and } \mathit{Contract}, \ldots \mid \textsf{or } \mathit{Contract}, \ldots \\
& \mid & \textsf{create_spec do } \langle \text{MSpec}\rangle^* \textsf{ end} \\
\\
\langle \text{MSpec}\rangle & ::= & \textsf{spec } \mathit{Symbol} \textsf{ do } \langle \text{SClause}\rangle^* \textsf{ end} \\
& \mid & \textsf{keyword } \mathit{Symbol} \textsf{ do } \langle \text{KClause}\rangle^* \textsf{ end} \\
\\
\langle \text{SClause}\rangle & ::= & \textsf{pre_cond } \mathit{String}?\ \mathit{Block} \\
& \mid & \textsf{pre_task } \mathit{String}?\ \mathit{Block} \\
& \mid & \textsf{post_cond } \mathit{String}?\ \mathit{Block} \\
& \mid & \textsf{post_task } \mathit{String}?\ \mathit{Block} \\
& \mid & \textsf{arg } \mathit{Nat}, \mathit{Contract} \\
& \mid & \textsf{opt } \mathit{Nat}, \mathit{Contract} \\
& \mid & \textsf{rest } \mathit{Nat}, \mathit{Contract} \\
& \mid & \textsf{ret } \mathit{Contract} \\
& \mid & \textsf{dsl do } \langle \text{MSpec}\rangle^* \textsf{ end} \\
& \mid & \textsf{dsl_from } \mathit{Contract} \\
\\
\langle \text{KClause}\rangle & ::= & \langle \text{SClause}\rangle \mid \textsf{action } \mathit{Block} \\
& \mid & \textsf{dsl } \mathit{Class}
\end{array}$$

Figure 8: RDL Contract DSL

4. Implementing RDL

In this section we describe our implementation of the formal model from Section 3.[3] The RDL library must perform three major tasks to check contracts: affirming that a DSL object contains the expected methods, checking uses of those methods against the corresponding method contract, and catching uses of non-DSL methods. We explain the first two by describing contract values in RDL and the last by illustrating how RDL checks a DSL during block evaluation.

Creating contracts. The implementation of the contracts in RDL follows the idea of contracts as pairs of projections [14]. Here, a projection maps values to similar values that behave the same modulo the addition of contract errors where appropriate. Figure 8 describes our contract language as a grammar. We use Kleene star to denote zero or more of the same item, and question mark to denote zero or one. Italicized terminals denote either block arguments or expressions that should return an object of the specified class. For example, *Symbol* denotes expressions that evaluate to symbols, and *Dsl* expressions that evaluate to a DSL contract value.

[3] Our implementation can be found at `https://github.com/plum-umd/rdl/tree/dsl_paper`.

The ⟨CSpec⟩ production describes our contract constructors. We allow some Ruby values, such as classes and ranges, to be used as contracts directly; internally they are converted to an appropriate projection. Any predicate procedure can be turned into a contract using the flat keyword, and the and and or keywords provide contracts that check values as being, respectively, in the intersection or the union of the listed contracts. Finally, create_spec creates a first-order contract that checks an object against a given DSL.

As we saw in Section 2, RDL also uses the keywords spec and keyword to add contracts to methods directly. Figure 8 shows these keywords (as ⟨MSpec⟩) and the full language allowed in their block arguments; ⟨SClause⟩ represents clauses allowed in uses of spec, and ⟨KClause⟩ represents clauses allowed in uses of keyword.

We have already seen almost all of RDL's features in section 2. The forms spec and keyword protect existing keywords or create new keywords, respectively. Inside spec, pre- and post-conditions can be given with pre_cond and post_cond. The forms pre_task and post_task are run before or after, respectively, the associated keyword, for their effect only. The form arg takes an argument number and applies the given contract to it, replacing the original argument. The opt form performs similarly, but allows the argument to be optionally provided. The rest form takes the number of the last required argument and applies the given contract to any subsequent arguments. The ret form applies the given contract to the return value of the keyword. The dsl form describes a DSL used in the block argument to the keyword. The dsl_from form also describes a DSL, but using an existing DSL contract value.

Inside of keyword, all the features of spec are available. Additionally, the user can supply either an action block given the implementation of the keyword or the form dsl $Class$ to indicate a fresh instance of that class should be the DSL for the keyword's block argument (as in Figure 4).

Contract Values. In addition to directly defining contracts in classes, RDL also supports first-class contract values. RDL::Dsl.new creates a new contract value and takes a block of method specifications using the same grammar ⟨MSpec⟩ defined above. Once a contract value is defined, it supports three operations: extension, merging, and application. The function RDL::Dsl.extend takes a contract to extend and a block of method specifications to add to that contract. Similarly, RDL::Dsl.merge combines multiple DSL contracts as we saw in Figure 3. When merged, method specifications for the same method are all applied in the merged contract, but only one keyword definition is allowed for a given name. Finally, RDL::Dsl.apply applies a DSL contract to a class.

When a DSL contract is applied, it first checks that the class has all the methods listed in spec clauses and that it does not contain any method listed in the keyword clauses. It then replaces the spec-described methods with shims that check the contract and call the original implementation. Any keyword-described methods are created with their action clause, if any, serving as their main body. If a keyword description contains a dsl clause that takes a class value like in ⟨KClause⟩, then the created method creates an instance of that class and executes the block argument on it. Any restrictions from {pre,post}_{cond,task} are then added either before or after the method is run. The pre_cond form, for example, becomes a check prior to calling the method that raises an error with the appropriate message if it fails.

The spec and keyword methods that can be mixed in by include RDL are implemented similarly. Their implementation can be thought of as taking a DSL contract that contains only that method contract and applying it to the current class.

Checking Contracts on Blocks. The dsl and dsl_from forms from ⟨SClause⟩ serve the same purpose as the **delay** contract in λ_{DSL}. Recall from (DELAY-PRE) from Section 3 that, when checking block contracts, we must delay the check on self until the block is called. In λ_{DSL}, blocks (a.k.a. premethods) are first-class values, but in Ruby, blocks are second-class values. Fortunately, Ruby allows blocks to be converted to instances of class Proc, and those instances are first-class. Thus, for dsl and dsl_from forms, RDL inserts code that creates a new Proc that serves as an η-expansion of the original block. This new Proc is passed to the original method instead of the original block.

When executed, this Proc first constructs a proxy that intercepts calls for the original self and adds delegates for each spec. Each delegate checks the clauses listed in the spec block and calls the original method as its main action. Analogously, uses of keyword create new methods in the proxy. To signal a contract failure when a non-listed method is called, the proxy also defines a method_missing method, which is Ruby's hook for handling calls to undefined methods. After constructing the proxy, the Proc then evaluates its block argument with the proxy bound to self.

5. Inferring DSL Contracts through Testing

To evaluate the expressiveness of RDL, we developed a pair of *contract inference* systems for discovering RDL contracts for existing DSLs. As mentioned earlier, these DSLs are currently implemented in complex ways that make static analysis difficult. Instead, we opt for a purely dynamic approach, using test cases to drive inference. We infer two kinds of contracts: first-order contracts involving types using TypeInfer, and structural contracts for lexical scoping using DSLInfer.

5.1 Contracts for Types

We use a simple approach to infer method types dynamically: we observe the values passed as parameters and returned as results, and record their classes. We then combine this information across all calls observed under a given test suite, and generate contracts that constrain types to at most those seen. More specifically, we contract the arguments (including optional and variable length arguments) and return types, as well as whether a block argument is allowed.

We implement this strategy by using RDL's pre_task to add the necessary hook to record the class of each argument and return, and whether a block was passed to the method. We use the following algorithm to combine information from multiple calls:

- The type of each argument position is the union of the observed types. If different invocations to the same method have different argument sizes, then we generate a contract for regular argument types for positions 0..n-1, where n is the smallest argument size. In this case we also add a contract on the "rest" argument restricting the type to be the union of all types observed in the other positions. If the size of the rest argument is 1, we change the contract on the rest argument type to a contract on the optional (default) argument type; we developed this heuristic because in our subject programs, there was at most one default argument.
- If a block argument is passed to all calls to the method, we add a contract that requires a block argument.
- We add a contract indicating the type of the return value is the union of all observed types of return values.

For a test suite, we use a set of *example programs* illustrating DSL usage, like the code in Figure 1. Most DSLs in Ruby include at least one such example, if not many.

5.2 Contracts for DSL Structure

Inferring DSL structural contracts is more complex. Since Ruby DSL implementations can avail themselves of the full power of Ruby, in theory the structure could be arbitrarily capricious. However, after examining a number of examples, we hypothesized that

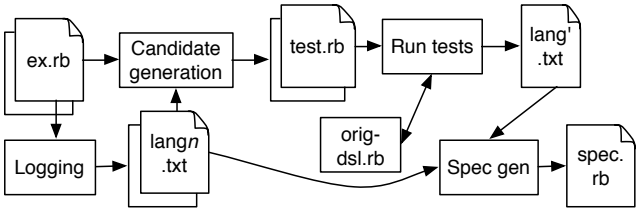


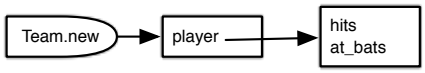
Figure 9: DSLInfer Architecture

```
1  D.entry do # self = A
2    outer1 do # self = B
3      inner1
4    end
5
6    outer2(0) { inner2(0) } # self = C
7    outer2(1) { inner2(1) } # self = C
8  end
```

Figure 10: Candidate generation example

in practice, most existing DSLs have a fairly simple structure: we assume DSLs can be described as a set of (possibly mutually recursive) RDL contracts without pre- and postconditions.

For example, given the actual Baseball DSL program in Figure 1, we want to be able to infer the DSL contracts in Figure 2, minus the pre_cond blocks and the type contracts. We can use a graphical representation of the baseball DSL contract to make the output of inference more intuitive:



Here the bullet-shaped node represents the method that enters the DSL (in this case, Team.new). The rectangular nodes represent RDL contracts for each sublanguage, e.g., the rightmost sublanguage allows calls to hits and at_bats. An arrow from a keyword k to a sublanguage l indicates l's keywords are in scope in the block argument to k. For example, hits and at_bats can be nested inside the player block.

We developed DSLInfer, a tool whose goal is to infer such a graph, i.e., to discover the language scopes, keywords, and nesting structure. Again, we use a dynamic approach. We begin with the same test suite as in Section 5.1, but we immediately face a problem: in our experience, while these examples provide useful type information, they often do not capture the full DSL structure.

Thus, DSLInfer systematically generates from the test suite a set of new candidate examples that may or may not be in the DSL. We then use the actual DSL implementation as an oracle to determine which candidates are valid. From the original examples and the valid candidates, we create an RDL specification.

The key to making the inference process efficient is to limit the number of candidates. DSLInfer exploits an observation we made of the DSL usage examples: while they may not show every possible keyword placement, they typically do use every keyword at least once. Thus, DSLInfer works by shuffling around keyword calls in the examples to determine if those keywords can appear in other places. Moreover, since those keywords are actually methods defined in classes, DSLInfer uses class information to decide which keywords might be in scope.

DSLInfer *Architecture.* Figure 9 illustrates DSLInfer in more detail. The first step of DSLInfer is to discover what classes contain each keyword, and in which block arguments those classes are bound to self. We extract this information dynamically by running the examples with inserted *logging* code. First, we manually combine the example test programs into a single file via concatenation (not shown).[4] DSLInfer uses the Ruby Intermediate Language (RIL) [19], a tool designed for exactly this kind of use, to parse in the example programs and replace each call m(args...) without an explicit receiver (i.e., a potential keyword call) with a call to dsl_log(:m, stmt_id, args...). Here, m is the method name, stmt_id is the unique RIL statement id, and args are the original parameters. When called, dsl_log records its arguments, current class and call stack information, and then calls m.[5]

The output of this process is a series of files lang$n$.txt, where each file contains four pieces of information about a single keyword k that runs its block in a DSL:

- method: k and the class that contains k,
- lang: the class of objects k uses to evaluate block arguments,
- used_methods: the methods used in block arguments to k, and
- lang_methods: the methods used in block arguments to all keywords sharing the same DSL (i.e., lang).

From this, we determine *candidate keywords* that might potentially appear in k's block argument, but did not in the examples. To do this, we first collect all the keywords in lang_methods and the methods in the lang class that appear in the lang_methods for any class that shares a common ancestor with lang. From that set, we remove all the keywords that appear in used_methods. The remaining keywords are our candidate keywords. Thus, for each candidate k', we now have a set of *keyword pairs* $k \rightarrow k'$, where k' is a candidate keyword that potentially could appear in the block argument to k (here, the arrow corresponds roughly to an edge in the graph representation of a contract).

Next, for each keyword pair $k_1 \rightarrow k_2$, DSLInfer copies one existing use of k_2 into a block of k_1; below, we illustrate the process and explain the choice of which k_1 and k_2 instances to use. Note we only construct one test case for each candidate keyword, even if multiple calls to the keyword are observed in the original example—this helps reduce the number of candidates. When DSLInfer copies a call to k_1, it also must copy any statements that the arguments depend on. The translation to RIL normalizes the code so that all method arguments are variables. Thus, DSLInfer also needs to copy the latest assignment $x_i = e_i$ to each variable argument x_i. If e_i contains uses of variables, those variables are copied and the process is repeated. More complex dependencies, e.g., involving block arguments, are currently ignored.

Candidate Generation Example. We illustrate the candidate generation process by example. Consider the DSL usage example in Figure 10. At the top-level, the call D.entry invokes the DSL. During that call, self is an object of class A while the block argument is evaluated. That block uses two keywords, outer1 and outer2. Inside outer1's block argument, self is an object of class B and the keyword inner1 is used. Inside outer2's block argument, self is an object of class C and the keyword inner2 is used.

If A, B, and C are all different classes, then there are no candidate keywords to generate—the only blocks that use the same class are the arguments to outer2, and they both use the same keyword, inner2.

[4] No example uses side effects such that concatenation would be unsafe.

[5] Note that not all receiver-less calls correspond to keywords, e.g., the printing method puts from the Kernel module, whose methods are mixed into Object as private methods. To eliminate such calls from being considered keywords, dsl_log ignores calls to private methods.

Now suppose that $B = C$. In this case, DSLInfer identifies two candidate keywords—inner2 could potentially appear within the block argument of outer1, and inner1 could potentially appear within the block argument of outer2. Thus, DSLInfer generates two candidate example programs. Each candidate takes one of the inner keyword uses and copies it to a block argument of the other outer keyword. Notice that there are two possible calls to inner2 that could be copied. DSLInfer chooses one such call at random among all calls whose dependencies can be fully resolved, Similarly, if DSLInfer has multiple possible blocks to copy into like with outer2, it chooses one at random.

Furthermore, suppose that $A = B = C$. Now, in addition to the previous candidates, there are four more: each outer$i$ could appear within the other outer$j$ or within itself (recursively). To generate these candidates, DSLInfer copies the entire call, e.g., it copies line 7 just before line 3. For the recursive cases, DSLInfer only copies a block within itself once. For example, lines 2–4 are copied to just before line 3 to form one candidate.

Specification Generation. Now that DSLInfer has candidate programs, it runs them by invoking the actual DSL implementation, and tests whether they execute successfully or raise an exception. For each passing test, it concludes the keyword pair added is valid and records that fact in lang$'$.txt. DSLInfer then combines the language description in the original examples, lang$n$.txt, with lang$'$.txt to calculate the graph structure of the DSL.

Once DSLInfer has the inferred graph structure of a given DSL, it is straightforward to translate that graph to a skeleton RDL specification that contains only structural information about the DSL. In that specification, DSLInfer creates a DSL contract for each language, where there is a spec contract for each keyword in the language. For keywords that run a block argument in a different DSL, the spec's block contains a call to dsl_from with the contract for the target language, otherwise the block is empty. After the DSL contracts are created, the specification then reopens each class containing an entry method and adds a spec contract for that entry method that calls dsl_from with the appropriate DSL contract.

6. Experimental Results

We ran contract inference on examples for eight Ruby DSLs to infer method types and structures. We selected the DSLs by starting with several Ruby gems (i.e., packages) that we use regularly, like Routing and Backup. We also searched for additional Ruby DSLs; in particular, ones that work with the latest version of Ruby with no external dependencies (e.g., no need for resources like an Amazon cloud account). In the end, we found these subject programs:

- *Backup*, a DSL for describing backup tasks;
- *Cardlike*, a DSL for developing and testing card games;
- *Graph*, a DSL for producing Graphviz dot documents;
- *Osheet*, a DSL for creating and specifying spreadsheets;
- *Rose*, a DSL for making report tables;
- *Routing*, the Ruby on Rails routing language, a DSL for mapping URLs to controller methods that handle requests;
- *StateMachine*, a DSL for finite state machines; and
- *Turtle*, a DSL for creating Resource Description Framework (RDF) documents.

For each DSL, we either combined the example programs included with the DSL into one example program or, if there were no such programs, constructed our own example from the DSL's documentation. We ignored certain example programs where methods are defined dynamically; we discuss this more later.

DSL	S	$\cup$	$\{\cdot\}$	$*$	o	F1	F2	F3	F4
Backup	8		✓			✓		✓	
Cardlike	7		✓			✓		✓	
Graph	23		✓	✓	✓	✓			✓
Osheet	26	✓	✓	✓	✓	✓	✓	✓	✓
Rose	11	✓	✓		✓	✓		✓	
Routing	11	✓	✓		✓	✓			✓
StateMachine	9	✓	✓		✓	✓	✓	✓	
Turtle	8	✓	✓	✓	✓	✓	✓	✓	✓

S = number of methods with contracts
$\cup$ = union types, $\{\cdot\}$ = block types, $*$ = vararg types, o = optional arg

Figure 11: Type Contract Inference Results Summary

6.1 Types

We ran the DSL method type contract inference on the same keywords used in the DSLInfer experiment. Figure 11 summarizes the results. We do not report performance because the overhead is negligible, as the DSLs are typically only run once during an execution.

For each DSL, the middle group of columns lists the number of DSL methods with inferred contracts, followed by columns indicating whether various type contract features occurred for that DSL: union types, block argument presence, varargs, and optional types. As expected based on reports for previous Ruby programs [27], these features are present across many examples, and the algorithm infers a wide range of useful types.

We then manually examined the example programs and the output of our contract inference to see where it discovers less precise information than possible. The rightmost column reports on the result. We found four different categories where our lightweight contract inference failed in some sense, as follows.

F1 - Misses Argument Types, F2 - Misses Return Types. In some test cases, the example programs only observe one type used as an argument or return, but other types are valid. This happened at least once in every program. One common case is only observing String or Symbol for an argument, but this argument can be any type with to_s defined on it because the method argument is too permissive.

F3 - Incorrectly Infers Block Argument is Required. We generate a contract that requires a block argument if all calls for the same method contain a block argument, but this may not be the case. The source code may have default behavior when no block is given or may not use the block on all paths. This was also a very common source of imprecision.

F4 - Incorrectly Infers Default Arguments / Variable Length Arguments. This category includes imprecision due to incorrectly inferring where the vararg starts, or inferring a default argument as a vararg or vice-versa. For example, suppose we have method definition foo(*args) with observed runs foo(1) and foo(1, 2). Then we will imprecisely infer that foo has a regular parameter at position 0 and a default parameter at position 1. In general, this category arises because contract inference looks only at the number of arguments passed to the method.

A more effective implementation can fix this type of error by using Ruby's parameter information and actual argument positions to match each argument with its corresponding formal parameter.

Discussion. In principle, the first three sources of errors can be eliminated when the example programs have complete path and type coverage. Thus, one potential approach to reduce these errors would to automatically generate additional test cases, like DSLInfer, to increase the number of method calls seen. However, it seems non-trivial to find argument values of the right type to make a method run successfully without having any prior knowledge.

DSL	Example LoC (#)	Init. Pairs	# Cands. Pass	# Cands. Fail	Final Pairs	Run Time	Miss. Pairs
Backup	39 (1)	6	·	·	6	1.28s	·
Cardlike	22 (1)	6	·	·	6	1.79s	·
Graph	126 (5)	27	16	1	43	25.1s	·
Osheet	644 (6)	27	11	112	38	9m49s	4
Rose	157 (1)	11	1	·	12	2.23s	·
Routing	55 (1)	15	23	12	38	2m33s	·
StateMach.	101 (4)	10	2	3	12	3.21s	3
Turtle	72 (5)	10	25	1	35	1.59s	1

Figure 12: DSLInfer Results Summary

6.2 DSL Structures

Effectiveness of Structural Inference. Figure 12 numerically summarizes the effectiveness of structural inference. Recall that a keyword pair $k_1 \rightarrow k_2$ indicates keyword k_2 may be used in the block argument of k_1. The second column in the table lists the total lines of code in the example programs with the number of example programs in parentheses. Most of the example programs are fairly small, ranging from 22 to 157 lines of code for all initial programs combined, with the exception of *Osheet*, whose example programs have 644 lines of code. The table also lists the number of keyword pairs found in the initial example program for the DSL, the number of passing and failing candidate programs for possible keyword pairs not found in the initial example programs, the total number of valid keyword pairs found by DSLInfer, and the running time of DSLInfer on that DSL—this includes the time to generate the test cases and to run them. The last column reports the number of missed pairs, i.e., those tested for by a failed candidate program, yet there exists some other program containing the pair that would succeed. We describe these cases in more detail below. Note that some DSLs have no failed candidate programs, and such DSLs do not contain any missed keyword pairs.

Overall, even with our unsophisticated candidate generation strategy, many candidate programs passed; this shows the tremendous flexibility in most Ruby DSLs. Our results also show that structural inference is highly effective for these examples. Our system inferred valid keyword pairs not covered by the original example programs in 6 of the 8 DSLs; the other two were completely covered by the original example programs. For a few of these DSLs, our system inferred a significant number of valid additional keyword pairs with a small number ($< 10\%$) of false negatives. For instance, in *Routing*, we took an example program that contained 15 keyword pairs and inferred 23 additional valid keyword pairs.

In addition, the running time for DSLInfer is generally low, with most experiments taking less than four seconds. *Osheet* takes the most time. The original example programs for *Osheet* contain many small uses of the DSL, and test only a few of all the possible combinations. In addition, all DSL methods are defined in one class. This means that there are a large number of possible pairings to try, so candidate generation is slow. *Routing* also takes a lot of time, because each test run requires loading the entire Rails framework, which is slow.

Inferred DSL Contracts. Here, we look at DSLInfer's results in more detail. Figure 13 shows three example contracts that were inferred for subject languages *Graph*, *Osheet*, and *Routing*. These are some of the larger DSLs, and they are still fairly small and simple. There are fewer than a dozen keywords in most DSLs (*Graph* is an outlier with 23 keywords, elided in the figure), and most DSLs have relatively few sublanguages. We next examine the graph for each DSL in turn, and then discuss the remaining DSLs.

Figure 13a shows the structure of *Graph*. This DSL contains one language with an entry method digraph that can contain nested calls to the DSL via subgraph. The example programs generated by our system show that digraph can contain calls to all the keywords, and subgraph can contain calls to all but save. The candidate program containing subgraph → save is the only failed candidate program in this DSL. This is not a missed pair because one of the dependent programs generates a syntax warning when this pair is used. Our inference splits the language into two sets of keywords that differ only in containing save. Of particular note is that subgraph can also contain nested subgraph calls.

Figure 13b shows the structure of *Osheet*. Investigating *Osheet* in more detail, we found it has some *context sensitivity*, meaning that whether or not a given keyword pair is valid depends on either ordering or argument values. We found two categories of context-sensitivity. The first category is keywords whose nesting behavior depends on their arguments. For example, worksheet has distinct behaviors depending on whether its regular and block arguments are both empty. In the graph, we represent these two distinct behaviors with worksheet and worksheet′. Recall that we only generate one candidate program for each missing keyword pair, no matter how many times the keywords are used in the original test program. In this DSL, DSLInfer luckily picked the right nested worksheet occurrences to copy and the right target locations so that the resulting candidate program did not fail.

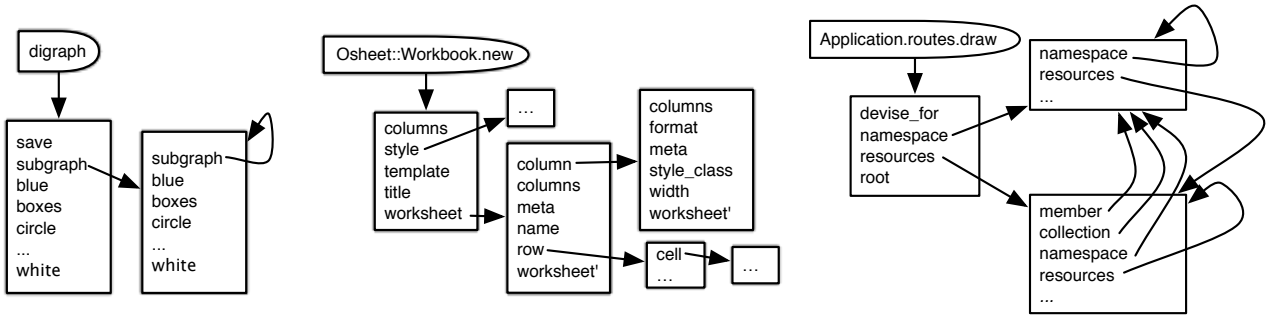
The second category is context-sensitive keywords that must appear before or after certain other keywords in the same block. While RDL can express these restrictions by appropriate preconditions, DSLInfer is not designed to discover them. *Osheet* has 112 failed programs, and 4 out of the failed programs missed keyword pairs. The failures included exception of the following forms: keyword method called from a non-nil object, indicating the candidate program did not miss any keyword pairs; keyword method called from a nil object; and undefined variable names caused by variable dependencies DSLInfer could not resolve.

Figure 13c contains a graph showing the partial inferred structure of the Rails routing language.[6] The routing language is particularly interesting because all the keywords are actually implemented as methods of the same class ActionDispatch::Routing::Mapper. However, the contract structure shows that particular keywords can only appear within the block arguments to certain other keywords. For example, most keywords, like resource, can contain any nested keyword (as shown in the contract in the lower-right of the figure), but collection, member, and namespace can contain any keywords *except* collection and member (as shown in the contract in the upper-right of the figure). Thus, we see that RDL allows the contract structure and the implementation structure to differ if needed. Also, in this particular case, the failed candidate programs contained exact error messages explaining why the candidate keyword pair was invalid. For example, "can't use member/collection outside resource(s) scope", and "missing :controller". Thus, we know the failed programs do not contain any missed keyword pairs.

We now discuss the remaining DSLs; the structure of these DSLs are similar to the ones we have seen, so we omit their graphs.

The *Turtle* DSL contains only one language with four entry methods for that language. Other than the one failed candidate program, DSLInfer generated example programs showing all DSL methods can be nested within any of the entry points. Recall that DSLInfer only runs one example program for each new potential edge pair for performance reasons. For this reason, we missed the valid edge predicate → subject because DSLInfer picked the subject call that passes a block with dependencies, since DSLInfer does not catch dependencies in block arguments.

[6] For those familiar with Rails, one surprising entry here is devise_for, which is not a standard method. However, the example we used came from a Rails app that uses the devise gem, which adds this keyword.



(a) Graph Spec Structure (b) Osheet Spec Structure (partial) (c) Routing Spec Structure (partial)

Figure 13: Example Inferred DSL Contracts

The *Rose* DSL uses inheritance in an interesting way: the class Proxy::Row, which implements the DSL used by the keyword rows, is a superclass of the class Proxy::Summary, which implements the DSL used by the keyword summary. Thus, all keywords in the former can be used in the latter. One such keyword, identity, is never used within summary in their examples, but DSLInfer is able to discover that pair.

In *StateMachine*, DSLInfer discovered two new keyword pairs. There are 3 failed candidate programs for this DSL, all due to invoking a method on nil. For each failed program, we constructed a new successful candidate that copied the original candidate, hence all 3 failed candidate programs contained missed keyword pairs.

The last two DSLs, *Backup* and *Cardlike*, do not generate any new example programs because their example programs include all possible keyword pairs.

Note that our study concerned only structural contracts. While DSLInfer cannot infer contracts involving certain types of structural contracts with context sensitivity, RDL can describe arbitrary contracts using pre_cond, post_cond, etc. These pre- and post-conditions have access to the full power of Ruby, and hence contracts can perform arbitrary reasoning and throw appropriate exceptions to signal contract failures. As future work, we plan to infer other kinds of contracts in addition to structural and type contracts.

6.3 Threats to Validity

There are several threats to the validity of our experimental results. One expected threat is whether our benchmark is representative; to mitigate this concern we selected eight DSLs from a variety of sources. The other main threats are:

Coverage Limitations. Type contract and structural inference operate only on methods in the examples we used, but these programs may involve only a subset of the DSL methods. Addressing this issue is difficult, because it involves divining programmer intent—DSL implementations often include many helper methods not intended for client use, and it is non-trivial to distinguish DSL methods from helper methods.

Another coverage limitation is that DSLInfer does not currently support certain classes of non-alphanumeric method names, such as [] for indexing. This caused us to miss exactly two keyword names across all eight DSLs.

Moreover, our testing process did not generate candidates that involved dynamically defining new keyword methods. For example, Cardlike has DSL keywords that, during execution, define additional keywords based on the provided arguments. These kinds of keywords can be supported by RDL using appropriate post_task calls, but we are unable to infer contracts involving them.

Context Sensitivity. In the DSLs we examined, DSL keywords can usually appear in any order in a block. However, as mentioned earlier, DSLInfer does miss valid edges in the rare cases when orderings or arguments are important.

Dependencies. As described above, DSLInfer only copies dependencies that are within the same block as the keyword call, not those from outer lexical scopes. In addition, dependencies used by the block argument are not resolved by DSLInfer, which could have caused one missed keyword pair in *Turtle*.

7. Related Work

Domain-Specific Languages. Languages provide support for DSLs in a variety of ways. Lisp [20] has a simple syntax that translates directly to data values, which can then be manipulated and executed. Scheme [9, 11, 29] and Racket [16–18] refine Lisp's technique by separating code from data and using that separation to provide additional features like "hygienic" handling of bindings. Tcl [24] allows the programmer to mark a given code block as uninterpreted, which reifies it as a string that can be manipulated and then interpreted as code at run time. In R [32], programmers create DSLs using a combination of lazy evaluation and first-class environments [35]. Mython [28] provides a front-end that can perform Lisp-style AST transformations on code before handing it over to the normal Python system for evaluation. Haskell DSLs [21, 22] typically use a *pure* embedding, that is, needing no preprocessor or macro-expander to translate the DSL down to the base language; monads provide encapsulation to separate the DSL from the base language. Leijen and Meijer [23] propose Haskell DSLs where the implementation of the embedded DSL serves as a compiler, rather than an interpreter. Discussions of many other DSLs can be found in an annotated bibliography [10].

To our knowledge, DSLs in Ruby have not previously been studied in the literature. We are also unaware of prior efforts to support contracts on DSLs.

Proxies. Both Strickland et al. [31] and Austin et al. [4] present systems for proxying primitive values and interposing on their operations in JavaScript and Racket, respectively. Method shims in our implementation are inspired by Strickland et al.'s description of chaperones and impersonators for functional values.

Specification and type systems for dynamic languages. Ren et al. [27] present a dynamic type checking system for the Ruby programming language. Their system handles normal type information about classes, methods, and values, but it cannot describe more complex interactions like the language available within the block argument to a DSL keyword.

We are aware of only one prior system that implements general-purpose contracts for Ruby [7], but it similarly has no special support for DSLs.

RubyDust [1] uses a constraint-based analysis at run-time to infer type information. In contrast to the simple type contracts we infer for RDL, RubyDust types are structural, e.g., RubyDust can infer that a method accepts any object with a `to_s` method, something that led to one category of imprecision in contract inference in Figure 11. However, RubyDust is also much more heavyweight—its performance overhead is significant, whereas our type contract inference has virtually no overhead. Additionally, RubyDust only works with an older version of Ruby.

Beyond Ruby, researchers have also explored type systems for other object-oriented dynamic languages such as Python [2, 5] and JavaScript [3, 34].

8. Conclusion

In this paper, we studied the problem of enforcing contracts for Ruby DSLs. We develop a formal model of Ruby DSLs, their specification and checking, and blame tracking. Our model supports first-order DSL contracts, including those for types; higher-order DSL contracts; and satisfies a contract erasure theorem. We have implemented our model as RDL, a DSL contract checking system for Ruby. The RDL library allows programmers to specify the behavior of existing DSLs and to create new DSLs using an abstraction layer over Ruby's metaprogramming facilities. To explore the applicability of RDL, we developed first-order type contract inference with TypeInfer and structural contract inference with DSLInfer. We ran contract inference on a number of examples, and found that it is able to successfully infer accurate specifications in a reasonably short amount of time.

Acknowledgments

We would like to thank the anonymous reviewers for their comments. This research was supported in part by NSF CCF-1116740 and NSF CCF-1319666.

References

[1] An, J., Chaudhuri, A., Foster, J.S., Hicks, M.: Dynamic Inference of Static Types for Ruby. In: Principles of Programming Languages (POPL). pp. 459–472 (2011)

[2] Ancona, D., Ancona, M., Cuni, A., Matsakis, N.: RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In: DLS 2007. pp. 53–64 (2007)

[3] Anderson, C., Giannini, P., Drossopoulou, S.: Towards Type Inference for JavaScript. In: European Conference on Object-Oriented Programming. LNCS, vol. 3586, pp. 428–452 (2005)

[4] Austin, T.H., Disney, T., Flanagan, C.: Virtual values for language extension. In: Object-Oriented Programming, Systems, Languages, and Applications. pp. 921–938 (2011)

[5] Aycock, J.: Aggressive Type Inference. In: International Python Conference (2000)

[6] Bentley, J.: Programming pearls: little languages. Communications of the ACM 29(8), 711–721 (Aug 1986)

[7] Bhargava, A.: contracts.ruby (2012), https://github.com/egonSchiele/contracts.ruby

[8] Claessen, K., Ljunglöf, P.: Typed logical variables in Haskell. In: Proceedings of the 2000 Haskell Workshop (2000)

[9] Clinger, W., Rees, J.: Macros that work. In: Symposium on Principles of Programming Languages. pp. 155–162 (1991)

[10] van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. SIGPLAN Not. 35(6), 26–36 (Jun 2000)

[11] Dybvig, R.K., Hieb, R., Bruggeman, C.: Syntactic abstraction in Scheme. LISP and Symbolic Computation 5(4), 295–326 (1993)

[12] Elliott, C.: Programming graphics processors functionally. In: Proceedings of the 2004 Haskell Workshop (2004), http://conal.net/papers/Vertigo/

[13] Felleisen, M., Friedman, D.P.: Control operators, the SECD-machine, and the λ-calculus. In: Formal Description of Programming Concepts III. pp. 193–217 (1986)

[14] Findler, R.B., Blume, M.: Contracts as pairs of projections. In: International Conference on Functional and Logic Programming. pp. 226–241 (2006)

[15] Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: International Conference on Functional Programming. pp. 48–59 (Oct 2002)

[16] Flatt, M.: Composable and Compilable Macros: You Want it *When*? In: International Conference on Functional Programming. pp. 72–83 (2002)

[17] Flatt, M., Culpepper, R., Darais, D., Findler, R.B.: Macros that Work Together. Journal of Functional Programming 22, 181–216 (2012)

[18] Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Inc. (2010), http://racket-lang.org/tr1/

[19] Furr, M., An, J., Foster, J.S., Hicks, M.: The Ruby Intermediate Langauge. In: Dynamic Languages Symposium (DLS). pp. 89–98. Orlando, Florida (October 2009)

[20] Guy L. Steele, Jr.: Common LISP: the Language, 2nd Edition. Digital Press, Newton, MA, USA (1990)

[21] Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys 28 (1996)

[22] Hudak, P.: Modular domain specific languages and tools. In: in Proceedings of Fifth International Conference on Software Reuse. pp. 134–142. IEEE Computer Society Press (1998)

[23] Leijen, D., Meijer, E.: Domain specific embedded compilers. In: Conference on Domain-specific Languages. pp. 109–122 (1999)

[24] Ousterhout, J.K., Jones, K.: Tcl and the Tk Toolkit, 2nd Edition. Addison-Wesley Professional, Boston, MA, USA (2009)

[25] Perrotta, P.: Metaprogramming Ruby. Pragmatic Bookshelf (2010)

[26] Reid, A., Peterson, J., Hager, G., Hudak, P.: Prototyping real-time vision systems: an experiment in dsl design. In: International Conference on Software engineering. pp. 484–493 (1999)

[27] Ren, B.M., Toman, J., Strickland, T.S., Foster, J.S.: The Ruby Type Checker. In: Symposium on Applied Computing. pp. 1565–1572 (2013)

[28] Riehl, J.: Language embedding and optimization in Mython. In: Symposium on Dynamic Languages. pp. 39–48. DLS '09 (2009)

[29] Sperber, M., Dybvig, R.K., Flatt, M., Van Straaten, A., Findler, R., Matthews, J.: Revised^6 Report on the Algorithmic Language Scheme. Journal of Functional Programming 19, 1–301 (2009)

[30] Strickland, T.S., Felleisen, M.: Contracts for first-class classes. In: Symposium on Dynamic Languages. pp. 97–111 (Oct 2010)

[31] Strickland, T.S., Tobin-Hochstadt, S., Findler, R.B., Flatt, M.: Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In: Object-Oriented Programming, Systems, Languages, and Applications. pp. 943–962 (2012)

[32] Team, R.C.: R Language Definition. http://cran.r-project.org/doc/manuals/r-release/R-lang.html

[33] Thiemann, P.: Modeling HTML in Haskell. In: International Workshop on Practical Aspects of Declarative Languages. pp. 263–277 (2000)

[34] Thiemann, P.: Towards a Type System for Analyzing JavaScript. In: European Symposium on Programming. LNCS, vol. 3444, pp. 408–422 (2005)

[35] Wickham, H.: Advanced R programming (Nov 2013), http://adv-r.had.co.nz/dsl.html