CS7480 Project Report

Strictness Analysis for R

Aviral Goel

9th December, 2016

Table of Contents

Acknowledgements

I would like to thank Dr. Jan Vitek for pointing out this problem to me. I had numerous discussions with him on solving this problem. I would also like to thank Olivier Flückiger and Benjamin Chung for stimulating discussions.

I would also like to show my gratitude to the instructor for this course, Dr. Frank Tip, for many useful insights, especially related to the correctness of this analysis and for encouraging me to continue working on this.

The R Programming Language

R is a dynamically typed domain specific language for statistical computing and graphics. Used by over 2 Million people worldwide, it has become a de facto standard for data mining and data analysis. R was designed as an open source alternative to S and has been continuously evolved to meet the demands of its users. R is

- Lexically Scoped
- Dynamically Typed
- Functional
- Object Oriented

The Problem

R evaluates function arguments lazily. Unevaluated arguments are wrapped in a data structure called Promise. The Promise also contains a reference to the calling context - the environment in which the argument has to be evaluated. When the formal parameter is used in a context where its value is needed, the Promise is *forced*. Forcing a Promise triggers the evaluation of the unevaluated argument in the calling context. Once evaluated, the result is memoized by the promise so that subsequent uses do not trigger a reevaluation. Promises are handled transparently by the language and are not visible to the R programmer.

Lazy evaluation has the potential to reduce work done by the interpreter. At runtime, certain control flow paths will not be taken, avoiding evaluation of certain arguments. However, research[1] indicates that this is rarely the case. On an average, 80% promises are evaluated by the function they are passed to and up to 99% promises can be evaluated in computationally intensive benchmarks. This makes promises redundant. Unfortunately, promises also:

- take up more memory which stresses the garbage collector.
- add a layer of indirection as the data structure has to be checked to know if it has already evaluated the argument and this can increase chances of cache misses.

[1] F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the design of the R language. In J. Noble, editor, ECOOP 2012 Object-Oriented Programming, volume 7313 of Lecture Notes in Computer Science, pages 104–131. Springer Berlin/Heidelberg, 2012

The Solution

One way to solve the problem is to rewrite the bytecode to prevent creation of promises when arguments are dispatched to functions. Now the arguments will be evaluated eagerly within the calling context and the result will be passed to the callee. No promises will ever be created. But this naive implementation will change program behavior. It's possible for the argument to be a side effecting expression. `print` function in R prints its argument to the console and returns it. By eagerly evaluating it, as a function argument, we can make the program print something to the console, which will otherwise not happen if the promise is not forced in the function. So, the naive approach will simply not retain the language semantics.

The Challenges

Removing promises while ensuring that the program behavior remains unchanged makes the problem hard. I have identified the following challenges in retaining language semantics:

- **Side Effects**: Function arguments can have side effects. If promises are evaluated eagerly, same side effects should be observed in same order.
- **Language Builtins**: R is a functional programming language and most of the core language features are implemented as functions. Loops take a block of code as a promise and evaluate it repeatedly. Conditionals are implemented as functions. The analysis will have to make special provisions to ensure that promises created by these “special” functions are not evaluated eagerly.
- **Metaprogramming and Reflection**: R sports an extreme form of dynamism. Environments can be introspected and definitions can be modified out of scope at runtime. `eval` can be used to evaluate abstract syntax trees which can make arbitrary changes to the state of the program. This behavior cannot be captured statically.

The Project

The goal of this project is to apply a static analysis technique to identify which arguments of a function can be evaluated eagerly and in which order. This information is referred to as the strictness state of a function.

The scope of this project is limited as follows:

- The bytecode will not be rewritten to eagerly evaluate the arguments.
- Reflection, metaprogramming and other dynamic features will not be taken into account.
- The language builtins relying upon promises will not be accounted for. This is relatively easy to accomplish as the relevant information can be retrieved programmatically from the interpreter.
- All functions are assumed to take the same arguments in the same order.
- The call graph can be determined statically.

The Algorithm

I have implemented a **flow-sensitive** and **path-insensitive CFA-0 interprocedural algorithm** based on **abstract interpretation**. The implementation has two phases which are described below.

Phase I - Intraprocedural Analysis

The intraprocedural analysis phase implements abstract interpretation of the bytecode of R functions. It simulates the stepwise execution of the function's bytecode and collects facts about argument evaluation in the strictness state. It completely ignores the effect of callees but creates a copy of the strictness state at each call site and associates it with the callee in a hash table. This information is used in the next phase of the analysis.

The strictness state of a function is comprised of two parts, which are described below.

Argument Evaluation Status

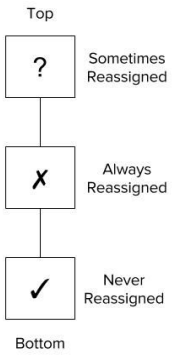
These fields provide information about whether a particular argument is evaluated or not. They are described below in detail.

Contains

This array contains information about formal parameter reassignment. If the corresponding promise is not forced upto the point of reassignment to a formal parameter, then it can

never be forced. It's also possible for the parameter to be reassigned only along some of the control flow paths. This makes it impossible to say ahead of time if the corresponding promise will be forced. These conditions are represented as values of a lattice which is depicted below.

Lattice	Merge Function

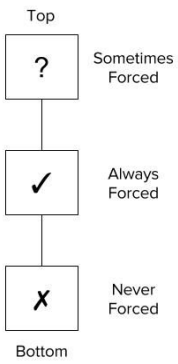


⊔	?	✗	✓
?	?	?	?
✗	?	✗	?
✓	?	?	✓

Forced

This array contains information about forcing of promises corresponding to the formal parameters of the function. It's possible for a promise to be forced only along some of the control flow paths. It's also possible for the corresponding argument to be reassigned. These conditions are represented as the values of a lattice which is depicted below.

Lattice	Merge Function



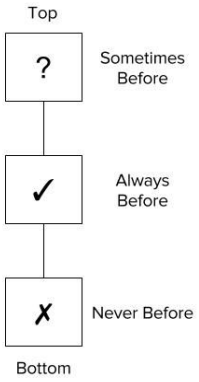
⊔	?	✓	✗
?	?	?	?
✓	?	✓	?
✗	?	?	✗

Argument Evaluation Order

This is a directed graph of the formal function parameters represented as an adjacency matrix. Each cell of the matrix contains the state of the directed edge from the row parameter to the column parameter. A directed edge from parameter a to parameter b implies that a gets evaluated immediately before b. An edge may never exist or it may exist

only along some of the control flow paths. These states of the edge are represented as the values of a lattice which is described below.

Lattice Values	Merge Function



⊔	?	✓	✗
?	?	?	?
✓	?	✓	?
✗	?	?	✗

All these fields are updated as the bytecode instructions are interpreted. An argument appearing in an evaluation context will result in the corresponding promise being forced if and only if the argument has not been reassigned. Simultaneously, an edge from the previously evaluated arguments to this argument will be created in the directed graph representing the argument evaluation order.

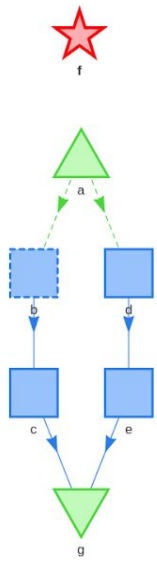
The most interesting operation is the merge operation performed at the end of conditionals. This corresponds to merging sets of facts obtained from the two branches of the conditional in a manner that the final facts do not violate either of the two set of facts. The merge operation is trivial as the values representing the information about parameter reassignment, argument evaluation and evaluation order form a lattice. As is clear from the merge table of all the lattices, merging will always result in a value equal to or greater than the highest of the values being merged. Merging facts at the end of conditionals makes the analysis **path-insensitive**.

An Example

The example below shows a function, the corresponding strictness state and its visualization as a directed graph (obtained from *signatuR*, described later on).

Function Definition	Abstract State Visualization

```
1  f1 <- function(a, b, c, d, e, f, g)
2  {
3      if(a) {
4          b + c
5      }
6      else {
7          b = 99
8          d + e
9      }
10     g
11 }
```



Strictness State

Argument	Forced	Contains
a	✓	✓
b	?	?
c	?	✓
d	?	✓
e	?	✓
f	✗	✓
g	✓	✓

Argument Evaluation Status

	a	b	c	d	e	f	g
a	✗	?	✗	?	✗	✗	✗
b	✗	✗	?	✗	✗	✗	✗
c	✗	✗	✗	✗	✗	✗	✓
d	✗	✗	✗	✗	✓	✗	✗
e	✗	✗	✗	✗	✗	✗	✓
f	✗	✗	✗	✗	✗	✗	✗
g	✗	✗	✗	✗	✗	✗	✗

Argument Evaluation Order

Phase II - Fixed point Iteration

The fixed point iteration phase implements a worklist algorithm[2] to flow in the strictness state from the callee to the caller. Each iteration improves the strictness signature of the function based on the strictness signature of the callees obtained from the previous iteration. The high level algorithm in pseudocode form is described below.

Pseudocode

Assume that the control flow graph of the program consists of n functions $f_1, \ldots, f_n$ and that the graph can be determined statically. All functions take the same arguments in the same order. s_i represents the strictness state of f_i. $s_{j,i}$ denotes the strictness state of f_j at the point it makes a call to f_i. Also assume that:

- `callers(`f_i`)` returns the set of functions that call f_i.
- `intraprocedural(`f_i`)` performs the intraprocedural analysis on f_i. It disregards callees and assumes all fields in the initial state to be at the bottom of their corresponding lattices.
- `interprocedural`$_i(s^0, s_1, \ldots, s_n)$ performs the interprocedural analysis on f_i assuming it's initial strictness state to be s^0. Whenever it encounters a call instruction in the bytecode of f_i, it updates the strictness state of f_i based on the strictness state of the callee.
- $\sqcup$ is the strictness state merge operation that merges the internal fields of all its argument states. The lattice and the corresponding merge operation for each field of the strictness state have been described earlier.
- w represents the worklist modeled as a queue data structure.

```
s1 = intraprocedural(f1)
...
sn = intraprocedural(fn)

w = {1, ..., n}
while (w ≠ ∅) {
    i = w.dequeue()
    s⁰ = ⊔{s_j,i | f_j ∈ callers(f_i)}
    y = interprocedural_i(s⁰, s1, ..., sn)
    if (y ≠ s_i) {
        for (f_j ∈ callers(f_i)) {
            w.enqueue(j)
        }
        s_i = y
    }
}
```

Worklist algorithm

[2] https://cs.au.dk/~amoeller/spa/

Correctness

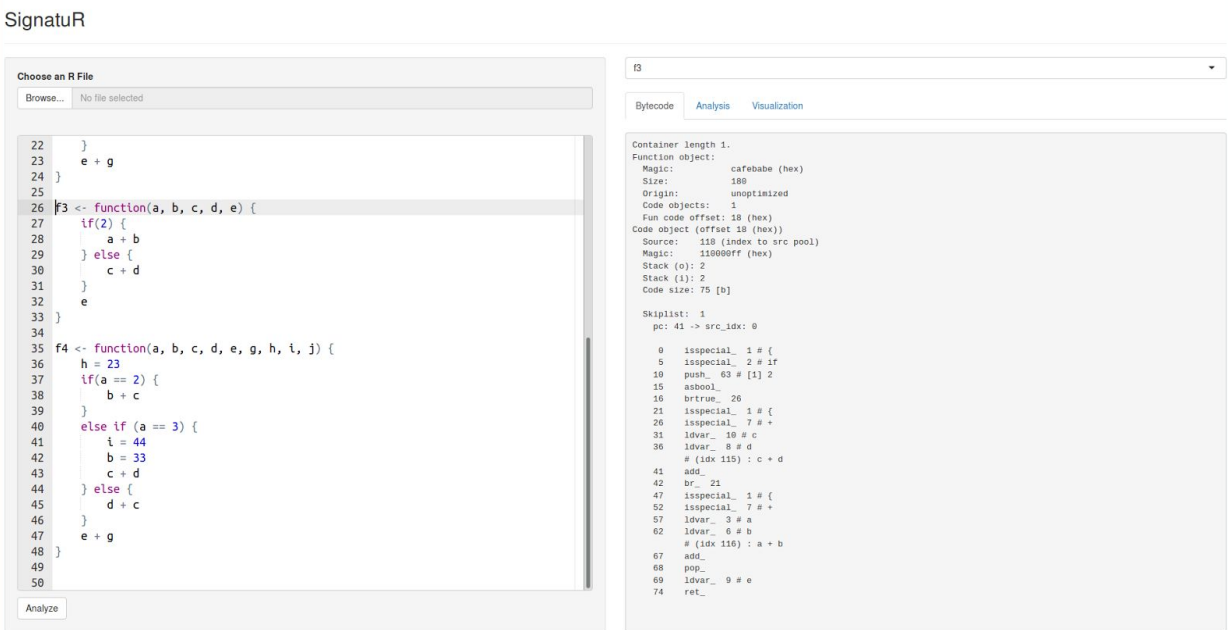
I have not worked out a formal proof of correctness of the algorithm yet. But I have made a couple of observations which will help me develop a proof. My analysis technique (CFA-0) fits into the traditional monotone data analysis framework. Information about forcing promises and their forcing order forms a lattice. My analysis always adds more information, it does not retract. Once a promise is forced, it can't be unforced. Only new edges are added to the directed graph representing the order of forcing of promises and previously added edges are never removed. This implies that once I reach the upper bound of my lattice, no additional facts can modify the results further. This means my fixed point iteration will terminate.

signatuR

While developing the algorithm for finding strictness signature of functions, I also designed a web application in R using the shiny library, called **signatuR**. This web application has two panels. The panel on the left has a code editor which allows one to write arbitrary code and run the Intraprocedural analysis. The panel on the right displays the result of the intraprocedural analysis in three tabs.

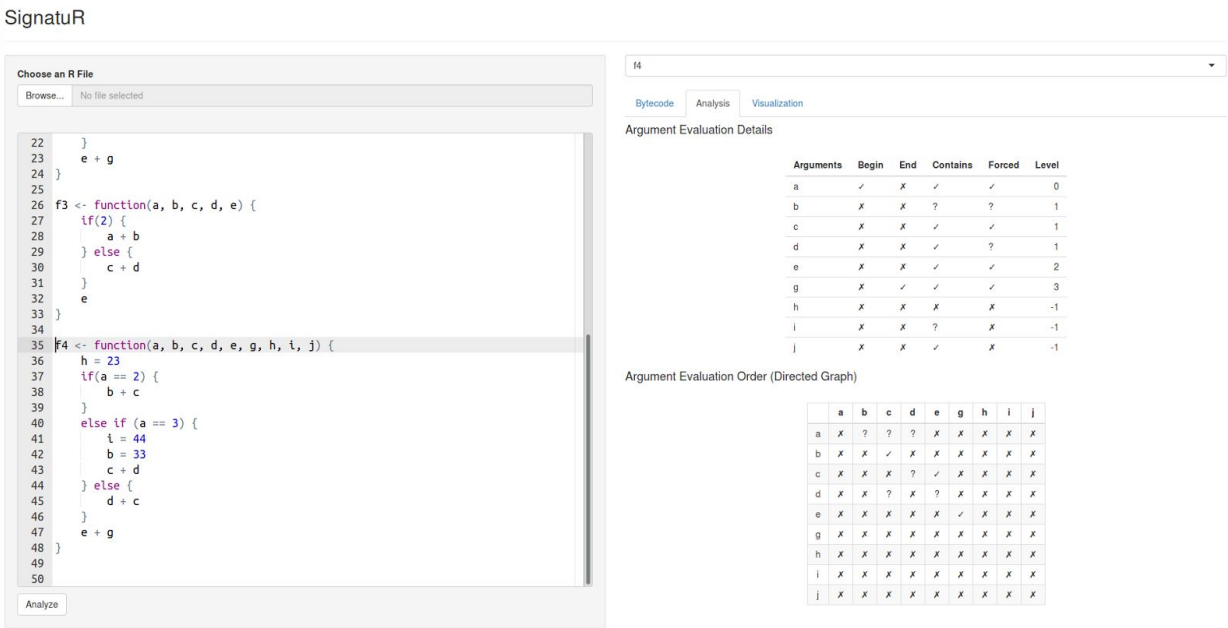
Bytecode Tab

This tab shows the bytecode of the compiled function.



Analysis Tab

This tab displays the strictness state obtained from the intraprocedural analysis in a neatly tabulated form.



Visualization Tab

This tab visualizes the directed graph representing the order of evaluation of arguments. The visualization encodes strictness state information in different properties.

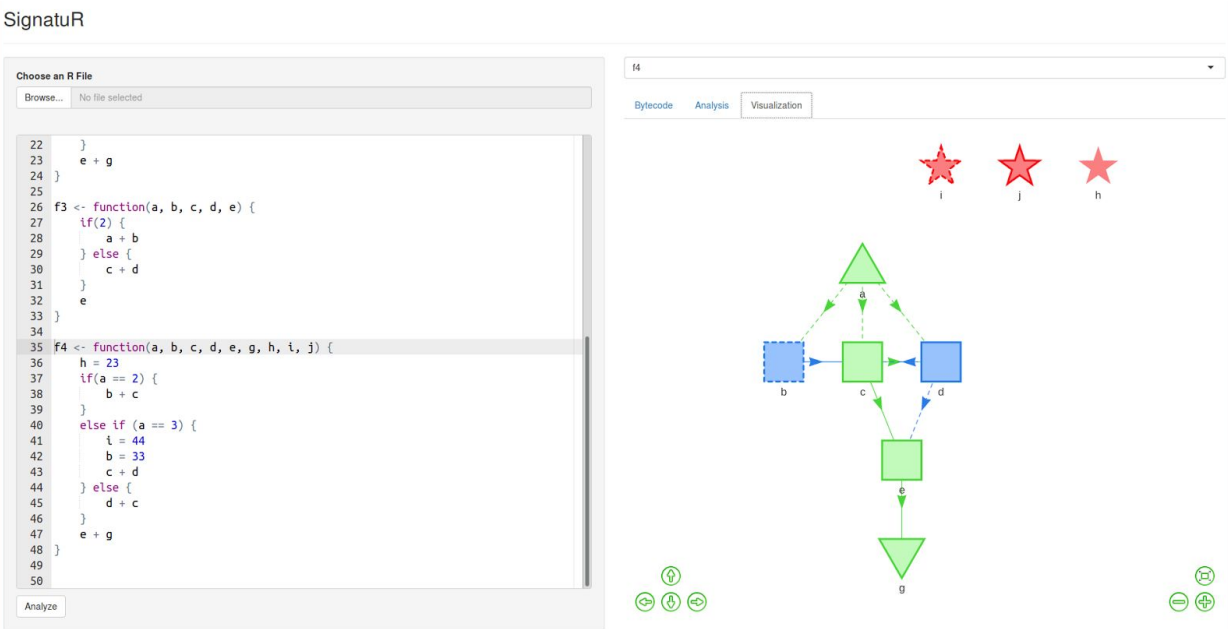
- **Color** represents the evaluation status of the arguments
  - *Green* arguments are always evaluated
  - *Blue* arguments are sometimes evaluated
  - *Red* arguments are never evaluated
- **Shape** identifies distinct positions of the directed graph
  - *Upward pointing triangles* represent arguments which may be evaluated first
  - *Downward pointing triangles* represent arguments which may be evaluated last
  - *Squares* represent arguments which are neither the first nor the last to be evaluated
  - *Stars* represent arguments which are never evaluated
- **Directed Edge** represents the order of argument evaluation
  - *Solid edge* from a to b indicates that a always gets evaluated immediately before b
  - *No edge* from a to b indicates that a never gets evaluated immediately before b

    - *Dashed edge* from a to b indicates that a may get evaluated immediately before b
- **Outline** represents the relationship between formal parameters and promises
    - *Solid outline* indicates that the formal parameter always refers to the promise
    - *Dashed outline* indicates that the formal parameter gets reassigned along some of the control flow paths, so it may or may not refer to the promise
    - *No outline* indicates that the formal parameter gets reassigned along all of the control flow paths, so it never refers to the promise
- **Position** represents the evaluation hierarchy
    - At the top are arguments never evaluated (stars)
    - Upward pointing triangles appear next
    - Squares appear after that in the order in which they will be evaluated
    - Downward pointing triangles appear in the end

Some of these combinations are redundant but some of them differentiate between fine shades of meaning. For example -

A dashed outline square can be green or blue in color. Green indicates that even if the parameter gets reassigned, the corresponding promise still gets forced. Blue indicates that the promise may or may not be forced.

A star is always red (redundancy) but can have a solid, dashed or no outline. The outline indicates whether the corresponding formal argument gets reassigned or not.



The Implementation

The code can be found in the private repository `CS7480-Fall2016/team8`

My analysis relies upon the dataflow analysis framework designed by the developers of rir to implement static and dynamic analyses on the R bytecode.

Strictness Lattice

`rir/rir/src/optimizer/`**`StrictnessLattice.h`**

This class implements the strictness lattice and merge operations for lattice values and sequences of lattice values.

Strictness State

`rir/rir/src/optimizer/`**`StrictnessState.h`**

This class implements the strictness state data structure. It contains members to represent information about forced promises and the order in which they are forced. It also contains the list of callees and a copy of it's own state at the point the callee was called.

Intraprocedural Analysis

`rir/rir/src/optimizer/`**`IntraproceduralStrictnessAnalysis.h`**

This class implements the abstract interpretation based intraprocedural analysis . It steps through the bytecode of a function and forces the promises corresponding to the arguments used in an evaluation context and ignores the effect of callees.

Interprocedural Analysis

`rir/rir/src/optimizer/`**`InterproceduralStrictnessAnalysis.h`**

This class implements Control Flow Analysis - 0 . It preprocesses all the functions using the Intraprocedural analysis class. It implements a worklist algorithm to do a fixedpoint iteration of the analyzed functions, flowing in the strictness information based on caller-callee relationship until all strictness signatures have been stabilized and the worklist becomes empty.

Tests

`rir/rir/tests/rir_analysis_strictness_intraprocedural.R`

`rir/rir/tests/rir_analysis_strictness_interprocedural.R`

The two files contain test cases for the intraprocedural and interprocedural analysis, respectively.

signatuR

`rir/apps/signatuR/app.R`

This script implements the web application which provides visual representation of the result of intraprocedural analysis.

dependR

`rir/apps/dependR/dependency-generator.R`

This script downloads metadata of all R libraries hosted on CRAN and creates a dependency graph. Unfortunately, the graph is represented as a data structure and there is no visual output facility yet. I intend to develop a proper API to expose this functionality and host this package on CRAN.

Conclusions

With the intention to optimize the inefficiencies arising due to R's lazy evaluation strategy, I have implemented a flow sensitive and path insensitive CFA-0 algorithm to identify function arguments in R which can be evaluated eagerly and the order in which they can be evaluated. However, I have barely scratched the surface of this problem. The implementation is severely limited because of the numerous simplifying assumptions I have made and is thus incapable of being run on real world R code. Furthermore, this analysis is purely static in nature. To solve the problem completely, I need to take into account the extremely dynamic nature of R. For this, I need to develop a dynamic counterpart that can intercept R's metaprogramming and reflective constructs and undo the eager evaluations suggested by my analysis. A bytecode rewrite pass to remove the promises and eagerly evaluate the arguments as dictated by this analysis also needs to be implemented to benchmark the effectiveness of this technique. I plan to continue working on this problem and hope to overcome the limitations. I am also interested in exploring the IFDS framework to solve this problem.