TransformeR: A DSL for Safe Variable Transformation

Yisu Remy Wang
Privacy Tools for Sharing Research Data
PI: Salil Vadhan **Mentors:** Stephen Chong, Marco Gaboardi, Victor Balcer

August 12, 2016

1 Overview

TransformeR is a domain-specific language embedded in the R programming language for writing safe variable transformations within the PSI private data sharing interface. In this document, we present the integration of TransformeR within the PSI prototype and the language's grammar, statistical operations and type system. We also attach a list of desiderata together with guidelines for future contributors.

2 Introduction

The PSI private data sharing interface [1] is a system of software tools for researchers, especially social scientists, to share and analyze datasets containing sensitive information while protecting the privacy of participants in the datasets. Its foundation, differential privacy, provides strong guarantee for privacy protection while preserving high utility of the protected datasets.

On the daily agenda of social scientists is the task of preparing datasets for statistical algorithms, be it removing outliers, selecting variables, or performing logarithm transformations. In this paper we refer to such process as variable transformation. Arbitrary variable transformations that work at the level of single individual data[1] can be safely applied before applying a differential private data analysis if an adversary only gets to observe the result of the differential private analysis. The current version of the PSI prototype offers support for writing variable transformations as R programs that can be run on the data before running the other private statistics. Arbitrary R programs can allow for leakage of information beyond the output, and "side-channel attacks" [2] where an adversary observes this additional leakage and thereby undermines the privacy guarantees.

TransformeR is a subset of the R language that can serve as a domain specific language useful to write the needed data transformations and at the same time be more manageable for preventing security weaknesses and side-channel attacks.[2]

3 Background

In this section we introduce the idea of differential privacy and the PSI interface, which is implemented based on the theory of differential privacy. Then we discuss about the compatibility problems between variable transformations and differentially private algorithms, as well as security vulnerabilities during variable transformations.

[1] For the sake of simplicity, we assume all datasets are tables where each row is an individual, e.g. a participant in a survey, and each column is a variable, e.g. age or weight.

[2] Part of this section is taken from the project description document Enhancing Support for Safe Variable Transformation.

3.1 Differential Privacy and PSI

The PSI prototype implements differential privacy, a mathematical formalization of privacy in statistical algorithms. Intuitively, differential privacy states that an algorithm has good privacy protection if its output stays almost the same when its input only changes by one individual. That is, an adversary would not be able to infer information about single individuals only from observing the change in the outcome of the algorithm.

The current implementation of PSI achieves differential privacy by adding random noise to the outcome of algorithms. The amount of noise is chosen with regard to, among other factors, the range of input to the algorithm. To comply with differential privacy, the outcome of any transformation should fall into the range of the differentially private algorithms chosen. We call that the compatibility problem between transformations and DP-algorithms, which we discuss in details in following paragraphs.

3.2 Compatibility between Variable Transformations and DP-Mechanisms

Current implementation of PSI allows variable transformations to be run on each row before differential private analyses. However, the transformed database might no longer be compatible with the differential private mechanism. For example, some DP-mechanism expects a database with ages of the participants, and a transformation takes an age and outputs "true" if it is lower than 21 and "false" otherwise. In this case, the original DP-mechanism cannot accept the transformed database, which holds booleans, because it expects numerical values. The system therefore needs to keep track of the type of the outcome of the transformations in order to select compatible DP-mechanisms. However, that is not enough. Consider a mechanism M which computes the average age of the participants in a database and adds some noise to its answer to achieve ϵ-differential privacy. Now, we double the age of each individual through variable transformations and apply the same mechanism to the transformed database. The mechanism would proceed successfully because it expects numerical values, which match the values in the transformed database. However, since we doubled the age of each individual, the potential effect of changing the age of one individual also doubles, therefore adding the same amount of noise as before may no longer achieve ϵ-differential privacy. To solve the above compatibility problems, we define a type system for TransformeR that refines range information into the types of expressions to provide information necessary for choosing compatible DP-mechanisms.

3.3 Side-channel Attacks

In computer security, side-channel attacks refer to malicious actions from an adversary who takes advantage of information beyond the direct outcome of some computation. In some existing private data sharing systems such as PINQ [3] and Airavat [4], an adversary can undermine the privacy guarantee by submitting queries with observable behavior beyond the direct outcome [2]. For example, a computation that saves the entire database to an external storage may return a useless result while blatantly violating any privacy guarantee. TransformeR is designed to eliminate such vulnerabilities by offering only functionality that is necessary for statistical transformations and nothing more. For example, we do not allow any file IO operations, therefore it is not possible for the adversary to save any part of the database to external storage.

4 Integration within PSI

TransformeR is to be integrated into the PSI prototype for performing variable transformations before differential private mechanisms, as Figure 1 demonstrates. During an analysis session in PSI, if a TransformeR program file `t.R` exists, PSI calls the `tyTrans` function provided by the `Types` module to type-check the transformation, which is parsed by calling `parse prog "t.R"` from PSI, where `prog` is the parser combinator defined in the `Parse` module[3]. If the program type-checks, then it does not access memory accessible to attackers and complies with differential privacy, and its type contains information necessary for choosing

[3]the TransformeR modules in Haskell are available for internal review at `https://github.com/HarvardPL/TransformeR`

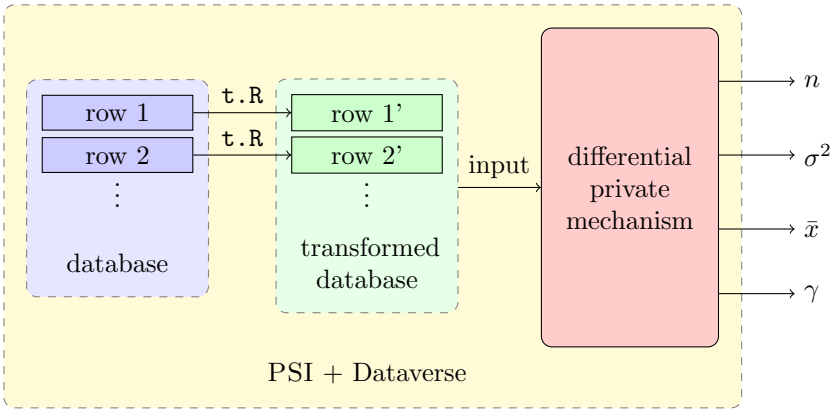

Figure 1: Integration of TransformeR in PSI. The analyst defines variable transformation in file `t.R`

```
transform (subject : { age          : N[0, 100]
                     , intelligence : C["smart", "dumb"]
                     , species      : C["human", "bot"]})

          { age          : N[50, 150]
          , intelligence : C["smarthuman", "smartbot", "dumbhuman", "dumbbot"]
          , species      : C["human", "bot"]}

  { subject[[age]]          <- +(subject[[age]], 50);
    subject[[intelligence]] <- join(subject[[intelligence]], subject[[species]]);
    subject }
```

Figure 2: Example TransformeR program

compatible DP-mechanisms. PSI then performs the transformation on each row of the queried database, and finally passes the transformed database to compatible differential private mechanisms chosen based on the information derived from the type checker and the specific queries evoked.

5 Language Components

To ensure safe variable transformation, TransformeR 1. has a well-defined grammar suitable for static analysis, 2. includes a set of operations adequate for many statistical transformations while barricading opportunities for side-channel attacks, and 3. uses a type checker to track changes in the range of values in the database to ensure compatibility between the transformation outcome and the differential private mechanisms. In this section we describe the above three components of TransformeR: its grammar, statistical operations and type system.

5.1 Grammar

Our goal is to make TransformeR easy to use for social scientists. The majority of PSI is integrated with the R programming language, therefore we aim to make TransformeR as close to R as possible. Here we present the definition of the grammar in BNF-form, along with explanations. Figure 2 shows an example of a TransformeR transformation , which will assist our presentation of the grammar.

$prog ::= \textsf{transform}(x : \tau_1) : \tau_2\ \{\ e\ \}$

A TransformeR transformation takes a named row as argument and one type annotation each for the argument and the result. The type annotations declare what kind of data the input / output of the transformation (numbers, categories, etc.) are as well as their ranges. Currently we allow for closed continuous ranges for fractions and finite sets of strings to be used as ranges for numerical data and categorical data, respectively. We formally define the types in section 5.4. The body is any expression.

$$e ::= v \mid \{l_1 = e_1, ..., l_n = e_n\} \mid e[[l]] \mid op(e_1, ..., e_n) \mid x \mid x \leftarrow e \mid x[[l]] \leftarrow e \mid e_1; e_2$$

As defined above in order, an expression can be a literal value, a row with named fields, a field access to a row, some operation on an arbitrary number of expressions, a variable, variable assignment, assignment to a field within a variable, and sequencing of expressions.

$$v ::= n \mid c \mid \{l_1 = v_1, ..., l_n = v_n\}$$

Finally, a value is a number, a category (e.g. a bin name in a histogram), or a record.

Though rudimentary, this definition is expressive enough to model a per-row transformation performed on a database defined in R. One can think of the field labels as column names in a database, and a field access `subject[[age]]` analogous to an R-style indexing with the identical syntax. One might be worried by the presence of assignments and mutations as they might expose information to side-channels. We decide to include imperative features based on the feedback of working social scientists, who suggest that imperative idioms are essential to the common style. To ensure mutations and assignments are safe, the type system carefully monitors environments to block side-channels. Since differential privacy limits variable transformations to be per-row transformations, Since TransformeR limits variable transformations to per-row transformations in order to comply with differential privacy, only a collection of simple statistical operations will suffice for writing most transformations. We introduce available statistical operations in the following section.

5.2 Statistical Operations

Based on the feedback from working statisticians and social scientists, we found just a handful of statistical operations will suffice to implement desired variable transformations. They fall into three categories: unary operations, binary operations, and conditionals, which we describe as follows:

1. Unary operations: `log, sqr, sqrt, reciprocal` on numbers, `not` on booleans (zero or non-zero numbers, more discussion in section 6).
2. Interactions: `+, -, *, /, ==` on a pair of numerical expressions, `and, or, xor` on a pair of booleans, and `==, concat` on a pair of categories.
3. Conditionals: `if` statements. In addition to booleans, numbers (zero or non-zero) are commonly used to represent `true` or `false` in statistics and social sciences.

More complicated operations appear in practice, such as the `melt` function in the `reshape2` package [5] that transforms data in short-form into long-form. However, it is not clear those transformations conform to the requirements of differential privacy, therefore we add them onto our wish list in section 6 to consider in the future.

5.3 Semantics

To relief TransformeR users the burden of learning a completely new language, we design TransformeR to closely resemble R, the language PSI uses as its back-end as well as parts of its front-end. Here we define the semantics of TransformeR in the form of operational large-step semantics, with the exception of one premise for OperatorApplication which we define with denotational semantics. There we denote an operation *op* with $[\![op]\!]$, for example $[\![+]\!]$denotes a mathematical function that sums its two arguments. The judgment $\langle e, m\rangle \Downarrow \langle v, m'\rangle$ declares "with store m expression e evaluates to value v and updates the store to

m'". $m[x \to v]$ stands for a store that is identical to m save for mapping x to v. Evaluation of literal values results in the values themselves and does not modify the store, and we leave out the rules. The rest of the semantics is rather straightforward, and we present them as follows.

$$\frac{\langle e, m\rangle \Downarrow \langle \{l_1 = v_1, ..., l_n = v_n\}, m'\rangle \qquad 1 \le i \le n}{\langle e[[l_i]], m\rangle \Downarrow \langle v_i, m'\rangle} \textsc{FieldAccess}$$

$$\frac{\forall i \in [1...n], \langle e_i, m\rangle \Downarrow \langle v_i, m_i\rangle \qquad [\![op]\!](v_1, ..., v_n) = v'}{\langle op(e_1, ..., e_n), m\rangle \Downarrow \langle v', m_n\rangle} \textsc{OperatorApplication}$$

$$\frac{\forall i \in [1...n], \langle e_i, m_{i-1}\rangle \Downarrow \langle v_i, m_i\rangle}{\langle \{l_1 = e_1, ..., l_n = e_n\}, m_0\rangle \Downarrow \langle \{l_1 = v_1, ..., l_n = v_n\}, m_n\rangle} \textsc{Record} \qquad \frac{m(x) = v}{\langle x, m\rangle \Downarrow \langle v, m\rangle} \textsc{Var}$$

$$\frac{\begin{array}{c} m(x) = \{l_1 = v_1, ..., l_n = v_n\} \qquad i \in [1...n] \\ \langle e, m\rangle \Downarrow \langle v', m'\rangle \qquad v = \{l_1 = v_1, ..., l_i = v', ..., l_n = v_n\} \end{array}}{\langle x[[l_i]] := e, m\rangle \Downarrow \langle v, m'[x \mapsto v]\rangle} \textsc{FieldAssign}$$

$$\frac{\langle e_1, m\rangle \Downarrow \langle v_1, m_1\rangle \qquad \langle e_2, m_1\rangle \Downarrow \langle v_2, m_2\rangle}{\langle e_1; e_2, m\rangle \Downarrow \langle v_2, m_2\rangle} \textsc{Sequence} \qquad \frac{\langle e, m\rangle \Downarrow \langle v, m'\rangle}{\langle x := e, m\rangle \Downarrow \langle v, m'[x \mapsto v]\rangle} \textsc{Assignment}$$

5.4 Type System

In this section we present TransformeR's type system which tracks the changes in the range of values during variable transformations. Formally, the type system enforces the following property, which states "every transformation that type-checks, if given an argument of, or is a subtype of type τ_1, will produce a result of, or is a subtype of type τ_2":

Property 1. *If* $\vdash$ *transform*$(x : \tau_1) : \tau_2$ { e }*, then* $\forall v$*. if* $v : \tau$ *and* $\tau \le \tau_1$ *and* $\langle e, [x \mapsto v]\rangle \Downarrow \langle v', m\rangle$ *then* $v' : \tau'$ *and* $\tau' \le \tau_2$*.*

where $\vdash$ denotes the transformation to follow type-checks; $v : \tau$ means value v has type τ; $\tau_1 \le \tau_2$ means τ_1 is a subtype of τ_2; $\langle e, [x \mapsto v]\rangle \Downarrow \langle v', m\rangle$ denotes "expression e evaluates to value v' with a store where the only present variable x maps to value v, while modifying the store to m", following the semantics established in section 5.3. In the rest of this section, we describe the syntax of types, the rule to check a valid transformation, type formation rules, sub-type rules, introduction rules and elimination rules in order. Since TransformeR supports variable assignments that modify the types of variables, we adopt a flow-sensitive type system with a type environment to track the change in the types of variables. In the following judgments we use $\Gamma \vdash e : \tau \rhd \Gamma'$ to denote "with a initial type environment Γ, expression e type-checks to τ and updates the type environment to Γ'". In order to check if types are well-formed, we use the judgment $\vdash \tau$wf to mean "type τ is well-formed".

Types

$$\begin{aligned} \tau &::= \{label : \sigma, ...\} \\ \sigma &::= N[n, n] \\ &\quad \mid C\{name, ...\} \end{aligned}$$

There are only two basic types, categories and numbers. We refine numerical types with a range and categorical types with a set of categories. Currently we allow for closed continuous ranges for fractions and finite sets of strings to be used as ranges for numerical data and categorical data, respectively. To avoid nested records, we separate the record type from basic types for expressions.

Valid Transformation

$$\frac{\vdash \tau_1 \text{ wf} \qquad \vdash \tau_2 \text{ wf} \qquad x : \tau_1 \vdash e : \tau' \rhd \Gamma \qquad \tau' \leq \tau_2}{\vdash \mathit{transform}(x : \tau_1) : \tau_2 \ \{\ e\ \}} \textsc{Transformation}$$

A valid transformation should have its body type-check to the declared outcome type when the argument is mapped to the declared input type in the type environment and should not have duplicate field labels anywhere.

Formation rules

$$\frac{n_1, n_2 \in \mathbb{R} \qquad n_1 \leq n_2}{\vdash N[n_1, n_2] \text{ wf}} \textsc{NumberFormation} \qquad \frac{\forall i \in \{1...n\}.\ c_i \in \text{String all distinct}}{\vdash C\{c_1, ..., c_n\} \text{ wf}} \textsc{CategoryFormation}$$

$$\frac{\forall i \in \{1...n\}.\ \vdash \sigma_i \text{ wf} \qquad l_i \in \text{String all distinct}}{\vdash \{l_0 : \sigma_1, ..., l_n : \sigma_n\} \text{ wf}} \textsc{RecordFormation}$$

Types are well-formed if they obey the following rules. We interpret intervals as closed for the moment for the sake of simplicity.

Sub-type rules

$$\frac{lb' \leq lb \qquad ub \leq ub'}{\vdash N[lb, ub] \leq N[lb', ub']} \textsc{NumericalSubtype} \qquad \frac{\forall i \in \{1...n\}.\ \exists k \in \{1...m\}.\ c_i = c'_k}{\vdash C\{c_1, ..., c_n\} \leq C\{c'_1, ..., c'_m\}} \textsc{CategoricalSubtype}$$

Types with narrower ranges are sub-types of those with broader ones. Sub-typing rules help us to reason about operations such as comparisons.

Introduction rules

$$\frac{n \in \mathbb{R}}{\Gamma \vdash n : N[n, n] \rhd \Gamma} \textsc{NumberLiteral} \qquad \frac{c \in \text{String}}{\Gamma \vdash c : C\{c\} \rhd \Gamma} \textsc{CategoryLiteral}$$

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma \rhd \Gamma} \textsc{RecordVariable} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \rhd \Gamma} \textsc{ExpressionVariable}$$

$$\frac{\forall i \in [1...n], \Gamma_{i-1} \vdash e_i : \sigma_i \rhd \Gamma_i}{\Gamma_0 \vdash \{l_1 = e_1, ..., l_n = e_n\} : \{l_1 : \sigma_1, ..., l_n : \sigma_n\} \rhd \Gamma_n} \textsc{Record}$$

With the help of sub-typing, we can safely use the literal values themselves as the bound of their ranges to achieve accuracy. Variables can have either primitive types or record types. Fields of a record expression type-checks from left to right while the type environment traces changes in variable types.

Elimination rules

$$\frac{\Gamma \vdash e : \{l_1 : \sigma_1, ..., l_n : \sigma_n\} \rhd \Gamma'}{\Gamma \vdash e[[l_i]] : \sigma_i \rhd \Gamma'} \textsc{Projection} \qquad \frac{\begin{array}{c}\forall i \in \{1...n\}, \Gamma_{i-1} \vdash e_i : \sigma_i \rhd \Gamma_i \\ op : \sigma_1 * ... * \sigma_n \to \sigma' \qquad \vdash \sigma' \text{ wf}\end{array}}{\Gamma_0 \vdash op(e_1, ..., e_n) : \sigma' \rhd \Gamma_n} \textsc{OperatorApplication}$$

$$\frac{\Gamma \vdash e : \sigma \rhd \Gamma'}{\Gamma \vdash x := e : \sigma \rhd \Gamma'[x \mapsto \sigma]} \textsc{Assignment} \qquad \frac{\begin{array}{c}\Gamma(x) = \{l_1 : \sigma_1, ..., l_n : \sigma_n\} \qquad \Gamma \vdash e : \sigma \rhd \Gamma' \\ \tau' = \{l_1 : \sigma_1, ..., l_i : \sigma, ..., l_n : \sigma_n\}\end{array}}{\Gamma \vdash x[[l_i]] := e : \tau' \rhd \Gamma'[x \mapsto \tau']} \textsc{FieldAssign}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \rhd \Gamma_1 \qquad \Gamma_1 \vdash e_2 : \sigma_2 \rhd \Gamma_2}{\Gamma \vdash e_1; e_2 : \sigma_2 \rhd \Gamma_2} \textsc{Sequence}$$

Finally, the type of a field access follows the type of the argument, and the arguments to as well as the outcome of an operation depend on the specific operator. Arguments to operations type-check from left to right and carry their effect on the type environment. Note that we use the notation $\text{op} : \sigma_1 * ... * \sigma_n \to \sigma'$ informally to indicate "op takes arguments of type $\sigma_1 * ... * \sigma_n$ and returns a result of type σ'." Assignment updates the type environment, and field assign remaps the variable to its type with the field mapped to the type on the right-hand side. A sequence of expressions thread each expression's effect on the type system and has the same type as the last expression.

6 Open Problems & Wish list

Since the current design of TransformeR is rather a proof of concept, there is plenty room for improvement. In this section we lay out the items on our todo-list as well as ideas out of the scope of the current project but worth considering in the future.[4]

6.1 "Types" for Statistical Operators

As we described in section 5.4, checking operations require knowledge of the input/output types of the specific operator used. Therefore we need to curate a collection of signatures ascribing type information to the statistical operators we decide to include in TransformeR.

6.2 Booleans

Practice in statistics and social science asks for special care when handling booleans. One common operation is to multiply numbers with booleans encoded by 0 or 1 in order to select a subset of a database. However, it does not make sense to attach a numerical range of [0, 1] to booleans, because they are discrete. One sensible option is to have booleans as a separate type, replace multiplications with "if" statements, and add extra inference rules to watch out for logical operations on 1 and 0.

6.3 Errors & Exceptions

Exceptions open another side-channel to undermine differential privacy. For example, an adversary may trigger an exception when some individual's data satisfies some premises, and by observing whether or not the transformation crashes the adversary can infer if the premise stands for that particular individual. Although it is undecidable to statically determine if a program may crash for Turing-complete languages, that might not be the case for TransformeR, which only supports simple statistical transformations. A type checker that guarantees successful execution of every type-checked program would block exception attacks.

[4] this list should be kept in sync with the (private) wiki at `https://github.com/HarvardPL/TransformeR/wiki`

6.4 Correctness of Type System

To demonstrate TransformeR enforces safety and privacy, we need to prove the correctness of the type system with regard to its property and to demonstrate the language is a proper subset of R. Proof assistants such as Coq can be used for the former, while empirical validations on an interpreter implemented from the semantics can achieve the latter.

6.5 A More General Requirement for Safe Variable Transformation

Differential private query tools like PINQ[3] implement a more general requirement for safe variable transformation of "stability". It is worth exploring how to extend TransformeR with that requirement.

6.6 Wish list

To make TransformeR more expressive, we consider the following features to implement in the future:

1. Include more sophisticated ranges, such as discontinuous ranges for numbers and infinite sets for categories.
2. Extend the type system to allow transformations between long-form and short-form rows. That is, have records as first-class expressions and allow nested records.
3. Enable type annotation by users to help infer more accurate ranges.
4. Explore user-defined operations.
5. Better error messages when a program fails to type check.
6. Extend TransformeR to full statistical procedures; automatically transform arbitrary programs into safe variable transformation, DP-mechanism, and post-processing.

Acknowledgments

We thank James Honaker, Dan Muise, Fanny Chow, Clara Wang, Jack Landry and Grace Rehaut for their valuable input on useful R operations in social science research.

References

[1] M. Gaboardi, J. Honaker, G. King, K. Nissim, J. Ullman, and S. Vadhan. PSI (Ψ): a private data sharing interface. In *Theory and Practice of Differential Privacy*, New York, NY, 2016 Working Paper.

[2] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 33–33, Berkeley, CA, USA, 2011. USENIX Association.

[3] F. McSherry. Privacy Integrated Queries: An extensible platform for privacy-preserving data analysis. *Commun. ACM*, 53(9):89–97, Sept. 2010.

[4] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 20–20, Berkeley, CA, USA, 2010. USENIX Association.

[5] H. Wickham. Reshaping data with the reshape package. *Journal of Statistical Software*, 21(1):1–20, 2007.

A Road Map for Future Contributors

Welcome to the PSI TransformeR project! This document aims to help you navigate through the TransformeR implementation and documentation and get everything up and running. If you have not already, go ahead and clone the git repo[5] at `https://github.com/HarvardPL/TransformeR` where everything lives. A few important files at project root:

- `app`: Right now TransformeR only works as a library with parsing and type-checking functions. For integration with PSI, one option is to do foreign function calls from system back end; another one is implement an executable for TransformeR and use system call to evoke it.
- `doc`: Documentations, including numerous presentations and the paper. The most up-to-date presentation is `final-presentation`. `syntax` also defines an (outdated) ANTLR grammar. There is extensive Haddock documentation in the source code, and you can generate Haddock HTML docs by `stack haddock`. Where it puts the files vary by system, for me it is `TransformeR/.stack-work/dist/x86_64-linux/Cabal-1.18.1.5/doc/html/TransformeR`. To compile this paper, cd to `doc/paper` and `make clean && make`. The LaTeXpresentation slides can be compiled with `latexmk -pdf`. There is room for improvement for the paper, and you can view the comments by setting the `\ifdraft` flag at the top of `designdoc.tex` to `\drafttrue`.
- `lots_of_R`: Corpus of R code. Was used to identify important statistical operations. The goal is to produce Rmarkdown documents for each project.
- `src` The source code, contains the parser and type checker. In the future an interpreter should be implemented from the semantics.
- `test` Tests, mostly QuickCheck tests on the parser and type checker. The idea is to 1. pretty print a bunch of syntax trees and parse them back in, to test parser. The pretty printer needs to be modified to be aware of fixities to make that work. 2. generate a bunch of TransformeR programs, type check them, run them in R, and make sure the result falls within the type.
- `TransformeR.cabal` and `stack.yaml`: TransformeR builds with Haskell Stack, which uses Cabal. Refer to their documentation for configuration. With Stack installed, to build: `stack setup && stack build`; to test: `stack test`.

Aside from the above files, you should check out the github wiki page for papers and other readings, and the github "issues" tab for things planned to be done, in addition to the Wish list in this paper. Have fun!

[5] Private repo, contact the author for access