A Typesafe Relational Extension of simPL: Relational-simPL (rSPL)

Oishik Ganguly

Contents

1 Introduction

This final project extends the simPL programming language to include tagged relational operations. Tagged relations provide a convenient mathematical framework for dealing with data sets (represented as tables) and operations on them. Before outlining the salient features of this project, we thus look at the formal definition for tagged relations.

If we have a set of tags $\{tag_1, tag_2, \cdots, tag_n\}$ and a set of sets $\{S_1, S_2, \cdots, S_n\}$, then a tagged relation R (heretofore referred to as a relation) is defined as:

$$R = \{\{tag_1 : d_{1i} \cdots tag_n : d_{ni}\} | d_{1i} \in S_1 \cdots d_{ni} \in S_n\}$$

where i represents a tagged tuple. This definition makes clear the representation of data sets as relations– the tags represent the data set columns, while the tagged tuples represent the data set rows. This mathematical definition has a further implication for data set representation, namely that all values under a particular column must belong to the same set, the properties of which are known. This feature gives rise to type safe relations which are a central feature of rSPL.

In our implementation of rSPL we first defined a syntax for (i) a type that represents relations , (ii) the relations themselves, and (iii) unary (selection and projection) and binary (join) operations on relations. Following this, we defined typing relations and a small step dynamic semantics for our relational extension. Finally, we implemented these programmatically by adding code to the existing sPL code base (written by the course facilitator(s) for YSC-3208).

2 rSPL Syntax

2.1 Types

For our relational extension, we used of sPL's *int* and *bool* types. Building on this, we defined a type to accurately capture relations as follows:

$$\frac{t_1 \cdots t_n \qquad tag_1 \cdots tag_n}{\{\texttt{relationtype} \quad tag_1 : t_2 \cdots tag_n : t_n \quad end\}} [\textbf{RelType}]$$

where t_i is either *int* or *bool*, and *relationtype* refers to the name of the relation. We further note that repetition of tags is not allowed.

2.2 Expressions

For rSPL, in addition to representing relations, we implement three operations on them, namely projection, selection, and join. In what follows, we define the syntax for relations and operations on them. We provide a qualitative explanation of the operations (the formal definitions for which may be found at here). Most of the following definitions were suggested by Professor Voicu.

1. **Relations**
   Relations are expressed in rSPL as follows:

$$t_r \qquad v_{11} \cdots v_{1n} \quad \ldots \quad v_{m1} \cdots v_{mn} \in \{int + bool\}$$

$$\text{relation } \{t_r\} \text{ row } v_{11} \cdots v_{1n} \text{ end } \ldots \text{ row } v_{m1} \cdots v_{mn} \text{ end end} \quad \text{[RelDef]}$$

   where t_r is a relation type.

2. **Projection**
   The projection of some relation R with respect to some subset of its tags, $\{tag_1 \cdots tag_k\}$ results in a new relation containing only those tags present in the subset. The syntax for projections of relation is as follows:

$$r \qquad tag_1 \cdots tag_k \in tags(r)$$

$$r|||(tag_1, tag_2 \cdots, tag_k) \quad \text{[ProjDef]}$$

   where r is a relation.

3. **Selection**
   Given some relation r, a tag $a \in tags(r)$, and a b which either belong to $tags(r)$ or is a value, and $\theta = \{<, <=, >, >=, =, != \}$, we define the syntax for selection as follows

$$r \qquad a \in tags(r) \qquad b \in tags(r) \textbf{or} \{int + bool\} \qquad \theta$$

$$[a\theta b]\$\$r \quad \text{[SelDef]}$$

   The resulting relation gives all tuples in the relation r that satisfy the predicate $(a\theta b)$.

4. **Join**
   The join of two relation r_1 and r_2 results in a relation that contains the union of the tags of $tags(r_1)$ and $tags(r_2)$. The tagged tuples so obtained in the join must have the same values for the common tags of r_1 and r_2. Syntactically, this operation is represented as follows:

$$\frac{r_1 \qquad r_2}{r_1| >< |r_2}\,[\textbf{JoinDef}]$$

Finally, we make use of sPL's `let in` construct to implement the above operations. Note that we have eliminated type annotation in the body of the said construct. This makes rSPL's code much less verbose.

3 rSPL Dynamic Semantics

rSPL's dynamic semantics are the same as sPL's. Given a program, we evaluate it by repeated one-step evaluation which conclude in a contraction to an rSPL value.The join operation follows the same one-step evaluation rules as binary operators in sPL. In similar vein, the selection and projection operations follows the same one-step evaluation rule as unary operators in sPL. We shall consequently not outline them in this report.

Our relational extension has only one type of value, namely, relations. These refer to any data structure having the relation type defined in **2.1**. The notion of an rSPL value, however, is slightly nuanced– relations that contain only integer or boolean constants are considered values in rSPL. If they contain sPL expressions (containing integers and booleans only), then they are considered further reducible, and these expression are subsequently evaluated. This point is further elaborated when we discuss the implementation of the rSPL interpreter.

4 rSPL Static Semantics

rSPL's primary focus is to ensure type safe data set representation and handling. Given this focus, we build on sPL's typing relation with the following rules.

1. **Relation**
   The following rule ensures that relations defined in rSPL are themselves type

safe, i.e., the values under a particular tag all belong to the same set to which the tag maps. In other words, each row (or tuple) in the relation should have the same type as the relation itself. The rule is defined as:

$$\Gamma|- \texttt{relation } \{t_r\} \texttt{ row } E_{11} \cdots E_{1n} \texttt{ end} \ldots \texttt{row } E_{m1} \cdots E_{mn} \texttt{end end : } t_r$$

$$\Gamma|-\texttt{row } E_{11} \cdots E_{1n} \texttt{ end : } t_r, \cdots, \Gamma|-\texttt{row } E_{m1} \cdots E_{mn} \texttt{ end : } t_r \quad \textbf{[RelT]}$$

2. **Projection**
The relation resulting from a projection must only contain those tags that were provided for the projection operation. The following rule thus ensure the type safety of relation projections:

$$\frac{\Gamma| - r \ : \ t \qquad tag_1, \cdots, tag_k \in tags(r)}{\Gamma| - r|||(tag_1, \cdots, tag_k) \ : \ t'} \textbf{[ProjT]}$$

where t' is a relation type with only the tags $tag_1, \cdots, tag_k$.

3. **Selection**
Since a selection operation alters the number of rows in a relation, and not the columns, the type of the resulting relation is the same as that of the original relation:

$$\frac{\Gamma| - r \ : \ t}{\Gamma| - [a\theta b]\$\$r \ : \ t} \textbf{[SelT]}$$

where $a \in tags(r)$ and $b \in tags(r)\textbf{or}\{int + bool\}$.

4. **Join**
Joining two relations results in a relation that contains the tags of both the operand relations. However, there *must* be common tags between the two operand relations for them to be joined. This is captured in the following type rule:

$$\frac{\Gamma| - r_1 \ : \ t_1 \qquad \Gamma| - r_2 \ : \ t_2 \qquad tags(r1) \cap tags(r2) \neq \emptyset}{\Gamma| - r_1| >< |r_2 \ : \ t'} \textbf{[JoinT]}$$

where t' is a relation type with tags $= tag(r_1) \cup tags(r_2)$

5 Implementating rSPL

To implement rSPL, we built upon sPL's code base, that comprises a type definition file, a lexer, a parser, and a type checker. In addition, we built an rSPL interpreter using the skeleton provided by the ePL interpreter. We chose to write our code on this preexisting code base for two reasons: (i) it enabled us to represent rSPL as a natural extension of sPL, building upon sPL's types and expressions, and (ii) it made debugging easier. In what follows, we walk through our implementation of the rSPL, starting with type definitions and ending with the rSPL interpreter.

5.1 Abstract Syntax Definitions

We add to sPL's abstract syntax of data types and expression types. The abstract syntax of the relation type is as follows:

```
1     | RelationType of (id * (id * sPL_type) list)
2
```

The leftmost `id` of the pair takes the name of the relation type, while the `(id*sPL_type)list` holds the respective tags and the sets they map to. These sets are of `sPL` type, and may either be *int* or *bool*.

The abstract syntax for relation based expressions (relations, projections, selections, and joins) is represented as follows:

```
1     | Relation of (sPL_type * (sPL_expr list) list)
2     | Projection of (sPL_expr * id list * sPL_type option)
3     | Join of (sPL_expr * sPL_expr * sPL_type option)
4     | Selection1 of (sPL_expr * id * op_id * id)
5     | Selection2 of (sPL_expr * id * op_id * sPL_expr)
6
```

Relations are stored as list of lists, each list in the bigger list representing a row. The relation operations are provided their own constructors, as opposed to using sPL's `UnaryPrimaryApp` and `BinaryPrimaryApp`, since we require greater flexibility to add the additional information required to perform these operations. For instance, although projection is essentially a unary operation on a relation, we still require a set of tags with repsect to which a relation is projected. Thus, using a separate constructor to deal with this operation allows us to add this additional information (stored in the `id list`). Note moreover that the `Projection` and `Join` constructors take an `sPL_type option` as argument; these store the inferred type of these operations, thus making interpreting the expressions easier. Finally, note that there are two constructors for the selection operation, `Selection1` and `Selection2`, to capture the two different types of predicates that may arise. We have thus far grouped them

together in the mathematical representation of the operation. `Selection1` deals with predicates containing two tags, while `Selection2` deals with predicates that contain one tag and one value.

5.2 Extending sPL parser for rSPL

The additions to the `token.ml` and `lexer.mll` files are trivial, namely the keywords `relation , row`, the operators `||| , |><| , $$` (for projection, selection, and join), and the separators, `':' , ';' , ','`.

Following this, we extended sPL's parser to capture the syntax we defined for rSPl types and expression in section 2. These extensions may be viewed in the `sPL_parser.ml` file. In particular, note that the precedence of the relation operators is as follows: selection (`$$`) followed by projection (`|||`), finally followed by join (`|><|`). This operator precedence is similar to any standard operator precedence rule inasmuch as unary operations have higher precedence over binary operations. Selection has a higher precedence that projection since user would more likely prefer to implement a selecting operation on a relation before projecting it. Finally, note that all our relational operators are left associative.

5.3 Type Safety in rSPL

In previous subsections, we defined rSPL's abstract sytax and extended sPL's parser to capture rSPL's syntax. In doing so, we obtain an abstract syntax tree of the input program that may be subsequently interpreted to yield an output. However, to ensure type safety, we first perform type checking and type inference based on our extension of sPL's typing relation (see **section 4**).

To infer the type of our program, we recursively traverse its abstract syntax tree. Our strategies for inferring the types of the different rSPL expressions that constitute the abstract syntax tree are as follows:

1. **Relations**
   As mentioned in **section 4**, we need to ensure that every row in a typing relation has the same type as the relation type. To that end, we extract only the tags of the relation type from the type annotation (that is syntactically required in the definition of a relation). Next, we check whether tags repeat or not. If there are no tag repetitions, then we take the first row of the relation, and infer an initial type (without the tags) based on the types of the data values in the rows. Following this, we infer the types of the remaining rows and compare it with this initial type. Should there be a mismatch, we have detected a typing

inconsistency, and the program is aborted. If this is not the case, we finally combine the tags and the inferred type to obtain a list of tag-type pairs. This final step is illustrated by the following code snippet, taken from `sPL_type.ml` :

```
      ....
        (* check if for all the type lists are equal to the preliminary type list*)
                    if (List.for_all
                    (fun el -> el = init_type)
                    remaining_types_lst)
                    (* If this is the case, then combine all the tags and inferred types for each column
                    into a pair list. Since the List.combine function throws errors when the lists are of unequal
                    lengths (implying an incorrect type declaration), we use a try-with construct. *)
                    then
                      try
                        let final_type =
                        List.combine
                        tags
                        ( List.map (fun (Some a) -> a) init_type ) (* This is the list of sPL types. *)
                        in
                        ( Some (RelationType (relation_name , final_type)), e)
                      with
                        | _ -> (None , e)
                    else
                      (None , e)
                      ....

```

Note that using a try-with construct allows us to detect a mismatch between the number of columns present, and the number of columns expected (depending on the number of tags in the relation type definition). While this inferential process may appear redundant given that we have type annotations for the relations, it is necessary because (i) the type annotation so provided may be incorrect, and (ii) this will allow us to do away with type annotations for future versions of rSPL. With this, we complete our type check and inference for relations.

2. **Projection**
   To infer the type of the projection of a relation, we first infer the type of the relation itself. Following this, we check if the tags with respect to which the projection is to be performed are present in the inferred relation type. If this is the case, then we us Ocaml's `List.partition` function to obtain the projected type as follows:

```
      ...
        (* If this is the case, then find the projected type*)
```

```
                then
                  let (projected_type_lst , _) =
                  List.partition
                  (
                  fun (id , _ ) -> List.mem id tag_lst
                  )
                  type_lst
                  in
                  let projected_type = RelationType (relation_name
^"_p" , projected_type_lst)
                  in
                  let new_expression = Projection (new_exp ,
tag_lst , Some projected_type)
                  in (Some projected_type , new_expression)
                  ...

```

Note that the `type_lst` refers to the tag-type pair list that is a part of the abstract sytax representation of the relation type.

3. **Join**
   To infer the type of a join operation, we first infer the types of the two operand expressions. In doing so, we utilize a try-with construct to catch expressions that do not have relation types. Provided both expressions have a relation type, we subsequently obtain a union of the two types using an auxiliary function, `SPL_type.construct_join_type` . Next, in keeping with the typing rule for join, we ensure the existence of common tags in both relations as follows:

```
      ...
      (* Type error if there are no common tags. *)
                if (List.length joined_lsts = List.length
rel1_ty_lst + List.length rel2_ty_lst)
                then (None , e)
                else
                ...

```

   We check if the length of the joined tag-type list is the sum of the length of the tag-type list for each of the operand relation types. If this is the case, basic properties of set theory inform us that no common tags could exist, and we fail to infer a type. However, if the predicate is not met, then the union type so obtained earlier is the inferred type.

4. **Selection**
   As mentioned in section **5.1**, we use two constructors to represent the abstract syntax of the selection operation. However, the type inference for both constructors is similar. As our typing rule specifies, the resulting relation of a selection operation should have the same type as the original relational expression. We thus infer the type of the relation expression, and the type so obtained is the type of the selection operation. In addition, for expressions with the `Selection1`

constructor, we check if the two tags provided belong to the relation type, and if the logical operator is valid or not:

```
      ...
      if ( List.mem id1 tag_lst && List.mem id2 tag_lst )
            then
            (* Finally check if a valid operator is used. *)
              if List.mem op selection_op_id_lst
              then
                let final_exp = Selection1 (new_rel_exp , id1 ,
op , id2)
                (* The selection will always have the same type
as the original relation*)
                in ( Some rel_ty , final_exp )
              else (None , e)
              ...

```

Likewise, for the `Selection2` constructor, we check if the first tag belongs to the relation type, and if value provided is an integer or boolean constant:

```
      ...
      if
            (
             List.mem op selection_op_id_lst
             &&
             (
               match (type_infer_x env value) with
                 | (Some IntType  , _ ) | (Some BoolType , _ )
 -> true
                 | _ -> false
             )
            )
            then
              let final_exp = Selection2 (new_rel_exp , id1 ,
op , value)
              (* The selection will always have the same type
as the original relation*)
              in ( Some rel_ty , final_exp )
              ...

```

In this manner, we infer (and check) the types of the four expression types that are part of rSPL. If the abstract syntax tree of an rSPL program passes the type inference, it may be finally interpreted using the small step dynamics that we briefly touch upon in **section 3**. Before this however, the rSPL program, which uses sPL's `let in` construct, is transformed into its function application equivalent (since sPL's core language specification does not contain the let construct). In the following section, we discuss how this function application is interpreted.

5.4 Interpreting rSPL programs

As outlined in **section 3**, we follow a small step dynamic semantics to implement our interpreter. Doing so involves evaluating an expression, which in turn is composed of numerous one-step evaluations concluding in a contraction to an rSPL value. Before interpreting our program however, we must remove the function application from it. To do so, we apply the function to the actual parameters and obtain the function body with the formal parameters substituted with the real ones. We now have a relational expression (without any variables) that may be interpreted:

```
...
 match e with
| Appln (func , ty , real_params) ->
    begin
      let Func ( _ , formal_params , body) = func
      in
      let env = (List.combine formal_params real_params)
      in
      (* We subsitute the formal parameters with
      the real parameters. This leaves us with an evaluable
      expression. Note that this strategy will only work
      for complete function application. *)
      let exp = substitute body env
      in
      if (reducible exp) then evaluate (oneStep exp)
      else exp
    end
    ...

```

Note that `substitute` is an auxiliary function. Finally, we check if the expression is reducible , and if this is the case, we move into a one-step evaluation of the expression.

Checking the reducibility of an expression is mostly trivial; however, as mentioned in **section 3**, the notion of reduction is slightly nuanced when it comes to relations. For a relation to be irreducible, all of its value must be boolean or integer constants; they cannot be boolean or integer expressions that may be further evaluated. To that end, we check for reducibility of relations as using an auxiliary function as follows:

```
let reduce_relation rel =
let Relation (rel_type , lol) = rel
in
not (
List.for_all
(
  fun a ->
    (* All values must be either ints or bools *)
    List.for_all
    (fun x ->
       match x with
```

```
            |IntConst _ | BoolConst _ -> true
            | _ -> false
        )
        a
    )
    lol
    )

```

The one-step evaluation process is similar to that of ePL's: given an expression wrapped in a constructor, we check whether the sub expressions of the expressions are reducible or not. This process continues recursively until a sub expression cannot be reduced any further, at which point we must contract it. For instance, the following code snippet illustrates one step evaluation for join:

```
    ...
     | Join (rel_exp1 , rel_exp2 , ty) ->
      if reducible rel_exp1
      then  Join (oneStep rel_exp1 , rel_exp2 , ty)
      else
      if reducible rel_exp2
      then  Join (rel_exp1 , oneStep rel_exp2 , ty)
      else contract e
      ...

```

As mentioned earlier, relations themselves may be reducible. A one-step evaluation of them consists of folding across its list of lists and evaluating each expression therein. The following code snippet illustrates this:

```
    ...
    (* One step evaluation for relational databases. *)
    | Relation (rel_type , lol) ->
        let new_lol =
        List.fold_right
        (
          fun a b ->
          (
             List.fold_right
             (
                fun x y ->
                (
                  oneStep x
                )::y
             )
             a
             []
          ) :: b
        )
        lol
        []
        in ...
```

```
23
```

Note that the one step evaluation of a relation with evaluable expressions results in a relation that does not require to be further contracted. At a practical level, this feature of relations allows us to include integer and boolean expressions in relations, which may prove to be useful in a variety of cases (refer to some of the test code to view this feature).

Finally, expressions that cannot be reduced further by one-step evaluation must be contracted. The contraction of relational expressions involving the selection, projection, and join operations are non-trivial, and are discussed in what follows:

1. **Projection**
   Since we verified whether a projection expression was valid or not during type inference, we are certain that the contraction will not run into exceptional situations (barring division by 0 errors). Consequently, we fold across the list of lists in the relation, and further, fold across each list, adding only those elements that belong to the projection tags. This is achieved using a `List.fold_right2` function as follows:

```
1        ...
2        (* fold_right2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b
     list -> 'c -> 'c *)
3                (* Check if the tag for a given column is
4                present in the tags list. If this is the case,
5                append the column value to the row, else don't. *)
6                List.fold_right2
7                  (
8                    fun x y z ->
9                      let ( id , _ ) = y
10                     in
11                     if (List.mem id tags)
12                     then x::z
13                     else z
14                 )
15               a
16               tag_type_lst
17               []
18               ...
19
20
```

   Finally, we returns a relation that contains the type inferred during the inference process, and the new list of lists obtained by the fold-right process described above.

2. **Join**
   To contract the join of two relations, r_1 and r_2 , we first obtain their common

tags. The existence of common tags is guaranteed by our earlier type inference process. Next, we iterate through the list of lists for the first relation, and combine each list with the tag-type pair list from r_1's type declaration. Next, within one iteration through r_1, we iterate through r_2's list of lists, and obtain the same combination as above for each of its lists. Combining the said lists in this manner enables us to keep track of the the tag/column to which a value belongs. Consequently, we may now check if for common tags/columns, the values are the same (note that `comb1,comb2` refer to the combinations of lists that we just described):

```
      ...
      (* For each elements of each row of the second relation,
                  check whether it is present in the corresponding row of
                  the first relation. If so, increase a counter. Check if
                  the counter is equal to the number of common columns
                  earlier computed. *)
          if (List.length common =
                List.fold_right
                (
                  fun f g ->
                    if List.mem f comb1
                    then (g+1)
                    else g
                )
                comb2
                0)
                (* If this is the case, filter
                the second list of the basis of the common tabs and attach it to the first row.
                This enables multiple usage of rows in the first relation. *)
                then
                  let joined_row = a @
                  List.map
                  (fun (var , _ ) -> var)
                  (
                  (* Removing those values from the second list that belong
                  to the common columns. This is because they are already present in the first
                  list. *)
                    List.filter
                    (
                      fun (_ , tag_ty_pair) ->
                        Pervasives.not (List.mem tag_ty_pair common)
                    )
                    comb2
```

```
                        )
                        ...

```

Note that because tags do not repeat, if the number of elements in `comb2` common with `comb1` is equal to the length of the common list, then a sound join is guaranteed. The joining of the rows is illustrated in the lower half of the code snippet above. Once we have combined all possible rows in this manner, we obtain a new list of lists that is returned along with the type inferred during type inference in the `Relation` constructor.

3. **Selection**
   For a selection operation involving two tags, we follow the strategy of combining each list in the list of lists of the relation with the tag-type list of the relation type as we iterate through the list of lists. Following this, we obtain the values corresponding to the two tags of the selection operation for each row, and check if the binary logical relation between them holds or not. If it does, then we add the row to our new list of lists. This is illustrated as follows:

```
      (* For each tuple, combine the elements with the
corresponding tag_type values.
       Then filter the list to retain only those tags that are
required in the
       selection. *)
         let comb = List.combine tag_ty_pair a
         in
         let [(_ , v1)] =
         List.filter
         (
           fun ( (tag , _ ) , _) ->
              tag = id1
         )
         comb
         and  [(_ , v2)] =
         List.filter
         (
           fun ( (tag , _ ) , _) ->
              tag = id2
         )
         comb
         in
         (* Finally check if the predicate is met. If this is
the case,
         attach the tuple to the new list. If not, continue. *)
         if (selection_predicate op v1 v2)
         then a::b
         else b

```

Note that `selection_predicate` is an auxiliary function that given a logical operator and two values, returns then result of applying to the operator to the

two values.
For a selection operation involving one tag and one value (`Selection2`), we follow the above process, but only for the one tag that is provided. We provide the `selection_predicate` function with the value so obtained by filtering through the combination list, and the value provided as part of the selection operation.

With contraction, we evaluate the meaning of the input rSPL program. In our case, relations containing integer and boolean constants constitute these meaning. The output relation is then printed in the terminal.

6 Testing rSPL Implementation

To test our rSPL implementation, we provide 14 test files. To run them, use the shell command `./spli file_name` . A brief description of the input files is as follows:

1. **r1.rspl , r1a.rspl , r2.rspl , r3.rspl**
   These rSPL programs deal with the most basic type of data sets, namely those that map user ids to age, gender, and phone numbers. The programs exhibit the functionality of the relational operators with these relations.

2. **r4.rspl , r5.rspl , r6.rspl , r7.rspl , r7a.rspl**
   These programs include slightly more sophisticated data sets that involve examination marks, and contain integer expressions that must be evaluated. These files show case rSPL's ability to evaluate expressions within relations. In addition, they exhibit the nested functionality of relational operations,

3. **r8.rspl , r9rspl**
   These programs contain data sets that are more closely aligned with real world data sets in terms of their tags and continue an exhibition of rSPL's operational capabilities.

4. **r10.rspl , r11.rspl , r12.rspl**
   These files contain data sets with type errors. They are to illustrate rSPL's type-safety feature.

7 Conclusion

This report has outlined the formalisms, implementation, and testing resources for the relational extension of sPL, rSPL. We defined a new syntax for relational types and expressions, discussed rSPL's dynamic semantics, extended sPL's typing relation

with new relation typing rules, and finally described in detail the implementation of the DSL. In particular, the implementation involved significant additions to sPL's parser and type checker, and writing an interpreter based on ePL's interpreter.

While the test files illustrate the functionality of rSPL, there are a few shortcomings, which could be fixed in future versions of rSPL. They are as follows:

1. **Absence of String Data**
   rSPL only handled boolean and integer data. This is a significant shortcoming, since most real world data sets will involve considerable usage of strings/factors.

2. **Few Relational Operations**
   rSPL provides only three relational operations. Many more operations are required for a comprehensive handling of data sets. We failed to add more operations because the three operaions took a signficant amount of time to implement.

3. **Inefficient Code**
   Some of our type-checking and interpreter code handles operators inefficiently by performing redundant iterations, or implementing naive algorithmic solutions. While these inefficiencies are not an issue with small data sets, large data sets (as those in the real world) will take significantly longer to process. We were unable to improve this owing to time constraints.

4. **Type Annotation Requirement**
   While we eliminated type annotations for the `let in` construct body, we have not done so for variable declarations. This makes rSPL code verbose and clunky. These could be eliminated in the future.

5. **Absence of Commenting Facility**
   We were unable to modify the sPL parser to include a commenting facility. This makes rSPL code relatively unreadable.

That said, designing and implementing rSPL has been a learning experience, and we have gained much from it, including improved programming skills, a better understanding of the design and implementation of programming languages, and a great deal of respect for those who professionally design languages. Hopefully, this project will serve as a starting point for future endeavours.