Course 02263 Mandatory Asssignment 1, 2018

Anne E. Haxthausen, DTU Compute

September 12, 2018

1 Practical Information

- The assignment should be solved in groups of two persons.
- The solution should be made in the form of a group report.
  - Full names and study numbers should appear on the front page.
  - The report must contain RSL specifications of the problems described in the assignment and informal explanations elaborating on the RSL specification.
  - It is strongly recommended to use LaTeX to produce your report. A LaTeX skeleton for the report is provided: `reportskeleton.tex`. It should be used together with a modified version of the listings package: `rsllisting.sty` that is also provided. You can find more documentation about LaTeX e.g. on `http://tex.loria.fr/english/general.html`
- It is a requirement that all specifications, types and values have names as required in the assignment text. (We need that when evaluating your solution, as we plan to run an automatic test on your specifications.)
- It is a requirement that all specifications have been type checked successfully with the RAISE tools, and that your test specification has been translated successfully with the RAISE tools. Solutions that do not fulfill these requirements will not be considered.
- One of the group members must upload (a) all your specifications (i.e. the files *Basics.rsl*, *Requirements.rsl*, *Design.rsl*, and *DesignTest.rsl*) and (b) a pdf-file containing the group report on DTU Inside under Assignment/Opgaver **not later than** Friday 12 October at 15:00. Remember to add your group partner to the DTU Inside group during the hand-in so all names of the group become visible. In addition to the electronic hand-in on DTU Inside, each group must also hand in a printed version of the report (inside a closed envelope on which you have written your signatures and study numbers) in the box labelled 02263 in front of my office (room 054) in building 303B.

2 Problem: a table planner

This assignment concerns the problem of allocating tables to the persons of a party in such a way that persons that are in the same family are not seated at the same table. People assigned to a table can seat as they want at that table.

In this assignment you are asked first to make a requirement specification for a function *plan* that (1) takes as input a set of families that are going to attend a party and (2) returns a table plan for all the persons in the families such that persons that are in the same family are not seated at the same table. Then you are asked to give an explicit (algorithmic) definition of this function such that it satisfies the requirement specification. It should be noted that for a given problem instance there are usually many solutions to this problem and consequently many algorithms for producing a solution.

On DTU Inside, among the assignment files, you will find skeletons for some of the specifications.

1. Complete the scheme *Basics* from DTU Inside (stored in a file named *Basics.rsl*).

   (a) The specification includes the following type declaration:

   type
   Person = **Text**,
   Family = Person-**set**,
   Families = Family-**set**,
   Table = Person-**set**,
   Plan = Table-**set**

   The types are used as follows: *Person* to represent identifiers of persons, *Family* to represent a family as a set of persons in the family, *Families* to represent all the families that are going to attend the party, and *Plan* to represent a table plan. A plan is modelled as a set of tables, and a table is modelled as a set of persons (that should be seated at that table).

   Example
   One out of the many possible values of type *Families* is the following one which represents four families:

```
{{"Elisabeth"},
 {"Lillian", "Erik"},
 {"Frederik", "Henrik", "Anne"},
 {"Lotte", "Torsten", "Camilla", "Jacob", "Pernille"}
}
```

   One out of many possible plans for this collection of families could be:

```
 {{"Torsten","Frederik"},
  {"Lillian","Camilla","Anne"},
  {"Jacob","Henrik"},
  {"Erik","Pernille","Elisabeth"},
  {"Lotte"}
}
```

In this case there are five tables. Note that there are many other possible plans for these families.

(b) The specification should include explicit definitions of auxiliary functions that can be applied in the specification of the *plan* function in item 3, below. E.g. specify explicitly a function, *areRelatives*, that can be used to test whether two persons are in the same family.

In the report you must informally explain the purpose of the auxiliary functions (in the same style as *areRelatives* was explained above).

(c) Ensure that your spec is within the translatable subset of SML as explained in appendix A.

2. Type check the *Basics* scheme.

3. Consider the scheme *Requirements* from DTU Inside. It extends the *Basics* scheme with a formal requirement specification of a *plan* function that is intended to take a set of families as argument and to return a plan for the given persons such that family members are not seated at the same table:

   plan : Families $\xrightarrow{\sim}$ Plan
   plan(fs) **as** p
   post isCorrectPlan(p, fs)
   pre isWellformed(fs)

   The pre condition should express requirements to the input of the *plan* function, i.e. which *Families* values *plan* is allowed to be applied to. The post condition should express the requirements in the form of a relation between input fs and output p. Any solution to the table assignment problem should satisfy these requirements, so the requirements must not be biased towards a particular solution.

   Add to the *Basics* scheme explicit definitions of the two functions having the following signatures:

   isWellformed : Families → **Bool**
   isCorrectPlan : Plan × Families → **Bool**

   You may also need to add definitions of more auxiliary functions that you need to define these two functions.

In the report you must also informally explain what the requirements you have formalized in $isCorrectPlan(p, fs)$ and $isWellformed(fs)$ are. Actually it is recommended to write the informal requirements before defining the two functions.

4. Type check the *Requirements* scheme.

5. Now complete the scheme *Design* from DTU Inside. It should be just like *Requirements* except that the *plan* function should now be defined explicitly, i.e. you have to select an algorithm that produces one of the solutions. The algorithm needs not to be optimal, but shouldn't be too trivial (e.g. making one table for each person). You may need to define auxiliary functions in *Design* in order to define the *plan* function. Ensure that your spec is within the translatable subset of SML as explained in appendix A. In the report you must also informally explain the idea behind your algorithm.

   Hint: There is an extension to RSL according to which the **hd** operation can be applied to a non-empty set and then it will return some (arbitrary) value from that set. You may need this in some of your function definitions.

6. Type check the *Design* scheme.

7. Translate *Design* to SML.

8. Complete the scheme *DesignTest* from DTU Inside. It should extend *Design* with some test cases that test your *plan* function and in particular test that *plan* creates output that satisfies the requirements that you stated in the requirement specification. One of the tests should create a plan for the four families of the example shown above (under item 1). Your plan might be different from the plan shown there. Also test the $isWellformed$ function and other auxiliary functions. You will be evaluated on how thoroughly you test your *Design*.

9. Type check *DesignTest*.

10. Translate *DesignTest* to SML and execute the test cases.

    The results of executing the test cases must be shown in your report. Also tell whether the results are as expected.

A Rewriting expressions into a translatable form

The tool `rsltc` can translate RSL specifications into SML programs provided that they are in a certain form as explained in the UNU/IIST user guide which can be found on DTU Inside in the Tools folder. Below, you find the required form for quantified expressions and comprehended expressions.

A.1 Universal quantification

Universal quantification can only be translated by `rsltc` if they take one of the forms:

$\forall$ x : type_expr • x $\in$ set_expr
$\forall$ x : type_expr • x $\in$ set_expr $\Rightarrow$ logical_expr
$\forall$ x : type_expr • x $\in$ set_expr $\wedge$ logical_expr_1 $\Rightarrow$ logical_expr_2

A.2 Existential quantification

Existential quantification can only be translated if they take one of the forms:

$\exists$ x : type_expr • x $\in$ set_expr
$\exists$ x : type_expr • x $\in$ set_expr $\wedge$ logical_expr
$\exists!$ x : type_expr • x $\in$ set_expr
$\exists!$ x : type_expr • x $\in$ set_expr $\wedge$ logical_expr

A.3 Nesting quantification

A quantified expression of the form

($\forall$ x, x$'$: type_expr • x $\in$ set_expr $\wedge$ x$'$ $\in$ set_expr$'$ $\Rightarrow$ logical_expr)

cannot be translated by `rsltc`. However, the expression can be rewritten into a translatable form:

($\forall$ x : type_expr • x $\in$ set_expr $\Rightarrow$
  ($\forall$ x$'$: type_expr • x$'$ $\in$ set_expr$'$ $\Rightarrow$
    logical_expr
)
)

A.4 Comprehended set expressions

Comprehended set expressions can only be translated if they take one of the forms:

{ expr | x : type_expr • x $\in$ set_expr }

{ expr | x : type_expr • x $\in$ set_expr $\wedge$ logical_expr }