Course 02263 Mandatory Asssignment 1, 2022

Anne E. Haxthausen
DTU Compute

September 20, 2022

1 Practical Information

- The assignment must be solved in the already registered groups of four (or five) persons. (The deadline for registration was 19 September.)
- The solution to the assignment should be made in the form of a group report.
  - Full names and study numbers of all persons 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 fulfil these requirements will not be considered.
- One of the group members must upload (a) a zip file containing all your specifications (i.e. the files *ColouringBasics.rsl*, *ColouringReq.rsl*, *ColouringEx.rsl*, and *testColouringEx.rsl*) and (b) a pdf-file containing the group report on DTU Learn under Assignment/Opgaver **not later than** Friday 14 October at 15:00, 2022.

2 Problem

This assignment concerns the problem of colouring the pieces of a puzzle such that pieces that are neighbours get different colours. Pieces can have any shape. Two pieces are considered as being neighbours if they have a common piece of border.[1] An example of a colouring of a puzzle is given in Figure 1. It should be noted that there are also many other possible colourings of this puzzle.

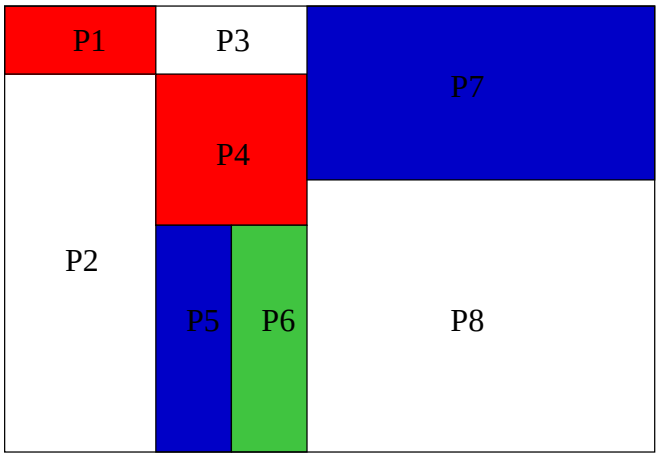

Figure 1: Example of colouring of a puzzle.

In this assignment you are asked first to make a requirement specification for a function *col* that given a neighbouring relation between the pieces of a puzzle produces a colouring of these pieces such that neighbours do not get the same colour. 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 puzzle there are usually many possible colourings and there are many different algorithms for producing a colouring. You should use one such algorithm for your function definition.

The specifications should be expressed without using imperative language constructs like loops.

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

[1] Note that, if two pieces (like P1 and P4 in Figure 1) touch each other in a single point, they are not considered to be neighbours.

2.1 Requirements Specification

1. Consider the scheme *ColouringBasics* from DTU Learn (stored in the file named *ColouringBasics.rsl*).

```
scheme ColouringBasics =
class
  type
    Piece = Text,
    Relation = (Piece × Piece)-set,
    Colour = Piece-set,
    Colouring = Colour-set

  value /∗ auxiliary functions ∗/

    isRelation : Relation → Bool
    isRelation(r) ≡ ...,

    areNb : Piece × Piece × Relation → Bool
    areNb(cn1, cn2, r) ≡ ...,

    isCorrectColouring : Colouring × Relation → Bool
    isCorrectColouring(cols, r) ≡ ...,

    ...
end
```

   (a) The scheme includes four type definitions.

   The idea is that *Piece* should be used to represent identifiers of pieces, *Relation* to represent neighbouring relations, and *Colouring* to represent colourings of pieces of a puzzle.

   A relation is represented as a set of pairs of neighbouring pieces. For instance, the neighbouring relation of the puzzle shown in Figure 1 can be represented by the RSL value

```
{("P1","P2"), ("P1","P3"),
 ("P2","P4"), ("P2","P5"),
 ("P3","P4"), ("P3","P7"),
 ("P4","P5"), ("P4","P6"), ("P4","P7"), ("P4","P8"),
 ("P5","P6"),
 ("P6","P8"),
 ("P7","P8")}
```

   Note that there are other possible ways of representing the neighbouring relation of the puzzle shown in Figure 1.

   A colouring is represented as a set of colours, where a colour is represented as a set of pieces that should be given the same unique colour. The colouring shown in Figure 1 can hence be represented by the RSL value

```
{{"P1","P4"}, {"P6"}, {"P2","P3","P8"}, {"P5","P7"}}
```

In this colouring "P1" and "P4" are given one colour, "P6" another colour, "P2", "P3" and "P8" a third colour, and "P5" and "P7" are given a fourth colour.

(b) *ColouringBasics* should also include explicit definitions of *auxiliary functions* that can be applied in the specification of the *col* function, e.g. a function, *areNb*, that can be used to test whether two pieces of a puzzle are neighbours. Complete the definition of this function and add definitions of any other auxiliary functions that you would like to use in later steps[2]. You will in later steps be asked to complete the definitions of two functions named *isRelation* and *isCorrectColouring*.

In the report you must informally explain the purpose of all auxiliary functions you have introduced. The explanation should be in the same style as shown for *areNb* above.

2. Consider the scheme *ColouringReq* from DTU Learn.

ColouringBasics

scheme ColouringReq =
extend ColouringBasics **with**
class
  value /∗ requirement spec ∗/
    col : Relation $\xrightarrow{\sim}$ Colouring
    col(r) **as** cols
    post isCorrectColouring(cols, r)
    pre isRelation(r)
end

It extends the *ColouringBasics* scheme with a requirement specification of a function *col* that is intended to take a neighbouring relation between the pieces of a puzzle as argument and to return a colouring of the pieces described by the neighbouring relation such that neighbouring pieces do not get the same colour.

The pre condition should express requirements to the input of the *col* function, i.e. which *Relation* values the *col* function is allowed to be applied to. In this case it is chosen to be expressed in terms of the *isRelation* function.

The post condition should express the requirements in the form of a relation

$$isCorrectColouring(cols, r)$$

that must hold between input r and output *cols*. Any solution to the colouring problem should satisfy these requirements, so the requirements must not be biased towards a particular solution.

3. In the report state informally the requirements a value r of type *Relation* must satisfy in order to be wellformed, i.e. for representing a neighbouring relation.

Now the informal requirements should be formalised: In the *ColouringBasics* scheme, complete the definition of the function

[2] You might wait making these additions until later steps where you find out what you need.

isRelation : Relation → **Bool**

such that the function can be used to test whether a value in the type *Relation* is wellformed. For instance, it should be posible to use the function to show that the neighbouring relation shown in item 1(a) for the puzzle in Figure 1 is wellformed.

4. In the report explain informally the required relation between input r and output $cols$ of the *col* function, i.e. explain what it means for $cols$ to be a legal colouring for r.

   In the *ColouringBasics* scheme, complete the definition of the function

   isCorrectColouring : Colouring × Relation → **Bool**

   such that $isCorrectColouring(cols, r)$ formalises the required relation between input r and output $cols$ of the *col* function.

5. Type check the *ColouringBasics* scheme and the *ColouringReq* scheme. If there are any error messages, you must fix the errors.

6. Translate *ColouringBasics* to SML (after having ensured that the specification only uses language constructs within the translatable subset of SML as explained in appendix A). If there are any error messages, you must fix the errors. Note that the *ColouringReq* scheme is not translatable to SML and should not be.

2.2 Explicit Specification

1. Consider the scheme *ColouringEx* from DTU Learn.

   ColouringBasics

```
scheme ColouringEx =
extend ColouringBasics with
class
  value
    col : Relation ~→ Colouring
    col(r) ≡ ...
    pre isRelation(r)
end
```

   You should complete this specification and show the result in the report. The *col* function should be defined explicitly, i.e. you have to select an algorithm that produces one of the possible colourings. The algorithm needs not to generate a colouring which is optimal in the sense that it uses as few colours as possible, but it should not be too trivial.

   You may need to define auxiliary functions in *ColouringEx* in order to define the *col* function. Ensure that your spec becomes 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.

2. Type check the *ColouringEx* scheme. If there are any error messages, you must fix the errors.

3. Translate *ColouringEx* to SML. If there are any error messages, you must fix the errors.

2.3 Testing by Translation to SML

1. Complete the scheme *testColouringEx* from DTU Learn and show the result in the report. The scheme should extend *ColouringEx* with some test cases that test your *col* function and in particular test that *col* creates output that satisfies the requirements that you stated in the requirement specification. One of the tests should create a colouring of the puzzle shown in Figure 1. Your colouring might be different from the colouring shown in the figure. Also test the *isCorrectColouring* function and other auxiliary functions. You will be evaluated on how thoroughly you test your specification.

   In the report you must explain your test strategy and make clear what the purpose of the individual test cases are.

2. Type check *testColouringEx*. If there are any error messages, you must fix the errors.

3. Translate *testColouringEx*. If there are any error messages, you must fix the errors.

4. 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 Learn 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 $\Rightarrow$ 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:

```
(∀ x : type_expr •  x ∈ set_expr ⇒
   (∀ x′ : type_expr • x′ ∈ set_expr′ ⇒
      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 }