Course 02263 Mandatory Assignment 1, 2016

Ali Chegini (s150939)
Quentin Tresontani (s151409)

March 16, 2016

Contents

1 Introduction

This document contains a solution for the problem of colouring the pieces of a puzzle such that no neighbours get same colors. The solution is proposed for unlimited number of colors.

2 Requirements Specification

This section is specifying the requirements for the colouring of a relation. *ColouringBasics* is defining functions which will be used in *ColouringReq*.

2.1 Auxiliary functions for requirements

The class *ColouringBasics* contains the methods *isRelation*, *isCorrectColouring* and the auxiliary functions we have used to implement them.

The function *isRelation* is used to check if a relation (i.e. (Piece >< Piece)-set) is well formed. We are checking that all couples of the relation are composed of two distinct pieces. If this is the case, the function will return true. We assume here that it is not a problem for us to have too many couples. For instance, having (P1, P2) and (P2, P1) in the same relation does not give more information and is not considered as an error.

The function *isCorrectColouring* is used to check if a coulouring (i.e. (Piece-set)-set) is well formed for a given relation. We are checking three different properties in the colouring:

- Unicity of each piece of the relation. Each piece can only have one colour, so each piece should be at most once in the colouring.
- Exhaustivity for each piece of the relation. Each piece of the relation should be coloured, so each piece should be at least once in the colouring.
- Neighborhood relations. Each piece of the relation should have a different colour from its neighbours.

Unicity and exhaustivity could be put together in a condition saying that each piece of the relation should be exactly once in the colouring. However, we are testing this properties separately in the code.

The following piece of code includes comments for each method:

scheme ColouringBasics =
class
    type
        Piece = **Text**,
        Relation = (Piece $\times$ Piece)**-set**,

```
                Colour = Piece-set,
                Colouring = Colour-set

        value

/* The point is to check if a piece is in the colouring (i.e. (Piece−set)−set)
 *
 * params
 *              cn1 − piece we want to check
 *              colouring − colouring which cotaining colours (Piece−set)
 *
 * returns
 *              true − if the piece is in the colour set
 *              false − if the piece is not in the colour set
 */
 isInColouring : Piece × Colouring → Bool
 isInColouring(cn1, colouring) ≡
  (∃colour : Colour • colour ∈ colouring ∧
                (∃p1 : Piece • p1 ∈ colour ∧ (p1 = cn1))),

/* Test if a couple in the relation has the same colouring
 *
 * params
 *              cn1 − first piece we want to test
 *              cn2 − second piece we want to test
 *              cols − colouring which contains the colours
 *
 * returns
 *              true − if Piece already in colour set
 *              false − if Piece not in color set
 */
isSameColour : Piece × Piece × Colouring → Bool
isSameColour(cn1, cn2, cols) ≡
    (∃colour1 : Colour • colour1 ∈ cols ∧ cn1 ∈ colour1
        ∧ (∃colour2 : Colour • colour2 ∈ cols ∧ cn2 ∈ colour2 ∧ (colour1 = colour2))),

/* Defining that for all pieces (p1, p2) defining couples in a relation,
 * we have p1 different from p2.
 * The point is to check that the relation doesn't have any couple defined
 * with the two times the same piece.
 * For instance: {p1, p1}
 *
 * params
 *              r − relation we want to test
 *
 * returns
```

```
 *                  true − if the relation is well−formated
 *                  false − if the relation is not well−formated
 */
isRelation : Relation → Bool
isRelation(r) ≡
   (∀p1 : Piece • p1 ∈ {let (p1, p2) = e in p1 end | e : Piece × Piece • e ∈ r} ⇒
                  (∀p2 : Piece • p2 ∈ {let (p1, p2) = e in p2 end | e : Piece × Piece • e ∈ r} ⇒
                          ((p1, p2) ∈ r ⇒ (p1 ≠ p2))
                  )
         ),

/* Defining two pieces p1 and p2 as parts of a couple which belongs to r,
 * then, we have the implication that if cn1 and cn2 are neighbour,
 * they should be equal to p1 and p2.
 *
 * The point is to check that it exists (or not) two pieces (p1, p2)
 * from the relation which are neighbours.
 * Then, we check if these pieces are equals cn1 and cn2.
 *
 * params
 *                  cn1 − first piece we want to check
 *                  cn2 − second piece we want to check
 *                  r − relation where are all the neighborhoods relation are defined
 *
 * returns
 *                  true − if pieces are neighbors according to the relation
 *                  false − if pieces are not neighbors according to the relation
 */
areNb : Piece × Piece × Relation → Bool
areNb(cn1, cn2, r) ≡
      (∃(p1, p2) : Piece × Piece • (p1, p2) ∈ r ∧
                  ((cn1 = p1 ∧ cn2 = p2) ∨ (cn1 = p2 ∧ cn2 = p1)))
pre isRelation(r),

/* A colouring is correct if:
 *                  − all couples of neighbours have different colours
 *                  − all pieces of the relation are only once in the colouring
 *                    since one Piece can have only one colour
 *                  − all pieces of the relation are there in the colouring at least once
 *
 * params
 *                  cols − colouring used on the pieces
 *                  r − relation defining neighborhood relations
 *
 * returns
 *                  true − if colouring is done correctly
```

```
 *                    false  − if colouring is not done correctly
 */
isCorrectColouring : Colouring × Relation → Bool
isCorrectColouring(cols, r) ≡
      neighborCondition(cols, r) ∧ unicityCondition(cols, r) ∧ exhaustivityCondition(cols, r)
pre isRelation(r),

/* Fonction used to check that all pieces
 * of the relation are in colouring */

exhaustivityCondition : Colouring × Relation → Bool
exhaustivityCondition(cols, r) ≡
  (∀p1 : Piece • p1 ∈ {let (p1, p2) = e in p1 end | e : Piece × Piece • e ∈ r} ⇒
                  (∀p2 : Piece • p2 ∈ {let (p1, p2) = e in p2 end | e : Piece × Piece • e ∈ r} ⇒
                           (∃col : Colour • col ∈ cols ∧ p1 ∈ col) ∧
                                    (∃col : Colour • col ∈ cols ∧ p2 ∈ col)
                  )
          ),

/* Function used to check that neighbour pieces have different colours */

neighborCondition : Colouring × Relation → Bool
neighborCondition(cols, r) ≡
       (∀p1 : Piece • p1 ∈ {let (p1, p2) = e in p1 end | e : Piece × Piece • e ∈ r} ⇒
                  (∀p2 : Piece • p2 ∈ {let (p1, p2) = e in p2 end | e : Piece × Piece • e ∈ r} ⇒
                           (areNb(p1, p2, r)) ⇒ ∼isSameColour(p1, p2, cols)
                  )
          ),

/* Functions used for the unicity of each Piece
 * from the relation in the colouring */

/* Unicity is proved by showing that for all pieces P in the relation,
 * it doesn't exist any couple of colours (c1, c2),
 * so that c1 and c2 are different and the piece P is in both of them.
 */
unicityCondition : Colouring × Relation → Bool
unicityCondition(cols, r) ≡
        (∀p1 : Piece • p1 ∈ {let (p1, p2) = e in p1 end | e : Piece × Piece • e ∈ r} ⇒
                  (∀p2 : Piece • p2 ∈ {let (p1, p2) = e in p2 end | e : Piece × Piece • e ∈ r} ⇒
                           (unicityConditionInside(cols, p1) ∧ unicityConditionInside(cols, p2))
                  )
          ),

/* Function used to prove that the piece cn1 is only in one colour of the colouring.
 */
```

unicityConditionInside : Colouring $\times$ Piece $\rightarrow$ **Bool**
unicityConditionInside(cols , cn1) $\equiv$
$\sim(\exists$colour1 : Colour $\bullet$ colour1 $\in$ cols $\wedge$ cn1 $\in$ colour1 $\wedge$
$(\exists$colour2 : Colour $\bullet$ colour2 $\in$ cols $\wedge$ cn1 $\in$ colour2 $\wedge$ (colour1 $\neq$ colour2)))

end

2.2 Colouring Requirements

The class *ColouringRes* contains an implicit function using the *isRelation* and *isCorrectColouring* methods.

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

The purpose of this class is to create a correct colouring for a well formed relation, as explained in the description of the functions *isRelation* and *isCorrectColouring*.
This class has been type checked, but it has not been translated to RSL and no tests have been performed on it since the SML translation does not support implicit functions.

3 Explicit Specification

3.1 Explanation on the algorithm created

The following section provides an explicit declaration of a function returning a correct colouring for a well formed relation.

The function starts by performing a recursive treatment on the relation in the method *treatmentOnRelation*. In order to apply some specific treatment to one couple of the relation (i.e. a (Couple)-set), we extract the first element of the relation and then perform *treatmentOnRelation* recursively on the remaining tail. We have added an empty set verification to stop this recursion. This way, we can make recursive calls without knowing the number of couples inside the relation.
The treatment made on couple is done inside *treatmentOnCouple*. This function is performing a recursive call of *treatmentOnPiece* which is putting the current piece at the right place in the colouring considering the neighbourhood relationships.

This class is so composed of three types of functions:

- auxiliary functions - used to perform small tasks or verifications
- treatment functions - used to perform treatment in a recursive way on Relations, Couples of pieces and Pieces
- the function col - the function starting the algorithm

In the code, comments are provided in the header of each function in order to explain its purpose:

3.2 ColouringEx.rsl

```
ColouringBasics

/* This class extends the ColouringBasics in order to
 * provide a method returning a correct colouring for a given relation.
 *
 * This class is composed of:
 *       - auxiliary functions for the small tasks
 *  - treatment functions which are used recursively on Pieces,
 *     Couples of Pieces and Relations
 *  - the main function col
 */

scheme ColouringEx =
extend ColouringBasics with
```

class

value

```
/* Auxiliary functions */

/* This functions checks if the piece can be added or not in the colour being tested.
 * We are checking if there is a neighbour of the piece in the colour
 * and if the piece is already in the colour.
 *
 * params
 *              cn1 − piece we want to check
 *              r − relation defining neighbourhood relations
 *              colour − current colour we want to test
 *
 * returns
 *              true − if Piece can be added to the colour
 *              false − if Piece cannot be added to the colour
 */
canAddPieceToColour : Piece × Relation × Colour → Bool
canAddPieceToColour(cn1, r, colour) ≡
   ∼((∃p1 : Piece • p1 ∈ colour ∧ areNb(p1, cn1, r))) ∧ ∼(cn1 ∈ colour),


/* This function checks if the piece can be added to the existing colours of the colouring
 *
 * params
 *              cn1 − piece we want to check
 *              r − relation defining neighbourhood relations
 *              colouring − Colouring to test
 *
 * returns
 *              true − if the piece can be added in a color of the colouring
 *              false − if the piece cannot be added to the colour of the colouring
 */
canAddPieceToExistingColours : Piece × Relation × Colouring → Bool
canAddPieceToExistingColours(cn1, r, colouring) ≡
 (∃colour : Colour • colour ∈ colouring ∧ canAddPieceToColour(cn1, r, colour))
      ∧ ∼pieceInColouring(cn1, colouring),


/* This function is used to add a colour to a new Colour (i.e. Piece−set)
 *
 * params
 *              cn1 − piece to add to the colour
 *              r − relation defining neighbourhood relations
 *              colour − current colour we want to add the piece
```

```
 *
 * returns
 *             colour − colour in the parameters with potentially the new piece
 */
addPieceToColour : Piece × Relation × Colour × Colouring → Colour
addPieceToColour(cn1, r, colour, colouring) ≡
    if canAddPieceToColour(cn1, r, colour) ∧ ∼pieceInColouring(cn1, colouring)
      then colour ∪ {cn1}
      else colour
      end,

pieceInColouring : Piece × Colouring → Bool
pieceInColouring(cn1, colouring) ≡
      (∃colour : Colour • colour ∈ colouring ∧ cn1 ∈ colour),

/* Treatment functions */

/* This fonction shows the treatment made on a Piece
 * that is added to the Colouring (.i.e. {Piece−set}−set)
 *
 * params
 *             cn1 − piece to add to the colour
 *             r − relation defining neighbourhood relations
 *             colouring − current colouring where we want to add the piece
 *
 * returns
 *             colouring − colouring from the input with potentially a new Piece
 */
treatmentOnPiece : Piece × Relation × Colouring → Colouring
treatmentOnPiece(cn1, r, colouring) ≡
    if canAddPieceToExistingColours(cn1, r, colouring)
      then
             let colour = hd { c | c : Colour • c ∈ colouring ∧ canAddPieceToColour(cn1, r, c)},
                    colouring = colouring \ {colour},
                    colouring = colouring ∪ {addPieceToColour(cn1, r, colour, colouring)}
             in
                    colouring
             end
      else elseCondition(cn1, colouring)
      end,


/* This function extends the treatmentOnPiece function in the case of a Piece
 * that cannot be added to an existing Colour (i.e. Piece−set)
 * This Piece should so either not be added at all (if the Piece is already in the Colouring),
 * or should be added in a new Colour.
```

```
 *
 * params
 *                cn1 − Piece to add to the colouring
 *                colouring − Current colouring where we want to add the piece
 *
 * returns
 *                colouring − Colouring from the input with potentially a new Piece
 */
elseCondition : Piece × Colouring → Colouring
elseCondition(cn1, colouring) ≡
   if ∼(∃colour : Colour • colour ∈ colouring ∧ cn1 ∈ colour) ∧ ∼pieceInColouring(cn1, colouring)
        then colouring ∪ {{cn1}}
        else colouring
        end,

/* This fonction shows the treatment made on a couple of Pieces
 * that should be added to the Colouring (.i.e. {Piece−set}−set)
 * This fonction is recursive and calls treatmentOnPiece.
 *
 * params
 *                couple − couple of Pieces to add to the colouring
 *                r − relation defining the neighborhood relationships
 *                colouring − Current colouring where the couple of Pieces should be added
 *
 * returns
 *                colouring − Colouring from the input with potentially a new couple of Piece
 */
treatmentOnCouple : (Piece × Piece) × Relation × Colouring → Colouring
treatmentOnCouple(couple, r, colouring) ≡
 let (p1, p2) = couple
        in treatmentOnPiece(p1, r, treatmentOnPiece(p2, r, colouring))
        end,

/* This fonction shows the recursive treatment made on a relation of Pieces
 * that should be added to the Colouring (.i.e. {Piece−set}−set)
 * First, we split the relation in a couple of Pieces and a tail using the hd method.
 * Then, we and apply the treatmentOnCouple on this couple.
 * Then, we apply recursively the treatmentOnRelation method on the remaining part of the relation.
 * This is done recursively until the relationSet is empty.
 *
 * params
 *                relationSet − relation used to define the set of pieces we are working on
 *                r − relation defining the neighborhood relationships
 *                colouring − Current colouring where the Pieces should be added
 *
 * returns
```

```
 *                  colouring  − Colouring from the input
 */
treatmentOnRelation : Relation × Relation × Colouring → Colouring
treatmentOnRelation(relationSet, r,  colouring) ≡
   if  relationSet  ≠ {}
          then let head = hd relationSet,
                             tail  = relationSet \ {head}
                    in treatmentOnRelation(tail, r,  treatmentOnCouple(head, r, colouring))
                    end
          else colouring
          end,


/* Main function */

tmpColour : Colouring = {{}},

/* This function returns a correct colouring for the relation in input.
 * This function is calling : treatmentOnRelation and starting the use of recursive functions.
 *
 * params
 *                  r − relation defining neighbourhood relations of piece we want to colour
 *
 * returns
 *                  Colouring − colouring considering correctly the neighbourhood relations
 */
col : Relation ~→ Colouring
col(r) ≡
  treatmentOnRelation(r, r, tmpColour)
pre isRelation(r)

end
```

4 Testings for by Translation to SML

The following section provides tests for all the functions presented in the previous files.

4.1 Test Specification

4.1.1 Tests for ColouringBasics

The purpose of the following class is to perform unit tests on all functions of *ColouringBasics*. For all these functions, we tried to test the possible scenarios. The tests are expressed as booleans returning true if the results are the expected ones.

ColouringBasics

```
scheme testColouringBasics =
extend ColouringBasics with
    class
        value
            P1 : Piece = "Piece 1 ",
            P2 : Piece = "Piece 2 ",
            P3 : Piece = "Piece 3 ",
            P4 : Piece = "Piece 4 ",
            r1 : Relation = {(P1,P2), (P1,P3), (P2,P4), (P3,P4)},
            r2 : Relation = {(P1,P1), (P1,P3), (P2,P4), (P3,P4)},
            c1 : Colouring = {{P1,P2}, {P3}},
            c2 : Colouring = {{P1,P2}, {P3}, {P4}},
            c3 : Colouring = {{P1,P4}, {P2}, {P3}}

        test_case
            [isARelation] isRelation(r1) = true,
            [isNotRelation] isRelation(r2) = false,

            [areNeighbours] areNb(P1, P2, r1) = true,
            [areNotNeighbours] areNb(P1, P4, r1) = false,

            [isInColouring] isInColouring(P1, c1) = true,
            [isNotInColouring] isInColouring(P4, c1) = false,

            [isSameColour] isSameColour(P1, P2, c1) = true,
            [isNotSameColour] isSameColour(P1, P3, c1) = false,

            [notCorrectedColoured] isCorrectColouring(c2, r1) = false,
            [correctedColoured] isCorrectColouring(c3, r1) = true
end
```

4.1.2 Tests for ColouringEx - Auxiliary functions

The purpose of the following class is to perform unit tests on all the auxiliary functions that we have used in order to implement the colouring algorithm. For all these functions, we tried to test the possible scenarios.
Most of our tests are expressed as boolean. They return true if the result is the expected one, they return false if not. If they are not, they return a colouring. Comments are put next to the test name in order to explain the expected behaviours and results.

ColouringEx

```
/∗ This class extends the class ColouringEx in order to perform unit tests on
 ∗ all auxiliary functions used for the colouring algorithm. ∗/

scheme testColouringExAuxiliaryFunctions =
extend ColouringEx with
        class
                value
P1 : Piece = "Piece 1",
P2 : Piece = "Piece 2",
P3 : Piece = "Piece 3",
P4 : Piece = "Piece 4",
P5 : Piece = "Piece 5",
r : Relation = {(P1, P2), (P2,P3), (P3, P4)},
r1 : Relation = {(P1, P2), (P2,P3), (P3, P4), (P1, P5)},
c1 : Colour = {P2},
c2 : Colour = {P3, P4},
colouring : Colouring = {{P1, P3}, {P2}},
colouring2 : Colouring = {{P1,P3}}

test_case

/∗ canAddPieceToColour ∗/

[canAddPieceToColour] /∗ False because P2 is in c1 and is neighbour with P1 ∗/
        canAddPieceToColour(P1, r, c1) = false,

[canAddPieceToColour2] /∗ False because P2 is in c1 ∗/
        canAddPieceToColour(P2, r, c1) = false,

[canAddPieceToColour3] /∗ Because no neighbour of P1 in c2 and P1 not in c2 ∗/
        canAddPieceToColour(P1, r, c2) = true,

/∗ addPieceToColour ∗/

[addPieceToColour] /∗ Should return {P2} because P1 already has neighbour in c1∗/
```

addPieceToColour(P1, r, c1, colouring) = {P2},

[addPieceToColour2] /* Should return {P2} because P2 already in c1 */
addPieceToColour(P2, r, c1, colouring) = {P2},

[addPieceToColour3] /* Should return {P2, P4} because P4 is added in c1 */
addPieceToColour(P4, r, c1, colouring) = {P2, P4},

/* canAddPieceToOneColour */

[canAddPieceToExistingColours] /* True because P4 can be added to P2's colour */
canAddPieceToExistingColours(P4, r, colouring) = **true**,

[canAddPieceToExistingColours2] /* False because P2 cannot be with P1 and P3 */
canAddPieceToExistingColours(P2, r, colouring2) = **false**,

/* treatmentOnPiece */

[treatmentOnPiece] /* Should add P4 with P2 */
treatmentOnPiece(P4, r, colouring) = {{P1, P3}, {P2, P4}},

[treatmentOnPiece2] /* Should not do anything since P2 is already inside */
treatmentOnPiece(P2, r, colouring) = colouring,

[treatmentOnPiece3] /* Should create a new colour for P2, colouring2 will be colouring */
treatmentOnPiece(P2, r, colouring2) = colouring,

[treatmentOnPiece4] /* Should not change anything */
treatmentOnPiece(P3, r, {{P1, P3}, {P2, P4}}) = {{P1, P3}, {P2, P4}},

[treatmentOnPiece5] /* Should add P3 with P1 or with P5 */
treatmentOnPiece(P3, r1, {{P2, P4}, {P1}, {P5}}),

/* treatmentOnCouple */

[treatmentOnCouple] /* should add P4 and P5 into the colouring */
treatmentOnCouple((P5, P4), r1, colouring),

[treatmentOnCouple1] /* Should P2 and P1 in two different colours */
treatmentOnCouple((P1, P2), r, {{}}),

[treatmentOnCouple2] /* Should add P3 with P1 */
treatmentOnCouple((P2, P3), r, {{P1},{P2}}),

[treatmentOnCouple3] /* Should add P4 into the colour of P2 */

```
        treatmentOnCouple((P3, P4), r, {{P1, P3},{P2}})
end
```

4.1.3 Tests for ColouringEx - Colouring Algorithm

The following class is used to test our colouring algorithm on:

- the relation of the assignment containing 9 pieces
- an other relation we created which contains 15 pieces

For each of these relations, we are performing two tests. The first one should return a colouring for the set. The second one should check if the colouring is correct -it should so return the boolean true.

ColouringEx

```
/∗ This class extends the class ColouringEx in order to
 ∗ perform unit test on the colouring algorithm ∗/

scheme testColouringEx =
extend ColouringEx with
        class
                value
P1 : Piece = "Piece 1",
P2 : Piece = "Piece 2",
P3 : Piece = "Piece 3",
P4 : Piece = "Piece 4",
P5 : Piece = "Piece 5",
P6 : Piece = "Piece 6",
P7 : Piece = "Piece 7",
P8 : Piece = "Piece 8",
P9 : Piece = "Piece 9",
P10 : Piece = "Piece 10",
P11 : Piece = "Piece 11",
P12 : Piece = "Piece 12",
P13 : Piece = "Piece 13",
P14 : Piece = "Piece 14",
P15 : Piece = "Piece 15",

/∗ Set presented in the assignment ∗/

r : Relation = {(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), (P7,P9), (P8,P9)},
```

```
/* Other set used to test our algorithm */

r1 : Relation = {(P1, P2), (P1, P3), (P2, P3), (P2, P4),
                 (P3, P4),(P4, P5), (P4, P6), (P4, P9),
                 (P4, P10),(P5, P6), (P7, P1), (P7, P8),
                 (P7, P14),(P8, P3), (P8, P9), (P8, P12),
                 (P8, P15),(P8,P14),(P9, P10), (P9, P12),
                 (P10, P12), (P10, P11), (P11, P6), (P11, P13), (P13, P12)}

test_case

[Assignment]
    col(r),

[isCorrectAssignment]
    isCorrectColouring(col(r), r) = true,

[Example]
    col(r1),

[isCorrectExample]
    isCorrectColouring(col(r1), r1) = true

end
```

4.2 Test Results

The results of excecuting the SML translation of the RSL test cases are:

4.2.1 Result for ColouringBasics

[isARelation] **true**
[isNotRelation] **true**
[areNeighbours] **true**
[areNotNeighbours] **true**
[isInColouring] **true**
[isNotInColouring] **true**
[isSameColour] **true**
[isNotSameColour] **true**
[notCorrectedColoured] **true**
[notCorrectedColoured2] **true**
[notCorrectedColoured3] **true**
[correctedColoured] **true**

4.2.2 Result for ColouringEx - Auxiliary functions

[canAddPieceToColour] **true**
[canAddPieceToColour2] **true**
[canAddPieceToColour3] **true**
[addPieceToColour] **true**
[addPieceToColour2] **true**
[addPieceToColour3] **true**
[canAddPieceToExistingColours] **true**
[canAddPieceToExistingColours2] **true**
[treatmentOnPiece] **true**
[treatmentOnPiece2] **true**
[treatmentOnPiece3] **true**
[treatmentOnPiece4] **true**
[treatmentOnPiece5] {{"Piece 1"},{"Piece 4","Piece 2"},{"Piece 5","Piece 3"}}
[treatmentOnCouple] {{"Piece 3","Piece 1"},{"Piece 4","Piece 2","Piece 5"}}
[treatmentOnCouple1] {{"Piece 2"},{"Piece 1"}}
[treatmentOnCouple2] {{"Piece 2"},{"Piece 1","Piece 3"}}
[treatmentOnCouple3] {{"Piece 3","Piece 1"},{"Piece 2","Piece 4"}}

4.2.3 Result for ColouringEx - Colouring Algorithm

[Assignment] {{"Piece 8","Piece 1"},{"Piece 4"},
{"Piece 3","Piece 9","Piece 2","Piece 6"},{"Piece 7","Piece 5"}}
[isCorrectAssignment] **true**

[Example] {{"Piece 11","Piece 12","Piece 2","Piece 14"},
{"Piece 4","Piece 8"},{"Piece 9","Piece 1","Piece 13","Piece 6","Piece 15"},
{"Piece 5","Piece 3","Piece 10","Piece 7"}}
[isCorrectExample] **true**

5 Conclusion

In this project, we started by specifying our requirements on relations and colourings using RSL. We used this two functions in an implicit function *Colour-Req* which could not be translated to SML. We implemented so an explicit function creating a correct colouring for a well formed relation.
To create this function, we had to create recursive methods and auxiliary ones to perform tasks. We also have been creating unit tests in order to test all of them and to detect problems while implementing these functions. The implementation of the *col* function in *ColouringEx* was beneficial for us since we were not used to functional programming and to the use of recursion in algorithm.
We made then tests to check this *col* function. From this test, we can conclude that our results are matching the requirements we had explained previously.