Informatics 2D. Course work 1: CSP Framework Reference Manual

Petros Papapanagiotou, Jacques Fleuriot

January 25, 2012

1 Introduction

This document describes the datatypes, functions and examples contained in the CSP framework for the Inf2D assignment. We will use the Map of Australia example as described in R&N §6.1.1 to demonstrate the framework's usage.

Note that you do not necessarily need to read this document from cover to cover. Use it as a helpful resource to look up information about particular parts of the available framework, such as its type definitions, functions and problems.

2 Datatypes (*CSPframework.hs*)

2.1 Var

The **Var** datatype is used to represent a named variable. It is defined as a String. For the Australia example, the list of variables is defined as follows:

aus_regions = [“WA”, “NT”, “Q”, “NSW”, “V”, “SA”]

ie. a list of Vars.

2.2 AssignedVar

The **AssignedVar** datatype is a custom datatype used to represent a variable that has been assigned a particular value. Note that only *Int* (integer) values are allowed in this framework. Therefore, in the map-colouring problem, for example, we use integers to represent the three colours (1 for red, 2 for blue, 3 for green).

You will not be required to use this datatype explicitly in the assignment.

2.3 Assignment

The **Assignment** datatype is a list of variable assignments. It represents the current state of the CSP. It may be either a partial or a full assignment.

The following functions can be used to manipulate this datatype:

- **lookup_var :: Assignment → Var → (Maybe Int)** This function looks up a variable in the assignment. It returns “Just” its assigned value or “Nothing” if it is unassigned.
- **is_unassigned :: Assignment → Var → Bool** This function returns “True” if the variable is **not** assigned within the given assignment or “False” if it is.
- **assign :: Assignment → Var → Int → Assignment** This functions returns a new assignment based on the existing one, by assigning a given value to a given variable. Note that if the variable is already assigned in the current assignment, its value is *replaced* with the new one.

2.4 Relation

The **Relation** datatype corresponds to a constraint *relation*. It defines a function with two arguments: A list of "neighbour" variables ([Var]) –the *scope*– and an Assignment.

$$type\ Relation\ =\ [Var]\ \rightarrow\ Assignment\ \rightarrow\ Bool$$

You can find some examples of Relation functions in §3.1.

2.5 Constraint

The **Constraint** datatype is a custom datatype used for constraint definitions within a CSP. It allows the definition of any type of constraint (unary, binary, or n-ary).
The following functions can be used to manipulate this datatype:

- **check_constraint :: Constraint → Assignment → Bool** Checks if the given assignment (partial or complete) is consistent with a (single) given constraint.
- **check_constraints :: [Constraint] → Assignment → Bool** Checks if the given assignment (partial or complete) is consistent with a given list of constraints.
- **scope :: Constraint → [Var]** Returns the scope of a constraint as a list of "neighbouring" variables.
- **is_constrained :: Constraint → Var → Bool** Returns "True" if a given variable is within the scope of the constraint, or "False" otherwise.
- **neighbours_of :: Constraint → Var → [Var]** Returns the list of variables (neighbours) that are in the same scope as a given variable. If the variable is not within the constraint's scope, an empty list is returned.

2.6 Domain

The **Domain** datatype represents the domain of a variable as a list of integers.

$$type\ Domain\ =\ [Int]$$

The following functions can be used to manipulate this datatype:

- **domain_add :: Int → Domain → Domain** Adds a value to a domain. Allows duplicate values.
- **domain_del :: Int → Domain → Domain** Deletes a value from a domain (once).

2.7 Domains

The **Domains** datatype is a list of domains for each variable of a CSP. It is defined as an associative list of variables and their respective domains.

$$type\ Domains\ =\ [(Var, Domain)]$$

The Domains list for the Australian Map problem is as follows:

aus_domains = [("WA",[1,2,3]), ("NT",[1,2,3]), ("Q",[1,2,3]), ("NSW",[1,2,3]), ("V",[1,2,3]), ("SA",[1,2,3])]

The values 1, 2, 3 correspond to the three available colours red, blue, and green.

2.8 CSP

The **CSP** datatype is a custom datatype used to represent a particular CSP problem. In order to construct a CSP, the following three elements are required:

1. A **String** as the name of the CSP.
2. A **Domains** list to define the variables of the CSP and their domains.
3. A list of defined **Constraint**s.

The following functions can be used to manipulate this datatype:

- **vars_of :: CSP → [Var]** Returns the list of all variables in the CSP.

- **domains :: CSP → Domains** Returns the **Domains** list (see §2.7) for the CSP.
- **constraints :: CSP → [Constraint]** Returns the list of constraints of the CSP.
- **set_domains :: CSP → Domains → CSP** Sets a new **Domains** list (see §2.7) for the CSP.
- **domain_of :: CSP → Var → Domain** Returns the domain of a given variable in the CSP or an empty list if the variable is not defined in the CSP.
- **set_domain :: CSP → Var → Domain → CSP** Sets a new domain for a given variable in the CSP.
- **add_domain_val :: CSP → Var → Int → CSP** Adds a value to the domain of the given variable in the CSP. Allows duplicate values.
- **del_domain_val :: CSP → Var → Int → CSP** Deletes a value from the domain of the given variable in the CSP (once).
- **get_unassigned_var :: CSP → Assignment → Var** Given a CSP and a (partial) assignment, it returns an unassigned variable. Throws an exception if the assignment is complete.
- **constraints_of :: CSP → Var → [Constraint]** Returns a list of constraints that have a given variable in their scope.
- **all_neighbours_of :: CSP → Var → [Var]** Returns all the variables (neighbours) of a given variable X that are found in the same scope as X of any constraint in the CSP.
- **common_constraints :: CSP → Var → Var → [Constraint]** Given a CSP and two variables X and Y, it returns the list of constraints that include both X and Y in their scope.
- **is_complete :: CSP → Assignment → Bool** Given a CSP and an assignment, returns "True" if the assignment is complete, or "False" otherwise.
- **is_consistent :: CSP → Assignment → Bool** Given a CSP and an assignment, returns "True" if the assignment is consistent, or "False" otherwise.
- **is_consistent_value :: CSP → Assignment → Var → Int → Bool** Given a CSP, an assignment (assumed to be consistent), a variable X, and a value y, this function returns "True" if y is a consistent value for X in the given assignment/state, or "False" otherwise.

3 Constraints

Some constraints have already been implemented for you in the file *Examples.hs*. They can be used as examples to demonstrate how constraints can be defined in this framework.

First, a Relation must be defined. Then, the corresponding constraint can be constructed as an instance of this Relation.

3.1 Relations

There are some implemented relations available for general use. The actual type of these relation functions is given in §2.4.

Note that if any of the variables in the scope of a Constraint is unassigned, the Relation must return True, thus allowing the algorithm to continue until all variables are assigned a value.

- **vars_diff :: Relation** A binary relation where the first two variables of the list must have different values.
- **all_diff_constraint :: Relation** A n-ary relation where all the given variables must have different values.
- **have_sum :: Integer → Relation** A n-ary relation where the sum of all the given variables is equal to the value of its parameter.
- **abs_diff_one :: Relation** A binary relation where the absolute value of the difference of the first two variables in the scope is equal to 1.
- **diff_one :: Relation** A binary relation where the difference of the first two variables in the scope is equal to 1.

3.2 Constraint constructors

Each of the above Relations is used in a constraint definition. In order to define a Constraint, three elements are required: a prettyprinting string, a list of variables as the scope, and the defined Relation function. We can create functions that will construct such Constraints from a given relation.

The following Constraint constructors have been defined for you, based on the above Relations:

- **vars_diff_constraint :: Var → Var → Constraint**
- **all_diff_constraint :: [Var] → Constraint**
- **sum_constraint :: [Var] → Int → Constraint**
- **abs_diff_one_constraint :: Var → Var → Constraint**
- **diff_one_constraint :: Var → Var → Constraint**

You can use these constructors to easily create Constraints for your CSPs. Their usage is demonstrated in the CSP examples (see §4).

For example, the Constraint *(vars_diff_constraint "WA" "NT")* creates a Constraint that enforces the "vars_diff" Relation on the two regions "WA" and "NT", and therefore enforces the constraint that they must have different values (colours).

Note that the constructors for the two Relations that you will implement for your assignment are already provided for you.

4 Example CSPs

There are four implemented examples of CSPs: The colouring of the Map of Australia, the colouring of the Map of Scotland, the 3x3 Magic Square, and the Sudoku from R&N Fig. 6.4.

4.1 Map of Australia

A detailed description of this example is given in R&N §6.1.1. The Map of Australia is divided into 6 regions. No two adjacent regions can have the same colour. Each region is represented a variable and each colour as a value: 1 for red, 2 for blue, 3 for green. The "vars_diff_constraint" is used to define the constraints for the adjacent regions. The CSP is defined as "**aus_csp**".

4.2 Map of Scotland

This is similar to the Map of Australia colouring problem but using the slightly bigger map of Scotland as depicted in Figure 1. Another difference is the fact that we consider the islands to be neighbours of the nearby land regions. The CSP is defined as "**map_csp**".

4.3 Magic Square

A Magic Square is a two-dimensional array of numbers. In our two example, the size is 3x3. The array must be filled using a different number for each location. We have achieved a Magic Square if all the rows, columns and diagonals of the array sum up to the same constant. The constant can be easily proven to be 15 for the 3x3 square. An example of a 3x3 Magic Square is given in Figure 2.

Each location of the square is represented by a different variable. "x1", "x2", "x3" are used for the first row, "y1", "y2", "y3" for the second and "z1", "z2", "z3" for the third row of the square, a total of 9 variables. There are 9 possible values for each variable so the list [1..9] is assigned as the domain of all of them.

The "all_diff" Relation is used to constrain the variables from obtaining the same value. The "sum_constraint" Relation is used to check that the sum of either a row, a column or a diagonal is correct. For each such constraint, the list of neighbours contains the list of variables involved in the row, column or diagonal. For example, the constraint for the first row of the 3x3 square is: *(sum_constraint 15,["x1", "x2", "x3"])*

Note that this problem is not normally applicable for the AC-3 algorithm, because of the ternary sum constraints.

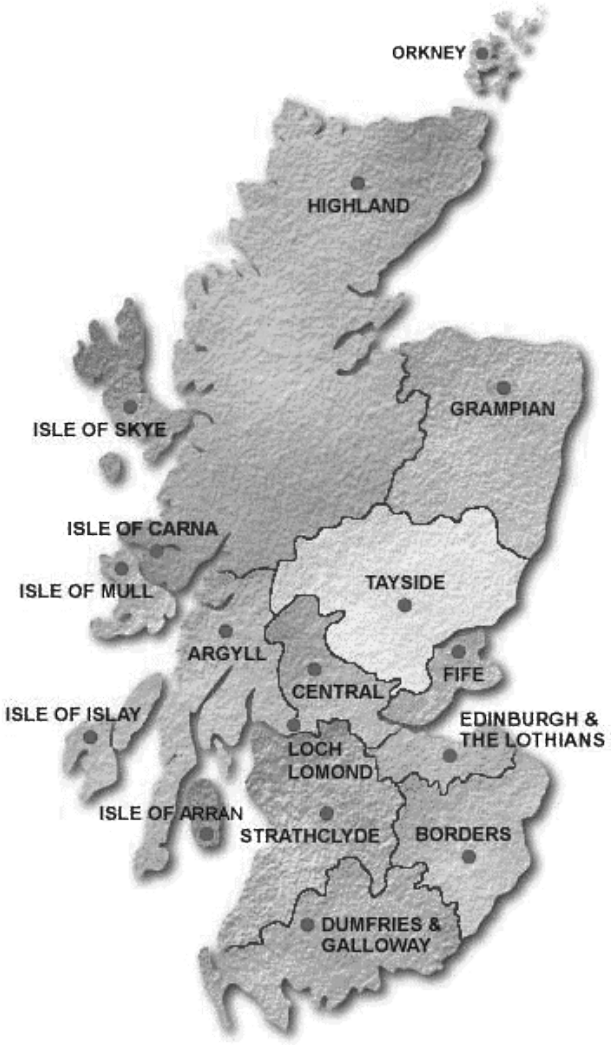

Figure 1: The map of Scotland divided in 15 regions.

2	7	6	→15
9	5	1	→15
4	3	8	→15
↓15	↓15	↓15	↓15 ↘15

Figure 2: Example of a 3x3 Magic Square.

4.4 Sudoku

The "sudoku_csp" defines a general Sudoku board, ie. a 9x9 grid of squares. The squares must be filled with digits 1-9 so that no digit must appears twice in any row, column, or 3x3 box.

The "sudoku_csp" defines the constraints on an empty grid. You can use the **sudoku** function to fill in the initial values. This function is used to define **sudoku1** which corresponds to the setup of R&N Fig.:6.4, but you can also use it to define your own sudoku boards.

Note that the implemented algorithms take too long to solve the given sudoku problems. It is strongly suggested that you only use the AC-3 algorithms, or at least FC+MRV to solve them.

5 The BT algorithm

The BT algorithm is a direct implementation of the pseudocode given in R&N Fig.: 6.5. The function **bt :: CSP → Maybe Assignment** corresponds to "BACKTRACKING-SEARCH" whereas **bt_recursion :: Assignment → CSP → Maybe Assignment** corresponds to "BACKTRACK".

The only notable difference is the pairing of the result with an Int value. The algorithm uses this value to return the number of visited search nodes. We consider a new search node is being visited every time a new value is assigned to one of the variables and recursively checked for consistency.

It is also worth noting that "bt_recursion" uses the internal recursive function "find_consistent_value" instead of the for-each loop in "BACKTRACK", since Haskell does not support such loops directly.

In order to execute the algorithm, you can use "bt" with one of the example CSPs as an argument, eg.:

bt ms3x3_csp

6 Further questions

Any further questions on this documentation and the CSP framework can be addressed to Petros Papapanagiotou by email (pe.p@ed.ac.uk) or in person at the lab.