GRACe tutorial

November 17, 2017

In this tutorial we introduce the GRACe DSL and demonstrate how to define GRACe components and write programs using those components. The GRACe DSL is divided into two Haskell modules: `GCM` and `CP`. The `GCM` module allows the user to define GRACeFUL Concept Map components and connect them to each other. The `CP` module contains primitives for constructing constraint programs, which describe the behaviour of an individual component.

1 GCM

1.1 Components and Ports

We model a `GCM` component by defining the ports it exposes. A `Port` represents a value that can be constrained. Two ports can be linked together to describe the connection between their respective components. Information contained in a component that we want to access in other parts of our model can be exposed through the component's ports. For instance, consider the following definition of a component that models a fixed amount of rain falling from the sky. It is parametrised on the amount of rain and has a port to expose that value to other components.

```
rain :: Float -> GCM (Port Float)
rain amount = do
  p <- createPort
  set p amount
  return p
```

The GRACe language is monadic which here means that we can use the `do`[1] notation to define our component in a manner which resembles a sequential program. The `createPort` command creates a new port, and the `set` command constrains the value of the port `p` to be equal to `amount`.

1.2 Goals

The command `createGoal` instantiates a `GCM` goal, which the constraint solver attempts to maximize.

Defining a helper function `maximize` helps us express goals in a clear way.

[1] `https://en.wikibooks.org/wiki/Haskell/do_notation`

```haskell
maximize :: Port Int -> GCM ()
maximize p = do
  g <- createGoal
  link p g
```

Using this function we can simply write:

```haskell
maximize p
```

to state that we would like to maximize the value at port `p`. Conversely, if our goal is to minimize a certain value we can define a similar helper function `minimize`:

```haskell
minimize :: Port Int -> GCM ()
minimize p = do
  g <- createGoal
  linkBy (fun negate) p g
```

The `link` command asserts that two values must be equal, while the `linkBy` command takes a function as a parameter to express a more complex constraint on the values.

The code for the `GCM` module is at `https://github.com/GRACeFUL-project/GRACe/blob/master/src/GCM.hs`.

2 CP

The `CP` module of GRACe supports reasoning about integer and floating-point arithmetic, Boolean expressions, and arrays. Computations in `CP` can be embedded in `GCM` using the `constrain` primitive. In this way we can embed constraints on a component's ports in the definition of the component.

Consider a `GCM` component representing a pump parametrised over the maximum flow through the pump:

```haskell
pump :: Float -> GCM (Port Float, Port Float)
pump maxCap = do
  inPort  <- createPort
  outPort <- createPort
  constrain $ do             -- This is in CP
    inflow  <- value inPort
    outflow <- value outPort
    assert $ inflow === outflow                -- no leakage
    assert $ inflow `inRange` (0, lit maxCap)  -- no back-flow, limited capacity
  return (inPort, outPort)
```

We define ports for the inflow and outflow of the pump, assert that their values must be equal, and that their values cannot exceed the pump's maximum capacity. The `value` command reads the value from a port, and the `assert` command allows us to express constraints. Note that we need to use `lit` to lift `maxCap`, which is a value in the host language Haskell, into the embedded language GRACe.

Finally we show a more complicated component, a water runoff area with an inflow, an outlet to which we may connect e.g. a pump, and an overflow. Here we can see some different constraints that are supported by the `CP` module.

```
runoffArea :: Float -> GCM (Port Float, Port Float, Port Float)
runoffArea cap = do
  inflow   <- createPort
  outlet   <- createPort
  overflow <- createPort
  constrain $ do
    currentStored <- createVariable
    inf <- value inflow
    out <- value outlet
    ovf <- value overflow
    sto <- value currentStored
    assert $ sto === inf - out - ovf
    assert $ sto `inRange` (0, lit cap)
    assert $ (ovf .> 0) ==> (sto === lit cap)
    assert $ ovf .>= 0
  return (inflow, outlet, overflow)
```

The code for the CP module is in `https://github.com/GRACeFUL-project/GRACe/blob/master/src/CP.hs`.

3 GRACe programs

In a GRACe program we define instances of available components and define their connections by linking their ports. As an example, we show a small GRACe program using the components defined earlier. We can think of it as modelling a rain runoff area, like a town square, which has been provided with a pump to alleviate possible flooding issues.

```
example :: GCM ()
example = do
  (inflowP, outflowP) <- pump 3
  (inflowS, outletS, overflowS) <- runoffArea 5
  rainflow <- rain 10

  link inflowP outletS
  link inflowS rainflow

  output overflowS "Overflow"
```

The `link` command links two ports together and asserts that their values are equal. The `output` command lets us inspect the resulting value at a `Port` after all constraints have been solved.

4 Component libraries and connection to visual editor

Although it is possible to define a GRACeFUL Concept Map directly using the GRACe DSL, most users will use the visual editor that is developed in work package 3. Using this visual editor users can draw concept maps using a palette of predefined components from a particular library. These components are also defined using the GRACe DSL. To create a library of components, we need to *annotate* them with some information, which is used by the visual editor to present the components in a proper way. For example, the `rain`, `pump`, and `runoffArea` defined above, can be annotated as follows:

```
exampleLib :: Library
exampleLib = Library "crud"
    [ Item "rain" ["description: Rain", "imgURL: ./data/img/rain.png",
                   "graphElement: nodal", "layer: domain"] $
         rain ::: "amount" # tFloat .->
         tGCM ("rotation: true" # "incomingType: none" # "outgoingType: arbitrary" #
               "rainfall" # tPort tFloat)

    , Item "pump" ["description: Pump", "imgURL: ./data/img/pump.png",
                          "graphElement: relational", "layer: domain"] $
       pump ::: "capacity" # tFloat.->
       tGCM (tPair ("rotation: true" # "incomingType: single" # "outgoingType: none" #
                    "inflow" # tPort tFloat)
                   ("rotation: true" # "incomingType: none" # "outgoingType: single" #
                    "outflow" # tPort tFloat))

    , Item "runoff area" ["description: Runoff", "imgURL: ./data/img/runOffArea.png",
                                 "graphElement: nodal", "layer: domain"] $
       runoffArea ::: "storage capacity" # tFloat .->
       tGCM (tTuple3 ("rotation: true" # "incomingType: single" # "outgoingType: none" #
                      "inflow" # tPort tFloat)
                     ("rotation: true" # "incomingType: none" # "outgoingType: single" #
                      "outlet" # tPort tFloat)
                     ("rotation: true" # "incomingType: none" # "outgoingType: single" #
                      "overflow" # tPort tFloat))
    ]
```

The example above creates a library called `exampleLib`. A library consists of an identifier, in this case `"crud"`, and a list of component items. Each `Item` has an unique identifier, a list of strings which contain meta data describing the component, and the actual component paired with a type annotation. The meta data is used by the visual editor for textual and visual descriptions, such as hover text. The items above have four meta data annotations, as explained below, but the list of strings can be extended with further meta data as is required by the front-end. The meta data strings should have the format `"name: value"` and will be added to the JSON object representing the library item to the visual editor.

The annotations for items used by the visual editor as of November 17th 2017 are:

1. `description` is a short description which appears as hovertext when the user hovers over the item.

2. `imgURL` is the location of the image used to represent the item in the visual editor.

3. `graphElement` has possible values 'nodal' and 'relational', and describes which sort of graph element better describes the item.

4. `layer` has possible values 'causal' (for elements of the CLD), 'domain' (for elements related to the domain knowledge, in our case CRUD), and 'problem' (for elements used to specify the problem the solver).

A type annotation of a component resembles a component's type signature, but has a slightly different syntax, namely `:::` instead of `::`, `.->` instead of `->`, and a `t` in front of the original type names (i.e. `tPort` to represent `Port`). Note that tuples need special handling: 2-tuples are annotated by `tPair` and 3-tuples by `tTuple3`.

Type annotations can be tagged with a string, using the `#` character, to give that entity an identifier. For example, on line 7 of above code snippet, the identifier `"capacity"` is tagged to the `tFloat` type annotation, meaning that the first parameter of the pump component is identified by `capacity`.

The ports can have a number of annotations that specify how they should be represented in the visual editor. The ports above have three annotations, which are explained below, but annotations with the format `"name: value"` can be added as required and will then be added to the JSON representation of the port.

The annotations for ports used by the visual editor as of November 17th 2017 are:

1. `rotation` can be `true` or `false`, For instance if the port is visually represented as an arrow it should be rotated to point in the right direction.

2. `incomingType` can be `none`, `single`, `multiple` or `arbitrary`, and describes how many incoming connections a port can have, where `multiple` means at least one and `arbitrary` zero or more.

3. `outgoingType` is the same as `incomingType`, but for outgoing connections.

The annotations in a library definition are used to automate the communication with the visual editor. These annotations contain all the necessary information that our web service needs to offer a library to the visual editor.

4.1 Connecting a new library to visual editor

When the GRACeServer web service starts it looks in the folder `libraries` in the GRACe directory for modules that export a function named `library` of type `Library`, interpret the library `library` and adds it to the list of available libraries.

To define a new library that can be accessed by the visual editor the library files should be located in the `libraries` folder. The header of the library file should contain the following lines:

```haskell
module Filename (library) where
import Library
```

and the library itself should have the name `library` and the type signature `library :: Library`

See examples of library files at `https://github.com/GRACeFUL-project/GRACe/tree/master/libraries`

4.2 Extending and combining libraries

Libraries can also be defined based on previously defined libraries, by combining such libraries or extending them with new items. We can use the functions `combine` and `combineList` to combine libraries and the function `insert` to extend them, as shown in the example below:

```
-- newlib1 has the id "newLib1" and
-- contains the items from libraries lib1 and lib2
newlib1 :: Library
newLib1 = combine "newLib1" lib1 lib2

-- newlib2 has the id "newLib2" and
-- contains the items from libraries lib1, lib2, lib3, and lib4
newlib2 :: Library
newLib2 = combineList "newLib2" [lib1, lib2, lib3, lib4]

-- newlib3 is an extension of newlib2.
-- it has the id "newLib2" and contains the items from newlib2
-- in addition to the items item1 and item2
newlib3 :: Library
newlib3 = insert [item1, item2] newlib2
```

5 Example: Vegetable Oil Production

The example in the previous section is rather small. We continue in this section with the explanation of a slightly larger example. The example we show is a simple optimization problem.

Let us assume we have an amount of farmland and three available crops, and would like to know how much of each crop to grow on the land to maximize our vegetable oil production. Each crop has parameters that state the yield of the crop, in tonnes, from one hectare of growing land, the amount of water required per hectare to grow the crop, and how much oil can be produced from one tonne of the crop. This example has been chosen because it is easy to imagine different stakeholders having different interests: one may strive to maximize production, another may aim to minimize water use, etc.

To model this problem in GRACe we define a component for each crop, using these parameters, with ports expressing the number of hectares, the oil yield, and the water consumption.

```
-- | GCM component for a single crop.
--
-- The component is parametrised on the crop's parameters and computes the
-- oil yield (in l) and water consumption (in Ml), given that we grow
-- so-and-so many ha of this crop.
crop :: CropParams -> GCM (Port Area, Port Water, Port Oil)
crop (y,w,o) = do
  -- Area (in ha) used to grow crop.
  areaPort  <- createPort

```

```haskell
  -- Amount of water used by crop.
  waterPort <- createPort
  -- Amount of oil produced from crop.
  oilPort   <- createPort

  -- Constrain the values at the ports.
  constrain $ do
    areaValue  <- value areaPort
    oilValue   <- value oilPort
    waterValue <- value waterPort

    -- Calculate values from data.
    assert $ oilValue   === lit y * lit o * areaValue
    assert $ waterValue === lit w * areaValue

  return (areaPort, waterPort, oilPort)
```

We also define components for the available farmland and water supply, and the oil production, which all have ports to link to each crop. We parametrised them on the number of different crops for the sake of generality.

```haskell
-- | GCM component for farmland.
--
-- The component is parametrised on the available amount of land (in ha)
-- and the number of different crops available to grow, and has ports
-- describing how the land is divided between the crops.
farm :: Area -> Int -> GCM [Port Area]
farm land numCrops = do
  -- Create a port for each crop.
  areaPorts <- mapM (\_ -> createPort) (take numCrops (repeat 0))
  constrain $ do
    areaVals <- sequence [value ap | ap <- areaPorts]
    --  The total area of crops is non-negative and is bounded by the available
    --  farmland. Each crop area is also non-negative.
    assert $ sum areaVals `inRange` (0, lit land)
    mapM_ (\x -> assert $ 0 .<= x) areaVals
  return areaPorts

-- | GCM component for water usage.
--
-- The component is parametrised on the available amount of water (in Ml)
-- and the number of different crops available to grow, and has ports
-- describing how the water is divided between the crops.
reservoir :: Water -> Int -> GCM [Port Water]
reservoir waterSource numCrops = do
  -- Create a port for each crop.
  waterPorts <- mapM (\_ -> createPort) (take numCrops (repeat 0))
  constrain $ do
    waterVals <- sequence [value wp | wp <- waterPorts]
    -- The total amount of water used is non-negative and is bounded by the
    -- available water reservoir. The amount for each crop is also non-negative.
```

```haskell
    assert $ sum waterVals `inRange` (0, lit waterSource)
    mapM_ (\x -> assert $ 0 .<= x) waterVals
  return waterPorts

-- | GCM component for oil production.
--
-- The component is parametrised on the number of different crops available
-- to grow, and has a list of ports describing how much oil is produced by
-- each crop as well as a port containing the total amount of oil produced.
oilProduction :: Int -> GCM ([Port Oil], Port Oil)
oilProduction numCrops = do
  -- Create a port for each crop.
  oilCrops <- mapM (\_ -> createPort) (take numCrops (repeat 0))
  oilOut <- createPort
  constrain $ do
    oilProduced <- value oilOut
    oilSources <- mapM value oilCrops
    -- The total amount of oil is the sum of the amounts from each crop.
    assert $ oilProduced === sum oilSources
  return (oilCrops, oilOut)
```

Our goal is to maximize the amount of oil produced, and we use the helper function `maximize`, as described in 1.2, to help us express this.

The full code for the vegetable oil example can be seen in Appendix A.

A Vegetable Oil Production - Full code

```haskell
module Main where

import Compile0 (runGCM)
import GCM      ( GCM, output, component, createGoal
                , Port, createPort, link, value
                )
import CP       ( assert, lit, (===), (.<=), inRange )


-- * Vegetable oil manufacturing
--------------------------------------------------------------------------------
-- We define a system to describe growing and producing vegetable oil from
-- different types of crops.

-- | We use type synonyms to keep track of the different resources
--   we are working with.
--
-- Farmland area is measured in ha
type Area  = Int
-- Crop yield is measure in t/ha
type Yield = Int
-- Water is measured in Ml
type Water = Int
```

```haskell
-- Oil is measured in l
type Oil   = Int

-- Each crop has parameters describing its yield in t/ha,
-- its water demand in Ml/ha, and its oil content in l/t.
type CropParams = (Yield, Water, Oil)

-- | GCM component for a single crop.
--
-- The component is parametrized on the crop's parameters and computes the
-- oil yield (in l) and water consumption (in Ml), given that we grow
-- so-and-so many ha of this crop.
crop :: CropParams -> GCM (Port Area, Port Water, Port Oil)
crop (y,w,o) = do
  -- Area (in ha) used to grow crop.
  areaPort  <- createPort

  -- Amount of water used by crop.
  waterPort <- createPort
  -- Amount of oil produced from crop.
  oilPort   <- createPort

  -- Constrain the values at the ports.
  component $ do
    areaValue  <- value areaPort
    oilValue   <- value oilPort
    waterValue <- value waterPort

    -- Calculate values from data.
    assert $ oilValue   === lit y * lit o * areaValue
    assert $ waterValue === lit w * areaValue

  return (areaPort, waterPort, oilPort)

-- | GCM component for farmland.
--
-- The component is parametrized on the available amount of land (in ha)
-- and the number of different crops available to grow, and has ports
-- describing how the land is divided between the crops.
farm :: Area -> Int -> GCM [Port Area]
farm land numCrops = do
  -- Create a port for each crop.
  areaPorts <- mapM (\_ -> createPort) (take numCrops (repeat 0))
  component $ do
    areaVals <- sequence [value ap | ap <- areaPorts]
    --  The total area of crops is non-negative and is bounded by the available
    --  farmland. Each crop area is also non-negative.
    assert $ sum areaVals `inRange` (0, lit land)
    mapM_ (\x -> assert $ 0 .<= x) areaVals
  return areaPorts
```

```haskell

-- | GCM component for water usage.
--
-- The component is parametrized on the available amount of water (in Ml)
-- and the number of different crops available to grow, and has ports
-- describing how the water is divided between the crops.
reservoir :: Water -> Int -> GCM [Port Water]
reservoir waterSource numCrops = do
  -- Create a port for each crop.
  waterPorts <- mapM (\_ -> createPort) (take numCrops (repeat 0))
  component $ do
    waterVals <- sequence [value wp | wp <- waterPorts]
    -- The total amount of water used is non-negative and is bounded by the
    -- available water reservoir. The amount for each crop is also non-negative.
    assert $ sum waterVals `inRange` (0, lit waterSource)
    mapM_ (\x -> assert $ 0 .<= x) waterVals
  return waterPorts

-- | GCM component for oil production.
--
-- The component is parametrized on the number of different crops available
-- to grow, and has a list of ports describing how much oil is produced by
-- each crop as well as a port containing the total amount of oil produced.
oilProduction :: Int -> GCM ([Port Oil], Port Oil)
oilProduction numCrops = do
  -- Create a port for each crop.
  oilCrops <- mapM (\_ -> createPort) (take numCrops (repeat 0))
  oilOut <- createPort
  component $ do
    oilProduced <- value oilOut
    oilSources <- mapM value oilCrops
    -- The total amount of oil is the sum of the amounts from each crop.
    assert $ oilProduced === sum oilSources
  return (oilCrops, oilOut)

-- | In our example problem we have 3 crops: Soybeans, sunflower seeds and
--   cotton seeds, parametrized by the following table:
--
--   Crop             Yield [t/ha]   Water demand [Ml/ha]   Oil content [l/t]
--   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--   Soybeans         3              5                      178
--   Sunflower seeds  2              4                      216
--   Cotton seeds     1              1                      433
--
-- Note the units.

-- | We define a type for our crops and a function to keep track of
--   parameters
data Crop = Soy | Sunflower | Cotton
cropTable :: Crop -> (Yield, Water, Oil)
```

```haskell
cropTable Soy       = (3, 5, 178)
cropTable Sunflower = (2, 4, 216)
cropTable Cotton    = (1, 1, 433)

-- | Help function for maximizing goal.
maximize :: Port Int -> GCM ()
maximize p = do
  g <- createGoal
  link p g

-- | GCM program for optimizing the area of land on which to grow each of the 3
--   available crops in order to maximize oil production.
--
--   In this example the following quantities are available:
--   Farmland: 1,600 ha
--   Water:    5,000 Ml
--
problem :: GCM ()
problem = do
  -- Create system
  (soyArea, soyWater, soyOil) <- crop $ cropTable Soy
  (sunArea, sunWater, sunOil) <- crop $ cropTable Sunflower
  (cotArea, cotWater, cotOil) <- crop $ cropTable Cotton

  [soy_a, sun_a, cot_a] <- farm 1600 3
  [soy_w, sun_w, cot_w] <- reservoir 5000 3
  ([soy_o, sun_o, cot_o], oilProduced) <- oilProduction 3

  -- Link the appropriate ports together.
  link soyArea soy_a
  link soyWater soy_w
  link soyOil soy_o

  link sunArea sun_a
  link sunWater sun_w
  link sunOil sun_o

  link cotArea cot_a
  link cotWater cot_w
  link cotOil cot_o

  -- Our goal is to maximize the amount of oil produced.
  maximize oilProduced

  -- We print the values we would like to see.
  output oilProduced "Oil produced"
  output soyArea "Soybean area"
  output sunArea "Sunflower area"
  output cotArea "Cotton area"
```

```
main :: IO ()
main = print =<< runGCM problem
```