

GRACe tutorial

June 30, 2017

In this tutorial we introduce the GRACe DSL and demonstrate how to define GRACe components and write programs using those components.

GRACe is divided into two modules, GCM and CP. The GCM module allows the user to define GRACeFUL Concept Map components and describe how they are connected. The CP module contains primitives for constructing constraint programs which describe the behavior of an individual component.

1 GCM components and ports

We model a GCM component by defining the ports it exposes.

A `Port` contains 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 program can be exposed through the component's ports.

For instance, consider the following definition of a component which models a fixed amount of rain falling from the sky. It is parametrized on the amount of rain and has a port to expose that value to another component.

```
1 rain :: Float -> GCM (Port Float)
2 rain amount = do
3   p <- createPort
4   set p amount
5   return p
```

Since the GRACe language is monadic we can use `do` notation and define our component in a sequential manner.

The `createPort` command creates a new port, and the `set` command constrains the value of the port `p` to be equal to `amount`.

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 `component` 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:

```

1 pump :: Float -> GCM (Port Float, Port Float)
2 pump maxCap = do
3   inPort  <- createPort
4   outPort <- createPort
5   component $ do -- This is in CP
6     inflow  <- value inPort
7     outflow <- value outPort
8     assert $ inflow == outflow
9     assert $ inflow 'inRange' (0, lit maxCap)
10  return (inPort, outPort)

```

We define ports for the inflow and outflow, 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 various different kinds of constraints that are supported by the CP module.

```

1 runoffArea :: Float -> GCM (Port Float, Port Float, Port Float)
2 runoffArea cap = do
3   inflow  <- createPort
4   outlet  <- createPort
5   overflow <- createPort
6   component $ do
7     currentStored <- createVariable
8     inf <- value inflow
9     out <- value outlet
10    ovf <- value overflow
11    sto <- value currentStored
12    assert $ sto == inf - out - ovf
13    assert $ sto 'inRange' (0, lit cap)
14    assert $ (ovf .> 0) ==> (sto == lit cap)
15    assert $ ovf .>= 0
16  return (inflow, outlet, overflow)

```

3 GRACe programs

In a GRACe program we define instances of available components and define their connections by linking their ports.

As an example, here is a small GRACe program using the components defined earlier. We can think of it as modeling a rain runoff area, like a town square, which has been provided with a pump to alleviate possible flooding issues.

```

1 example :: GCM ()
2 example = do

```

```

3   (inflowP, outflowP) <- pump 3
4   (inflowS, outletS, overflowS) <- runoffArea 5
5   rainflow <- rain 10
6
7   link inflowP outletS
8   link inflowS rainflow
9
10  output overflowS "Overflow"

```

The `link` command links two ports together and asserts that their values must be equal.

The `output` command lets us inspect the resulting value at a `Port` after all constraints have been solved.

4 Example: Vegetable Oil Production

As an example we consider a simple optimization problem.

We have a set amount of farmland and three available crops, and would like to know 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 tons, 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 ton of the crop.

We define a component for each crop, using these parameters, with ports expressing the number of hectares, the oil yield, and the water consumption.

```

1  -- | GCM component for a single crop.
2  --
3  -- The component is parametrized on the crop's parameters and computes the
4  -- oil yield (in l) and water consumption (in Ml), given that we grow
5  -- so-and-so many ha of this crop.
6  crop :: CropParams -> GCM (Port Area, Port Water, Port Oil)
7  crop (y,w,o) = do
8    -- Area (in ha) used to grow crop.
9    areaPort <- createPort
10
11    -- Amount of water used by crop.
12    waterPort <- createPort
13    -- Amount of oil produced from crop.
14    oilPort <- createPort
15
16    -- Constrain the values at the ports.
17    component $ do
18      areaValue <- value areaPort
19      oilValue <- value oilPort
20      waterValue <- value waterPort
21
22    -- Calculate values from data.
23    assert $ oilValue == lit y * lit o * areaValue
24    assert $ waterValue == lit w * areaValue

```

```

25
26     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 parametrize them on the number of different crops for the sake of generality.

```

1  -- / GCM component for farmland.
2  --
3  -- The component is parametrized on the available amount of land (in ha)
4  -- and the number of different crops available to grow, and has ports
5  -- describing how the land is divided between the crops.
6  farm :: Area -> Int -> GCM [Port Area]
7  farm land numCrops = do
8      -- Create a port for each crop.
9      areaPorts <- mapM (\_ -> createPort) (take numCrops (repeat 0))
10     component $ do
11         areaVals <- sequence [value ap | ap <- areaPorts]
12         -- The total area of crops is non-negative and is bounded by the available
13         -- farmland. Each crop area is also non-negative.
14         assert $ sum areaVals `inRange` (0, lit land)
15         mapM_ (\x -> assert $ 0 <= x) areaVals
16     return areaPorts
17
18 -- / GCM component for water usage.
19 --
20 -- The component is parametrized on the available amount of water (in Ml)
21 -- and the number of different crops available to grow, and has ports
22 -- describing how the water is divided between the crops.
23 reservoir :: Water -> Int -> GCM [Port Water]
24 reservoir waterSource numCrops = do
25     -- Create a port for each crop.
26     waterPorts <- mapM (\_ -> createPort) (take numCrops (repeat 0))
27     component $ do
28         waterVals <- sequence [value wp | wp <- waterPorts]
29         -- The total amount of water used is non-negative and is bounded by the
30         -- available water reservoir. The amount for each crop is also non-negative.
31         assert $ sum waterVals `inRange` (0, lit waterSource)
32         mapM_ (\x -> assert $ 0 <= x) waterVals
33     return waterPorts
34
35 -- / GCM component for oil production.
36 --
37 -- The component is parametrized on the number of different crops available
38 -- to grow, and has a list of ports describing how much oil is produced by
39 -- each crop as well as a port containing the total amount of oil produced.
40 oilProduction :: Int -> GCM ([Port Oil], Port Oil)
41 oilProduction numCrops = do
42     -- Create a port for each crop.
43     oilCrops <- mapM (\_ -> createPort) (take numCrops (repeat 0))

```

```

44 oilOut <- createPort
45 component $ do
46   oilProduced <- value oilOut
47   oilSources <- mapM value oilCrops
48   -- The total amount of oil is the sum of the amounts from each crop.
49   assert $ oilProduced == sum oilSources
50   return (oilCrops, oilOut)

```

Our goal is to maximize the amount of oil produced, and we define the helper function `maximize` to help us express this:

```

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

```

The command `createGoal` instantiates a GCM goal, which the constraint solver will attempt to maximize. Using this helper function we can simply write

```
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`

```

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

```

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

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

A Vegetable Oil Production - Full code

```

1 module Main where
2
3 import Compile0 (runGCM)
4 import GCM      ( GCM, output, component, createGoal
5                  , Port, createPort, link, value
6                  )
7 import CP       ( assert, lit, (==), (<=), inRange )
8
9
10 -- * Vegetable oil manufacturing
11 -----
12 -- We define a system to describe growing and producing vegetable oil from
13 -- different types of crops.
14
15 -- / We use type synonyms to keep track of the different resources

```

```

16  -- we are working with.
17  --
18  -- Farmland area is measured in ha
19  type Area = Int
20  -- Crop yield is measure in t/ha
21  type Yield = Int
22  -- Water is measured in Ml
23  type Water = Int
24  -- Oil is measured in l
25  type Oil = Int
26
27  -- Each crop has parameters describing its yield in t/ha,
28  -- its water demand in Ml/ha, and its oil content in l/t.
29  type CropParams = (Yield, Water, Oil)
30
31  -- | GCM component for a single crop.
32  --
33  -- The component is parametrized on the crop's parameters and computes the
34  -- oil yield (in l) and water consumption (in Ml), given that we grow
35  -- so-and-so many ha of this crop.
36  crop :: CropParams -> GCM (Port Area, Port Water, Port Oil)
37  crop (y,w,o) = do
38    -- Area (in ha) used to grow crop.
39    areaPort <- createPort
40
41    -- Amount of water used by crop.
42    waterPort <- createPort
43    -- Amount of oil produced from crop.
44    oilPort <- createPort
45
46    -- Constrain the values at the ports.
47    component $ do
48      areaValue <- value areaPort
49      oilValue <- value oilPort
50      waterValue <- value waterPort
51
52    -- Calculate values from data.
53    assert $ oilValue == lit y * lit o * areaValue
54    assert $ waterValue == lit w * areaValue
55
56    return (areaPort, waterPort, oilPort)
57
58  -- | GCM component for farmland.
59  --
60  -- The component is parametrized on the available amount of land (in ha)
61  -- and the number of different crops available to grow, and has ports
62  -- describing how the land is divided between the crops.
63  farm :: Area -> Int -> GCM [Port Area]
64  farm land numCrops = do
65    -- Create a port for each crop.

```

```

66   areaPorts <- mapM (\_ -> createPort) (take numCrops (repeat 0))
67   component $ do
68     areaVals <- sequence [value ap | ap <- areaPorts]
69     -- The total area of crops is non-negative and is bounded by the available
70     -- farmland. Each crop area is also non-negative.
71     assert $ sum areaVals 'inRange' (0, lit land)
72     mapM_ (\x -> assert $ 0 <= x) areaVals
73   return areaPorts
74
75   -- / GCM component for water usage.
76   --
77   -- The component is parametrized on the available amount of water (in Ml)
78   -- and the number of different crops available to grow, and has ports
79   -- describing how the water is divided between the crops.
80   reservoir :: Water -> Int -> GCM [Port Water]
81   reservoir waterSource numCrops = do
82     -- Create a port for each crop.
83     waterPorts <- mapM (\_ -> createPort) (take numCrops (repeat 0))
84     component $ do
85       waterVals <- sequence [value wp | wp <- waterPorts]
86       -- The total amount of water used is non-negative and is bounded by the
87       -- available water reservoir. The amount for each crop is also non-negative.
88       assert $ sum waterVals 'inRange' (0, lit waterSource)
89       mapM_ (\x -> assert $ 0 <= x) waterVals
90     return waterPorts
91
92   -- / GCM component for oil production.
93   --
94   -- The component is parametrized on the number of different crops available
95   -- to grow, and has a list of ports describing how much oil is produced by
96   -- each crop as well as a port containing the total amount of oil produced.
97   oilProduction :: Int -> GCM ([Port Oil], Port Oil)
98   oilProduction numCrops = do
99     -- Create a port for each crop.
100    oilCrops <- mapM (\_ -> createPort) (take numCrops (repeat 0))
101    oilOut <- createPort
102    component $ do
103      oilProduced <- value oilOut
104      oilSources <- mapM value oilCrops
105      -- The total amount of oil is the sum of the amounts from each crop.
106      assert $ oilProduced == sum oilSources
107    return (oilCrops, oilOut)
108
109   -- / In our example problem we have 3 crops: Soybeans, sunflower seeds and
110   -- cotton seeds, parametrized by the following table:
111   --
112   -- Crop                Yield [t/ha]   Water demand [Ml/ha]   Oil content [l/t]
113   -- ~~~~~
114   -- Soybeans            3                5                178
115   -- Sunflower seeds    2                4                216

```

```

116 -- Cotton seeds      1          1          433
117 --
118 -- Note the units.
119
120 -- / We define a type for our crops and a function to keep track of
121 -- parameters
122 data Crop = Soy | Sunflower | Cotton
123 cropTable :: Crop -> (Yield, Water, Oil)
124 cropTable Soy      = (3, 5, 178)
125 cropTable Sunflower = (2, 4, 216)
126 cropTable Cotton   = (1, 1, 433)
127
128 -- / Help function for maximizing goal.
129 maximize :: Port Int -> GCM ()
130 maximize p = do
131   g <- createGoal
132   link p g
133
134 -- / GCM program for optimizing the area of land on which to grow each of the 3
135 -- available crops in order to maximize oil production.
136 --
137 -- In this example the following quantities are available:
138 -- Farmland: 1,600 ha
139 -- Water:    5,000 Ml
140 --
141 problem :: GCM ()
142 problem = do
143   -- Create system
144   (soyArea, soyWater, soyOil) <- crop $ cropTable Soy
145   (sunArea, sunWater, sunOil) <- crop $ cropTable Sunflower
146   (cotArea, cotWater, cotOil) <- crop $ cropTable Cotton
147
148   [soy_a, sun_a, cot_a] <- farm 1600 3
149   [soy_w, sun_w, cot_w] <- reservoir 5000 3
150   ([soy_o, sun_o, cot_o], oilProduced) <- oilProduction 3
151
152   -- Link the appropriate ports together.
153   link soyArea soy_a
154   link soyWater soy_w
155   link soyOil soy_o
156
157   link sunArea sun_a
158   link sunWater sun_w
159   link sunOil sun_o
160
161   link cotArea cot_a
162   link cotWater cot_w
163   link cotOil cot_o
164
165   -- Our goal is to maximize the amount of oil produced.

```



```

166     maximize oilProduced
167
168     -- We print the values we would like to see.
169     output oilProduced "Oil produced"
170     output soyArea "Soybean area"
171     output sunArea "Sunflower area"
172     output cotArea "Cotton area"
173
174     main :: IO ()
175     main = print =<< runGCM problem

```