PROGRAMMING LANGUAGES AND COMPUTATION

Problem Sheet 1: Transition Systems

* 1. Setup a clean Haskell environment on your own machine according to the instructions at https://github.com/uob-coms20007/labcode. Verify your installation by checking the following at the command prompt:

```
$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 9.0.1

$ cabal --version
cabal-install version 3.4.0.0
```

Clone the `labcode` repository into your workspace using:

```
$ git clone https://github.com/uob-coms20007/labcode.git
Cloning into 'labcode'...
```

For this, you will need to use git, so if you don't have it already, please install it. Verify that you can build the project using:

```
$ cd labcode/
$ cabal build all
Resolving dependencies...
```

This will produce a lot of output on the terminal because it must first build any dependencies of the project that are not already built.

The rest of this week's problems concern the file reg/src/TrSys.hs. The file defines a module `TrSys`, which concerns implementations of transition systems and some associated functions for working with them.

* 2. In the same terminal, start up a repl session in GHCI and immediately bring into scope all the entities exported by the `TrSys` module.

```
$ cd reg
$ cabal repl
...
GHCi, version 9.0.1: https://www.haskell.org/ghc/ :? for help
[1 of 5] Compiling NonDet       ( src/NonDet.hs, interpreted )
[2 of 5] Compiling TrSys        ( src/TrSys.hs, interpreted )
[3 of 5] Compiling Automata     ( src/Automata.hs, interpreted )
[4 of 5] Compiling RegExp       ( src/RegExp.hs, interpreted )
[5 of 5] Compiling Vis          ( src/Vis.hs, interpreted )
Ok, five modules loaded.
λ import TrSys
```

It is something of a tradition among Haskell programmers to set the GHCI prompt to be a coloured λ. In terminals where it is supported, this can be achieved by a variation on the ghci command `:set prompt "\x1b[38;5;228m\0x3bb\x1b[0m "`*, which can be also placed into your ghci.conf for more permanency.*

Libraries and their import

The first part of `TrSys` defines the imports we (will) need to make the functions work:

```
import qualified Data.List as List

import Data.Set (Set)
import qualified Data.Set as Set

import System.Random (StdGen)
import qualified System.Random as Random
```

The syntax `import qualified Data.List as List` imports all the entities exported by the `Data.List` (library) module but requires that you prefix their names by `List` when you use them. For example, `intercalate` is a function exported by the `Data.List` module, so if we were to use that in our module here, we would need to refer to it as `List.intercalate`. Specifically, the "qualified" keyword is requiring the prefix, i.e. that the names are qualified by the module they come from, and the `as List` part is giving a local name `List` for that module (otherwise we would need to write e.g. `Data.List.intercalate`).

The syntax `import Data.Set (Set)` imports the single entity `Set` from the `Data.Set` module, which happens to be the *type constructor* for finite sets – i.e. `Set a` is the type of finite sets whose elements have type `a`. There is no "qualified" keyword here, so we can use the type constructor `Set` without prefix in the definitions that follow. Note: the compiler does not mind that we subsequently `import qualified Data.Set as Set`, so that `Set` now refers both to the type constructor and to the module.

You can find documentation for the entities that are imported by looking up the packages that contain them on Hackage. `Data.List` comes from the base package, `Data.Set` comes from

the containers package and `System.Random` from the random package, but googling the module names is usually enough to get you to the correct documentation.

The type of transition systems

Perhaps the single most important definition in the module is:

```
type TrSys a = a → Set a
```

This defines the type of *transition systems over configurations of type* `a`, written `TrSys a`, as a synonym of `a → Set a`. This means that writing the type `TrSys a` is essentially the same as writing the type `a → Set a` as far as the compiler is concerned.

This may seem like a strange choice, given the definition of transition systems that we have used in lectures. In lectures we said that a transition system consists of a set of configurations and a transition relation describing which configurations transition to which other ones. The idea is that the type `a` tells us what all the possible configurations are and the function of type `a → Set a` tells us the transition relation. More specifically, we use a function `f` of type `a → Set a` to describe the successors of a given configuration: a pair of configurations `c` and `d` of type `a` are in the transition relation just if `d` is an element of the set `f c`.

The traffic light example

Take a look at the traffic light example. A traffic light configuration is described by the type `Light`, defined as:

```
data Light =
  Red | Amber | Green | RedAndAmber
deriving (Eq, Ord, Show)
```

You may remember that the syntax `deriving (Eq, Ord, Show)` asks the compiler to automatically generate instances of the named typeclasses for this datatype, which ensures that traffic lights can be tested for equality, ordered and written as a `String` respectively.

The traffic light transition system is given by:

```
traffic :: TrSys Light
```

which says exactly that `traffic` is a transition system over configurations of type `Light`. Remember that `TrSys Light` is just a synonym for `Light → Set Light` so, to define our transition system, we will need to give a function of this type.

In the traffic light example, each configuration only has a single successor, and that successor is given by the UK traffic light sequence, e.g red is followed by red and amber in combination. So, to keep the code neat, we factor out this sequence as a function `tfic :: Light → Light` and then apply the singleton set constructor to the result.

The While program example

The main reason that we use a Haskell function to represent a transition system is that Haskell functions are an example of a finite representation of an infinite structure. For example, a function like `f(x) = x+1` is a finite description – just 10 characters in your text editor – but it unambiguously represents an infinite set of mappings $\{0 \mapsto 1, 1 \mapsto 2, -3 \mapsto -2, \ldots\}$.

We can't even fully imagine this set due to its fathomless size, but we can nevertheless *use* the finite representation – program with it – in order to compute with the infinite set just as if we really had it in our hand. For example, if I want to know if there is a mapping $n \mapsto m$ in this infinite set (for some particular numbers n and m then I can write a program expression `f(n) == m` and evaluate it.

Ultimately, computations can only manipulate finite representations – there is only so much memory – but many interesting transition systems are infinite: there are infinitely many configurations and infinitely many transitions. We will see several examples of finite representations of infinite structures in this unit, but functions are perhaps the most useful.

Consider the While program example. Here we have a transition system with infinitely many configurations – all possible triples of program counter, integer value of x and integer value of y – and infinitely many transition between them (think of one of the infinite traces). However, we can represent these infinitely many transitions by the function:

```
prog :: TrSys (Int, Int, Int)
```

which gives, for each possible configuration `(pc,x,y)`, its finite set of successors `prog (pc, x, y)`. For example, the fact that `prog (3,4,6)` evaluates to the singleton set containing `(4,2,6)` encodes the transition made when executing the code on line 3: the value of x is decreased by 2, y stays the same and the program counter moves to line 4.

Our modelling of the While program example as a function of type `TrSys (Int, Int, Int)` is not perfect because, in reality, there are only 4 possible values for the program counter – it can't really be any integer. This is a compromise we are willing to make in the interests of ease of modelling. The advantage is that we can just use an existing type, `Int`, as the type for the program counter component of the configuration, but a disadvantage is that we require the last clause of the definition, stating that in all other cases the set of successors is empty.

* 3. You are provided with a complete definition of the function `theTrace` which can be used to obtain a trace of execution. Look at its type and read the associated comments.

(a) What are the first 15 configurations of the unique trace of the While program example when started with $x = 2$ and $y = 3$?

The function `theTrace` works correctly only for deterministic transition systems. When a transition system is nondeterministic, we would still like to get an idea of the traces, but obtaining all possible traces can be a bit overwhelming.

(b) Using the definition of `theTrace` as a guide, implement the function `rngTrace`, which carries out a random walk through the transition system to generate a trace.

Hint you will want to familiarise yourself with the behaviour of the function Data.List.unfoldr and the module System.Random.

Solution

(a)

```
λ take 15 (theTrace prog (1,2,3))
  [
    (1,2,3),
    (2,5,3),
    (3,5,3),
    (4,3,3),
    (2,3,5),
    (3,3,5),
    (4,1,5),
    (2,1,9),
    (3,1,9),
    (4,-1,9),
    (2,-1,17),
    (3,-1,17),
    (4,-3,17),
    (2,-3,33),
    (3,-3,33)
  ]
```

(b) See labcode.

** 4. Implement the chameleons example as a transition system `chameleons` and the circuit example as a transition system `circuit`. In a separate terminal, you can test your code using:

```
$ cabal test --test-show-details=direct --test-option="--pattern=<P>"
Running 1 test suites...
Test suite test: RUNNING...
...
```

You will need to replace the `<P>` by the code of the test group you want to exercise. In this case, you want to run the group "T2: Random Trace Tests" so you should supply code `T2`:

```
$ cabal test --test-show-details=direct --test-option="--pattern=T2"
```

Note: the first time this is run it will first have to download and build the dependencies of the test suite, which may take a few minutes. If all of the Random Trace Tests pass, then you are in good shape.

Reachability computation

The final part of this problem sheet concerns computing a finite set of reachable configurations. Given a set of initial configurations I, the *fixed-point algorithm* computes the set of all configurations reachable from I whenever that set is finite. The algorithm is shown to the right of this text.

The idea is that the set C will be enlarged on every iteration until it eventually contains all the configurations reachable from I. We initialise C with exactly the configurations in I and, on each iteration, we add in to C all configurations that can be reached from one of the configurations in C in exactly one step.

```
REACH(I)
1  C ← I
2  D ← ∅
3  while C ≠ D
4        do D ← C
5           C ← C ∪ {d | ∃c ∈ C. c ⇒ d}
6  return C
```

** 5.

(a) Implement the fixed point algorithm as the function `reachable`. Test your code as above: if all of *T3: Reachability Tests* pass then well done, you are in great shape!

(b) Give a *loop invariant* involving C and the number of iterations and explain why it justifies the correctness of the algorithm. You will need to appeal to your algorithms training for this.

Solution

(a) On iteration n, C is exactly the set of configurations reachable from I in n transitions. If the set of reachable configs is finite, then there is a maximum number of transitions required to

reach any reachable config, say k. So after k iterations of the loop, C will already contain all reachable configurations and the statement $C \leftarrow C \cup \{d \mid \exists c.\ c \Rightarrow d\}$ will not change C. Hence, the following test $C \neq D$ will fail and the loop will terminate.

(b) See labcode.

*** 6. Prove that $P := \{(r, b, g) \mid r + b + g = 9\}$ is an inductive invariant for the chameleons transition system with initial configuration $(2, 3, 4)$. Your proof should follow the same structure as that given in lectures.

Solution

Base. We have to show that $\{(2, 3, 4)\} \subseteq P$, i.e. $(2, 3, 4) \in P$. By definition of P, we just need to check $2 + 3 + 4 = 9$ which is straightforward.

Step. We have to show that $(r, b, g) \in P$ and $(r, b, g) \Rightarrow (r', b', g')$ together imply $(r', b', g') \in P$. So, assume the statements on the left hand side of the implication:

(i) $(r, b, g) \in P$

(ii) $(r, b, g) \Rightarrow (r', b', g')$

and then we try to prove the statement on the right: $(r', b', g') \in P$. First, let's observe that by (i) and the definition of P, it must be that:

(iii) $r + b + g = 9$

Now, to show $(r', b', g') \in P$, by the definition of P, we need to show that $r' + b' + g' = 9$. It follows from the definition of the transition relation $\Rightarrow$, that the transition we have in (ii) could only have occurred if one of the following is true:

(1) $r' = r + 2$, $b' = b - 1$ and $g' = g - 1$

(2) $r' = r - 1$, $b' = b - 1$ and $g' = g + 2$

(3) $r' = r - 1$, $b' = b + 2$ and $g' = g - 1$

We don't know which one is responsible, so we need to show that $r' + b' + g' = 9$ in all cases:

- In case (1) is true, we have that:

  (iv) $r' = r + 2$, $b' = b - 1$ and $g' = g - 1$

  That means that $r' + b' + g' = (r + 2) + (b - 1) + (g - 1)$ and this is just $r + b + g$ which we know from (iii) is 9.

- In case (1) is true, we have that:

  (iv) $r' = r - 1$, $b' = b - 1$ and $g' = g + 2$

  That means that $r' + b' + g' = (r - 1) + (b - 1) + (g + 2)$ and this is just $r + b + g$ which we know from (iii) is 9.

- In case (1) is true, we have that:

(iv) $r' = r - 1$, $b' = b + 2$ and $g' = g - 1$

That means that $r' + b' + g' = (r - 1) + (b + 2) + (g - 1)$ and this is just $r + b + g$ which we know from (iii) is 9.

Since we have $r' + b' + g' = 9$ in all cases, we are done.

Optional extension problems

The fixed point algorithm given above is not very efficient because configurations that are added to C in some iteration $n+1$ will be pointlessly added again in every iteration $n+1$, $n+2$, $n+3$ thereafter. To understand this, observe that the algorithm still works when line 5 is replaced by $C \leftarrow I \cup \{d \mid \exists c \in C.\, c \Rightarrow d\}$. A more efficient approach is to maintain a *frontier*, which is the set of configurations that were added only in the previous iteration. With such a frontier, say F, we could replace line 5 by $C \leftarrow C \cup \{d \mid \exists c \in F.\, c \Rightarrow d\}$ and the expensive equality test on sets in line 3 with $F = \emptyset$.

** 7. (Optional) Following the above idea, implement a more efficient version of the fixed-point algorithm as function `freachable`.

**** 8. (Optional) Prove that the following set Q is an invariant for the While program transition system with initial configurations $I = \{(1, x, y) \mid x \in \mathbb{Z}, y \in \mathbb{Z}\}$:

$$Q := \{(pc, x, y) \mid pc = 5 \text{ implies } (x = 0 \wedge y \text{ odd})\}$$

To do this, describe a smaller set P, i.e. such that $P \subseteq Q$ and show that P is an *inductive* invariant.

Solution

If you attempt to prove that Q is an inductive invariant, you will soon discover that you need to know that, when the program reaches line 5, it had better be that y is already odd.

So we set P as the following smaller set:

$$P := \left\{ (pc, x, y) \;\middle|\; \begin{array}{ll} & (pc = 5 \text{ implies } x = 0 \wedge y \text{ odd}) \\ \wedge & (pc = 2 \wedge x = 0 \text{ implies } y \text{ odd}) \end{array} \right\}$$

and we prove the two cases for being an inductive invariant. To help keep track of the facts we can use in the proof, we will label every fact that we either assume or otherwise deduce to be true by a roman numeral.

Base. We prove that $I \subseteq P$. So, assume that:

(i) $(pc, x, y) \in I$

and we aim to show that $(pc, x, y) \in P$. By (i) and the definition of I, this means that $pc = 1$, and x and y are some integers. Then it follows immediately that (pc, x, y) because $pc = 1 \neq 5$ and $pc = 1 \neq 2$, so the two implications in the definition of P are vacuously satisfied – P contains *all* configurations whose pc is different from 5 and 2.

Step. We prove that $(pc, x, y) \in P$ and $(pc, x, y) \Rightarrow^* (pc', x', y')$ implies $(pc', x', y') \in P$. So we assume both of the statements on the left hand side of the implication:

(i) $(pc, x, y) \in P$

(ii) $(pc, x, y) \Rightarrow (pc', x', y')$

and attempt to show that the statement on the right hand side, $(pc', x', y') \in P$, necessarily follows. By definition of P, showing that $(pc', x', y') \in P$ means showing two things:

(a) $pc' = 5$ implies $x = 0$ and y odd

(b) $pc' = 2$ and $x = 0$ together imply that y is odd

So we prove each of them separately.

- To prove (a), we assume the statement:

  (iii) $pc' = 5$

  that is on the left-hand side of the implication and attempt to prove the statements $x = 0$ and y odd that are on the right. First, lets observe that, by combining (ii) and (iii) we get:

  (iv) $(pc, x, y) \Rightarrow (5, x', y')$

  and, by the definition of the transition relation, the only way that such a transition can come about is if:

  (v) $pc = 2$, $x = 0$, $x' = x$ and $y' = y$

  Hence, we now know that $x' = 0$, which was the first of our two statements to prove. The second statement we need to prove is that in this part is that y' is odd. In light of (iv) and (v), our original assumptions (i) and (ii) can be refined as:

  (vi) $(2, 0, y') \in P$

  (vii) $(2, 0, y') \Rightarrow (5, 0, y')$

  From (vi) and the definition of P, we deduce that y' is odd. That's all for this case.

- To prove (b) we assume the statements on the left hand side of the implication:

  (iii) $pc' = 2$ and $x' = 0$

  and we have to show that the statement on the right: y odd, necessarily follows. In light of (iii) we can refine our assumption (ii) as:

  (iv) $(pc, x, y) \Rightarrow (2, 0, y')$

  By the definition of the transition relation $\Rightarrow$, this can only happen in two ways, either because:

  - $pc = 1$, $x' = x * x + 1$ and $y' = y$
  - or $pc = 4$, $x' = x$, $y' = 2 * y - 1$

  However, the first of these is impossible because we know from (iii) that $x' = 0$. Therefore, the transition must have occurred due to the second bullet, so everything in that bullet is necessarily true:

  (v) $pc = 4$, $x' = x$, $y' = 2 * y - 1$

  But, we can observe that $2 * y - 1$ will always be odd, no matter the value of y, and thus we have that y' is indeed odd.
