Technical University of Munich
Chair for Logic and Verification
Prof. Tobias Nipkow, Ph.D.
J. Rädle, L. Stevens, K. Kappelmann, MC M. Eberl

WS 2019/20
17.01.2020
Deadline: 28.01.2020, 23:59

Functional Programming and Verification

Sheet 13

Tutorial Exercises

Exercise T13.1 Huffman Coding

In the lecture, you were introduced to Huffman's compression algorithm. In this exercise you will write a program that uses this algorithm to compress files. Concretely, we ask you to implement the following functions:

```
compress :: String -> FilePath -> IO ()
decompress :: FilePath -> IO String
```

`compress s file` should compress the string `s` and store the result in `<file>.huff`. The tree used for decompression should be saved as `<file>.code`.

The invocation `decompress file` should read `<file>.code` to decompress `<file>.huff` and return the decompressed string.

Hint: You can convert the trees and bitlists defined in `Huffman.hs` to and from strings using `show` and `read`, respectively.

Note that no actual compression is achieved here, because we print the bitlists and the encoding tree quite verbosely. To write compact representations of these datastructures, one could e.g. use the `bitstring` library.

Exercise T13.2 Arithmetic Expressions

In this exercise, we consider the datatype `AExp` which models addition and multiplication on integers:

```
data AExp = Val Integer | Add AExp AExp | Mul AExp AExp
  deriving Eq
```

We define a function `eval` to evaluate an expression to an integer, and a function `simp` that simplifies expressions of the form `0 + e` to `e`:

```
eval (Val i) = i
eval (Add a b) = (eval a) + (eval b)
eval (Mul a b) = (eval a) * (eval b)

simp (Val i) = Val i
simp (Mul a b) = Mul (simp a) (simp b)
simp (Add a b) = if a == Val 0 then simp b else Add (simp a) (simp b)
```

Your task is to prove that this simplification preserves the value of an expression, i.e. that the following equation holds:

```
eval (simp e) = eval e
```

You may use these familiar axioms, no further rules for arithmetic should be required:

```
axiom addZero: x + 0 = x
axiom zeroAdd: 0 + x = x
```

Note that CYP cannot handle this proof, because it supports neither case distinction on integers nor rewriting with Haskell equalities (`==`). For the purposes of this exercise (and the exam), you may use case distinctions on `==` and use expressions of the form `x == y` to rewrite `x` to `y`. Here is a generic example:

```
To show: ... (if x == 10 then y else z) ... = ...
Proof by case analysis on x == 10
  Case True
    Assumption: x == 10
    Proof
                            ... (if x == 10 then y else z) ...
      (by Assumption) = ... y ...
      ...
      ...               = ... x ...
      (by Assumption) = ... 10 ...
    QED
  Case False
    Assumption: x /= 10
    Proof
                            ... (if x == 10 then y else z) ...
      (by Assumption) = ... z ...
      ...
    QED
QED
```

Homework

You need to collect 2 out of 3 points (P) to pass this sheet.

Exercise H13.1 Isn't that obvious?! [1P]

In this homework you will work with a datatype that models the natural numbers, which is defined as

```
data Nat = Z | Suc Nat
```

Here, `Z` represents 0, and all other natural numbers n are represented as successors of $n-1$. That is, 1 is represented as `Suc Z`, 2 as `Suc (Suc Z)` and so on.

We can define addition on this type recursively

```
add Z m = m
add (Suc n) m = Suc (add n m)
```

Your task is to prove that this definition is commutative:

```
goal add n m .=. add m n
```

Hint: You will need at least two lemmas.

Exercise H13.2 Serial Vectoriser [1: 1P, 2: 1P]

In preparation for this weeks competition, your tasks is to generate a series of Scalable Vector Graphics (SVGs). The series of SVGs can then be converted to series of JPEGs which we can concatenate to a video with ffmpeg. The Übungsleitung generously provides the code for this conversion in the template so you can focus all your attention on handling the SVG files. In particular we consider only a minimal subset of the SVG standard:

- rectangles whose corresponding XML-tag is `<rect>` and which have the attributes `width` and `height`,
- ellipses whose tag is `<ellipse>` and which have attributes `rx` and `ry` describing their radii in the x and the y direction, and
- groups whose tag is `<g>` and contain rectangles and ellipses. (In general groups may also contain other groups, but you will not be required to handle this)

All of the above have two additional attributes in common, namely `id` and `transform`. The former will be used to address specific SVG elements in a SVG while the latter is useful for manipulating elements to generate animations. A `transform` consists of the three components `rotate`, `scale`, and `translate`. While rotate is a single float, scale and translate are a pair of floats to account for both the x and the y direction. In Haskell, we can represent `transform` as a record:

```
data Transform = Transform {
    rotate :: Double,
    scale :: (Double, Double),
    translate :: (Double, Double)
  }
```

Records are similar to ordinary data constructors but with the advantage that we automatically obtain accessor functions for the fields of a record, e.g. we can access the first field of a specific `Transform t` using `rotate t`. When constructing an SVG element the transform should initially be set like follows:

```
Transform{rotate=0, scale=(1,1), translate=(0,0)}
```

Your task is to implement the functions

1. `paint :: String -> String`
2. `animate :: [(String, Transform -> Transform)] -> String -> [String]`

`paint` takes geometric objects encoded as a `String` (see below for the input format) and returns a string encoding of the corresponding SVG. You should write data types that represent SVGs in order to accomplish this. (You may use records as shown in the definition of `Transform`)

`animate` takes a list of tuples `ts` that represent transformations that should be applied to parts of the SVG in order to produce an animation. Like `paint` it also takes a string `s` that encodes geometric objects which should be used as the initial SVG for the animation.

The input format looks as follows:

```
group 1:
  ellipse 2 27.7031 36.4382
  rectangle 3 31.5652 34.9351
group 4:
  ellipse 5 14.5585 38.6296
```

Every ellipse or rectangle belongs to exactly one group, which is printed as `group <id>:`. The elements belonging to the group are shown on the subsequent lines in the format `ellipse <id> <rx> <ry>` or `rectangle <id> <width> <height>`.

Each tuple `(i, f)` in `ts` contains a function f which is to be applied to all SVG elements with the id i in the current SVGs. By applying the functions in `ts` iteratively to the initial SVG `s`, the function `animate` creates a list of strings representing SVGs. An example invocation could look like follows:

```
>>> mapRotate f Transform{rotate=r, scale=s, translate=t} =
        Transform{rotate=f r, scale=s, translate=t}
>>> mapScale f Transform{rotate=r, scale=s, translate=t} =
        Transform{rotate=r, scale=f s, translate=t}

>>> -- Print all elements of the list to the console
```

```
>>> mapM_ putStrLn $ animate [
    ("1", mapScale (\(sx, sy) -> (2 * sx, 0.5 * sy))),
    ("2", mapRotate (+ 90))
    ] ...

<svg viewBox="-25 -25 50 50" xmlns="http://www.w3.org/2000/svg">
    <rect id="1" width="20" height="15"
        transform="rotate(0) scale(1 1) translate(0 0)"/>
    <g id="2" transform="rotate(0) scale(1 1) translate(0 0)">
        <ellipse id="1" rx="2" ry="3"
            transform="rotate(0) scale(1 1) translate(0 0)" />
    </g>
</svg>

<svg viewBox="-25 -25 50 50" xmlns="http://www.w3.org/2000/svg">
    <rect id="1" width="20" height="15"
        transform="rotate(0) scale(2 0.5) translate(0 0)"/>
    <g id="2" transform="rotate(0) scale(1 1) translate(0 0)">
        <ellipse id="1" rx="2" ry="3"
            transform="rotate(0) scale(2 0.5) translate(0 0)" />
    </g>
</svg>

<svg viewBox="-25 -25 50 50" xmlns="http://www.w3.org/2000/svg">
    <rect id="1" width="20" height="15"
        transform="rotate(0) scale(2 0.5) translate(0 0)"/>
    <g id="2" transform="rotate(90) scale(1 1) translate(0 0)">
        <ellipse id="1" rx="2" ry="3"
            transform="rotate(0) scale(2 0.5) translate(0 0)" />
    </g>
</svg>
```

Exercise H13.3 The art of ending the Wettbewerb

Your task in this very last competition exercise is to generate a pixel image, a vector image, or a video in one of the customary formats (z.B. PNG, SVG, MP4). To realise that you can use and extend the code from the previous exercise. Furthermore, we allow and encourage you to utilise third-party libraries. Some starting points are given by the links below.

> `http://www.haskell.org/haskellwiki/Applications_and_libraries/Graphics`
> `https://hackage.haskell.org/package/JuicyPixels`

The generated imagery will be graded by all the MCs with the criteria being aesthetics (nice picture) and technology (impressive code). As an inspiration for aesthetics consider the last iteration of this competition exercise and for technology you can consider using lenses to make modifying nested data structures elegant.

Important: Your submission to the Wettbewerb should contain WETT tags and an example image created with your program. Furthermore, it should contain instructions on how to generate the image(s). If your code uses libraries that are not enabled on the server, you may send it directly to fpv@in.tum.de.

> If you truly love Nature, you will find beauty everywhere.
>
> — Vincent van Gogh