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

WS 2020/21
04.12.2020
Deadline: 14.12.2020, 23:59

Functional Programming and Verification

Sheet 5

Exercise T5.1 Be More General

We define functions `sum :: Num a => [a] -> a` and `(++) :: [a] -> [a] -> [a]` as follows:

```
sum xs = sum_aux xs 0

sum_aux [] acc = acc
sum_aux (x:xs) acc = sum_aux xs (acc+x)

[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Use structural induction to show that

```
sum (xs ++ ys) = sum xs + sum ys
```

Exercise T5.2 Computation Induction

We define the functions `sum :: Num a => [a] -> a` and `sum2 :: Num a => [a] -> [a] -> a`:

```
sum [] = 0
sum (x:xs) = x + sum xs

sum2 [] [] = 0
sum2 [] (y:ys) = y + sum2 ys []
sum2 (x:xs) ys = x + sum2 xs ys
```

Use computation induction to show that `sum2 xs ys = sum xs + sum ys`.

The CYP syntax differs slightly from the one presented in the lecture. As an example, given the following definition

```
myFunc xs [] = xs
myFunc [] (y:ys) = myFunc (y:ys) []
myFunc (x:xs) (y:ys) = myFunc (y:xs) ys ++ myFunc xs ys
```

CYP expects `Proof by computation induction on xs ys with myFunc` to start a computation induction proof. Also note that computation induction was only added to CYP a few days ago. You hence might have to reinstall cyp by first pulling the newest changes from the repository.

Exercise T5.3 Type Inference

Determine the most general types of the following definitions:

```
f u v = u < 0 && u < v
f u v w = u + v
f u v = min (head u) v
f = 0
f u v = let w = u && v in []
f u v = let w = u && v in if w then [u] else []
f u v = concat [u | (d,_)<-v, u==d]
f u = [x | ((x:xs):zs)<-u]
```

Use the type-inference algorithm presented in the lecture once. Solve the other cases informally, collecting and solving type-equality and typeclass constraints using your intuition and ad-hoc reasoning.

Homework

You need to collect 2 out of 3 points (P) to collect a coin.

Exercise H5.1 (**Wettbewerb**) Cubing the Cuboid [1P]

> **Note:** While this exercise is not difficult to implement as such (the MC Sr's solution is 3 lines of easy Haskell), it is a bit of a brain teaser. If you're just looking for a few quick homework points and don't care about the *Wettbewerb*, you might want to start working on the other homework exercises first.

The MC Sr wants to build a model of the logo of his favourite proof assistant out of wooden cubes. To this purpose, he purchased a cuboid-shaped piece of wood (the dimensions are all multiples of 1 cm). He now wants to completely carve up this piece of wood into cubes. However, since he vastly overestimated how much wood he needs, he wants to cut it up in such a way that the total number of cubes is as small as possible. To make things easier, he only wants cubes whose sizes are powers of two in centimetres (i.e. 1 cm, 2 cm, 4 cm, 8 cm, etc.)

Your task is to write a function `decompose :: [Integer] -> [Integer]` that, given the dimensions of the piece of wood, returns how many cubes of each type the optimal decomposition has (in ascending order by size, with no trailing zeroes). Because the MC Sr is a big fan of pointless generalisation, he also requires your function to work not just in 3 dimensions, but in *any* dimension ≥ 1. See Figures 2 and 3 for an illustration.

Example:

```
decompose [123] = [1,1,0,1,1,1,1]
decompose [6, 5] = [6, 2, 1]
decompose [9, 6] = [6, 4, 2]
```

```
decompose [8, 4, 4] == [0, 0, 2]
decompose [10, 10, 10] == [0, 61, 0, 1]
decompose [10, 5, 5] == [90, 4, 2]
decompose [6, 5, 4] == [24, 4, 1]
decompose [9, 18, 27, 36] == [22680, 1512, 48, 24]
```

Hint: It is not recommended to brute-force all decompositions. You have to use your geometric intuition to come up with a clever recursion. You can try starting in 1 or 2 dimensions and then generalise your approach to n dimensions. The generalisation should be straightforward.

Wettbewerb only: For the *Wettbewerb*, the MC is looking for optimal performance. For reference, the MC Sr's solution takes 5 s for a 10^4-dimensional cube of size 10^{12}. Should several submissions have similar performance, he will rank them according to the somewhat ~~arbitrary~~ subjective criteria of *brevity* and *elegance* (*not* number of tokens).

Additional Wettbewerb challenge: Geometric intuition is very useful, but it can easily mislead you. You have the *Artemis* tests to guide you, of course (if they go through, your algorithm is probably correct). However, the MC Sr likes proofs even better than tests. In accordance with the current proof-heavy focus of the lecture, he will therefore award an appropriate number of bonus *Wettbewerb* points to the first few students to submit a convincing proof that their approach is correct. You can text the MC Sr. on Zulip or send him an e-mail. Beware though: the MC Sr is an awful pedant!

Exercise H5.2 Map and Territory [1P]

An anonymous MC loves accumulators and has written this slightly inelegant (and inefficient) version of the `map` function:

```
accum f [] ys = ys
accum f (x:xs) ys = accum f xs (ys ++ [f x])
```

Prove that this function is indeed equivalent to plain old `map`:

```
accum f xs [] .=. map f xs
```

We have provided you with two helpful lemmas about the `++` function in the template.

Figure 1: An optimal decomposition of a one-dimensional cuboid of size 23 into one-dimensional cubes of sizes 16, 4, 2, and 1. The type of the decomposition is `[1, 1, 1, 0, 1]`.

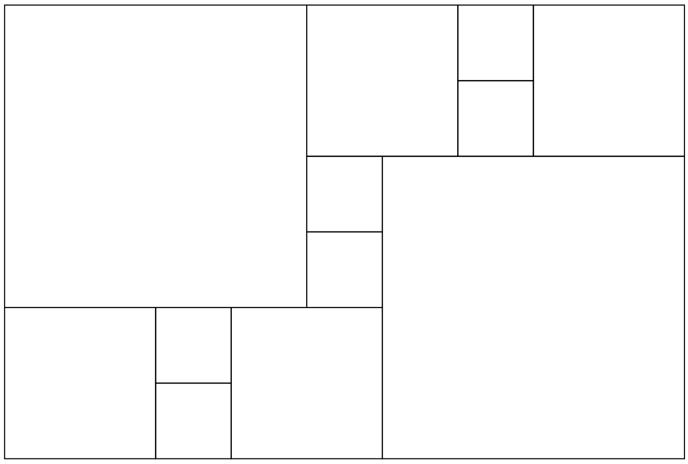

Figure 2: An optimal decomposition of a 9×6 cuboid. The type of the decomposition is `[6, 4, 2]`.

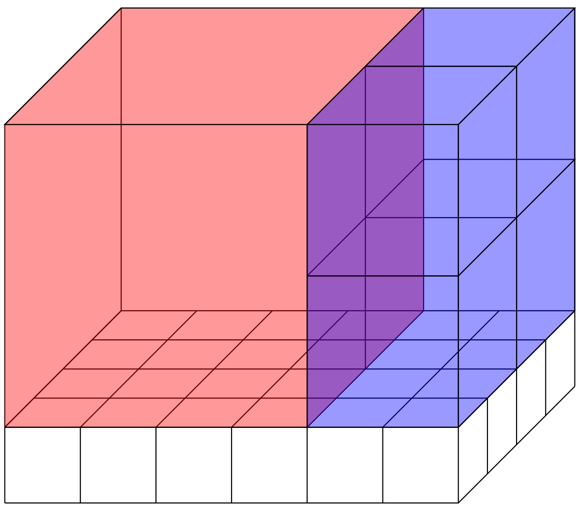

Figure 3: An optimal decomposition of a $6 \times 5 \times 4$ cuboid into one cube of size 4 (red), four cubes of size 2 (blue), and 24 cubes of size 1 (white). The type is therefore `[24, 4, 1]`.

Exercise H5.3 Haarspalterei [1P]

In the lecture, function `splice` was introduced that splices two lists together like a zipper. Using another function from the lecture, namely `drop2`, we now define the inverse function `unsplice` that unzips a list into a pair of lists:

```
unsplice [] = Pair [] []
unsplice (x : xs) = Pair (drop2 (x : xs)) (drop2 xs)
```

We write `Pair a b` instead of `(a, b)` due to technical limitations imposed by CYP. Your task is to prove that `unsplice` behaves as a right-inverse to `splice`, i.e. using computation induction prove that

```
splice (fst (unsplice xs)) (snd (unsplice xs)) .=. xs
```

You will also need case analysis on the datatype of lists for which the syntax in CYP looks as follows:

```
Proof by case analysis on List xs

Case []
  Assumption: xs .=. []
  Proof
      ...
      (by ...)   .=. f xs
      (by Assumption) .=. f []
      ...
  QED

Case x:xs'
  Assumption: xs .=. x:xs'
  Proof
    ...
    (by ...) .=. f xs
    (by Assumption) .=. f (x : xs')
    ...
  QED
QED
```

> Ich mag Philosophen nicht, die das Haar auf fremden Köpfen spalten. Noch dazu mit einem Beil.
>
> — Stanisław Jerzy Lec