Assignment 6, INF222, Spring 2023

Compulsory: Deadline 2023-03-23 at 12:15

There are many sources of coding errors that can have devastating effects. Static analysis, e.g., strong typing, is one of the safeguards against coding errors. We may deliberately improve this by using fine-grained typing: having separate types for length, mass, time etc will discover more errors then using just a number type, even though all of these distinct types may be implemented by the same 64bit floating-point numbers (commonly known as `Double`). Introducing units, such as meters, kilograms and seconds, allows us to infer appropriate types for such literals.

We may even want to separate large lengths, moderate lengths and small lengths as separate types. The former for astronomical scale data, the middle one for human scale data, while the latter for atomic scale data. So numbers in light years or parsec would be large lengths, in km or mm will be moderate lengths, while nanometer or picometer are small lengths. Demanding that all data, also inputs, is given a unit will allow our tools to provide additional code safety.

This assignment explores fine grained typing with units within the calculator object language from the pamphlets (if you have not completed the pamphlets, it can be a good start to look at previous pamphlets). Based on the framework of Pam8T you will implement a calculator for yarn. Pam8T supports multiple types, overloading (see PAm8O) and can provide decent error messages when a function is applied to inappropriate arguments. There are various strategies for extending Pam8T to handle units.

6.1 What to turn in

You should submit at link to your GitLab repo to the MitUiB Assignment 6c. In your GitLab repo we expected to find:

- All files your solution needs, i.e., those provided by us (edited or not by you) and any new files you create by you.
- Your solution should use the file names given in the task descriptions. Each file should contain a header explaining which problem it is solving, who wrote it, and the date.
- A file called README.md containing text (non-code) answers to questions. The text should be in markdown (suffix `.md`) or plain text (suffix `.txt`) or PDFs (suffix `.pdf`). Other formats may be acceptable, but this must be agreed with your group leader.

Start early and go to the labs to get help!

- Solving the tasks can give you up to 30 points.
- You can also gain an additional 5 bonus points.

Bonus points are given when you solve more tasks than needed, when your solution is particularly elegant, and when you have extensively documented your solution. Summarised: you get a bonus for high-quality software engineering.

In this compulsory assignment, you will be handing in your solution by yourself. Hence all students have to complete all tasks (except bonus tasks) for a maximum score.

6.2 Handing in the assignment

You will all be assigned a GitLab repo where you will work on the assignment.

We found that automatic testing is unsuitable for this assignment, so there will be no automated testing this time.

You can find your repository on `https://git.app.uib.no/`

NOTE! If you've never used git before, the following link has information on how to use it. `https://docs.gitlab.com/ee/gitlab-basics/start-using-git.html`

6.3 Tasks

Your task is to implement a series of yarn calculators, reflect on the implementations, and implement some improvements to the calculator. Reuse code from the modules, e.g., `Pam8IntrinsicsReal` and `IntrinsicsYarn`, where possible, by importing the relevant types and functions into your solution.

6.3.1 Implement a yarn calculator with units in the literals, 10 points

Implement a yarn calculator following the idea discussed in section 6.4.2

For this calculator the example input strings (the one with a unit error and the correct one) will look like the following.

```
SetVar "w" (Fun "Mult" [Lit 0.35 "NOK/meter",Fun "Mult" [Lit 160.0 "gram",Lit 4.0
    "amount"]])
SetVar "a" (Fun "Mult" [Lit 0.35 "NOK/meter",Fun "Slash" [Fun "Mult" [Lit 160.0
    "gram",Lit 4.0 "amount"],Lit 0.2 "gram/meter"]])
```

Here you will need to upgrade exisiting code from P8T. Do this by copying the module into a new file P8U, e.g., copy `Pam8TSignatureAST` into `Pam8USignatureAST` before doing the upgrade.

Hint! Sometimes the changes may be as little as removing the type annotations on functions (and updating the module names).

6.3.2 Implement a yarn calculator with units in the semantic domain, 5 points

Implement a yarn calculator following the idea discussed in section 6.4.3

For this calculator the example input strings (the one with a unit error and the correct one) will look like the following.

```
SetVar "w" (Fun "Mult" [Lit (0.35, "NOK/meter"),Fun "Mult" [Lit (160.0, "gram"),Lit
    (4.0, "amount")]])
SetVar "a" (Fun "Mult" [Lit (0.35, "NOK/meter"),Fun "Slash" [Fun "Mult" [Lit
    (160.0, "gram"),Lit (4.0, "amount")],Lit (0.2, "gram/meter")]])
```

6.3.3 Implement a yarn calculator with typed semantic domain, 5 points

Implement a yarn calculator following the idea discussed in section 6.4.4

For this calculator the example input strings (the one with a unit error and the correct one) will look like the following.

```
SetVar "w" (Fun "Mult" [Lit (UnitCost 0.35),Fun "Mult" [Lit (Weight 160.0), Lit (Amount
    4.0)]])
SetVar "a" (Fun "Mult" [Lit (UnitCost 0.35),Fun "Slash" [Fun "Mult" [Lit (Weight
    160.0), Lit (Amount 4.0)], Lit (Density 0.2)]])
```

6.3.4 Discuss the implementation strategies, 5 points

The three implemetations above have different propeties from a software evolution viewpoint (from a software engineering perspective), and the resulting calculators may have different useability properties (from a calculator user perspective).

Here you should discuss and evaluate calculators from these perspectives.

6.3.5 Help message improvements, 5 points

If a user types a wrong expression to the calculator, the calculator will provide a help message giving the general form of the caclulator expressions. This help message does not include adice on correcting literal syntacs, which is somewhat diferent in the three caclulator implementations.

Implement improved help messages to the user. This will require extensions to the calculator template.

There are bonus points if the implementation strategy is reusable across caclulators. That is, the upgraded calculator template should be reusable for all intrinsics without requiring further modification of the calculator template. Thus any further adaption should be local to the intrinsics module.

6.3.6 Usability improvements, 5 bonus points

The calculator notation is fairly horrible from a user perspective.

Suggest a better calculator notation, both for user inputs and for the feedback messages.

Implement these improvements. This requires replacing the haskell based input parser (`read` function) with a specifically developed rudimentary parser for user input.

6.4 Background and motivation for a yarn calculator

Some of your nerdy friends have taken up knitting during the Corona shutdowns the last few years. They started to experiment with mixing different types of yarns from different sources, and established a network of suppliers from around the world. Boasting about their designs on social media, other people from the knitting community got interested in what your friends were doing and how to get hold of the relevant yarns. Your friends are now preparing to start a business buying and selling yarn and knitting designs.

To run the business they need a spreadsheet calculator to compute yarn, cost and design parameters when discussing with customers. For now they use the real number calculator from INF222 pamphlet 8. Gaining some experience, your friends have realised that it is very easy to mix up the various numbers involved.

- Total cost (NOK), typically 50-1500 NOK.
- Density of thread (gram/meter), typically 0.2-0.4g/m, related to the thickness and type of yarn.

- Length of a roll of yarn (meters), typically 150-600m.
- Unit cost of yarn (NOK/meter), typically 0.3-0.5 NOK/m, related to thread density and type of yarn.
- Weight of a roll of yarn (grams), typically 50-200g.

As an example, consider an expensive yarn with rolls at 160 gram, density 0.2 gram/meter, unit cost 0.35 NOK/meter, and an order for 4 rolls. Entering the calculation $0.35*160*4$ yields the plausible, but wrong, cost of 224 NOK. The correct calculation is $0.35*(160*4/0.2)$, yielding a cost of 1120 NOK. In the real number calculator from P8s on gitlab these computations, for customers Ole and Kari, could be entered as below.

```
¢ SetVar "cost_ole" (Fun "Mult" [Lit 0.35, Fun "Mult" [Lit 160, Lit 4]])
SetVar "cost_ole" = 224.0
¢ SetVar "cost_kari" (Fun "Mult" [Lit 0.35, Fun "Slash" [Fun "Mult" [Lit 160, Lit 4],
    Lit 0.2]])
SetVar "cost_kari" = 1120.0
```

Here Ole is getting a real bargain on the yarns because of this mistake, while Kari is charged the regular price. The business cannot afford making many such mistakes.

Your friends are therefore thinking of making the calculator use units and fine-grained typing. Then an incorrect calculation would yield an error message instead of the wrong value. Assuming a type checking yarn calculator with units, the computations above would be entered as below.

```
¢ SetVar "cost_ole" (Fun "Mult" [Lit 0.35 "NOK/meter",Fun "Mult" [Lit 160.0 "gram",Lit
    4.0 "amount"]])
Syntax error, undeclared functions / variables : ("Mult",["UnitCost","Weight"],"?")
¢ SetVar "cost_kari" (Fun "Mult" [Lit 0.35 "NOK/meter",Fun "Slash" [Fun "Mult" [Lit
    160.0 "gram",Lit 4.0 "amount"],Lit 0.2 "gram/meter"]])
SetVar "cost_kari" (Lit 1120.0 "NOK")
```

Now the error in computing Ole's cost is detected, while the computation for Kari goes through.

6.4.1 Yarn calculator signature

For this purpose your friends have thought of using the following signature for a yarn calculator. The accompanying measurement units are listed in parenthesis in the documentation.

```
-- | The number of rolls of yarn (amount)
type Amount
-- | The total cost for a purchase of yarn (NOK)
type Cost
-- | Density of thread (gram/meter)
type Density
-- | Length of a roll of yarn (meter)
type Length
-- | The unit cost for yarn (NOK/meter)
type UnitCost
-- | Weight of a roll of yarn (gram)
type Weight
-- | Add two costs
Add :: Cost, Cost -> Cost
```

```
-- | Subtract two costs
Sub :: Cost, Cost -> Cost
-- | Compute weight from density and length
Mult :: Density, Length -> Weight
-- | Compute length from weight and density
Slash :: Weight, Density -> Length
-- | Compute density
Slash :: Weight, Length -> Density
-- | Add two lengths
Add :: Length, Length -> Length
-- | Subtract two lengths
Sub :: Length, Length -> Length
-- | Multiply length by amount
Mult :: Length, Amount -> Length
-- | Compute amount
Slash :: Length, Length -> Amount
-- | Compute cost based on length
Mult :: UnitCost, Length -> Cost
-- | Compute unit cost from cost and length of yarn
Slash :: Cost, Length -> UnitCost
-- | Add two weights
Add :: Weight, Weight -> Weight
-- | Subtract two weights
Sub :: Weight, Weight -> Weight
-- | Multiply weight by amount
Mult :: Weight, Amount -> Weight
```

The types and units chosen for this calculator langugae is very specific for the yarn domain. They limit the operations for each type of the yarn calculator, thus limiting the mistakes that will be silently accepted by the calculator. The operation names are overloaded to give a similar association as in the real calculator. In the yarn calculator each literal must in addition to its value be given its unit of measurement. These units are mapped to the corresponding calculator type, and the expressions can be type checked and warning messages provided if the types do not match. The actual values and computations should use the same semantics as for the real number calculator.

Your task is to implement a Yarn calculator, and ensure all safety checks are in place. Attached is a real calculator `XXX` based on P8T. It also contains some related declarations for a Yarn calculator, including the signature listed above.

There are several ways to extend P8T with units. Three of them are discussed below.

6.4.2 Calculator with units in the literals

This idea is to enhance the calculator modules with units on the literals in the AST.

```
data CalcExprAST valuedomain
  = Lit valuedomain UnitName
  | Fun FunName [CalcExprAST valuedomain]
  | Var VarName
  deriving (Eq, Read, Show)
```

For the real calculator, `(Lit 5)` is no longer valid, while `(Lit 5 "")` and `(Lit 5 "parsec")` should work.

This seemingly small change has repercessions in most of the modules. Many of the support

functions need to be updated to take into account the new literal syntax. In other cases the only change needed is to include the unit based modules rather then the regular one.

The `typeOfValue` function needs to map the units to their appropriate domain types.

6.4.3 Calculator with units in the semantic domain

This idea is to change the calculator's semantic domain to include units. For example, a real calculator would change domain from `Double` to `(Double,UnitName)`, where `UnitName` is a string.

For a yarn calculator the units will be like `"gram/meter"`. The `typeOfValue` function must map the value-unit pairs to their appropriate domain types, e.g., `"Density"`.

The repercessions here should limit themselves to minor changes in the intrinsic module.

6.4.4 Calculator with typed semantic domain

This idea is to change the calculator's semantic domain to a sum of products, where each case reflects a type from the domain. This is how the personnel calculator from its P8T module is implemented. Here the units become secondary, as the values are typed explicitly by the cases.

The repercessions here are limited to the intrinsic module.