Documentation: Dimension Analysis

Beerend Lauwers Frank Wijmans

July 6, 2012

1 Introduction

This document describes the source code for the dimension analysis on a simple language. The analysis we do is inspired by a thesis[1] by Andrew Kennedy. He wrote his Ph.D. on the topic of dimension analysis in 1996. In his thesis, he argues that in the field of physics it is required to specify the dimensions, while in programming it is normal to leave numbers dimensionless, allowing very severe bugs to creep in the code that a normal typing system is unable to detect.

The rest of this document is structured as follows. Section 2 elaborates on the problem at hand. In section 3 we briefly show the target language on which we analyse. Next we will talk about the implementation of our analysis in section 4. After which we will conclude in section **??**.

2 Problem description and approach

In programs, numbers are used all the time for multiple purposes. The original programmer knows the context within which they are used and maybe even by looking at the variable's name, knows that some number is a length, some currency value or even a compound value (say, meters per second). This context (for a single number: its specific dimension) is implicit. This implicit context is not always so clear for others, and allows the creation of hard-to-find bugs.

We analyse a given piece of code, coupled with an environment that specifies what function or variable maps to what dimensional value. Dimensional values can either be monomorphic or polymorphic. For instance, $3m^2$ would map to the dimension multiplication of two meter dimensions. Meter is a unit inside the Length base. But a function $id :: a \to a$ works for every monomorphic dimensional value: hence, it is a polymorphic dimensional value.

In the thesis it is made clear that a dimension analysis can be done by extending the programming language itself with dimension types. We show with our analysis that if the dimension types are given externally, so not embedded in the language itself, that check is possible as well. Future work could be implementing the dimension types in the language itself.

Using the extra information given, we can suggest coercions between, for instance, the metric and imperial systems for measurement of lengths. Furthermore, we generate errors that inform the programmer of incorrect function application. For example, if you have a function that takes two values in meters (or some other length value) and creates a surface (or a multiplication of two

lengths), it would be wrong to give it anything but two lengths. Like a type system, it will give more information to the compiler which in its turn informs the programmer of any problems encountered during compilation.

Next, let's take a look at the language.

3 Language

The target language is the simply typed Lambda calculus with non-recursive let statements (see figure 1).

```
data Term
  | Var String
  | Lam String Term
  | App Term   Term
  | Let String Term Term
```

Figure 1: Simply typed lambda calculus with let-constructs.

With these four language constructs, we are able to model a subset of Haskell. In our analysis, we build up dimensions as follows:

- `Var` - Fetches a `DimVar` from the environment or generates a fresh one.
- `Lam` - Generates a `DimArrow` to indicate a dimensional function.
- `App` - Application takes the right hand side of the `DimArrow` of the function and returns that as its dimension. (e.g. (f::a→b) x, hence dimension of (f x) is b)
- `Let` - `Let` returns the dimension of its body.

A `Let` construct adds a new constraint equality to the environment and passes it down to its body.

4 Implementation

The Hindley-Milner typing code was recycled from the last CCO assignment by Augusto Martins and João Alpuim. We were allowed to use their code with their permission.

We have used attribute grammars to define how the constraints are built. The file `DimensionTyping.ag` contains the code for building up constraint equalities and dimension inference. Constraint equalities are tuples of two dimensions, a constraint tells us two dimensions were found to be equal.

Dimensions are defined as follows from figure 2, note that this is AG notation.

Respectively the constructors formulate: functions, bases, variables, polymorphic variables, inverse and multiplication. Inverse is used to define division.

The `Base` datatype describes five main physical quantities.

Although we are aware that there are more quantities to add, for simplicity we only implemented Mass and Length fully. It is enough to show how the

```
data Dimension
   | DimArrow a1 :: Dimension a2 :: Dimension
   | DimBase b :: Base
   | DimVar v :: {String}
   | DimPoly v :: {String}
   | DimInverse a :: Dimension
   | DimMult a1 :: Dimension a2 :: Dimension
```

Figure 2: Dimension as defined in our implementation in AG notation.

```
data Base
   | Mass s :: MassScale
   | Length s :: LengthScale
   | Time s :: TimeScale
   | ElectricCharge s :: ElectricScale
   | AbsoluteTemperature s :: TemperatureScale
```

Figure 3: Bases for dimensions in AG notation.

analysis works. It is always possible to extend the amount of bases, such as adding currency and other possible contexts for numbers.

For now, the analysis is based on the dimensions and bases as given above. In the case of an error, we figured we want to tell the programmer what went wrong, and where. The collection of constraints in the implementation is a list of `AnnConstraint`, which is a tuple of a single constraint equality, its position in the source code and the code term from which it originated.

Next, we built a constraint solver. The code can be found in `ConstraintSolver.hs`. At the end of the analysis we have built up a large list of constraints, which we should try to rewrite such that it finds that it either is completely correct or there is some error. In the last case we will return the message to help the programmer.

The solver consists of a couple of steps.

- `flattenArrows` This function is applied on the collection of constraints, where it checks if the constraint is an equality between two `DimArrows`. If this is the case, it splits up the `DimArrow`, creating new unique constraints.
- `solve` and `solve'` The first is the entry function, it applies the flattening function and calls `solve'`. `solve'` checks a single constraint equality with the `rewrite` function, which results in either an error (after which analysis halts), or a (possibly empty) list of warnings, after which the next constraint equality in the list is evaluated.
- `rewrite` This function does three things:
  - Reordering the list by swapping the sides of an equality and pushing down not yet solvable multiplication equations.
  - Rewriting the constraint list by replacing variables.
  - Calling the validation functions if a solvable equality is found.

- `valid`, `validBases` and `validMult`
  - The function `valid` checks whether a single constraint is a valid constraint. This is usually called when the analysed code ends up generating a collection of constraints that solve to an equality like `DimVar a = DimBase (Length Meter)`, which is not really wrong, but is not a valuable conclusion.
  - `validBases` checks whether the dimensions are coercible: a constraint equality of `DimBase a = DimBase b` generates either a(n) (empty) coercion message, or a coercion error to indicate the dimension types `a` and `b` cannot be coerced into each other.
  - `validMult` checks dimensional multiplications on their validity. Multiplications are special because they should first be simplified and reduced: a dimension type of the form $(m * s)/(m/(m * s))$ will be simplified to the form $(m*s)*(m*(m*s)^{-1})^{-1}$, which further simplifies to $m*s*m^{-1}*m*s$. This form can then be reduced to $m*s*s$. It is these simplified and reduced forms that are used to verify if a multiplication equality is valid. A multiplication equality is valid if the "base count" is the same, that is, if each side of the equality has the same amount of bases for each type of base. For example, the equality `m*s = s*m` holds because each side has the same amount of bases for each type of base. If a multiplication equality is valid, the bases of each side are flattened to a list, sorted, zipped up, and compared if they are coercible.

5 Examples

The `examples` folder provides a few examples of the analysis in action. To analyze an example, go to `src/hm` and run `runtest.sh` with the path to an example file as a parameter, like so: `runtest.sh examples/id.hm`. Four examples have been provided:

- `id.hm` - a dimension of `m*cm` is passed to a length function `lengthfunc`, which generates a `foot*timevar`, which is passed to the identity function and finally getting applied to `lengthgramfunc`, which expects a `foot*gram`, so an error is raised. Remove `lengthgramfunc` to see the coercion suggestions in action.
- `inverse.hm` - gives an example of how a dimensionally polymorphic function `invert`, which inverts its argument, applied twice, will be simplified to the original argument. A coercion is suggested as well.
- `divmult.hm` - here, a dimension of the type `m / (g*m)` is built up and passed to a function that expects a mass base. The dimension type is simplified to a mass type by removing the `m`. A coercion is suggested.
- `complex.hm` - as the name suggests, this one does some elaborate things behind the scenes. A dimension type of the form `(m*g) * (m / (m*g))` is built up and passed to a function. It is simplified to the form of `m*g*g`. The function itself expects a `(mass*length*mass)`, so the bases are reordered

to allow the dimension type to be accepted by the function, with some coercions placed in the right spots.

6 Future work

There are still several loose ends that were not addressed due to time constraints:

- Dimensions for functions and variables are defined through an external environment instead of written down as extra signatures in the program code.
- Dimensionless constants (such as the number 20) are currently unsupported. If we had gone for a different datatype (for example, a 5-tuple) that captures the bases and their exponentials in the following form,

  $$[X] = L^{\alpha} \cdot M^{\beta} \cdot T^{\gamma} \cdot I^{\delta} \cdot \Theta^{\epsilon}$$

  we could have defined the dimension type of a dimensionless constant as (0,0,0,0,0).
- We started with a very simple functional language which we hoped to extend later on with things like recursive lets.

References

[1] A. Kennedy, "Programming languages and dimensions," tech. rep., University of Cambridge.