Semantic Search For Quantity Expressions

Bachelor Thesis DRAFT*

EdN:1

Tom Wiesing
Supervisor: Michael Kohlhase
Co-supervisor: Tobias Preusser
Jacobs University, Bremen, Germany

May 21, 2015

Abstract

In this paper we want to give an approach to Semantic Search for Quantity Expressions.

A Quantity Expression is an expression that represents a quantity such as $25\frac{\mathrm{m}}{\mathrm{s}}$. Different Quantity Expressions can represent the same quantity, for example the expression can also be refered to by $90\frac{\mathrm{km}}{\mathrm{h}}$. In this case we are only using metric units, however using different units can become a problem when having to convert between them constantly.

In a Semantic Search Engine for Quantity Expressions we want to be able to search for quantities independent of their representations. The implementation we have designed is very modular. We use a meta-mathematical formalisation for the unit system. This brings the advantage of being easily extensible with new units. Because of the modularity, the spotting of Quantity Expressions within documents is taken on in [**?**].

Despite this modularity, our implementation has some problems, one of them being that we can not translate between *Celsius* and *Kelvin*. Nonetheless, if expanded and improved properly, this implementation has potential to solve the problem of different units entirely.

*EdNote: Remove draft status

Contents

1 Introduction

1.1 The Problem Of Units

Units are everywhere. We encounter units and quantity expressions in everyday life wherever we go. When driving on the road we see a speed limits on signs (for example $30\frac{\text{km}}{h}$). When we go shopping there are different shoe sizes. When we buy something, we pay a currency of 30€. Everything is being quantified. This is also the case in science papers. Many, if not all, papers have 1 or more quantity expressions in them. Approximatly 1% of all letters in scienficic papers belong to a quantity expression.

This in itself is not a problem. The problem occurs when different units are used to describe the same quantity. In most of the world, the metric system is used to describe most quantities. Some countries still use non-SI units. These different units sometimes make it very difficult to talk about Quantities.

One notable example for this is the *Mars Climate Orbiter* which was destroyed in 1999 when it entered the atmosphere of Mars because it received the non-standard units of *pound-seconds* instead of the expected *newton-seconds* [Boa].

1.2 State Of The Art: Unit Converters And The SI System

Out of convenience new units ones are constantly being invented. Hence there are many hundred units that already exist. Just converting between them is easy, googleing "unit converter" reveals about 12000000 results. Even Google itself has implemented a unit converter into its search engine.

However, all of these tools have a problem. They only convert units directly, meaning they require the user to enter the original units and desired units. This reuires the user to identity that there is a problem with units in the first place.

There is no tool which directly incorperates this translation of units into search results. Such a tool should find quantities independently of which units they are expressed in. It would make search efforts much easier, as fewer user input is required. It would no longer be required for the user to recognise that there is a problem.

In addition to this problem unit converters are usually very restricted in the units they support. They commonly store translation formulae for any pair of units to convert between. Hence they are difficult to extend with new units.

Their superficial handling of units usually does not take the underlying meaning, the semantics, into accoun On the other hand, The SI specification [dPeM] provides a very good insight into how units can be handled. It is precisely defined what each unit means and what kind of quantities can be expressed. It is a very formal approach. This approach does not take into account that it is sometimes less practical to user general units instead of specfic ones.

Neither this very formal nor the previous approach with unit converters can be efficiently used to solve the problem.

1.3 Our Approach: A Semantic Quantity Expression Search

That is why we want to build a search engine that unifies these approaches. It should be capable to find occurrences of quantity expressions within documents, no matter of their representation. For this we need several components: (1) an extensible system flexible enough to

convert between units when needed, (2) a so-called spotter that finds occurrences of quantity expressions within documents, (3) a search algorithm that can take a quantity expression from the user and find equivalent quantities in the results from the spotter and (4) a front-end that allows the user to enter a quantity expression and receive the results.

For (1) we want to use a meta-mathematical theories approach which is implemented by a software called MMT. With the help of MMT this allows us to build an extendible unit system. This uses a concept of a theory graph in which units are related via so-called views and imports. These can be used to translate between them. Additionally, we can define a new unit and easily link it to any of the ones which have been defined previously. Point (2) will be taken on by Stiv Sherko in a seperate effort [She15]. The spotter finds quantity expressions inside documents which can almost directly be used by our system. For our search algorithm (3) we use a simple trick: We normalise spotted units. A normal form is then used to efficiently index the harvest delivered from the previous step. Finally for (4) we built a frontend which allows the user to enter quantity expressions. It is deployed at [Wie]. A basic sketch of the system can be found in Figure 1.

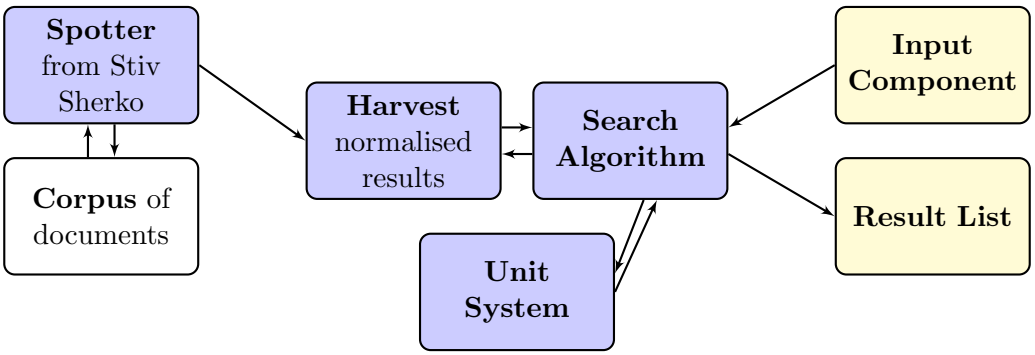


Figure 1: Basic Architecture Of The System. Components Of The Backend Are Drawn In Blue, Components Of The Frontend In Yellow.

1.4 Overview

This thesis is organised as follows: In section 2 we start by giving an introduction to mathematical theory modeling. We proceed in section 3 to talk about how quantity expressions can be formalised and in section 4 we apply these insights in order to start building in our search engine. In section 5 we present the implementation in detail by describing how a search query is processed and how it is presented to the end user. We finally conclude with discussing the limits of this implementation as well as future work in section 6.

2 The Structure Of Mathematics: Theories, Views And Imports

Before we start looking at Quantity Expressions and how to build a search engine for them, we want to give an introduction to meta-mathematical structures. We will use this knowledge later to build a better search engine.

To model mathematics we need a proper logical foundation. For this we use the *LF Logical Framework* [HHP93]. LF provides a typed logic, that is a logic in which each object has a so-called type. It also allow has a function type via the $\to$ constructor.

Next we can take a look at the concept of theories.

2.1 Modeling Mathematics With The Help Of Theories

Theories, in this sense, are simply sets of symbols. Each of the symbols can optionally have a type and a definition. Within each theory, we can then use these symbols to write down terms (or expressions) within this theory. Types and definitions of these symbols are terms themselves[1]. As a simple example of this, let us consider the theory of semigroups as seen in Figure 2.

Semigroup		
G	:	type
$\circ$	:	$G \to G \to G$
assoc	:	ded $(\forall x \in G. \forall y \in G. \forall z \in G. (x \circ y) \circ z = x \circ (y \circ z))$

Figure 2: The Theory Of Semigroups.

In this theory, we define 3 symbols: G, $\circ$ and assoc. In the first line we define a type G. Next we define a function $\circ$ that takes 2 terms of type G as arguments and returns another term of type G. We could also denote this function by $G \times G \to G$ but LF does not provide the $\times$ symbol. Hence we just use the equivalent notation $G \to G \to G$. In the last line, we make the statement that associativity holds. This is achieved via the *ded* (short for deduction) symbol. When a statement (such as associativity formulation in this case) is passed to it, it asserts that the statement is provable by returning a proof for it. This is commonly used for axiomes.

2.2 Extending Theories Using imports

Sometimes we want to extend theories without having to define everything again. For example, we want to say that a Monoid is a semi-group along with an identity element. In the semi-group example above, we have also used terms from other theories to define G as a type.

We can model this concept by using imports. An import from one theory into another makes symbols of the imported theory available in the target theory. In Figure 3 we use this to define a monoid.

[1] They are not terms over the same theory however.

Monoid		
import Semigroup		
e	:	G
id	:	ded $(\forall x : G.x \circ e = e \circ x = x)$

Figure 3: The Theory Of Monoids Which Imports The Theory Of Semigroups As Defined In Figure 2.

2.3 Views As Truth-Preserving Mappings Between Theories

However imports are not the only way how theories can be related. If we have two theories, we sometimes want to have a map between them. In addition to the theory of monoids above we define the theory of non-negative integers in Figure 4.

Non-negative integers		
$\mathbb{Z}_0^+$	:	type
0	:	$\mathbb{Z}_0^+$
$+$	:	$\mathbb{Z}_0^+ \to \mathbb{Z}_0^+ \to \mathbb{Z}_0^+$
assoc	:	ded $\left(\forall x : \mathbb{Z}_0^+ . \forall y : \mathbb{Z}_0^+ . \forall z : \mathbb{Z}_0^+ . (x \circ y) \circ z = x \circ (y \circ z)\right)$
id	:	ded $\left(\forall x \in \mathbb{Z}_0^+ . x + 0 = 0 + x = x\right)$

Figure 4: The Theory Of Non-Negative Integers.

A map from the theory of monoids to the theory of positive integers should map all symbols from the theory of monoids to symbols from the theory of positive integers. Furthermore, such a map should be truth preserving, i. e. if I write down a true statement as a term over the theory of monoids and translate this term, it should still be true in the theory of positive integers. Such a mapping is called a *View* from the theory of monoids to the theory of Positive integers. Such a view ϕ could look as follows:

$$\phi = \left\{ \begin{array}{l} G \mapsto \mathbb{Z}_0^+ \\ e \mapsto 0 \\ \circ \mapsto + \\ \mathsf{assoc} \mapsto \mathsf{assoc} \\ \mathsf{id} \mapsto \mathsf{id} \end{array} \right\}$$

If we take a closer look at this view, we notice that we also have to map the imported symbols. This is needed so that we can translate any term or statement in theory of monoids to a term or statment into the theory of non-negative integers.

2.4 Building Theory Graphs

We have seen in the examples above that we can model mathematics with the help of theories, views and imports. To make this structure even more obvious, we can represent it in a graph, a so-called theory graph. We consider the theories as vertices of such a graph and the views and imports as edges. An example can be found in Figure 5.

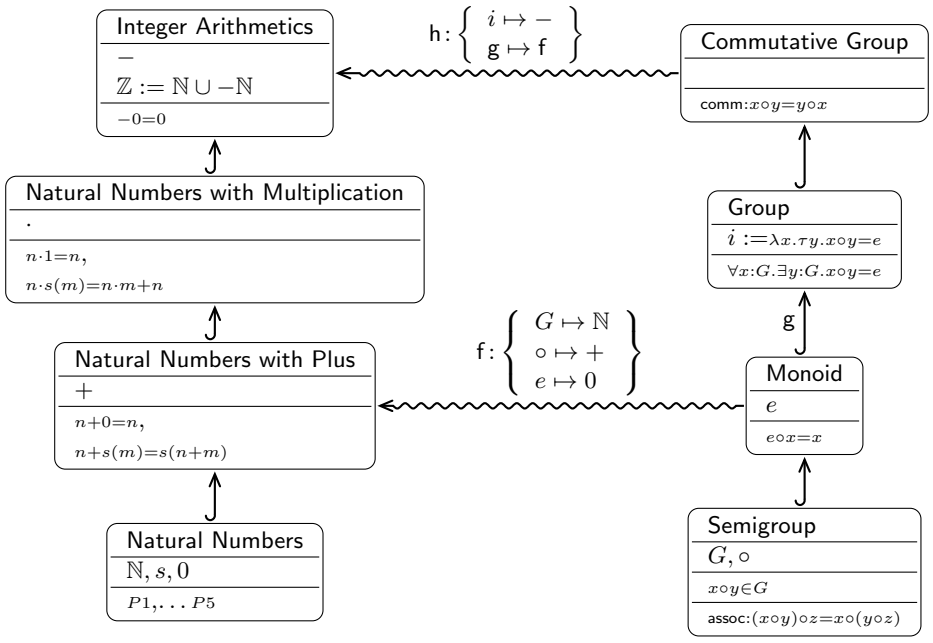


Figure 5: A Simple Theory Graph. Imports Are Represented As Solid Edges And Views As Wavey Edges. The Mappings Of The Views Are Given Explicitly. We Omit The Complete Definitions Of The Peano Axiomes For The Natural Numbers.

2.5 Using MMT To Write Down Terms And Theories

MMT is a **M**odule system for **M**athematical **T**heories [RK13]. With the help of MMT we can represent theories, views and imports in *.mmt* files. It is easy to write these files and anyone without programming knowledge can easily extend existing ones. The objects defined in these files can then be used via an API to write down terms and transform these using definitions and views.

This furthermore allows us to easily model an extensible system for units for use within a search engine. We will come back to this later in section 4.1.

3 The Structure Of Quantity Expressions

The first step in developing a good search engine for Quantity Expressions is to take a closer look at Quantity Expressions.

When looking at Quantity Expressions, we need to distinuish between two perspectives, a notational one (i. e. the way they are written down) and a semantic one (i. e. what they mean).

3.1 Compositional Behaviour Of Quantity Expressions

Let us start by the behaviour of the notation. For this purpose let us take a look at:

$$x = 25\ \frac{\mathrm{m}}{\mathrm{s}}$$

We notice that x consists of 2 parts, a scalar (25) and a scalar-free $\frac{\mathrm{m}}{\mathrm{s}}$ one. Furthermore, the unit consists of two primitive units m and s. Since they divide one another, we can conclude that x describes a velocity.

In general, the *unit component* is not always scalar free. While m and s certainly do not contain scalars, this is no longer the case in the following example:

$$y = \frac{\mathrm{L}}{100\ \mathrm{km}}$$

We could certainly call y a *unit*. It consists of 2 sub-expressions, L and 100 km. The first one is a primitive unit and the second one a multiplication of a number and the unit km. This, first of all, leads us to define the concept of a *primitive unit* as a simple Quantity that is not a Number.

Abstracting from the examples above, we can now considder the semantics of Quantity Expressions. It makes sense to define Quantity Expressions as one of the following 6 cases:

1. A *primitive unit*, such as *Meter*.
2. The *Product* of a Quantity Expression with a scalar, such as 5Meter
3. The *Quotient* of a Quantity Expression by a (non-zero) scalar[2].
4. The Multiplication $\cdot$ which takes 2 existing quantity expression and generates a new one, for example $\cdot(100, \mathrm{m}) = 100\ \mathrm{m}$
5. The division / which again takes 2 quantity expressions and generates a new one, for example $/(\mathrm{m}, \mathrm{s}) = \frac{\mathrm{m}}{\mathrm{s}}$
6. The addition of two quantity expressions [3]

This allows us to easily generate Quantity Expressions (like the examples x and y from above) starting at the primitive units m, s, L and km.

[2] Strictly speaking, this is equivalent to the case above. For simplicity we define it anyways.

[3] This is also not strictly needed, but can become useful later on when we want to translate between different units.

3.2 Dimensions Of Quantity Expressions

Next let us briefly examine the dimensions of Quantity Expressions. The dimension of a Quantity Expression is the type of quantity it expresses. For example 5 m describes some length. According to the International System of Units [dPeM] there are seven basic dimensions:

1. length
2. mass
3. time
4. electric current
5. temperature
6. luminous intensity
7. amount of substance.

In addition to these we found that there are commonly two other dimensions: The *count* dimension (used for counting of objects) and the *none* dimension for dimensionless quantities.

Similar to the compositional behaviour of Quantity Expressions, dimensions can be multiplied and divided. In this the *count* dimension behaves like a multiplicative identity element.

Unlike Quantity Expressions however dimensions can not be multiplied with numbers. Furthermore, when multiplying to Quantity Expressions of dimensions a and b their resulting dimension is $a \cdot b$, the product of the dimensions. The same goes for division. In this regard the dimension of a quantity expression behaves like a type.

3.3 A Mathematical Theory Of Quantity Expressions

The realisation that dimensions are types of quantity expressions leads us to our first formalisation of Quantity Expressions. We first define a theory of dimensions in Figure 6 and then import it to define a theory of Quantity Expressions in Figure 7.

Figure 6 defines the 9 basic dimensions and then dimension compositon via multiplication and division. Then we move on in Figure 7 to define quantity expressions. Each quantity expression has a dimension (via the *QE* constant). This allows us to define basic units (which we will actually do in the next Figure). With the *QENMul* and *QENDiv* symbols we can multiply and divide quantity expressions by numbers (for this case we actually need to import some theory of numbers to allow us actually write this down as a Term). Then we define multiplication and division of quantity expressions in such a way that dimensions multiply and divide appropriatly.

Now we need to introduce some basic units. Let us start by just defining *Meter* in Figure 8. We can now write a term in this theory that expresses any number of meters.

3.4 Transforming Quantity Expressions From One Form Into Another

This provides a basis for a unit system. Let us furthermore define a few more units of non-metric lengths. In Figure 9 we show a few non-si units. Here we first define thoug as a quanity expression of length and then one-by-one define more units in terms of the previous one.

Dimension		
dimension	:	type
none	:	dimension
count	:	dimension
length	:	dimension
mass	:	dimension
time	:	dimension
current	:	dimension
temperature	:	dimension
luminous	:	dimension
amount	:	dimension
$\cdot$	:	dimension $\rightarrow$ dimension $\rightarrow$ dimension
$/$	:	dimension $\rightarrow$ dimension $\rightarrow$ dimension

Figure 6: A Formalisation Of The Theory Of Dimensions.

Quantity Expression		
import Dimension import Number		
QE	:	dimension $\rightarrow$ type
QENMul	:	$\forall x$: dimension.number $\rightarrow$ QE $(x) \rightarrow$ QE (x)
QENDiv	:	$\forall x$: dimension.QE $(x) \rightarrow$ number $\rightarrow$ QE (x)
QEMul	:	$\forall x$: dimension.$\forall y$: dimension.QE $(x) \rightarrow$ QE $(y) \rightarrow$ QE $(\cdot\,(x, y))$
QEAdd	:	$\forall x$: dimension.QE $(x) \rightarrow$ QE $(x) \rightarrow$ QE (x)
QEDiv	:	$\forall x$: dimension.$\forall y$: dimension.QE $(x) \rightarrow$ QE $(y) \rightarrow$ QE $(\setminus\,(x, y))$

Figure 7: The Choosen Formalisation Of The Theory Of Quantity Expressions.

We now want to relate quantity expressions with units from the *Meter* theory to units from the *Non SI Lengths* theory. It is known that $1\mathsf{Thou} = 0.0000254\mathsf{Meter}$. This can be easily expressed with a view ψ between these two theories:

$$\psi = \left\{ \ \mathsf{Thou} \mapsto \mathsf{QENMul}\,(0.0000254, \mathsf{Meter}) \ \right\}$$

Even though the view just maps the symbol *Thou* to some Term in the Meter theory, we can also use it to transform any other term from the Thou Theory. Since all units are defined in terms of the previous one, we can just expand all definitions to get an expression containing only numbers and the unit *Thou*. Then we can use the view as normal to get a Quantity Expression in the *Meter* theory.

Meter		
import QuantityExpression		
Meter	:	QE (length)

Figure 8: A Theory Defining The Primitive Unit Meter.

Non SI Lengths		
import QuantityExpression		
Thou	:	QE (length)
Foot	:	QE (length) = QENMul (1000, Thou)
Yard	:	QE (length) = QENMul (3, Foot)
Chain	:	QE (length) = QENMul (22, Yard)
Furlong	:	QE (length) = QENMul (10, Chain)
Mile	:	QE (length) = QENMul (8, Furlong)

Figure 9: A Theory Of Some Non-SI Units Of Length.

4 Making Quantity Expressions Searchable

Now that we know what quantity expressions are and how we can convert between equivalent representations, it is time to start about concrete algorithms for our search engine.

When querying this search engine with a quantity expression, we want to be able find occurrences of equivalent Quantity Expressions. For this we need several components:

1. a component that allows the user to enter quantity expressions,
2. a software to find quantity expressions within documents
3. a way to send quantity expressions from the user to the search engine
4. an algorithm to convert between different forms of a quantity expressions (or at least determine if 2 quantity expressions are equivalent) and finally
5. a unit system that is flexible enough to handle different kinds of units.

4.1 An Idea For An Extensible System

Let us start with point (5), a unit system that is flexible enough to handle all different of units. In Figure 10 you can find a small part of the unit graph we developed. At first, in the gray area, we define a basic version of our system as defscribe in section 3.3. Next we continue in the top left corner to define all the basic SI units. For the following we only look at the theories descrobing length and area units (however there are also some for all of the other dimensions).

We start with defining the non-SI unit *Thou* in the *Imperial Lengths A1*. In order to be able to convert between these units we apply what we learned in section 3.4 and create a view that maps this unit back to standard SI units. We then expand into *Imperial Lengths A2* and define a few more units incrementally based on *Thou* (we use an import to be able to do this).

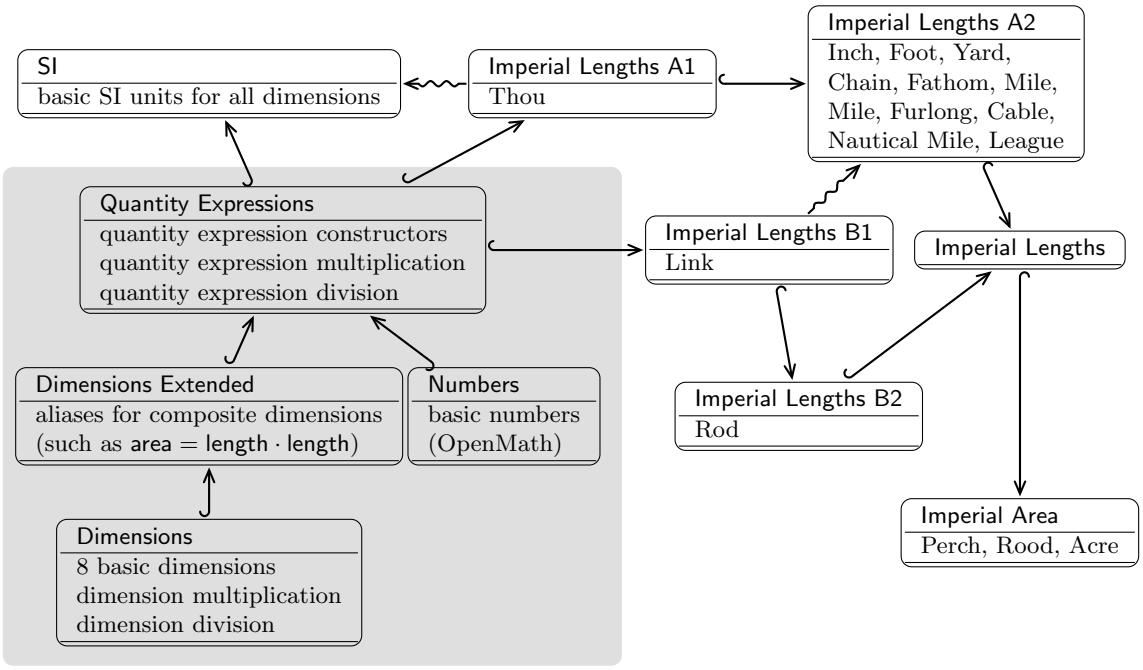


Figure 10: A Small Part Of The Graph Containing Unit Theories. Imports Are Represented As Solid Edges And Views As Squiged Edges. The Gray Area Contains Basic Theories That Are Used To Define Quantity Expressions.

Next, we define a new unit *Link* in the *Imperial Lengths B1* theory and define the unit *Rod* in *Imperial Lengths B2*. We then also make a view from *Imperial Lengths B1* to *Imperial Lengths A2*. Since *Imperial Lengths B2* imports *Imperial Lengths B1* this allows us to easily transform both of the new units back to SI if needed.

Next we want to define some units with the dimension Area. For this we first collect all known lengths and make them available in a *Imperial Lengths* theory via imports. Then we import this theory into a new *Imperial Area* theory where we can then define *Perch*, *Rod* and *Acre*.

But why do we define so many different theories? Why do we not simply define all the units of length in a single theory? Obviously, this one length theory would be very big. Furthermore, it would require anyone who wants to add more units to the system to add to this one theory, which can cause conflicts if there are 2 units with the same name (such as *Mile* and *(nautical) Mile*).

Defining units in the way we do it allows us to incrementally add new units to the system. If we have a view back to already known theories, it is easy to compare them back to the existing units.

4.2 Normalisation Of Quantity Expressions

To build a search engine however, it is inefficient to try to directly compare a search query with known quantity expressions. One way to compare quantity expressions is to normalise them to a suitable normal form. The normal form we came up with works in 2 steps.

We first turn a Term representing a Quantity Expression into a Term over the theory

containing only standard units. In general we can choose any theory we want for this. We choose the *SI* theory for this so that the normal forms can be understood easily.

In order to transform a term to standard units, we just expand each unit (represented by constants) into a Term over the standard theory. This is achieved by translating it along a path in the theory graph. A path in the theory graph can consist out of 2 types of edges, Views and Imports. For views we can obviously use the mappings to translate the unit from the source to the target theory of the respective view. For imports however, it is not that simply. We can translate terms along them via definition expansion. This takes us to the imported theory from the theory that imports. We can use these edges as described and use standard graph theory algorithms to find a path for a unit to be recursively translated upon.

In the second step, we seperate the scalar from the units. For this, we first compute the scalar value of a quantity expression and then cancel out units which occur both in numerator and denominator of the quantity expression. This step is much easier then the first one, as we no longer have to use the theory graph. For this reason, we compute the normalisation of each unit only once by caching the result. Then we no longer need to look at the graph for our normalisation procedure at all.

Let us illustrate the process of normalisation a bit more. We will try to normalise the quantity expression

$$q = 42\frac{\text{Furlong}}{\text{Fortnight}}$$

First, we need to normalise *Furlong* (a quantity expression of length) to *Meter*. For this we first do definitional expansion:

$$\text{Furlong} = 10\ \text{Chain} = 10\,(22\ \text{Yard}) = \cdots = 10\,(22\,(3\,(12\,(1000\ \text{Thou}))))$$

Next, we need to use a view to turn this into meters:

$$10\,(22\,(3\,(1000\ \text{Thou}))) = 10\,(22\,(3\,(12\,(1000\,(0.0000254\ \text{Meter})))))$$

Similary, we use definitional expansion to find that:

$$\text{Fortnight} = (2\,(7\,(24\,(60\,(60\ \text{Second})))))$$

Substituting into q gives us the normal form of the quantity expression after the first step:

$$42\frac{10\,(22\,(3\,(1000\,(0.0000254\ \text{Meter}))))}{2\,(7\,(24\,(60\,(60\ \text{Second}))))}$$

Next, we want to extract the scalar component and compute it:

$$42\frac{10\,(22\,(3\,(12\,(1000\,(0.0000254))))))}{2\,(7\,(24\,(60\,(60))))} = 0.006985$$

We also extract the unit component:

$$\frac{\text{Meter}}{\text{Second}}$$

Finally we compose these again to get the normal form:

$$0.006985\ \frac{\text{Meter}}{\text{Second}}$$

4.3 Serialising Quantity Expressions To XML

Now that we have a normal form of units, we want to be able to exchange quantity expressions between the user, the search engine and the harvesting component. The W3C has written a note on how to handle Units in MathML [Gro]. MathML is an XML serialisiation of Mathematical expression. We orient ourselves on the format. For this serialisation we start with a term over some unit theory. The serialisation of the term

$$0.006985 \ \frac{\text{Meter}}{\text{Second}}$$

can be found in Figure 11. Let us take a closer look to understand this serialisation. We

```
<apply>
  <times />
  <cn type='float'>0.006985</cn>
  <apply>
    <divide />
    <csymbol cd='SIBase'>Meter</csymbol>
    <csymbol cd='SIBase'>Second</csymbol>
  </apply>
</apply>
```

Figure 11: Serialisation Of A Simple Quantity Expression.

translate each component of the Term individually and then assemble them together.

We translate the unit (constant) *Meter* to `<csymbol cd='SIBase'>Meter</csymbol>`. This XML Tag expresses that we are talking about the symbol *Meter*. The cd attribute stands for content directory. In our case this is the name of the theory were the symbol was declared.

We translate multiplication and division by using `<apply><times />...</apply>` and `<apply><divide />...</apply>` respectively. We can use quantity expressions directly here. If we want to multiply with or divide by numbers, we use `<cn type='float'>...</cn>` instead of the `<csymbol>` tags.

This format allows us to easily exchange quantity expressions. It is also easy to send quantity expressions from the user to the backend (the core component of the search engine).

4.4 Finding Units Inside Documents

Apart from the frontend (which we will describe in detail in Section 5), there is only one major component left: Finding and matching quantity expressions inside documents. As we have outlined in the previous sections, we can easily compare them with the help of normalisation. If we have a list of quantity expressions and where they occur, we can use the following algorithm to easily find them:

1. First find quantity expressions inside documents and store them in a list along with their origin.

2. Normalise each of them as described before.

3. Store the normalised forms in an efficient index along with their original versions and origins.

4. When a query is sent, normalise the query.

5. Next look in the index for the normalised query.

6. Then deliver the original versions and their origins from the index.

This algorithm allows us to easily and efficiently find quantity expression in documents after we have built up an index. In order to build up this index we need a list of quantity expressions and occurrences inside documents. A software that fullfills this task is called a Spotter. In our case the spotter is developed by Stiv Sherko in a seperate effort [She15].

5 The Implementation: The Search Process

Now that we have discussed all the theoretical components of the search engine, it is time to move on to the actual implementation. In this section we describe in detail how a search query is processed and how it is presented to the end user.

5.1 Entering A Query And Getting Results

As to give an overview of what exactly happens during a typical search, we will show what happens during a normal search process. From the user perspective, everything happens inside a web browser. When the user visits the demo page [Wie], they are presented with a frontend as shown in Figure 12.

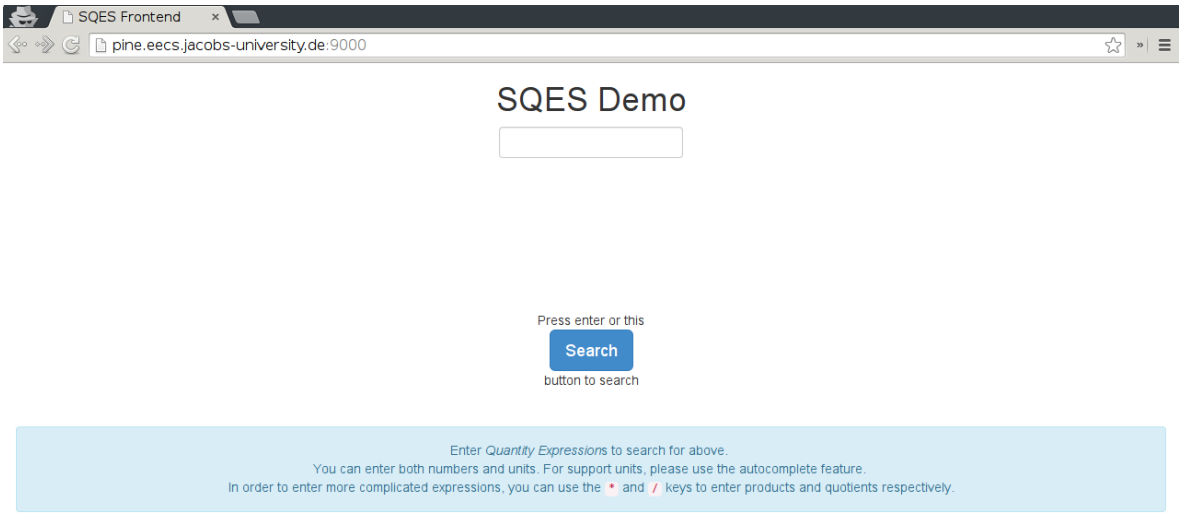


Figure 12: The main page of the frontend.

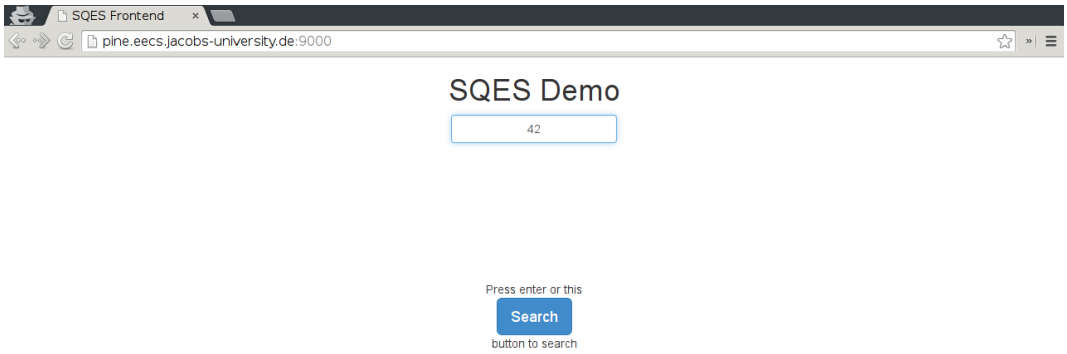
Because the frontend is running inside a Webbrowser it is written using HTML, CSS and JavaScript. Furthermore it uses the frameworks jQuery [jQu15] and Bootstrap [Twi15]. As seen in the figure, the main components are a text field to enter a quantity expression as well as a search button.

With the text field it is easily possible to enter composed quantity expression.

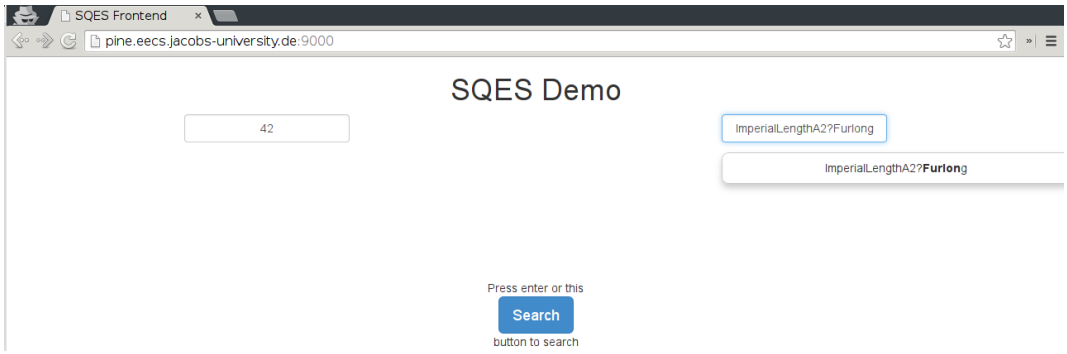
When searching for a certain unit it is neccessary to tell the backend which *unit theory* this unit comes from. In order to make this easier, the GUI has an autocompletion feature. It is only required to enter the name of the unit itself and the possible theories this unit can come from will be suggested automatically.

Apart from entering a simple unit it is possible for the user to compose quantity expressions by multiplication and division. If the keys "*" and "/" are pressed, the text field splits into two different fields. In each of them another quantity expression can be entered.

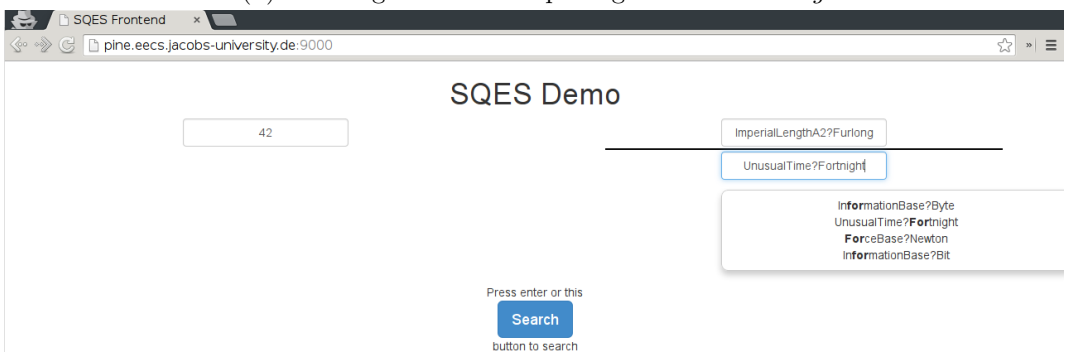
In order to facilitate entering division and multiplication by numbers, it is also possible to enter numbers into these fields. With this input unit it is also possible to enter all the quantity expressions as described by the formalism above. In Figure 13 the process of entering the query $42\frac{\text{Furlong}}{\text{Fortnight}}$ is shown.



(a) Entering a number



(b) Entering and autocompleting the unit *Furlong*



(c) Entering the final component.

Figure 13: Process of entering the quantity expression $42\,\frac{\text{Furlong}}{\text{Fortnight}}$ into the frontend.

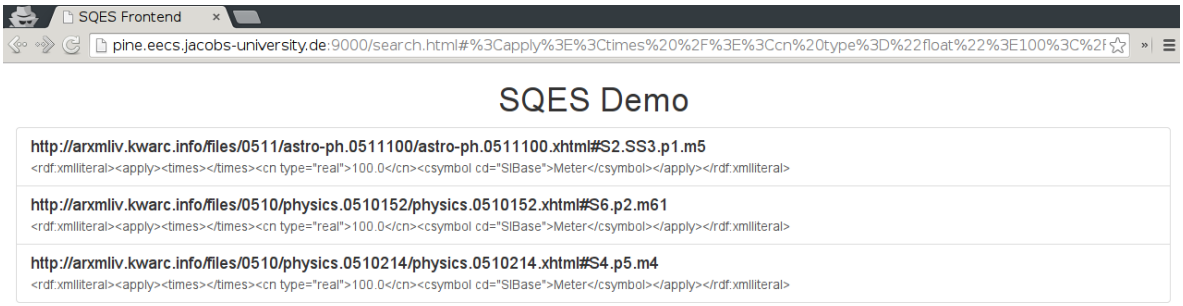


Figure 14: The result page when searching for 100 Meters.

After the quantity expression has been entered, the search can be started by pressing the *Search* button or hitting the *Enter* key. Once the results have been received from the backend they are displayed to the user. The result page lists the found Quantity Expressions in their original forms[4] and links to the locations where they were found inside the actual document in the spotter. An example can be found in Figure 14.

5.2 Preparing The Backend

After the query has been entered into the frontend, it is serialised to XML as shown in Section 4.3. This XML is then sent to the backend.

The backend is written in Scala and exposes its functionality via HTTP. Before the backend is ready to receive queries, it has to normalise the quantity expressions found by the Spotter. These queries are also provided in XML form. This XML is parsed and deserialised into an internal data structure.

Next we want to normalise these quantity expressions using the algorithm described in Section 4.2. In order to achieve this the backend first parses the MMT theories and views and builds up a theory graph. Next, it normalises all units found inside the unit graph and caches these normal forms in memory.

We can then use the forms to quickly normalise all the quantity expression found inside the harvests. The normalised forms are then used to store them in an appropriate structure on disk. In this structure all equivalent quantity expressions are stored inside the same folder. This allows very quick access to all quantity expressions in preparation for processing a query.

5.3 Processing A Typical Query

Upon receiving the query, the backend deserialises and normalises the XML as it did for all the harvests. Since the normal forms of each *basic unit* are still cached inside memory, this process is very quick. Because the normal forms are already cached on disk, the process of finding equivalent quantity expressions can be efficiently done. As described above, all equivalent quantity expressions are in the same folder. The backend now just reads all the

[4]Currently they are only displayed as serialised XML, this is discussed in more detail in section 6.6.

quantity expressions stored inside the folder the query falls in. This delivers all equivalent quantity expressions and can be sent back the frontend.

Because all the normal forms are simply stored on disk inside a folder it is easily possible to add more quantity expressions to an existing harvest. This is even possible during query timee. For this reason the implementation of the search engine has two seperate modes, a *harvest* mode and a *query* mode which allow preprocessing of harvests and actual querying. Since the first mode does not require network interaction, only the query mode has an actual HTTP interface.

6 Current Limits And Future work

The current implementation provides a proof that the choosen approach indeed works. There are several problems with it. In the following we discuss several improvements for existing features as well as missing features.

6.1 Integration With MathWebSearch: Ranged Queries and Query Variables

The MathWebSearch system (MWS) is an existing semantics-aware system to search (LaTeX[5]) documents for mathematical formulae [HKP14]. Similar to the system we built it is also semantics-aware. It abstracts away from variable names and allows *variables* insides queries such as $x + \sqrt{x}$. For this query MWS would deliver forumlae of the form as above where x has been substituted with any sub-formula. In addition MWS supports ranged queries, i. e. queries where we search for a variable that is contained within a certain range.

Integrating our system with MathWebSearch could potentially allow some of these features to be used when searching for Quantity Expressions as well. Additionally the system of harvesting, indexing and searching formulae is much more elaborate and well tested than our system. An enhancement would be to just expose a normalisation API to MathWebSearch and then allow it to handle the remainder of the search process.

Support for *query variables* and *ranged queries* would not come for free however. A suitable normal form would have to be developed. When we currently have an expression, all non-standard units are removed. Thus if we just want to answer queries by just comparing normal forms, those units have to be removed from queries as well. However if we have a query variable we do not know if this is just a scalar variable (in which we do not have to remove it) or if it is a unit (in which case we would have to keep it and restrict the scalars in some way.). This could potentially be solved by enforcing types for each query variable. This argument shows that it is not easy to just extend our system with these features.

6.2 Absolute Vs Relative Units

The current unit system is easily extensible and can express a lot of different units easily.

However there also is a major limitation that stops it from being able to translate between *Kelvin* and *Celsius* as Units of Temperature. Since *Views* are maps *Constants* of Theories and not between *Terms* over certain theories, the system can only be used to translate between units with linear maps.

In [SD08] Davenport distinguishes between two types of units: Absolute and relative[6]. When adding two Quantity Expressions with the same unit we can usually just factor out the unit and then add up the scalars, in other words the set of these QEs forms an (abelian) monoid. Davenport discovered that this is not always the case. We have $2\text{ Kelvin} + 3\text{ Kelvin} = 5\text{ Kelvin}$ but $2\text{ Celsius} + 3\text{ Celsius} \neq 5\text{ Celsius}$. *Celsius* for example is not defined via a single quantity expression but is just $\frac{1}{100}$ of the difference between two other Quantity Expressions. Hence it is not an absolutely defined unit, but rather a *relative* one.

[5] Technically, MWS itself can only search XHTML documents. However with the help of LaTeXml [Mil] it also handles LaTeX documents.

[6] In the paper these are simply called non-absolute units.

Our implementation can currently only handle *absolute units*. One possible solution to this problem is to choose a different formalisation of units. At this point a unit is only some Quantity Expression. If we consider a unit as a map from *Real numbers* to *Quantities of a certain dimension* we can easily translate between unit theories with the existing views by using functions:

$$\text{Celsius}(x) = \text{Kelvin}(x - 273.5)$$

6.3 Handling Of SI Prefixes

So far we have treated units like *Kilometer* and *Meter* as two different units. *Kilometer* is actually just the SI Prefix *Kilo* added to the unit *Meter* and it is thus clear that Kilometer = 1000 Meter. We could improve the system by adding support for these. Since something like *Kilo-Kilometer* makes no sense, a formalisation of them will require a distinction between prefix-free units and prefixed units.

Prefixes have also been treated by Davenport in [SD08]. He gives two types of units: *units* and *prefixed units*. He treats resolutions of prefixes as maps:

$$\text{prefix} \times \text{unit} \to \text{prefixed unit}$$

Hence in order to handle SI Prefixes properly, we will need to add a new type of units, the prefixed unit. Since *Kilogram* is the standard SI unit for mass, this will likely result in normalisation to *Gram* instead.

6.4 MMT And Type Equalities

In Section 3 we defined our structure of dimensions and quantity expressions. However at no point we have defined multiplication to be associative or commutative. Hence MMT is not aware of this. Furthermore we did not explicitly define cancellation of dimensions and basic units anywhere. Because we did not define it, MMT is not aware of it.

When MMT type checks views this can cause errors. In the current implementation we have a workaround for this by explicitly giving MMT the result type of a multiplication and division (as an additional argumemt to the multiplication function.) This makes MMT terms significantly longer, in particular when multiple multiplication or divisions are involved. This problem can obviously be solved by adding these rules to our formalisation. That alone is not sufficient, because MMT does not implement any kind of type checking of this form at the moment.

6.5 The Current State Of The Spotter

Currently the spotter extracts simple Quantity Expressions of the form $\mathit{Scalar} \cdot \mathit{Unit}$ from XHTML documents. This is done in two steps, first the presentational markup is extracted and then the semantics are infered from this representation. The markup extracter currently achieves an f-measure of 0.4465 and the semantics extracter achieves an f-measure of0.6546. The spotter is described in detail in [She15].

In order for the search engine to function properly, it is neccessary to have a proper spotter. For early testing the current implementation is sufficient, but for proper development it will have to be improved. Obviously we want to be able to search for velocities as well. With the current implementation this is not yet possible. Because the spotter only provides harvests to

the search engine and is not directly integrated however, it can be developed independently of the rest of the system.

6.6 Improvement To The Frontend

The current frontend, especially the result page, needs some improvement.

Currently the result page just shows a list of links as well as the XML representation of the found quantity expressions. Instead it should show the human-readable representation of the Quantity Expression as well. Such an improvement could be delivered by the spotter. As described above, it extracts both markup and semantics of Quantity Expressions. Instead of only returning the latter it could return both, which could then easily be shown to the frontend.

Furthermore, the text field used to enter Quantity Expressions could use an improvement. Currently this is rendered manually. It makes sense to take advantage of presentation MathML or similar technologies here. Presentation MathML is an XML-based language that can be used to describe the layout of math formulae.

6.7 Conclusion

We wanted to build a search engine for Quantity Expressions that unifies both the presentational and semantic approach. As shown by the problems outlined above, our work is not yet finished. Several components still require improvements to be practical. Furthermore, the system should be subjected to some kind of evaluation by end users to measure the actual usability of it. Nonetheless, the prototype implementation we have designed demonstrates that this is indeed a viable approach. Our system, if expanded and improved properly, has potential to solve the problem of different units entirely.

References

[Boa] Mishap Investigation Board. Mars Climate Orbiter, Phase I Report. ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf. 10 November 1999.

[dPeM] Bureau International des Poids et Mesures. The International System of Units (SI). http://www.bipm.org/utils/common/pdf/si_brochure_8_en.pdf. 8th edition, 2006.

[Gro] W3C Working Group. Units in MathML. http://www.w3.org/TR/2003/NOTE-mathml-units-20031110/. Note 10 November 2003.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

[HKP14] Radu Hambasan, Michael Kohlhase, and Corneliu Prodescu. MathWebSearch at NTCIR-11. In Noriko Kando and Hideo Joho andKazuaki Kishida, editors, *NTCIR 11 Conference*, pages 114–119, Tokyo, Japan, 2014. NII, Tokyo.

[jQu15] jQuery Foundation. jQuery. https://jquery.com/, 2015.

[Mil] Bruce Miller. `LaTeXML`: A LaTeX to XML converter.

[RK13] Florian Rabe and Michael Kohlhase. A scalable module system. *Information & Computation*, 0(230):1–54, 2013.

[SD08] Jonathan Stratford and James H. Davenport. Unit knowledge management. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, *Intelligent Computer Mathematics*, number 5144 in LNAI, pages 382–397. Springer Verlag, 2008.

[She15] Stiv Sherko. Extracting quantity and unit semantics from technical documents. Bachelor thesis in computer science, Jacobs University - School of Engineering and Science, 2015.

[Twi15] Twitter, Inc. Bootstrap. http://getbootstrap.com/, 2015.

[Wie] Tom Wiesing. SQES Frontend. http://pine.eecs.jacobs-university.de:9000/.