Units of Measure for Hybrid Programs

15-424 term paper

Vincent Huang (vincenth@andrew.cmu.edu)

May 2, 2016

Abstract

In this paper, we discuss our augmentation of dℒ with a monomorphic unit-of-measure type system with a top type, and our work done to modify the theorem prover KeYmaera X to support this type system. We present examples of how this system can be useful.

1 Introduction

1.1 dℒ

Differential dynamic logic (dℒ) is a logic for specifying and verifying hybrid systems. A hybrid system is a system that exhibits both continuous and discrete dynamic behaviour; some examples are self-driving cars, aeroplane autopilots, and robotic surgeons.

dℒ can be used as a logic to specify correctness properties for hybrid systems written down operationally as hybrid programs. A proof sequent calculus exists for dℒ, and this can be used to verify the aforementioned correctness properties. However, performing and checking large sequent calculus proofs by hand can be daunting.

1.2 KeYmaera X

KeYmaera X is a theorem prover for hybrid systems based on dℒ. One can see from the examples of hybrid systems given earlier that it is important that we are able to trust such systems, and what better way to develop trust in something than to *prove it correct*?

With a theorem prover like KeYmaera X, the task of performing large proofs is made easier, and checking is automatically done by a computer. Even so, with our limited resources and capabilities, the systems we can prove, and most importantly, *what* we can prove about the systems we prove are also limited. Almost always, we make simple, idealised models of systems of interest, and verify those — anything beyond that is both

far too difficult to model, and, even if that were doable, would be far too difficult to prove. Better an idealised proof than no proof, of course.

However, this opens work on hybrid systems verification up to various missteps. One example is modelling a system in such a way that the model is vacuously valid because it has no possible executions. Another example, although not having come up in the context of hybrid systems verification proper to the best of my knowledge, is unit errors. One might imagine that if the Mars Climate Orbiter team were to have tried to verify their spacecraft, they might still have made the fatal error of neglecting to take into account foot/meter conversions.

1.3 Contribution

In this work, we developed a monomorphic unit-of-measure type system with a top type for $\mathsf{d}\mathcal{L}$, and built a working implementation in the current version of KeYmaera X. Our version of KeYmaera X is entirely backwards-compatible with the existing version — the presence of $\top$ in the type system means that we can assign any unannotated variables to $\top$ and hence programs written without explicit units still typecheck.

1.4 Related work

The theory behind type systems for units of measure has been understood for many years now, since at least 1994 [2], and implementations based off this work have made their way into at least one mainstream programming language, F# [3]. There are also libraries providing such features for a variety of languages — for instance, Boost.Units for C++, units for Haskell, Squants for Scala, and dimensioned for Rust.

Anyone who has done even elementary physics also knows the usefulness of dimensional analysis — by staring at the quantities available, one can either eyeball or use a more sophisticated method like the Buckingham Pi Theorem [1] to figure out the appropriate way to combine them (up to dimensionless factors) to get a sensible answer. Not only is dimensional analysis useful in helping people figure out what to do, it is also extremely useful in catching errors — it can prevent silly mistakes like trying to add a quantity with dimension L to a quantity with dimension L^2. Catching errors in reasoning at this stage can prevent both the needless agony of trying to prove something that might not work out because of incommensurate quantities being mashed together, or the horror of discovering that a model, while verified, was actually incorrect.

2 Units of measure in $\mathsf{d}\mathcal{L}$

2.1 Unit types (not the unit type)

Being monomorphic, the type system introduced in this work does not make use of any sophisticated concepts from type theory. For a reader less familiar with ideas from type

theory, perhaps a useful analogy is with (a safe version of) `typedef` in C, also known as the *newtype pattern* in a number of other programming languages.

`typedef` allows one to declare synonyms for types. For instance, `typedef int length_t` declares `length_t` as a synonym for `int` that can be used interchangeably with it from that point on. The primary use of this is to remind oneself of one's semantic intention when using a value of that type (so in this instance, using `length_t` would be an indication to oneself that one meant this value to represent a length instead of any generic integer). However, C does not prevent one from, say, using an `int` where the type signature indicates that a function expects a `length_t` instead. This is similar to reasoning about units in one's head, where one makes arguments to convince oneself of correctness, but there is nothing to enforce the reasoning.

The newtype pattern is like a safe version of `typedef`, in that one can declare "synonyms" for types, but said "synonyms" can no longer be used interchangeably with the original type (hence the quotes). So if one defined `newtype int length_t`, appropriating a C-ish syntax, one would not be able to pass a function expecting a `length_t` an `int`. Nor, if one also defined `newtype int temperature_t`, would one be able to use `temperature_t` interchangeably with `length_t`.

One can again see an analogy here with units of measure. This is like having to explicitly tell KeYmaera X about the units of all variables in a hybrid program. Then, just like the compiler of a newtype language would detect at compile-time misuse of type "synonyms", KeYmaera X will detect during its parsing/static analysis phase that a unit error has occurred if a user does something untoward with incommensurate quantities. The analogy is especially apt given that dℒ (and hence KeYmaera X) has only 2 interesting types[1] in the core language, Real and Bool.

2.2 Adding units to dℒ

Given what we said in section 2.1, and given that we would like to maintain compatibility with hybrid programs/dℒ proofs written without a unit type system in mind, it seems like we want to take an extrinsic view of unit of measure types. That is, we would like every formula that is a valid formula of dℒ in the core language to still be valid in the presence of types. We achieve this by allowing for *unit annotations* on variables, and automatically inferring type $\top$ for any unannotated variables/numeric constants. The type $\top$ (pronounced "top", or in our specific case also "any unit") is simply a type that matches against any other type. Since our type system is simple, our typechecking rules are also simple (we use the convention that τ is a type, x is a variable, r is a real number, n is a natural number, and $\oplus$ is a generic binary operation). A representative sample of these rules can be found in figure 1.

We use a judgement *ok* to capture the notion of a dℒ formula being acceptable within the scope of the unit of measure type system. Taking the convention that t stands for a term, P stands for a program, and f stands for a formula, we give some

[1]Known as "sorts" in the KeYmaera X codebase and dℒ literature.

$$\text{Var-T}\ \frac{\Upsilon(x) = \tau}{\Upsilon \vdash x : \tau} \qquad \text{Real-T}\ \frac{}{\vdash r : \top} \qquad \text{Plus-T}\ \frac{\Upsilon \vdash x_1 : \tau \qquad \Upsilon \vdash x_2 : \tau}{\Upsilon \vdash x_1 + x_2 : \tau}$$

$$\text{Times-T}\ \frac{\Upsilon \vdash x_1 : \tau_1 \qquad \Upsilon \vdash x_2 : \tau_2}{\Upsilon \vdash x_1 \times x_2 : \tau_1 \cdot \tau_2} \qquad \text{Div-T}\ \frac{\Upsilon \vdash x_1 : \tau_1 \qquad \Upsilon \vdash x_2 : \tau_2}{\Upsilon \vdash x_1 \div x_2 : \tau_1 \cdot \tau_2^{-1}}$$

$$\text{Pow-T}\ \frac{\Upsilon \vdash x : \tau}{\Upsilon \vdash x^n : \tau^n} \qquad \text{Prime-T}\ \frac{\Upsilon \vdash x : \tau}{\Upsilon \vdash x' : \tau \cdot \mathrm{s}^{-1}}$$

$$\frac{\Upsilon \vdash x_1 : \top \qquad \Upsilon \vdash x_2 : \tau}{\Upsilon \vdash x_1 \oplus x_2 : \top}$$

Figure 1: Representative examples of rules for typing $\mathsf{d}\mathcal{L}$ terms

$$=\text{-ok}\ \frac{\Upsilon \vdash t_1 : \tau \qquad \Upsilon \vdash t_2 : \tau}{\Upsilon \vdash t_1 = t_2\ ok}$$

$$\neg\text{-ok}\ \frac{\Upsilon \vdash f\ ok}{\Upsilon \vdash \neg f\ ok}$$

$$=\text{-ok}\ \frac{\Upsilon \vdash f_1\ ok \qquad \Upsilon \vdash f_2\ ok}{\Upsilon \vdash f_1 \wedge f_2\ ok}$$

$$\text{Prog-ok}\ \frac{\Upsilon \vdash P\ runs \qquad \Upsilon \vdash f\ ok}{\Upsilon \vdash [P]f\ ok}$$

Figure 2: Representative examples of rules for validating $\mathsf{d}\mathcal{L}$ formulas

representative examples of rules for formula validity in figure 2.

We use a judgement *runs* to capture the notion of program validity. Taking the convention that D stands for an ordinary differential equation, we give some representative examples of rules for program validity in figure 3.

3 Units of measure in KeYmaera X

We implemented unit of measure types and a unit-checker in KeYmaera X in accordance with the rules given in section 2.2. Since the KeYmaera X core is soundness-critical and our unit type system is an extrinsic layer over it, we tried as far as possible to avoid modifying code in the `keymaerax/core` directory.

$$:=\text{-runs}\ \frac{\Upsilon \vdash x : \tau \qquad \Upsilon \vdash t : \tau}{\Upsilon \vdash x := t\ \mathit{runs}}$$

$$'\ :=\text{-runs}\ \frac{\Upsilon \vdash x : \tau \qquad \Upsilon \vdash t : \tau \cdot \mathrm{s}^{-1}}{\Upsilon \vdash x' := t\ \mathit{runs}}$$

$$;\text{-runs}\ \frac{\Upsilon \vdash P_1\ \mathit{runs} \qquad \Upsilon \vdash P_2\ \mathit{runs}}{\Upsilon \vdash P_1;\ P_2\ \mathit{runs}}$$

$$\text{ODE-runs}\ \frac{\Upsilon \vdash x : \tau \qquad \Upsilon \vdash t : \tau \cdot \mathrm{s}^{-1}}{\Upsilon \vdash \{\ x' = t\ \}\ \mathit{runs}}$$

$$\text{ODE-system-runs}\ \frac{\Upsilon \vdash D\ \mathit{runs} \qquad \Upsilon \vdash f\ \mathit{ok}}{\Upsilon \vdash \{\ D\ \&\ f\ \}\ \mathit{runs}}$$

Figure 3: Representative examples of rules for validating d$\mathcal{L}$ programs

3.1 Modifications made to the core

As mentioned above, we tried to make as few modifications to the core as possible. In fact, the only modifications to the core were in the file `Expression.scala`, which contains datatype[2] definitions for abstract syntax of hybrid programs accepted by KeYmaera X. The changes made were

- Addition of a new sort, `UnitOfMeasure` ("sort" is the KeYmaera X codebase's term for the types in the core language, the important ones being Bool and Real).
- Addition of a new datatype, `MeasureUnit`, representing the unit of measure types. This datatype has constructors
  - `UnitUnit of String`, which represents a simple unit of measure like 'm' or 's' (the String stores the identifier provided by the user)
  - `ProductUnit of Map[String, Int]`, which represents a unit of measure that is a product of multiple units of measure, e.g. $\mathrm{m} \cdot \mathrm{kg}^2$, which would be stored as `Map(m->1, kg->2)`.
  - `AnyUnit`, which represents the type $\top$.
  - `NoUnit`, which is the type of dimensionless quantities.

3.2 Modifications made to the lexer/parser

Before we talk about modifications to the lexer/parser, it behooves us to discuss our choice of syntax for unit-type-using hybrid programs.

[2]Or whatever the equivalent Scala terminology is.

```
ProgramUnits.
 U m.
 U n.
End.
```

Figure 4: An example of a `ProgramUnits` section

```
ProgramVariables.
 R trackr.
 R xx.
 R roguer.
 R wtr.
 R t.
 R T.
 R v.
End.
```

Figure 5: An example of a `ProgramVariables` section without unit annotations

A KeYmaera X program file currently begins with a section `ProgramVariables` (and an optional section `Functions`) that specifies the variables that the program will use and their types (Real or Bool). We decided to add a section before this, aptly titled `ProgramUnits`, that specifies the units that the program will use. For instance, the code in figure 4 specifies that the program that follows may make use of 2 units, 'm' and 'n', as well as an implicitly present third unit 's'. The unit 's', meant to represent seconds, a unit of time, is necessary in every program and is therefore always implicitly included[3] (it is not an error, although unnecessary, if the programmer includes a `U s.` declaration).

We also made changes to the `ProgramVariables` section. Previously, a `ProgramVariables` section might look something like figure 5. We decided to support unit annotations using the fairly conventional syntax of `variable:type`, or, in this case, `variable:unit`. Unannotated variables are treated by our KeYmaera X fork as having type `AnyUnit`, which, as given by the rules in section 2.2, typechecks against any other unit, so this scheme is perfectly compatible with pre-unit KeYmaera X programs.

An example of a unit-annotated `ProgramVariables` section can be seen in figure 6. Assuming the units 'm' and 'kg' were declared earlier, this program will be able to use variables `trackr` with unit m, `xx` with unit m^2, `v` with unit m s^{-1} et cetera, as well as variable `t` with unit `AnyUnit`.

We modified `KeYmaeraXLexer.scala` and `KeYmaeraXProblemParser.scala` in the `parser` directory to add this support. The lexer now lexes `COLON` as a token, and supports a `PROGRAM_UNITS_BLOCK`.

The parser now returns a `KeYmaeraXProblemParserResult`, which contains a `variableMap`

[3]This actually exposes a weakness of our approach, which will be addressed in section 5.

```
ProgramVariables.
 R trackr : m.
 R xx: m^2.
 R roguer : m.
 R wtr : kg.
 R t.
 R T : s.
 R v : m/s.
End.
```

Figure 6: An example of a `ProgramVariables` section with unit annotations

and a `unitMap`. The `variableMap` is a map from variable identifiers to their sorts, which the parser used to return, but now with the addition of a `MeasureUnit`. The `unitMap` is a map from unit identifiers to a `Map[MeasureUnit, Term]` where the `Term` should be either a `Number` or a binary operation on ultimately numerical subterms. Currently, the `unitMap` is used essentially as a set — an empty `Map` is always stored as the corresponding value. But we chose to use a map rather than a set to support future improvements that we intend to do (see section 5).

3.3 The unit-checker

The bulk of the unit-checker is in a new file in the `parser` directory, `UnitChecker.scala`. In this file, we define a sensible equality on the `MeasureUnit` datatype (for instance, `UnitUnit(s)` is equal to `ProductUnit(m)` where `m` maps `s` to 1 and everything else to 0). We define a number of mutually recursive functions that implement the rules from section 2.2.

4 Usefulness of units in KeYmaera X programs

Hopefully the usefulness of units and dimensional analysis in any system are not lost on the reader at this point, especially after section 1.4, but we will provide a couple of concrete examples anyway.

4.1 The weirdly-accelerating car

Consider the hybrid program given in figure 7, meant to model a car moving in a straight line that has to stop before a stop sign. Notice that in the ODE, instead of `x' = v` as the programmer probably intended, it has `x' = x`. This is clearly very wrong and the model does not in any way model a car moving in a straight line, but it passes muster in KeYmaera X 4.2b1, whether the resulting model turns out to be provable or

```
ProgramVariables.
 R x.
 R S.
End.
Problem.
 [{ x' = x }]x <= S
End.
```

Figure 7: An incorrect model of a 1D car

```
ProgramUnits.
 U m.
End.
ProgramVariables.
 R x : m.
 R S : m.
End.
Problem.
 [{ x' = x }]x <= S
End.
```

Figure 8: An incorrect model of a 1D car with unit annotations

not[4]. Assuming it is not provable, the programmer might be caused countless hours of anguish as they try to prove it, and then grief as they eventually realise their mistake.

Now say the programmer has decided to put unit annotations on their hybrid program, as in figure 8. Now life is good, as KeYmaera X will tell them

```
Unit analysis error
x'@DifferentialSymbol had unit Units[,m**1,s**-1
but x@Variable had unit UnitUnit m
```

allowing them to look for a problem in their model before expending huge amounts of futile effort in the search for a proof!

4.2 The neat new distance metric

In lab 3 of 15-424, we had to model a robot travelling around a circular track, verifying that our controller would have it stop before it ran into an obstacle on said track. It is possible that a student in the class submitted a model that hinged on this test

```
?((ox-x)^2 + (oy-y)^2 - v*T - (a*T^2)/2 >= -((v + a*T)^2)/(2*B));
```

which ends up subtracting a quantity of distance from a quantity of distance squared. This is problematic. The problem is isolated in figure 9.

[4]We have not tried to prove it.

```
ProgramUnits.
 U m.
End.
ProgramVariables.
 R ox : m.
 R x : m.
 R oy : m.
 R y : m.
 R a : m/(s*s).
 R T : s.
 R v : m/s.
 R A : m/(s*s).
 R B : m/(s*s).
End.
Problem.
 [?((ox-x)^2 + (oy-y)^2 - v*T - (a*T^2)/2 >= -((v + a*T)^2)/(2*B));]x=x
End.
```

Figure 9: Incorrectly treating distance and distance squared as commensurate

Fortunately, KeYmaera X catches this problem for us!

```
Unit analysis error
unit error in term on LHS of <=
 Problematic term is (ox-x)^2+(oy-y)^2-v*T-a*T^2/2
```

5 Future work

Having units in dℒ and KeYmaera X is fresh and exciting, and our work leaves some obvious paths to follow from here. In this section, we outline further contributions that we believe can be made in the area of unit of measure types in KeYmaera X, or avenues inspired by or made possible because of this work.

5.1 Short-term

In the short term, we would like to rectify some obvious shortcomings, including:

- The bugs in our implementation. For instance, the example in section 4.2 was intentionally pared down because the full example exposed a number of bugs in our fork of KeYmaera X.
- Type annotations for function symbols. Currently, arguments to functions and function codomains are treated as all being of type `AnyUnit`, which is suboptimal, especially since we have the machinery to support annotation and unit-checking

for them already in place. Given more time, we would support units for functions as well.

- Unit conversions. As mentioned earlier (when we talked about the implicit 's' unit), one of the shortcomings of our current work is that it does not clearly distinguish the concept of *dimension* and *unit*. As an example, there is a single *dimension* of length, but multiple *units* of length (e.g. meters and feet). We believe a useful feature for KeYmaera X would be it taking a series of specifications of unit abbreviations/equivalences/conversion factors (perhaps in a `UnitRelationships` section), and automatically inserting the relevant conversion factors into a hybrid program where necessary.

5.2 Medium-term

In the medium term, we would like to:

- Properly handle non-integral exponents. Non-integral exponents, while not part of core $\mathtt{d}\mathcal{L}$, are useful when doing actual work in KeYmaera X, so we would like to handle them suitably.
- Support "libraries" of common units, so users don't have to reproduce the same `ProgramUnits` sections over and over.

5.3 Long-term

These are goals made possible by the addition of unit-of-measure types, and would be very nice to have.

- Proof search constrained by units. The question here is: can knowing the units help constrain, say, the `master` tactic in what branches it tries? Another avenue of attack here is seeing if KeYmaera X's invariant generation can be aided by unit of measure constraints.
- An improved interactive interface for model writing and proving. We would like to have something along the lines of agda-mode for Agda. A user of KeYmaera X stuck on what to write in a particular place in a model should be able to summon a dialog that presents the expected units at the current location of the model, plus a list of all salient quantities and their units, to aid them in their thinking (basically, "let the types guide you"). Similarly, when working in the theorem-proving mode, they should be able to ask for unit help from KeYmaera X. What we would like here is something along the lines of KeYmaera Xanalysing the units in open goals, and presenting quantities that look like they might be helpful in constructing quantities with the desired units.

6 Acknowledgements

(No more humble we!) I would like to acknowledge:

- The staff of 15-424 for a fantastic course
- Andre Platzer for useful discussions and great suggestions that helped push me in the right direction when searching for a project
- Nathan Fulton for guidance with the KeYmaera X codebase, and for helping me flesh out our plan for the project
- Brandon Bohrer for all his protips about $\mathsf{d}\mathcal{L}$, KeYmaera X, and life in general
- Esther Wang for putting up with my constant complaining

References

[1] Edgar Buckingham. "On Physically Similar Systems; Illustrations of the Use of Dimensional Equations". In: *Physical Review* 4 (Oct. 1914), pp. 345–376. DOI: 10.1103/PhysRev.4.345.

[2] Andrew Kennedy. "Dimension Types". In: *Proceedings of the 5th European Symposium on Programming: Programming Languages and Systems*. ESOP '94. London, UK, UK: Springer-Verlag, 1994, pp. 348–362. ISBN: 3-540-57880-3. URL: http://dl.acm.org/citation.cfm?id=645390.651419.

[3] Andrew Kennedy. *Units of Measure in F#: Part One, Introducing Units*. 2008 (accessed March 28, 2016). URL: https://blogs.msdn.microsoft.com/andrewkennedy/2008/08/29/units-of-measure-in-f-part-one-introducing-units/.