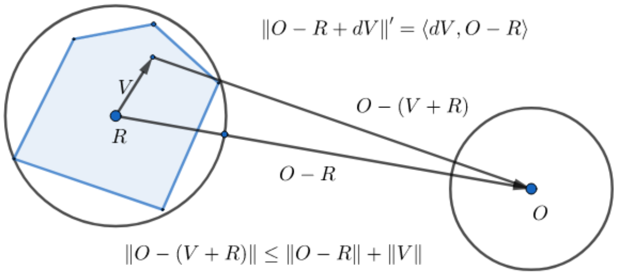
Towards Vector Reasoning in KeYmaera X

cslamber, akabra

November 2020

Vectors for KeYmaera X


$\|O - R + dV\|' = \langle dV, O - R\rangle$
V
R
$O - (V + R)$
$O - R$
O
$\|O - (V + R)\| \leq \|O - R\| + \|V\|$

Abstract

KeYmaera X has proven itself to be a useful tool in the formal verification of cyber-physical systems, a field that studies the intersection of code physics, modeling robots and the autonomous products that have made headlines repeatedly in the past years. We explore avenues to extend the software with support for vectors—from simple syntactic sugar to a concrete type system with in-build generators of axioms—and discuss their feasibility for mainline inclusion into the software, with considerations for implementation detail, maintenance required, and ease-of-use for both end users as well as power users who prove using the tactic language in the source directly.

1 Introduction

Hybrid systems are those systems where continuous evolution and discrete dynamics meet. Many safety-critical real world dynamical systems, such as airplanes, trains, and autonomous vehicles are best modeled as hybrid systems. Differential Dynamic Logic is a scheme to specify and verify hybrid systems [3]. Using first order logic, differential equations and modal operators it allows the description of these systems. KeYmaeraX is a hybrid program theorem prover that uses a practical extension of dL to prove hybrid systems safe [4]. The extension has been formalised as $\mathcal{DL}_l$[5]. However the language does not use first-class vector objects.

The dynamics of most physical systems are most naturally expressed in terms of vectors. In order to enable cleaner expression and more convenient proving of such models we explore the addition of vectors as first class types. We demonstrate how such an addition would make model expression cleaner with examples. Then, we propose a syntax and semantics for vector terms, discussing soundness, completeness and expressivity. We describe the technical considerations involved in adding vector types, exploring multiple implementation strategies.

2 Related Work

In [3], A. Platzer introduced differential dynamic logic as a powerful means to specify and formally verify hybrid systems. In [2] he shows that it is a sound and complete axiomatisation relative to continuous dynamical systems and discrete dynamical systems. This logic, however, is a minimal theoretical framework that does not use vector types.

KeymaeraX is a hybrid system theorem prover that uses Differential dynamic logic as its modeling language [4]. KeymaeraX, being a practical tool, grew to have convenient sound constructions outside of DL such as division and tuples. Bohrer et al. formalize this extended DL that KeymaeraX uses in [5]. The logic is called $\mathcal{DL}_l$. While this work dicusses the usage of tuples to express vectors, it does not actually discuss implementation of vectors in KeYmaeraX, or formalize them.

Our work discusses an implementation of vector types in KeYmaeraX. It describes the model simplifications that arise from first-class vectors, proposes the syntax and semantics of vector types, and describes implementation choices. It also evaluates a partial implementation. To the best of our knowledge, vector types have not been added to KeYmaeraX before.

3 Motivation

consider the following KeYmaeraX model, which models being able to brake before hitting the origin, going in an arbitrary direction

```
Definitions
    Real B;
    Real StoppingDistance(Real v) = (v^2 / (2*B));
End.

ProgramVariables
    Real x;
    Real v;
    Real d;
```

```
End.

Problem
    B > 0 & v > 0 & StoppingDistance(v) < abs(x) ->
        [{
            {d := 1 ++ d := -1};
            {x' = d*v, v' = -B & v >= 0}
        }*]x != 0
End.
```

and then its generalization into two dimensions

```
Definitions
    Real B;
    Real StoppingDistance(Real v) = (v^2 / (2*B));
End.

ProgramVariables
    Real x;
    Real y;
    Real v;
    Real dx;
    Real dy;
End.

Problem
    B > 0 & v > 0 & StoppingDistance(v) < (x^2 + y^2)^(1/2) ->
        [{
            dx := *; dy := *;
            ?dx^2 + dy^2 = 1;
            {x' = dx*v, y' = dy*v, v' = -B & v >= 0}
        }*](x != 0 | y != 0)
End.
```

While the former is possible to prove out of considerations of casing over whether x is positive or negative, the latter is not so simple, since the Euclidean distance between a point and the origin—and dealing with distances in general in the Euclidean metric—is largely nontrivial and comes from consequences of the Cauchy-Schwarz inequality. This means that even though the latter is just a simple step up from the former conceptually, proving it involves considerably more work, and it is easily possible currently to run into short arithmetic-only formulae that take an unreasonable amount of time to solve. Namely, using QE on Cauchy-Schwarz and the Euclidean triangle inequality in 2 dimensions did not complete after running for over 7 hours on a **TODO what's your computer's specs?**.

This problem is not a mere contrivance either, as measuring the change in distance between two points is intuitively an important thing to be able to quickly do in modelling cyber-physical systems and controllers, and comes up frequently. Consider this problem: a car travels along a circle with a fixed center and radius. It can accelerate and decelerate. It must never collide into an obstacle, should there be one along its track.

A KeYmaeraX model and proof for this question can be found in the appendix A.

This is a large proof, and it still makes a very conservative approximation—it uses the Chebyshev distance between the vehicle and the obstacle. Yet, this conservation results in branching 4 times (due to the 4 differential cuts (dc)) on nearly identical expressions to handle each possible case of the Chebyshev distance, two for each coordinate potentially being the largest in magnitude due to it being negative or positive. Fortunately, the proof following each of these cuts is immediate, but as KeYmaera starts to scale up to solve larger and harder problems, this might not be the case, and in three dimensions this sort of approximation would lead to six nearly-identical branches, each of which might take some time to properly prove.

We would instead want to mirror the classical logic counterpart of how a proof of a system would proceed, using the intuitive idea that even if the car were moving straight towards the obstacle, it would not travel far enough to collide.

Consider a more strict version of the original system, where instead of moving predictably around a circle, we can move in any direction any amount of times. Define the stopping distance of a given velocity to be the distance travelled in a straight line before stopping with break power B, so $SD(v) = \frac{v^2}{2B}$. We can claim that so long as we are always breaking and our stopping distance is less than the shortest-path distance between the car and the obstacle, we will never collide with the obstacle regardless of what direction we move in. This claim would be modelled in dynamic logic (extended with a bit of vector notation) as

$$B > 0 \wedge \mathrm{SD}(v) < \|x\| \implies [\{d := *; ?\|d\| = 1; x' = vd, v' = -B \,\&\, v \geq 0\}^*] x \neq 0$$

where $v \in \mathbb{R}$, and $x, d \in \mathbb{R}^N$ for some positive N—so really this holds in any Euclidean space. Since $x \neq 0 \iff \|x\| > 0$, we can rewrite the above equivalently as

$$B > 0 \wedge \mathrm{SD}(v)^2 < x \cdot x \implies [\{d := *; ?d \cdot d = 1; x' = vd, v' = -B \,\&\, v \geq 0\}^*] x \cdot x > 0$$

As a lemma, we can show the following corollary of the "leave within closed" derived axiom (`leaveWithinClosed` in `Ax.scala` in KeYmaera X's codebase),

$$
\begin{array}{cl}
* & \\ \hline
\vdash f(x) \geq 0 \rightarrow (\langle x' = g(x) \& Q \rangle f(x) \leq 0 \iff \langle x' = g(x) \& Q \& f(x) \geq 0 \rangle f(x) \leq 0) & \text{leave within closed} \\ \hline
\vdash f(x) > 0 \rightarrow (\langle x' = g(x) \& Q \rangle f(x) \leq 0 \iff \langle x' = g(x) \& Q \& f(x) \geq 0 \rangle f(x) \leq 0) & \text{strengthen} > \text{to} \geq \\ \hline
\vdash f(x) > 0 \rightarrow \langle x' = g(x) \& Q \rangle f(x) \leq 0 \rightarrow \langle x' = g(x) \& Q \& f(x) \geq 0 \rangle f(x) \leq 0 & \text{strengthen} \rightarrow \text{to} \iff \\ \hline
f(x) > 0, \langle x' = g(x) \& Q \rangle f(x) \leq 0 \vdash \langle x' = g(x) \& Q \& f(x) \geq 0 \rangle f(x) \leq 0 & \rightarrow\text{R} \\ \hline
f(x) > 0, \neg \langle x' = g(x) \& Q \& f(x) \geq 0 \rangle f(x) \leq 0 \vdash \neg \langle x' = g(x) \& Q \rangle f(x) \leq 0 & \neg\text{L/R} \\ \hline
f(x) > 0, [x' = g(x) \& Q \& f(x) \geq 0] f(x) > 0 \vdash [x' = g(x) \& Q] f(x) > 0 & \langle\rangle \text{ duality} \\ \hline
f(x) > 0 \vdash [x' = g(x) \& Q] f(x) > 0 & \text{cut}
\end{array}
$$

where the branch that would be created by that first cut becomes the goal we have to prove (namely that $f(x) > 0 \vdash [x' = g(x) \& Q \& f(x) \geq 0] f(x) > 0$).

In dL extended with some facts about vectors to be discussed later, this stronger system can be proven without any major branching and with a tighter approximation than the Chebyshev distance as follows, omitting the constant assumption that $B > 0$ (and eventually that $d \cdot d = 1$ since d is unchanging in the ODE). Let $\epsilon = \frac{\|x\| - SD(v)}{2} > 0$ in the initial state for brevity.

$$
\begin{array}{cl}
* & \\ \hline
x > 0, \|d\| = 1, v \geq 0 \vdash v \geq v\|d\| & \mathbb{R} \\ \hline
x > 0, \|d\| = 1, v \geq 0 \vdash v \geq v \dfrac{\|x\|\|d\|}{\|x\|} & \mathbb{R} \\ \hline
x > 0, \|d\| = 1, v \geq 0 \vdash v \geq v \dfrac{|\langle x, d \rangle|}{\|x\|} & \text{C-S} \\ \hline
\mathrm{SD}(v) + \epsilon \leq \|x\|, \|d\| = 1, v \geq 0 \vdash \dfrac{-Bv}{B} \leq \dfrac{x \cdot vd}{\|x\|} & \text{(properties of real inequalities)} \\ \hline
\mathrm{SD}(v) + \epsilon < \|x\|, \|d\| = 1, v \geq 0 \vdash [x' := vd][v' := -B] \dfrac{2vv'}{2B} \leq \dfrac{x}{\|x\|} \cdot x' & [:=] \\ \hline
\mathrm{SD}(v) + \epsilon < \|x\|, \|d\| = 1 \vdash [\{x' = vd, v' = -B \,\&\, v \geq 0 \& \mathrm{SD}(v) + \epsilon \leq \|x\|\}] \dfrac{v^2}{2B} + \epsilon < \|x\| & \text{dI} \\ \hline
\mathrm{SD}(v) + \epsilon < \|x\|, \|d\| = 1 \vdash [\{x' = vd, v' = -B \,\&\, v \geq 0\}] \dfrac{v^2}{2B} + \epsilon < \|x\| & \text{lemma} \\ \hline
\mathrm{SD}(v) + \epsilon < \|x\| \vdash [d := *; ?\|d\| = 1; \{x' = vd, v' = -B \,\&\, v \geq 0\}] \mathrm{SD}(v) + \epsilon < \|x\| & [;],[:=],[?] \\ \hline
\mathrm{SD}(v) + \epsilon < \|x\| \vdash [\{d := *; ?\|d\| = 1; x' = vd, v' = -B \,\&\, v \geq 0\}^*] \mathrm{SD}(v) + \epsilon < \|x\| & \text{ind} \\ \hline
\mathrm{SD}(v) + \epsilon < \|x\| \vdash [\{d := *; ?\|d\| = 1; x' = vd, v' = -B \,\&\, v \geq 0\}^*] \|x\| > 0 & \text{M} \\ \hline
B > 0, v > 0, \mathrm{SD}(v) + \epsilon < \|x\| \vdash \ldots & \text{WL} \\ \hline
\vdash B > 0 \wedge v > 0 \wedge \mathrm{SD}(v) < \|x\| \rightarrow \ldots & \rightarrow\text{R},\mathbb{R}
\end{array}
$$

where we needed that lemma to have that $\|x\| > 0$ in the differential invariance, so we were free to divide by it when taking the derivative while keeping a necessarily well-defined expression. For branch INIT we get

$$\dfrac{*}{\left(\frac{v^2}{2B}\right)^2 < x \cdot x, d \cdot d = 1 \vdash \left(\frac{v^2}{2B}\right)^2 < x \cdot x}\,\text{id}$$

Note that for the M (monotonicity) step, we know that $SD(v)^2 \geq 0$, so $x \cdot x > SD(v)^2 \implies x \cdot x > 0$, which would have to be proven in a different branch via a differential cut, but this is not important for the mechanics of this proof so we omitted that single, quick branch for brevity.

The (relative) simplicity, straightforwardness, and generality of this proof we believe to be sufficient motivation in itself to pursue first-class vectors in KeYmaera X. The concepts of this proof are much closer to a proof of the statement in a statement in an analysis or introductory vector calculus course, there is no unnecessary branching, and the core derived lemmas were chosen in such a way that the same ideas can be applied with generality whenever a similar problem arises. Of course, making sure that we are not dealing with the case where $\|x\| = 0$ is not particularly nice, but it does follow from the more intuitive reason that we should “be enough away” from zero, and does not rely on anything particularly clever.

4 Approach

4.1 Syntax

At the base level, vectors can be thought of as merely ordered list of expressions. So, our model of vectors was simply taking some number of terms (or variables), and binding them into a single object that is a term or variable respectively. We propose the syntax that follows.

Vector terms v produce as below:

$$v := \{\{a_1, a_2 ... a_n\}\} \,|\, v_n + u_n \,|\, v_n - u_n \,|\, v * a \,|\, -v \,|\, v' \,|\, \text{var}$$

where a_i is a scalar real term, and v_n and u_n are vector terms of n dimensions, and var is a variable name. We have chosen the syntax of double braces to denote the start and end of vector literals as to avoid conflicts with modal operators [and <.

In turn, scalar terms are extended from $\mathcal{DL}_l$with the following extra possibilities:

$$a := a_{\mathcal{DL}_l} \,|\, v.u \,|\, \text{norm}(v)$$

Where the syntax used for norm(a) is $|a|$, written in function form above only to avoid confusion with the overloaded | symbol. $a_{\mathcal{DL}_l}$ should be read as a macro for “all the productions in $\mathcal{DL}_l$”.

Since a could be a differential, it is possible that the component of a vector be a differential.

Expressions compose into programs largely the same way as in $\mathcal{DL}_l$. Terms, however, compose with the natural minor differences. Magnitude comparison operators like $\leq$ and $\geq$ require that the type of term on either side is be scalar, eg., $\{\{1,2\}\} < \{\{2,1,3\}\}$ is a meaningless operation. Other binary operators like $=$ and $:=$ require that the terms on either side be of the same type, eg., $\{\{a,b\}\} := 3$ is undefined.

4.2 Semantics

$\mathcal{DL}_l$is capable of expressing vectors [5]. We translate our source vector-extended $\mathcal{DL}_l$to target regular $\mathcal{DL}_l$, providing a translational semantics for our vector extension.

Translation proceeds by structural induction. All terms and expressions already defined in $\mathcal{DL}_l$remain the same- the translation function is the identity. Thus the only new non-trivial inductive cases are for expressions and terms involving the newly defined term forms. We present the translations of interest below.

First, define the operational procedure that reduces vector terms of any for to the form of $\{\{a_1, a_2...a_n\}\}$.

$$[\![\text{var}]\!] \Rightarrow [\![\{\{a_1, a_2...a_n\}\}]\!] \quad \text{where } \sigma(\text{var})=\{\{a_1, a_2...a_n\}\}$$

sigma is the context the stores the value of variables.

$$[\![v \oplus u]\!] \Rightarrow [\![\{\{v_1 \oplus u_1, ...v_n \oplus u_n\}\}]\!]; \quad \text{where } \oplus \text{ represents } + \text{ and } -$$

$$[\![-v]\!] \Rightarrow -[\![\{\{v_1, ..., v_n\}\}]\!];$$

$$[\![v']\!] \Rightarrow [\![\{\{v_1', ..., v_n'\}\}]\!]';$$

$$[\![v * a]\!] \Rightarrow [\![\{\{a * v_1, ...a * v_n\}\}]\!];$$

Having defined a "flattening" transformation from an arbitrary vector to a vector of the form $\{\{a_1, a_2...a_n\}\}$ we define translational semantics for this form only. The idea is that with the function *flatten* defined above, given an arbitrary vector $v + u$ and a translation rule, say.

$$[\![\{\{a_1, a_2...a_n\}\} := \{\{b_1, b_2...b_n\}\}]\!] \triangleq [\![a_1 := b_1]\!]; ...[\![a_n := b_n]\!];$$

The translation $[\![a := v + u]\!]$ can be obtained as $[\![\text{flatten}(v + u)]\!]$, which on expansion, would be

$$[\![a := v + u]\!] \triangleq [\![a_1 := v_1 + u_1]\!]; ...; [\![a_n := v_n + u_n]\!];$$

Now, the rules for new real terms are:

$$[\![\{\{v_1, ...v_n\}\}.\{\{u_1, ..., u_n\}\}]\!] \triangleq [\![v_1]\!].[\![u_1]\!] + ... + [\![v_n]\!].[\![u_n]\!]$$

$$[\![|\{\{v_1, ..., v_n\}\}|]\!] \triangleq ([\![v_1^2]\!] + ... + [\![v_n^2]\!])^{1/2} = ([\![|\{\{v_1, ..., v_n\}\}| \cdot |\{\{v_1, ..., v_n\}\}|]\!])^{1/2}$$

The definition of norm uses the square root, which fortunately, is allowed in $\mathcal{DL}_l$.

Finally:

$$[\![\{\{v_1, ...v_n\}\} \sim \{\{u_1, ...u_n\}\}]\!] \triangleq [\![v_1]\!] \sim [\![u_1]\!]; ...[\![v_n]\!] \sim [\![u_n]\!]; \quad \text{where} \sim \text{represents} = \text{and} :=$$

This translational semantics assumes that vectors are of finite dimension, an assumption that holds true in most physical models.

4.3 Typing Implementation

Consider the expression $a + b$. At parsing time, without maintaining a typing context, there is no good way to distinguish the situations where a and b are real variables, or vector variables. In order to catch the situation where one is real but the other is a vector, we would need to perform type checking. The cleanest way to do this would be to introduce a type checking pass.

We found workarounds: while parsing, you could have type variables lazily, not setting a value until there is enough information. Alternately, it is possible to maintain a typing context while parsing whole programs. However, both solutions are inferior to introducing a typing pass, especially since complexity will increase if new types such as matrices and tensors are introduced in the future.

5 Completeness and Expressivity

$\mathcal{DL}_l$is at least as expressive as vector extended $\mathcal{DL}_l$, as we were able to define translational semantics. vector extended $\mathcal{DL}_l$, being a strict superset of $\mathcal{DL}_l$, is at least as expressive as it; any program that can run in $\mathcal{DL}_l$will run unchanges in vector-extended $\mathcal{DL}_l$. As a consequence, vector extended $\mathcal{DL}_l$, like $\mathcal{DL}_l$, is not complete. Vector-extended $\mathcal{DL}_l$possesses no new axioms, and every new expression translates uniquely to a $\mathcal{DL}_l$expression. Defining the semantics of the vectors from their translations, its soundness follows from that of $\mathcal{DL}_l$.

6 New Lemmas

There is a set of lemmas that apply very naturally to vector expressions. After having first class vectors, their inclusion makes many proofs easier.

- Cauchy-Schwarz Inequality: This inequality is the foundation out of which much vectorial-reasoning is derived, simply stating that in Euclidean space, $|x \cdot y| \leq \|x\|\|y\|$, or equivalently without square roots it is $(x \cdot y)^2 \leq (x \cdot x)(y \cdot y)$. This proof already exists as part of the underlying derivation of vectorial differential ghosts in KeYmaera X, and the result is easily extracted to make the same statement about vector variables in our formulation.

- Triangle Inequality for the Euclidean Norm: Potentially the most useful property of a norm—being the sole important property of a metric even—this is used in countless analytical and physical proofs of properties. Specifically, this allows users to prove upper bounds on distances between two points via the distance between those points and another, which might be substantially easier to show. Follows using the usual derivation from Cauchy-Schwarz.

- Basic properties of inner product spaces: While not particularly flashy, simply knowing that $\langle ax+z, y\rangle = \langle y, ax+z\rangle = a\langle x, y\rangle + \langle z, y\rangle$ for $a, b \in \mathbb{R}$, $x, y, z, \in \mathbb{R}^N$ is so important that it goes without stating in most cases. Similarly for other properties of $\mathbb{R}$-modules and IP spaces such as $\|ax\| = |a|\|x\|$, $(ab)x = a(bx)$, etc. All follow directly from definitions of the vector operations, and will be proven as derived lemmas in KeYmaera X in a similar manner to how Cauchy-Schwarz is—only substantially simpler.

7 Implementation Possibilities

We initially identified two avenues for attempting to integrate vectors into KeYmaera X: as syntactic sugar and as fixed-length lists, and began working on implementing to a reasonable extent to understand where in the core we would face difficulties. While we initally were partial to the fixed-length list implementation, during the process we identified a third, much more promising option of letting these vectors be inductively-defined within KeYmaera X's axiom logic.

All three options relied on the same parsing code, which we built on top of the experimental alternative parser DLParser. While not prepared for full integration as it fails to parse lemmas involving interpreted functions, it allows for rapid prototyping at the cost of not being able to extend it fully to the web UI. This turned out to not be a bottleneck as we have identified other features that would be necessary the integration of proper vector proofs into the web UI.

7.1 As syntactic sugar

Motivation We easily rewrite every formula, program, or term that contains vectors without them by applying the above rules recursively on the expression—rewriting terms of the vector sort to a list of terms of the real sort that do not possess any vector components and rewriting terms of the real sort to a single term that does not. Formulae and programs can be written to single formulae and programs, recursively applying the relevant rules to all component parts. Because of this equivalence between things we already could express and things we can express with vectors, it is tempting to rewrite a formula as above immediately after parsing since it removes the trouble of dealing with newly-defined operators in proofs.

Benefits This approach is by far the most straightforward to implement, and is nearly guaranteed to not cause any issues interacting with the other parts of the system. Existing proof rules and tactics are optimized to work with problems phrased using just reals, and there has been a lot of work already internally to provide tactics with some useful lemmas of Euclidean space, such as the equality of norms and the Cauchy-Schwarz inequality. Since this does not affect the interactive portion of the theorem prover at all, we do not need to keep external data as to what is a vector variable if we were to add in vector variables as some preprocessing that would convert a declared vector with length "X" to $\{\{x_1, x_2, x_3, \cdots, x_n\}\}$—i.e. using n real variables always in that form to represent a variable vector.

Drawbacks As mentioned above, this does not interact with the interactive portion at all, and as such will be largely useless in providing a nice user experience in solving multidimensional problems. Furthermore, the lemma seems to work best when matching against lemmas that are shorter, so expressing something like Cauchy-Schwarz as was useful in the above inequality would take many variables, rather than the two needed to state it in terms of vectors. While not offering any potential issues to the soundness of the system, it does not have enough power to provide the user with anything nicer than a slight shortcut in writing models, which hardly makes progress on the problem we identified.

7.2 As fixed-length lists

Motivation In most treatments of vectors in a physics or analysis course, they are represented as functions from $[N] = \{1, 2, \ldots, N\}$ to $\mathbb{R}$ for a given dimension $N \in \mathbb{N}$, and all primitive operations (addition, subtraction, scalar multiplication) are performed coordinate-wise. By creating a new sort of expressions in KeYmaera X and a suite of operators that deal with them that are defined in terms of just applying their underlying operation coordinate-wise, we can keep vectors together and deal with them as holistic objects during proving. To provide these definitions to the prover, rather than having them as unavoidable transformations that occur after parsing, they are encoded as axioms in the core of KeYmaera X that can be applied at will. Of course, since each of those equations above would need to be listed for every dimension, we cannot explicitly encode all of the infinitely many axioms that would be required, so instead we added a module to the core that would output the axioms for a given dimension.

Benefits This style is much closer to the style of proof that we identified as our goal earlier, since we do not have to decompose vectors immediately into their component parts, and in fact we can leave them together throughout the proof so long as we have the lemmas to close out our branches. This is the most straightforward way of considering vectors, and so there is nothing particularly surprising that would arise when using this form.

Drawbacks KeYmaera's core axiom structure is very dependent on having only a fixed number of axioms, with tactics being defined manually through macros in a single file and not having any structure to allow for 'requesting' axioms from the core should they be needed. Therefore, without restructuring or changing a large portion of the code, all of the definitions would be hardcoded for whatever dimensions the programmer believes that the user would potentially need. While not particularly bad in itself since it is unlikely for anybody to need a 5-dimensional physical vector, let alone 10, this sort of quick hack to get it working would likely prevent any amount of useful extension on the idea and implementation, say to support matrices, tensors, or other very physically-useful constructions, without changing the way in which they are implemented in the first place.

With the vector constructor taking anywhere from one to an unbounded many arguments, it conflicts with most of the assumptions made about operators throughout the codebase. Much deeper, however, is that this implementation requires that any dimension-generic statements be made in the source of KeYmaera X itself as it is a function with integer argument, rather than potentially having a user capable of proving it. Not only is this a barrier just in terms of effort, but also in terms of understanding the less visually-intuitive system that the tactic language of the internals. We believe that this implementation could work and be effective, however, if enough lemmas were built-in to the source already and the clashes with other design assumptions were resolved. But, the next construction and implementation provides a much more promising, clean solution that would expand upon more fruitful areas of the core of the software.

7.3 As an inductively-defined structure

Motivation In functional programming languages and areas of constructive logic in general, types or sets are generally explicitly defined in terms of the finitely-many ways there are of constructing a member of such a set. Peano naturals, for examples, are defined to be either zero or the successor of a natural number (applied arbitrarily but finitely many times), with explicit equality for two members constructed in the same way and inequality for two that are not. In these fields, finite lists of elements of a given type are defined as either the empty list, or a pair of an element of that type and a list (again applied arbitrarily but finitely many times). This sort of construction gives rise to an immediate way of proving things about members of these sets via structural induction over the constructors being applied to generate elements of the set. KeYmaera X, being inspired by and existing in the realm of functional programming and formal logic, would make sense to have a the new data type being defined in this way. Notice that vectors will remain distinct from tuples, which have a similar inductive definition because of there are more assumptions on the nature of vectors, so that certain lemmas and properties like inner product already apply.

We could define vectors as either being the 0-dimensional zero vector, say `Nil`to follow in the spirit of Scala, or it is a real-valued term 'cons'ed onto a vector to add a new coordinate (represented by the operator `::`, taking in a real term on the left and a smaller vector on the right), in the same way that the pair of an element and a list made up a new list. Then, we can axiomatize each of the above equations that defined our definition of vectors as follows for all a, b being vectors of reals, $x, y, v \in \mathbb{R}$, and propositions p, q:

- $\text{Nil} = \text{Nil} \iff \text{true}$
- $(x :: a) = (y :: b) \iff (x = y)\&(a = b)$
- $\text{Nil} \oplus \text{Nil} = \text{Nil}$
- $\text{Nil} \cdot \text{Nil} = 0$
- $-\text{Nil} = \text{Nil}$
- $[\text{Nil} := \text{Nil}]p \iff p$
- $v * \text{Nil} = \text{Nil} = \text{Nil} * v$ for any $v \in \mathbb{R}$
- $(x :: a)' = (x') :: (a')$
- $(x :: a) \oplus (y :: b) = (x + y) :: (a \oplus b)$
- $(x :: a) \cdot (y :: b) = (xy) + (a \cdot b)$
- $v * (x :: a) = (vx) :: (v * a) = (x :: a) * v$
- $-(x :: a) = (-x) :: (-a)$

and then for programs, which I will discuss the difficulty with later

- $[\text{Nil} := \text{Nil}]p \iff p$
- $[\{\text{Nil}' = \text{Nil}\&q\}]p \iff (q \implies p)\}$
- $[(x :: a) := (y :: b)]p \iff [x := y; a := b]p$
- $[\{(x :: a)' = (y :: b)\&q\}]p \iff [\{x' = y, a' = b\&q\}]p$

Note that while we mentioned the dimension of the base case, in this construction there is no explicit need to keep track of the dimension of vectors, as formulae with mismatched vector sizes will be unprovable regardless, since during the unraveling of the vector structure while applying the above structurally-inductive definitions, both arguments that have to have matched dimensions will have to reach the Nil case in the same number of steps, since Nil cannot be compared to something that is not Nil. That being said, many important lemmas require some requirement that two vector lengths are equal, so keeping some track of this will be useful.

Benefits Unlike in the previous idea, we only need finitely many axioms to cover every case of this implementation of vectors. Furthermore, all operations are unary or binary, which allows for immediate unification, printing, and areas of the codebase that assume all operators are unary or binary to work properly. Given the sort of matching that would allow us to express the program axioms, ideally it would be possible for a user to define and prove their own inductive lemmas that apply for all values of the dimension without modifying the codebase directly. For example, Cauchy-Schwarz (and by immediate consequence the triangle inequality in Euclidean space) admits a very short inductive proof for our system. Namely, we are trying to prove that for all dimensions $n \in \mathbb{N}$ we have that for all $x, y \in \mathbb{R}^N$ that

$$(x \cdot y)^2 \leq (x \cdot x)(y \cdot y) \iff (x \cdot x)(y \cdot y) - (x \cdot y)^2 \geq 0$$

Which is trivial in the base case when $x = y = \text{Nil}$, and so $(\text{Nil} \cdot \text{Nil})^2 = 0 = (\text{Nil} \cdot \text{Nil})(\text{Nil} \cdot \text{Nil})$. In the inductive step, we want to show it for $a :: x, b :: y$ where $a, b \in \mathbb{R}$ and we already have the inequality on just x and y. This can be shown since since

$$\begin{aligned}
&((a :: x) \cdot (a :: x))((b :: y) \cdot (b :: y)) - ((a :: x) \cdot (b :: y))^2 \\
&= (a^2 + x \cdot x)(b^2 + y \cdot y) - (ab - x \cdot y)^2 \\
&= a^2b^2 + a^2(x \cdot x) + b^2(y \cdot y) + (x \cdot x)(y \cdot y) - a^2b^2 - 2ab(x \cdot y) - (x \cdot y)^2 \\
&\geq a^2b^2 - a^2b^2 + (x \cdot y)^2 - (x \cdot y)^2 + a^2(y \cdot y) + b^2(x \cdot x) - 2ab(x \cdot y) \\
&= (a * y - b * x) \cdot (a * y - b * x) \geq 0
\end{aligned}$$

where the second to last step holds by induction hypothesis, and the last step holds by a clever rearrangement using the linearity of the dot product, as $(a * y - b * x) \cdot (a * y - b * x) = (a * y) \cdot (a * y) - 2(a * y) \cdot (b * x) + (b * x) \cdot (b * x) = a^2(y \cdot y) - 2ab(x \cdot y) + b^2(x \cdot x)$.

Of course, this argument is contingent upon having already shown the linearity of the inner product and having a notion of induction built-in to KeYmaera X, since we are inducting on the lengths of two different vectors (that we require have the same length). However, with some sort of clever tactic it would be possible to automatically run all of the induction up to where you need as desired without making drastic changes to the core (and potentially its soundness), making this a much more feasible option for the extensible implementation of the features of vectors that we propose in this paper.

Drawbacks Unfortunately, it appears that the current expression unifier and substitution system in KeYmaera X does not have the ability to match on any variable-like expression, which would be needed in order to match the 'tail's of the assignee in the assignment and ODE cases. In order to resolve this, we would need to do more research into how we could extend the uniform substitution axiom to allow for this sort of "match anything that looks like a variable" expression, and more crucially we would need a deeper investigation into the soundness of any such extension. Intuitively, these matches should be sound though since we *could* have equivalently written out the expression in a vector-free way, and these axioms just allow us to convert back and forth between those representations. Therefore, if the original system was sound, allowing for the matching of variable-like expressions at least in the above cases should be sound, but of course in allowing for extensibility within the proof assistant, doing it as general and as safely as possible will be a much more sustainable option.

8 Future work

Having investigated implementation choices, it remains to choose and execute a complete implementation of first class vectors. It would then be interesting to explore case studies of systems that are more easily expressed and proved with vectors. Another useful contribution would be identifying more lemmas that would make vector intuition easier to capture in proof. Exploring the possibility and use cases of adding matrices and tensors is a potential future extension.

References

[1] Platzer, A. Differential Dynamic Logic for Hybrid Systems. J Autom Reasoning 41, 143–189 (2008). https://doi.org/10.1007/s10817-008-9103-8

[2] A. Platzer, "The Complete Proof Theory of Hybrid Systems," 2012 27th Annual IEEE Symposium on Logic in Computer Science, Dubrovnik, 2012, pp. 541-550. doi: 10.1109/LICS.2012.64

[3] Platzer, A. "Differential Logic for Reasoning About Hybrid Systems", Conference Proceedings, Hybrid Systems: Computation and Control, Berlin, Heidelberg, 2007, pp. 746-749.

[4] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp and André Platzer (2015). KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems. In CADE (pp. 527–538). Springer.

[5] Bohrer, Brandon, Fernández, Manuel and Platzer, André (2019). Definite Descriptions in Differential Dynamic Logic. In Automated Deduction – CADE 27 (pp. 94–110). Springer International Publishing.

9 Appendix

9.1 A: KeymaeraX proof for car traveling in circle

```
Definitions
     Real r; /* Racetrack radius */
```

```
  Real T; /* Maximal time interval between consecutive triggerings of
       the controller */
  Real A; /* The robot accelerates with acceleration A */
  Real B; /* The robot brakes with acceleration -B */

  Real ox; /* Cartesian coordinate x of the obstacle */
  Real oy; /* Cartesian coordinate y of the obstacle */

  /* a formula (d) that computes the cartesian distance between the
      robot
     and the obstancle, and a function that gives the distance that
         will
     be travelled before breaking at acceleration -B given a current
     velocity (in a straight line) */
  Bool absBigger(Real x, Real y) <-> (x > y | -x > y);
  Bool safeBy(Real x, Real y, Real m) <-> (absBigger(x-ox, m) |
      absBigger(y-oy, m));
  Real breakD(Real v) = (v^2/(2*B));
End.

ProgramVariables
  Real t; /* Elapsed time since the last controller triggering */
  Real x; /* Cartesian coordinate x of the robot */
  Real y; /* Cartesian coordinate y of the robot */
  Real v; /* Linear velocity (ground speed) of the robot */
  Real a; /* Linear acceleration of the robot */
End.

Problem
  (
    /* Constant assumptions and initial conditions */
    (T > 0 & A > 0 & B > 0 & r > 0 & ox = -r & oy = 0) & v >= 0 &
    /* obsD(x, y) > breakD(v) */
    safeBy(x,y, breakD(v)) &
    x^2 + y^2 = r^2
  ) -> [ {
      /* Robot controller */
      { { ?(
            /* When is it safe to accelerate? */
            /*obsD(x, y) > breakD(v+A*T) + v*T + 1/2*A*T^2 */
            safeBy(x,y, breakD(v+A*T) + v*T + 1/2*A*T^2) );
          a := A; } ++ { a := -B; } }
      /* Continuous dynamic */
      t:=0;
      { x'=-(v/r)*y, y'=(v/r)*x, v' = a,
      t' = 1 & v >= 0 & t <= T }
    }* @invariant(
      /* obsD(x, y) > breakD(v) */
      safeBy(x,y, breakD(v)) &
      x^2 + y^2 = r^2 &
      v >= 0 )  ]
  /* Safety condition: the robot stays on the track and never hits
      the obstacle */
  ((x^2 + y^2 = r^2) & !(x = ox & y = oy))
End.
```

With proof

```
Tactic "Lab 3 Linfty: Proof"
implyR(1) ; loop("safeBy(x,y,breakD(v))&x^2+y^2=r()^2&v>=0", 1) ; <(
  expandAllDefs ; auto,
  expandAllDefs ; auto,
  composeb(1) ; choiceb(1) ; andR(1) ; <(
    expandAllDefs ; unfold ; dC("x^2+y^2=r()^2", 'R) ; <(
      boxAnd('R) ; andR(1) ; <(
        hideL(-9=="(x-ox()>v^2/(2*B())|-(x-ox())>v^2/(2*B()))|y-oy()>v^2/(2*B())|-(y-oy())>v
             ^2/(2*B())") ; orL(-7) ; <(
          orL(-7) ; <(
            dC("x-ox()>(v+A()*(T()-t))^2/(2*B())+v*(T()-t)+1/2*A()*(T()-t)^2", 'R) ; <(
              auto,
              auto
              ),
            dC("-(x-ox())>(v+A()*(T()-t))^2/(2*B())+v*(T()-t)+1/2*A()*(T()-t)^2", 'R) ; <(
              auto,
              auto
              )
            ),
          orL(-7) ; <(
            dC("y-oy()>(v+A()*(T()-t))^2/(2*B())+v*(T()-t)+1/2*A()*(T()-t)^2", 'R) ; <(
              auto,
              auto
              ),
            dC("-(y-oy())>(v+A()*(T()-t))^2/(2*B())+v*(T()-t)+1/2*A()*(T()-t)^2", 'R) ; <(
              auto,
              auto
              )
            )
          ),
        auto
        ),
      auto
      ),
    expandAllDefs ; assignb(1) ; composeb(1) ; assignb(1) ; unfold ; dC("x^2+y^2=r()^2", 'R)
         ; <(
      auto,
      auto
      )
    )
  )
End.

End.
```