Inductive Extensions to dL

Ananya Kumar

May 9, 2017

1 Introduction and Goals

Our work aims to add inductive structures to dL. Possible applications of our work include,

1. A group of robots cooperating to achieve a task (the number of robots is variable).
2. A robot trying to clean a room shaped as a polygon (the number of sides of the polygon is variable).
3. A single robot avoiding a static obstacle in high dimensional space (the number of dimensions is variable).
4. Proving properties about increasingly good Taylor series approximations to a function (the number of terms in the approximation is potentially infinite). This could be useful when agents interact with other agents based on approximations (for example by approximating the shape of an arc).

More specifically, we sketch out how to add two inductive structures to dL: symbol trees and lists. Symbol trees represent computation trees which can contain variables. For example, Times(x, 3), is a symbol tree representing $3x$, and Plus(Times(x, x), Plus(x, 4)) represents $x^2 + x + 4$. Using symbol trees, we can represent things like sequences and series of symbols. Importantly, our symbol trees can be symbolically differentiated, which allows for proofs that cannot be done using dL or QdL. We also sketch out how to add lists of real numbers to dL. We add corresponding proof rules, and investigate possible applications.

I think the results presented in this paper are exciting. However, they are not fully developed. I was playing around with some inductive structures, and thought about

introducing the notion of symbol trees to solve some cool problems. However, I significantly underestimated the amount of effort it would take to complete this work. At that point, there were 2 possibilities. Either I could rigorously do something very mundane (like regular lists) that do not have any applications to cyberphysical systems. Or I could race ahead in developing some much more exciting, albeit not yet fully rigorous, theory. I chose the latter.

2 Previous Work

One way of describing variable or infinite collections of objects is Quantified Differential Dynamic Logic. QDL contains a finite number of sorts. A sort C is just a finite or infinite collection of symbols like $c_1, c_2, c_3, ...$ The sorts are not allowed to overlap. In some sense, each sort can define a new class of things, like the set of cars. Functions in QdL can map sorts to other sorts (including $\mathbb{R}$). For example, the function x could map cars (elements of C) to their x-coordinate $\in \mathbb{R}$. In many cases, x can also be thought of as a list of values that are indexed by elements of C.

QdL is an elegant extension of dL, and has been used in many applications (for example in proving that a variable number of cars driving on a highway, even with cars switching lanes and entering the highway, is safe). However, the functions in QdL are not inductively defined over the structure. There are many situations that are naturally inductive. For example, if we want the cars to be initially separated by distance exactly 1, or want the car positions to be in the sequence $(1, 2, 4, 7, 11, 16, ...)$, the QdL expression is fairly complicated. We would have to essentially implement a linked list by keeping a 'next' pointer. Additionally, after setting up the car positions, I might choose to apply different controls to each car, where the controls are also based on the inductive structure.

A running example that we focus on, is proving properties about Taylor series. For example, we might want to prove that $e^x > 1 + x + x^2/2 + ... + x^i/i!$. We can write a dL formula that describes this:

$$\begin{aligned} x = 0, y = 1 \vdash [(y' = y, x' = 1);& \\ & s := 0; t := 1; i := 0; \\ & (s := s + t; i := i + 1; t := (t \cdot x)/i)^*] \\ & s \leq y \end{aligned}$$

Basically, this model sets $s = 1 + x + x^2/2 + ... + x^i/i!$. In each iteration of the loop, it generates the next term t of the Taylor series, and adds it to s.

Unfortunately, it is not clear how to prove this model using existing rules in dL. In fact, I think it is unlikely that we can prove this model using dL. To motivate this intuition, let us look at a possible way we might try to prove this. Let $s_i = 1+x+x^2/2+...+x^i/i!$. In Written assignment 5, we proved that $s_3 \leq e^x$. We used an inductive argument. We first showed that $s_2 \leq e^x$, and then cut that into our differential equation. Then, we used dI to show that $s_3 \leq e^x$, exploiting the structure of the Taylor Series of e^x.

Crucially, we had to show a connection between s_2, which we cut into the differential equation, and the derivative of s_3 which we were proving a property about. However, remember that in the above model $s' = 0$. We want to somehow capture how s' evolves after each iteration of the loop, but there is no way to do this in dL. The point is that x, s, t, y are simply real numbers, we do not store any connection between them.

In some sense, our problem was that we computed $1 + x + x^2/2 + ... + x^i/i!$ after the differential equation completed, but did not keep track of how $1+x+x^2/2+...+x^i/i!$ changed during the differential equation. To do this, we need to support some form of symbol trees that can be symbolically differentiated.

3 Typing System

In this project we will introduce multiple different kinds of structures. We want to distinguish their types. We will use syntax to distinguish between types. If s is a string consisting of alphabets and underscores, then $R(s)$ is a real variable, $L(s)$ is a finite list of reals, and $S(x)$ is a symbol tree. Note that we may have variables $R(x)$, $L(x)$, and $S(x)$ can all be distinct variables. A store ω maps variables of the form $R(x)$ to a real number, $L(x)$ to a list, etc. A dL formula e is valid if $\omega \models e$ for all ω. This method is a little hacky, but lets me get started on the meat of this work.

Ideally, we would take inspiration from PFPL (Harper, 2016). We would keep both a type context τ and a store ω. τ will map each variable x to a pre-defined type. ω will map each variable x to a value of type $\tau(x)$. We would then define what it means for a dL expression is valid under a certain typing contex. We could separate out the static (typing) semantics from the non-deterministic transition semantics. This can conveniently facilitate higher order types, but we might not get to it in this project.

4 Symbol Trees

4.1 Syntax

Given a string s that does not contain brackets, a real term r, and symbol tree terms T_1, T_2, a symbol tree term is defined inductively as,

$$\text{SLeaf}(\text{“}R(s)\text{”}) \mid \text{RLeaf}(r) \mid \text{Plus}(T_1, T_2) \mid \text{Times}(T_1, T_2) \mid \\ \text{Div}(T_1, r) \mid \text{Neg}(T_1) \mid \text{Deriv}(T_1)$$

SLeaf(“$R(s)$”) is one of the most important constructs, because it constructs a tree with an unevaluated variable. For example, SLeaf(“$R(x)$”) represents a tree containing the variables x. However, you can think of this as a lazy tree, because the string “$R(x)$” is stored, as opposed to the value of x. A lot of the other constructs are used to store compositions of trees. Again, these are simply stored as syntax trees, they are not actually evaluated. The denominator in Div is a real number as opposed to a tree, so that we can guarantee in the semantics that the denominator is non-zero. Deriv(T_1) is slightly from the rest, because it takes a symbol tree and output a new symbol tree that is the symbolic derivative.

We also modify the syntax of real terms (that is, regular terms in dL). Given real terms e_1, e_2, a string s that does not contain brackets, a rational constant c, and a symbol tree term T, a real term is defined inductively as,

$$R(s) \mid c \mid e_1 + e_2 \mid e_1 \cdot e_2 \mid e_1/e_2 \mid E[T]$$

The only difference from dL is that we introduced $E[T]$ which evaluates the symbol tree T.

We also augment the definition of formulae, to allow comparisons on symbol trees. Specifically, if T_1 and T_2 are symbol tree terms, we allow comparisons $T_1 = T_2$ and $T_1 \equiv T_2$. Intuitively, the former is stronger and means the trees are syntactically the same, and the latter means the trees are equivalent (so $\equiv$ will capture commutativity of $+$, for example).

Lastly, we augment the definition of assignment in hybrid programs. We support 2 kinds of assignments, where r is a real term and s is a symbol tree term,

$$R(x) := r$$

$$S(x) := s$$

4.2 Semantics

In the metatheory, a symbol tree will be defined inductively, with a few differences from the inductive definition of a symbol tree term. Given $r \in \mathbb{R}$, a string S that does not contain brackets, and symbol trees T_1, T_2, a symbol tree is defined inductively as,

$$\text{SLeaf}(\text{“}R(s)\text{”}) \mid \text{RLeaf}(r) \mid \text{Plus}(T_1, T_2) \mid \text{Times}(T_1, T_2) \mid \text{Div}(T_1, r) \mid \text{Neg}(T_1)$$

Next, we define the semantics of a symbol tree term, given a state ω.

$$\omega[[\text{SLeaf}(\text{“}R(s)\text{”})]] = \text{SLeaf}(\text{“}R(s)\text{”})$$

$$\omega[[\text{RLeaf}(r)]] = \text{RLeaf}(\omega[[r]])$$
$$\omega[[\text{Plus}(T_1, T_2)]] = \text{Plus}(\omega[[T_1]], \omega[[T_2]])$$
$$\omega[[\text{Times}(T_1, T_2)]] = \text{Times}(\omega[[T_1]], \omega[[T_2]])$$
$$\omega[[\text{Div}(T_1, r)]] = \text{Div}(\omega[[T_1]], \omega[[r]])$$
$$\omega[[\text{Neg}(T_1)]] = \text{Neg}(\omega[[T_1]])$$

The semantics of Deriv basically replicate product rule, sum rule, etc, of derivatives.
So, for example, if $\omega[[T]] = \text{Plus}(T_1, T_2)$, then $\omega[[\text{Deriv}(T)]] = \text{Plus}(\omega[[\text{Deriv}(T_1)]], \omega[[\text{Deriv}(T_2)]])$.
Of note are the 2 base cases:
If $\omega[[T]] = \text{SLeaf}(\text{“}R(s)\text{”})$, then $\omega[[\text{Deriv}(T)]] = \text{SLeaf}(\text{“}R(s)'\text{”})$. In other words, if we had a real variable x, its derivative would simply be x'.
If $\omega[[T]] = \text{RLeaf}(r)$, then $\omega[[\text{Deriv}(T)]] = \text{RLeaf}(0)$. In other words, the derivative of a real constant is 0.

Next, we define the semantics of $E[T]$ where T is a tree term.
If $\omega[[T]] = \text{SLeaf}(\text{“}R(s)\text{”})$, then $\omega[[E[T]]] = \omega[[R(s)]]$
If $\omega[[T]] = \text{RLeaf}(r)$, then $\omega[[E[T]]] = \omega[[r]]$
If $\omega[[T]] = \text{Plus}(T_1, T_2)$, then $\omega[[E[T]]] = \omega[[E[T_1]]] + \omega[[E[T_2]]]$
The same applies for times, divide, and negation. Note that derivative will not appear in $\omega[[T]]$ because of the way we defined the semantics (we simply compute the symbolic derivative).

The transition semantics for hybrid programs are defined the same way as in regular hybrid programs. For assignment, $(\omega, \omega^{\omega[[l]]}_{L(x)}) \in [[L(x) := l]]$ and $(\omega, \omega^{\omega[[r]]}_{R(x)}) \in [[R(x) := r]]$. The transition semantics of the other constructs remain the same.

4.3 Equality

We define two types of equality for symbol trees, $=$ and $\equiv$. $T_1 = T_2$ is stronger and means that the two trees are syntactically identical. $T_1 \equiv T_2$ means that the two trees mean the same thing. Given two symbol trees T_1 and T_2 in the metatheory, we defined $T_1 = T_2$ by structural induction. Example rules are shown below.

$$\frac{T_1 = T_1' \quad T_2 = T_2'}{\text{Times}(T_1, T_2) = \text{Times}(T_1', T_2')} \qquad \frac{T_1 = T_1' \quad T_2 = T_2'}{\text{Plus}(T_1, T_2) = \text{Plus}(T_1', T_2')}$$

$$\frac{T_1 = T_1' \quad r = r' \quad r, r' \neq 0 \in \mathbb{R}}{\text{Divide}(T_1, r) = \text{Divide}(T_1', r')}$$

We also have reflexivity, symmetry, and transitivity of $=$.

Note, however, that $\text{Plus}(\text{RLeaf}(0), \text{RLeaf}(1)) \neq \text{Plus}(\text{RLeaf}(1), \text{RLeaf}(0))$. For that, we need another notion of equality. We define $T_1 \equiv T_2$ using structural induction, and it has all the same rules as $=$, but more rules. In particular, the rules capture things like commutativity of Plus, Times, associativity, etc. We do not list all the rules, but give some examples,

$$\frac{}{\text{Times}(T_1, T_2) \equiv \text{Times}(T_2, T_1)} \qquad \frac{}{\text{Plus}(T_1, T_2) \equiv \text{Plus}(T_2, T_1)}$$

$$\frac{}{\text{Times}(T_1, \text{RLeaf}(0)) \equiv \text{RLeaf}(0)} \qquad \frac{}{\text{Plus}(T_1, \text{RLeaf}(0)) = T_1}$$

$$\frac{}{\text{Deriv}(\text{Plus}(T_1, T_2)) \equiv \text{Plus}(\text{Deriv}(T_1), \text{Deriv}(T_2))}$$

Now, given 2 symbol tree terms, T_1, T_2, we say that $\omega \vdash (T_1 = T_2)$ iff $\omega[[T_1]] = \omega[[T_2]]$, where we use the metatheory definition of equality for the latter equality. Similarly, we say that $\omega \vdash (T_1 \equiv T_2)$ iff $\omega[[T_1]] \equiv \omega[[T_2]]$. We can derive rules that are pretty much identical to the semantics in the metatheory, for example,

$$\frac{\gamma \vdash T_1 = T_1' \quad \gamma \vdash T_2 = T_2'}{\gamma \vdash \text{Times}(T_1, T_2) = \text{Times}(T_1', T_2')} \qquad \frac{\gamma \vdash T_1 = T_1' \quad \gamma \vdash T_2 = T_2'}{\gamma \vdash \text{Plus}(T_1, T_2) = \text{Plus}(T_1', T_2')}$$

$$\frac{}{\vdash \text{Times}(T_1, \text{RLeaf}(0)) \equiv \text{RLeaf}(0)} \qquad \frac{}{\vdash \text{Plus}(T_1, \text{RLeaf}(0)) = T_1}$$

4.4 Evaluation

The main power of the $\equiv$ relation is it gives us the following rule,

$$\frac{\gamma \vdash T_1 \equiv T_2}{\gamma \vdash E[T_1] = E[T_2]}$$

This can be shown by structural induction on the semantics of $\equiv$. The cases are fairly straightforward, for each case we simple use the semantics of E.

4.5 Derivative

The usual dI rule can be applied in our extension. However, we note an additional definition,

$$(E[T])' = E[\text{Deriv}(T)]$$

The proof that dI is still sound with this additional definition is beyond the scope of this work.

4.6 Application: Taylor Series

Armed with the power of symbol trees, we now return to our original quest of proving that $e^x > 1 + x + x^2/2 + ... + x^i/i!$ for all i and $x \geq 0$. Our model is as follows:

$$
\begin{array}{l}
R(x) = 0, R(y) = 1, R(i) = 0, \\
S(s) = \text{RLeaf}(0), S(t) = \text{RLeaf}(1) \\
\vdash \\
[(S(s) := \text{Plus}(S(s), S(t)); \\
\quad R(i) := R(i) + 1; \\
\quad S(t) := \text{Divide}(\text{Times}(S(t), \text{SLeaf}(\text{“}R(x)\text{”})), \text{RLeaf}(R(i))))^*] \\
[(R(y)' = R(y), R(x)' = 1)] \\
E[S(s)] \leq R(y)
\end{array}
$$

Our model first constructs the Taylor expansion $1 + x + x^2/2 + ... + x^i/i!$, then runs the differential equation, and compares the evaluated value of the Taylor expansion with the value of y. Our syntax is a little messy, so we explain the how we construct the Taylor Series. Basically, we are setting s, which is a symbol tree, to be $s + t$, where s and t are symbol trees. This adds the latest term $x^i/i!$ to the Taylor series sum. Then, we set $t = tx/i$, where t is a symbol tree, x is a variable, and i is a real number. This gives us the next term in the Taylor Series sum.

A full formal proof is beyond the scope of this work, but we sketch out the main ideas. These main ideas can be transformed into a proper proof where we apply rules. First,

we apply the loop induction rule. Our loop invariant is as follows:

$$
\begin{aligned}
&[(R(y)' = R(y), R(x)' = 1)]E[S(s)] \leq R(y) \;\&\\
&R(i) \geq 0 \;\&\\
&((i = 0 \;\&\; E[S(t)] = 1) \mid (i > 0 \;\&\; E[S(t)] = 0) \;\&\\
&((i = 0 \;\&\; E[S(s)] = 0) \mid (i > 0 \;\&\; E[S(s)] = 1) \;\&\\
&\text{Deriv}(\text{Plus}(S(s), S(t))) \equiv S(s) \;\&\\
&\text{Deriv}(\text{Divide}(\text{Times}(S(t), \text{SLeaf}(\text{“}R(x)\text{”})), \text{RLeaf}(R(i) + 1))) \equiv S(t)
\end{aligned}
$$

The first predicate is simply the postcondition. The second predicate, stating that $i \geq 0$ is needed to guarantee that we never divide by 0. The third predicate says that if I evaluate the Taylor series term t given the preconditions, then its 1 in the first iteration of the loop, and 0 after. This is because in the preconditions $x = 0$, and all terms of the Taylor series except the first have x in them (the terms are 1, x, $x^2/2$, ...), and are therefore 0. The fourth predicate says that if I evauate the Taylor series s given the preconditions, then its 0 in the first iteration (because $s = 0$) and 1 after (because $s = 1 + x + x^2/2 + ...$). The fifth predicate says that if I take a partial Taylor series s, and add the next term of the Taylor series t, and take the derivative, I get back s. This is a property of the Taylor series for e^x. The last predicate basically says that $(tx/(i + 1))' = t$.

We note that the loop invariant implies the postcondition since the loop invariant contains the postcondition.

We start by showing that the precondition implies the loop invariant. We omit the second, third, and fourth predicate, which are fairly easy to show using the rules for evaluation E. For the third predicate, we want to show, $\text{Deriv}(\text{Plus}(\text{RLeaf}(0), \text{RLeaf}(1))) \equiv \text{RLeaf}(0)$. We apply axioms for $\equiv$ to get that $\text{Deriv}(\text{Plus}(\text{RLeaf}(0), \text{RLeaf}(1))) \equiv \text{Plus}(\text{Deriv}(\text{RLeaf}(0)), \text{Deriv}(\text{RLeaf}(1)))$. We then get that this is $\equiv \text{Plus}(\text{RLeaf}(0), \text{RLeaf}(0)) \equiv \text{RLeaf}(0)$. Basically, we repeatedly apply the rules for $\equiv$ that describe when symbol trees are equivalent. We can do the same for the third predicate. For the first predicate we want to show that the preconditions imply $[(y' = y, x' = 1)]E[\text{RLeaf}(0)] \leq R(y)$. We first apply the rules for E in predicates. So it suffices to show that the preconditions imply $[(y' = y, x' = 1)]0 \leq R(y)$. This is a standard differential ghost proof.

Next, we show that the loop invariant is maintained by the updates. The second, third, fourth predicates are easy to show using the rules for evaluation E. The fifth and sixth predicate just involve the rules for $\equiv$, where we use the fact that $R(i) \geq 0$ so $R(i) + 1 > 0$. The most interesting part is to show the first predicate,

$[(R(y)' = R(y), R(x)' = 1)]E[S(s)] \leq R(y)$. Given the inductive hypothesis, we want to show that, $[(R(y)' = R(y), R(x)' = 1)]E[\text{Plus}(S(s), S(t))] \leq R(y)$. We first cut in the first predicate from the inductive hypothesis into the differential equation. Then we apply dI (see the augmented version of dI in the previous subsection). We then use the fifth and sixth predicate, with the rules for $\equiv$, to show that the claim holds.

5 Lists

5.1 List Terms

Given a real term r, and list l, a list-term is defined inductively as:

$$L(x) \mid \text{nil} \mid \text{cons}(r, l)$$

In the meta-theory, a list will also be defined inductively. A list is the smallest set of elements closed under the following rules:

$$\frac{}{\text{nil list}} \qquad \frac{r \in \mathbb{R} \quad l \text{ list}}{\text{cons}(r,\ l) \text{ list}}$$

Then, we define the semantics of list terms inductively as follows:

$$\omega[[L(x)]] = \omega(L(x))$$

$$\omega[[\text{nil}]] = \text{nil}$$

$$\omega[[\text{cons}(r, l)]] = \text{cons}(\omega[[r]], \omega[[l]])$$

We might also add operators "first" and "tail" to list terms, which give us the real term and the tail list respectively. Together, they serve as the inverse of "cons".

5.2 Hybrid Program Syntax

We augment the syntax of hybrid programs to support assignments of list-terms. In particular, we support two kinds of assignments, where l is a list term and r is a real term,

$$L(x) := l$$

$$R(x) := r$$

The transition semantics are defined the same way as in regular hybrid programs. That is, $(\omega, \omega^{\omega[[l]]}_{L(x)}) \in [[L(x) := l]]$ and $(\omega, \omega^{\omega[[r]]}_{R(x)}) \in [[R(x) := r]]$. The transition semantics of the other constructs remain the same.

5.3 Formulae

We augment both the first order formulae (that we can use inside hybrid programs inside conditionals) and dL formulae, to contain an inductively defined predicate. If f_1 is a formula, and l is a list, then $\text{pred}(f_1, l)$ is also a formula. We have the following associated inductive rule,

$$\frac{\mu \vdash [r/R(x)]f_2, \gamma \quad \mu \vdash \text{pred}(f_1, l), \gamma}{\mu \vdash \text{pred}(f_1, \text{cons}(r, l)), \gamma}$$

This allows us to, for example, check that all the elements of the list are greater than 0. We can also introduce inductive predicates that operate over multiple lists to, for example, check if two lists are equal, and to have predicates that allow us to check if a list is sorted.

6 Conclusion

In this work, we proposed 2 inductive extensions to dL. I particularly like the symbol tree direction, because I spent many hours trying to do certain proofs in dL (e.g. the Taylor series proof), or trying to add rules to dL to make such proofs possible, but wasn't able to. I think this is a pretty cool way to enable such proofs.

A big TODO is to make this work more fleshed out.