Signal Temporal Logic Formalisation in Isabelle

Mark Chevallier

June 2023

Project Overview

- The goal of the project is to build a pipeline that implements provably correct logical constraints in deep learning via theorem proving.
- We are aiming to begin by formalising signal temporal logic (STL).
- Once formalised, we prove a loss function is sound with respect to it, and generate code reflecting the loss function in OCaml.

Which STL to formalise?

- STL is discussed in a variety of papers.
- Each paper has some slight variation in how it is defined.
- For example, in some papers the μ operator is defined using a predicate function, and in others a differentiable real function.
- We choose to formalise the version of STL discussed in the STLCG paper[1] as this paper explicitly deals with using STL in gradient descent based learning methods via computation graphs.

[1] Leung, K., Aréchiga, N., & Pavone, M. (2020). Backpropagation through signal temporal logic specifications: Infusing logical structure into gradient-based methods. The International Journal of Robotics Research, 0278364921082115.

State Representation

- A signal is a measure of some state and how it varies over continuous time.
- We represent the state in Isabelle by using a type variable `'v`.
- We previously asserted this type variable must fulfil the obligations of a `real_vector` type class.
- We weren't using these properties for anything so I have dropped them for the moment.
- The precise type used for the state is left general. This could be interpreted as the real tensor type once it is formalised.

Signal Representation

- The state is sampled at certain times and these samples, taken together, form the signal.
- In Isabelle, we represent the signal as `(real x 'v) list`.
- In other words, a list of 2-tuples (t, s) where t represents the time the state was sampled, and s the sample itself.
- Note that this index of an element of the list does not necessarily represent its natural order in time.
- We define a predicate `valid_signal` which is true for a signal `S` if and only if the first element of the tuples within it are distinct (ie, each time is sampled only once). Considering insisting it is sorted too.

STL deep embedding

A **deep embedding** here is a formal representation of a logical system as a type. This allows us to reason about soundness, completeness etc. as well as efficiently generate code using the type to build constraints.

- We formalise STL constraints as the type `'v constraint`, with four constructors taken from the STLCG paper:
- `cMu f r`, where `f` is a function from `'v` to `real` and `r` is a real number. There are no constraint parameters for this constructor – it represents the atomic proposition of an STL constraint.
- `cNot c`, where `c` is a `'v constraint`.
- `cAnd c1 c2`, where `c1` and `c2` are of the `'v constraint` type.
- `cUntil x y c1 c2`, where `c1` and `c2` are of the `'v constraint` type, and both `x` and `y` are real numbers.
- We do not define the True constructor used in the paper, as this can be represented by `cMu (λx. 1)` $\sqsubset$ 0.

Evaluation of an STL constraint over a signal

We build an `evals p t c` function in Isabelle that will evaluate the truth value of a constraint `c` over signal `t` starting at time point `p`.

- `evals` is defined over each constructor of the `'v constraint` type separately, using the definitions of the STLCG paper.
- Thus, `evals p t (cMu f r)` is true if and only if there is a sample state `v` in signal `t` at point `p` such that `f v > r`.
- `evals p t (cNot c) = ¬(evals p t c)`
- `evals p t (cAnd c1 c2) = ((evals p t c1) ∧ (evals p t c2)`
- `evals p t (cUntil x y c1 c2)` is true if and only if there exists a sample at some time `t'` such that `t'`$\geq$`p+x` $\wedge$ `t'`$\leq$`p+y`, with `evals t' t c2`. For every sampled time `t''` such that `t''`$\geq$`p` $\wedge$ `t''<t'`, `evals t' t c1` must hold. (It is defined differently, recursively, but we prove equivalence.)

Proving STL equivalencies

Using the definitions for extended logical operators given in the STLCG paper, we prove that the definitions are equivalent to the normal understanding of the logical operator.

- `cTrue` is defined as `cMu (λx. 1) 0`. We show it is always evaluated as true at any sampled point in time.
- `cOr c1 c2` is defined as `cNot (cAnd (cNot c1) (cNot c2))`. We show it is true if and only if either `c1` or `c2` are.
- `cEventually x y c` is defined as `cUntil x y cTrue c`. We prove that it is true if and only if there exists a sample at some time between `x` and `y` such that `c` holds.
- `cAlways x y c` is defined as `cNot (cEventually x y (cNot c))`. We prove that it is true if and only if, for all samples between `x` and `y`, `c` holds.

Robustness

We define a robustness function `robust p t c` that will evaluate the robustness of a constraint `c` over signal `t` starting at time point `p`.

- Again we follow the logic given in the STLCG paper – except for the `cUntil` operator as we cannot define a procedure akin to that described for that operator.
- Instead, we recurse down the signal using logic similar to that used in our implementation of LTL.
- Complicating this recursion is the fact that the list we are using is not necessarily ordered by time.
- For $\text{Until}_{[x,y]}\ \rho\ \tau$ at time p, every entry in the signal is checked to see if τ is true at a time between x and y. If so, we check the entire list to verify ρ is true between p and the found time, inclusive.
- There are two main issues with this: first, it's very computationally expensive; secondly, it's not differentiable as we use a predicate function that cannot be converted to use the robust function easily.

Soundness

We show the `robust` function is sound wrt the logic of the `evals` function.

- If the `robust` function returns a value of 0, we can say nothing about the truth value of `evals`. But if it is positive, `evals` must be true, and if negative, `evals` must be false.
- This is easily provable by induction over the `'v constraint` parameter for all operators except `cUntil`.
- Proving the soundness for this operator is more complicated but given our recursive definition over both `evals` and `robust`, works.

Next steps

- If we can assume an ordered signal, we can write the recursive Until more naturally.
- We wouldn't need to check the entire signal for each element within it when we think we might have verified the Until condition, but only the prior portion.
- If we can do it this way, we should be able to rewrite robust to be fully differentiable too.
- There are two ways to do this - either extend `valid_signal` to check for both distinct and ordered time indices on the signal, or create a Signal type that is the list type with the required ordered, distinct, index. This second option will only be needed if there are termination issues doing it the first way.