TLC in Coq

Jeremiah Griffin

11 March 2019

1 Introduction

This project is an implementation of the *Temporal Logic of Composable Distributed Components* (TLC) in Coq. The core of TLC is its assertion language and associated program logic. The assertion language includes simple terms of variables, constants, external functions, and function application, and assertions comprising predicates, propositional operators, first-order quantifiers, and temporal operators. The program logic is derived from the system LK sequent calculus and Manna and Pnueli's temporal logic, extended with domain-specific inference rules for reasoning about distributed components.

The Coq implementation of TLC has been under development for several months. A number of different embedding approaches have been attempted without success, including a variety of deep and shallow embeddings. Most attempts failed at the same point: evaluating component functions written in Coq to produce terms usable in the embedding. The current iteration seeks to solve that problem by extending the internal term language to an enriched lambda calculus capable of implementing the components directly. The purpose of this document is to describe the techniques used in this implementation of TLC.

The first phase of this project, for the purpose of this class, is the implementation of the enriched lambda calculus. It includes types for the syntax of terms and a subset of the assertion language, definitions of decidable equality on those types, and a semantics implementing an evaluation engine for terms. The interaction between terms and the logic is also explored via a sequent calculus extended with inference rules for atomic assertions.

The second phase is the implementation and partial verification of a single distributed component: the stubborn link. This involved extending the assertion language and logic to include temporal logic, and adding the program logic and types and constructors specific to it. The term language was extended with literal terms for unit, natural, and Boolean types. The constructors for these types were removed and the pattern matching facilities were changed to implement matching on literal terms. In addition to the temporal operators, the assertion language was extended with a separate constructor for predicate assertions.

The third phase is the refinement of the proof system. It changes the way in which proofs are written, reduces the dependency on Coq for proving assertions within the logic, and introduces Coq tactics for manipulating proofs. It refines the syntax and semantics of pattern matching expressions to enable case analysis in proofs. It also includes the addition of the lowering transformation and its inner restrict and push transformations. Finally, the functional definition of the second distributed component, the perfect link, is included. The previous two additions will serve as the basis for the next phase of the project: proving the perfect link.

In addition to the high-level overview of the implementation provided in this document, the source code is thoroughly commented.

2 Syntax

The combined syntax of the term and assertion languages is described as follows:

$$
\begin{array}{rcl}
\text{Constructor } c & := & \mathsf{CPair} \mid \mathsf{CLeft} \mid \mathsf{CRight} \mid \mathsf{CNil} \mid \mathsf{CCons} \\
 & \mid & \mathsf{CFLSend} \mid \mathsf{CFLDeliver} \\
 & \mid & \mathsf{CSLSend} \mid \mathsf{CSLDeliver} \mid \mathsf{CPLSend} \mid \mathsf{CPLDeliver} \\
\text{Literal } l & := & \mathsf{LUnit}\ u \mid \mathsf{LBoolean}\ b \mid \mathsf{LNatural}\ n \\
 & \mid & \mathsf{LOrientation}\ o \mid \mathsf{LPeriodic}\ pe \\
\text{Function } f & := & \mathsf{FEqual} \mid \mathsf{FNot} \mid \mathsf{FOr} \mid \mathsf{FSucc} \mid \mathsf{FAdd} \\
 & \mid & \mathsf{FConcat} \mid \mathsf{FCount} \mid \mathsf{FUnion} \mid \mathsf{FMap} \\
\text{Term } t & := & \mathsf{TFailure} \mid \#(i,j) \mid v \mid c \mid l \mid f \mid t\ \$\ t \mid \mathbf{fun}\ t \\
 & \mid & \mathbf{match}\ t\ \mathbf{with}\ Cs \\
\text{Cases } Cs & := & \{\{\ \}\} \mid \{\{\ C \mid .. \mid C\ \}\} \\
\text{Case } C & := & p \rightarrow t \\
\text{Pattern } p & := & \% \mid \# \mid c \mid p\ p \\
 & \mid & \mathsf{PLUnit}\ u \mid \mathsf{PLBoolean}\ b \mid \mathsf{PLNatural}\ n \mid \mathsf{PLSucc}\ p \\
 & \mid & \mathsf{PLOrientation}\ o \mid \mathsf{PLPeriodic}\ pe \\
\text{Predicate } P & := & \mathsf{False} \mid t = t \mid t \in t \mid t \subseteq t \mid \mathsf{correct}\ t \\
\text{Assertion } A & := & P \mid \neg A \mid A \land A \mid \forall v.A \\
 & \mid & \hat{\Box} A \mid \hat{\boxminus} A \mid \hat{\Diamond} A \mid \hat{\Diamond\!\!\!\!-} A \mid \bigcirc A \mid \ominus A \mid \circledS\ A
\end{array}
$$

The extensions to the untyped lambda calculus are clear from this grammar. Pattern matching is accomplished with *match* terms. From this term constructor, syntactic sugar can be defined to encode *let* and *if* terms.

External functions are represented by atomic *function* terms. These terms have special treatment within the evaluation engine described in the semantics section of this paper. Several derived functions are defined in terms of these core functions. Only the names of the functions are shown here; additional infix notations for functions are provided internally.

2.1 Free and bound variables

In addition to the described extensions to the untyped lambda calculus, this grammar uses the *locally nameless* representation of bound variables. Free variables are represented as *variable* terms, defined by their names as strings, and bound variables are represented as *parameter* terms, defined by their scope and binding indices. This representation is adopted from a paper by

Arthur Chargueraud, which presents an extension of the classical de Bruijn indexing approach to include multi-binders and named free variables.

This representation for bound variables was chosen for two reasons. First, substitution of free variables in this system is non-capturing. The namespace for free variables is global and independent of binders. This is important because the traditional capture-avoiding substitution algorithm does not satisfy Coq's requirements for structurally decreasing recursion. Second, syntactic equality of terms is equivalent to equality under α-conversion.

The *abstraction* and *match* terms are binders and create a new binder level for their inner terms. Abstraction binds a single parameter and matching binds as many parameters as there are bindings in the matched pattern. Binders are indexed starting at 0 for the closest binder. Bindings are indexed starting at 0 for the leftmost binding in its binder.

Parameters are accessed with the syntax $\#(i,\ j)$, referring to the jth binding in the ith enclosing binder. Within a function abstraction, typeset here at **fun** t, the parameter is accessed as $\#(0,\ 0)$, representing the leftmost binding in the closest binder: in this case, the abstraction. The syntactic sugar $\#j$ is defined as $\#(0,\ j)$, to access the jth binding in the immediate binder. For example, the identity function would be written **fun** $\#0$ and the constant function would be written **fun fun** $\#(1, 0)$.

Matching abstractions produce multiple bindings: one for each binding appearing in the pattern. In **match** t **with** $\{\{\ (\#, \#) \rightarrow \#0 + \#1\ \}\}$ the parameter $\#1$ refers to the right-hand binding within the matched pair. If the left-hand binding were instead a wildcard, as in the pattern $(\%,\ \#)$, the right-hand element of the pair would instead be accessed as $\#0$.

3 Semantics

The top-level semantics for terms manifest in two algorithms: variable substitution and term evaluation. Substitution is, as noted in the previous section, trivial in the locally nameless representation. The `instantiate_term` function takes an environment, which is a partial mapping from variables to terms, and substitutes all instances of free variables appearing in that environment with their associated terms. Related to variable substitution is the `term_free` function, which computes the set of free variables occurring in a term. This function is used to define the `is_term_closed` predicate, which is true when a term contains no free variables. These functions are also defined for predicates and assertions.

3.1 Evaluation

Evaluation is similar to traditional β-reduction of lambda calculus terms, with three changes: term opening, external functions, and pattern matching. The evaluation engine, called `evaluate_term`, takes a term and produces a result, which is a monad whose value is either success, containing the reduced term, or failure, containing an error status.

Internally the engine takes an additional parameter in the form of recursion fuel, a natural number that decreases with each recursive evaluation. This allows the evaluation engine to be implemented as a simple fixpoint in Coq despite the engine not strictly recursing on structurally smaller terms. If a term requires more than 5000 recursive calls to evaluate, the engine will fail with a fuel error. This limit can be raised by calling the internal function directly.

3.1.1 Term opening

Traditional β-reduction is defined in terms of capture-avoiding substitution. In the locally nameless representation, this operation is instead defined with term opening. A term $(\lambda.t_1)\ t_2$ is reduced by opening t_1 and replacing its parameter, $\#0$, with t_2. The term opening function, `open_term_at`, takes a binder depth k, where 0 constitutes the top level, a list of terms us to replace bindings with, and a term t to open. The function returns the opened term upon success.

The function recurses on t. If t is a parameter term $\#(i,j)$ and $i = k$, then the jth term in us is returned; if there is no such term in us, a parameter error is returned. If the binding depth is different, the term is returned unchanged. Failure, variable, constructor, and function terms are also returned directly. Application terms are opened by opening both subterms. Function abstractions are opened by opening the subterm at the next depth, $k + 1$. Matching abstractions are opened by opening the argument term at the same depth and the then- and else-terms at the next depth.

3.1.2 External functions

When an application term is evaluated, if the left-hand term does not evaluate to an abstraction term as in β-reduction, external function evaluation is attempted if both the left- and right-hand terms are closed. Terms matching the expected form of an external function, such as the application of two terms to the add function, are evaluated with specialized rules. Typically, this involves lifting the argument terms into Coq, performing some operation, then converting the result back into a term.

For example, the add function lifts its two arguments into Coq nat values, adds them, and returns the sum as an embedded natural term. The map function lifts its second argument into a Coq list of terms, applies its first argument to every term in the list, then returns the evaluation of the resulting list. This pattern of lift, compute, and convert is repeated in all of the evaluators for external functions.

The lifting operation is defined for each embedded type. For natural numbers this operation is called `lift_natural`. It takes a term and either returns its inner value if the term is a natural literal or fails if the term is not a natural literal. For lists this operation is called `lift_list` and is recursive. It takes a term and either returns the empty list if the term is a nil constructor, returns the cons of the head term and the lifted tail if the term is a cons constructor, or fails if the term is not a well-formed list constructor.

3.1.3 Pattern matching

Match terms are evaluated by evaluating their argument and performing pattern matching on the evaluated argument and the set of cases. The cases are matched in order. The first case with a pattern that matches correctly is then evaluated. If none of the cases match the argument, a matching error is produced.

Pattern matching is done by a separate algorithm, called `match_pattern`. This algorithm takes a pattern and a term and returns a result that contains a list of bound terms on success and a match error on failure. It accomplishes this by simultaneously matching on the pattern and the term. Wildcard patterns (%) match any term and produce an empty binding list. Binding patterns (#) match any term and produce a binding list containing the term.

Each constructor is implemented separately. For example, the unit constructor matches when both the pattern and the term are `⊤`. It produces an empty binding list. The pair constructor matches when both the pattern and the term are well-formed pairs. It matches the first term against the first pattern then the second term against the second pattern. If both sub-matchings were successful, it returns the concatenation of the first and second binding lists.

4 Logic

The logic is defined in the `derives` inductive relation. If an assertion A can be derived from a set of assumptions Γ with bound variables Δ for a component C then it is stated that $(\Delta, \Gamma) \vdash C, A$. The derives relation includes the standard inference rules from sequent calculus, temporal logic axioms and inference rules, axioms and inference rules from the program logic, and additional implementation-specific rules. The implementation-specific rules enable term evaluation and substitution, assertion rewriting, predicate rules, constructor injectivity rules, and case analysis rules.

Derived rules and lemmas for sequent, predicate, temporal, and program logics are provided in separate files. These lemmas, such as the sequent DSAssumption and DSModusPonensC lemmas, are built upon the basic axioms and rules defined in the derives relation. The temporal and program derived rules and lemmas are implemented directly from the TLC paper. The sequent and predicate derived rules and lemmas are made specifically to ease the writing of proofs in this embedded logic. They emulate behaviors available when writing proofs directly in Coq.

The pair (Δ, Γ) is referred to as the context of a proof. It can be thought of as an emulation of Coq's proof context, which consists of terms that may be either values or propositions. Without the benefit of a type system, it is necessary to separate untyped terms from assertions. Δ is a set of variable names that are universally quantified within the premises and conclusion of the proof. Γ is a set of assertions that are assumed to be true. Variables are introduced to the context by sequent logic rules for universal and existential quantifiers.

4.1 Rules

These rules are specific to this implementation of the logic. They largely exist to emulate the behavior of certain Coq tactics that are helpful for writing proofs.

- DAEvaluateP: Evaluates all terms in the head premise.
- DAEvaluateC: Evaluates all terms in the conclusion.
- DASubstituteP: Substitutes all term equalities that appear in the tail premises within the head premise.
- DASubstituteC: Substitutes all term equalities that appear in the premises within the conclusion.
- DARewriteIfP: Replaces all positive occurrences of A_p in the proven assertion $A_p \Rightarrow A_c$ with A_c within the head premise.
- DARewriteIfC: Replaces all positive occurrences of A_p in the proven assertion $A_p \Rightarrow A_c$ with A_c within the conclusion.
- DARewriteIffPL: Replaces all occurrences of A_{pl} in the proven assertion $A_{pl} \Leftrightarrow A_{pr}$ with A_{pr} within the head premise.
- DARewriteIffPR: Replaces all occurrences of A_{pr} in the proven assertion $A_{pl} \Leftrightarrow A_{pr}$ with A_{pl} within the head premise.
- DARewriteIffCL: Replaces all occurrences of A_{pl} in the proven assertion $A_{pl} \Leftrightarrow A_{pr}$ with A_{pr} within the conclusion.
- DARewriteIffCR: Replaces all occurrences of A_{pr} in the proven assertion $A_{pl} \Leftrightarrow A_{pr}$ with A_{pl} within the conclusion.
- DAPEqual: Proves that two terms are syntactically equal.
- DAPIn: Proves that a term is a member of a list term.
- DAPExtension: Proves that a list term is an extension of another list term.
- DADestructP: Destructs a match term within the head premise, generating case analysis assertions for each case.
- DADestructC: Destructs a match term within the conclusion, generating case analysis assertions for each case.

The rewriting rules for implication and bicondition are repeated for their strong temporal counterparts.

4.2 Tactics

These tactics are provided for working with the proof system in a more natural, Coq-like way. Not listed here are tactics for ergonomic use of the sequent rules and axioms.

- d_evalp, d_evalc, d_eval: Similar to Coq's simpl tactic.
- d_substp, d_substc, d_subst: Similar to Coq's subst tactic.
- d_destructp, d_destructc: Similar to Coq's case tactic.
- d_false: Immediately completes a proof when False is the head premise.
- d_clear: Removes the head premise from the context.
- d_swap: Exchanges the first two premises.
- d_have: Adds a premise to the context and generates a Coq goal for proving its truth.

5 Components

Components are currently defined as tuples of terms. At a conceptual level they contain other data, such as the types of states and events. However, due to the untyped nature of the term language, this information is not needed. Instead, components are defined by four function terms:

1. `initialize`: A function term taking a node and returning an initialized component state. This function is called when a node is initialized.
2. `request`: A function term taking a node, component state, and input request event and returning a tuple of updated component state, emitted output requests, and emitted output indications. This function is called when an input request is processed on a node.
3. `indication`: A function term taking a node, component state, and input indication event and returning a tuple of updated component state, emitted output requests, and emitted output indications.
4. `periodic`: A function term taking a node and component state and returning a tuple of updated component state, emitted output requests, and emitted output indications.

5.1 Stubborn Link

This component is implemented in this phase. It specifies two assertions that characterize the fundamental behavior of the component. The first is the stubborn delivery specification: if a correct node n sends a message m to a correct node n', then m will be delivered to n' infinitely often. The second is the no-forge specification: if a message m is delivered to a node n from node n', then m was previously sent to n by n'. Both of these properties are partially proven true.

6 Usage

To use the features implemented in this phase, build and install the Coq modules with the included makefile and set up a Coq file with the following prelude:

```
Require Import mathcomp.ssreflect.all_ssreflect.
Require Import tlc.all.

Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

Alternatively, simply build the modules with make and open the `derives.v` file in CoqIDE. This file is at the top of the module hierarchy and can be used to experiment with the evaluation engine and logic by placing commands at the bottom of the file.

To evaluate a term, the [[tt]] notation can be used:

```
Eval compute in [[t
  (fun: match: #0 with: {{ #.+1 -> #0 | % -> 0}}) <$> [0, 1, 2, 3]
]].
(* Output:
= Success
(CCons $ 0 $
 (CCons $ 0 $
  (CCons $ 1 $
   (CCons $ 2 $
    CNil))))%term
: result error term *)

Eval compute in [[
  (if: 0 \in [1, 2] then: 0 else: 1, 0 = 1)
]].
(* Output:
= Success (CPair $ 1 $ false)%term
: result error term *)
```

To enter proof mode on an assertion within the logic, the $(\Delta, \Gamma) \vdash C, A$ notation can be used:

```
Example example1 C :
  Context [::] [::] |- C, {A: ATrue \/ AFalse}.
Proof.
  d_left. d_notc. d_head.
Qed.

Example example2 C :
  Context [::] [::] |- C, {A: x = 5 -> "x" + "x" = 10}.
Proof.
  d_ifc. d_substc. d_evalc. by exact: DAPEqual.
Qed.
```