Topics

This part consists of various topics that don't fit neatly into the existing structure. As the hyperbook evolves, the sections in this part will be moved elsewhere.

18 Variable Hiding and Auxiliary Variables

A specification usually has some "visible" variables that describe real parts of the system that must be implemented and other "internal" variables that are used only to help describe the behavior of the visible variables. In a philosophically correct specification, the internal variables should be hidden.

In TLA$^+$, variables are hidden using the temporal existential quantification operator $\exists\!\!\exists$. For any temporal formula F, the formula $\exists\!\!\exists\, x : F$ essentially asserts that there is some way of choosing values of x that makes F true. The operator $\exists\!\!\exists$ is temporal existential quantification. It differs from the operator $\exists$ of ordinary (non-temporal) logic because $\exists\!\!\exists\, x : F$ asserts not that there is a single value of x that makes F true, but rather a sequence of values—one for each state of the behavior. The $\exists\!\!\exists$ operator is defined precisely in Section 16.2.4 of *Specifying Systems*.

Engineers are not interested in philosophical correctness, so they don't worry about hiding variables. All you probably need to know about the operator $\exists\!\!\exists$ is that it obeys most of the rules of the existential operator $\exists$ of ordinary math. In particular, we prove $(\exists\!\!\exists\, x : F) \Rightarrow G$ and $F \Rightarrow (\exists\!\!\exists\, x : G)$, for temporal formulas F and G, the same way we prove the corresponding formulas for ordinary existential quantification:

1. If x is not a free variable of G, then $F \Rightarrow G$ implies $(\exists\!\!\exists\, x : F) \Rightarrow G$.

2. If $\overline{G}$ is the formula obtained from G by substituting the expression $\overline{x}$ for x, then $F \Rightarrow \overline{G}$ implies $F \Rightarrow (\exists\!\!\exists\, x : G)$.

If F and G are specifications, we restate the second rule as follows: to show that F implies $\exists\!\!\exists\, x : G$, it suffices to show that F implements G under a refinement mapping$^{\square}$ that is the identity on all variables of G except x—that is, a refinement mapping such that $\overline{v} = v$ for all variables v of G other than x.

The concept of variable hiding plays another role in reasoning about specifications. Sometimes, we can't prove correctness as implementation of a higher-level specification because the implementation state doesn't contain enough information to define the necessary refinement mapping. We solve this problem by adding *auxiliary variables* (sometimes called *dummy variables*) to the implementation. Adding an auxiliary variable a to a specification S means writing a new spec S^a containing the additional variable S such that, hiding a in S^a produces a specification equivalent to S. More precisely, S^a is obtained from S by adding the auxiliary variable a iff $\exists\!\!\exists\, a : S^a$ is equivalent to S. We prove correctness of S by showing that S^a implements the desired higher-level specification under a refinement mapping.

The most common form of auxiliary variable is a *history* variable. A history variable essentially records information about what happened earlier in an execution, without altering the values that other variables assume. There's an easy way to add a history variable h to an existing PlusCal algorithm—assuming h is not already a variable of the algorithm. Add the declaration of h and assignment statements of the form $h := \ldots$. As long as you don't add or remove labels or add an h to any existing part of the code, the TLA$^+$ translation of the new algorithm will be obtained by adding the history variable h to the translation of the original algorithm.

TLC does not handle properties that contain the $\exists$ operator. The TLAPS proof system might some day handle them, but it doesn't now. Also, note that the TLA$^+$ formula $\exists\, x : F$ is not legal in any context in which the variable x is already declared. This means that if *Spec* is defined in a module M to be a specification containing the variable x, then $\exists\, x : Spec$ is not a legal formula in that module. To hide x in *Spec*, we have to create a separate module containing a statement such as

$$H(y) \triangleq \text{INSTANCE } M \text{ WITH } x \leftarrow y$$

and write $\exists x : H(x)!Spec$ in that module.

19 Reduction

Reduction is a concept introduced in

> Richard J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Communications of the ACM*, 18(12):717-721, December 1975.

This section will be based on this paper.

20 Debugging With TLC

This section is about how to locate the source of the problem when TLC reports an error or is having a performance problem. A TLC performance problem is when TLC takes a long time to evaluate an expression. This can manifest itself by TLC taking a long time to compute new reachable states, right from the beginning. (However, it's normal for TLC's rate of generating new states to decrease dramatically after it has found enough states that it must start writing their fingerprints onto disk.)

The standard *TLC* module provides some special operators that are useful for debugging a specification. Some of them are outside the domain of mathematics and should not be used as part of a specification.

20.1 Print Statements

The standard *TLC* module defines two operators that cause TLC to print something when it evaluates them:

- The *PrintT* command is defined so $PrintT(o)$ is equal to TRUE, but evaluating it causes TLC to print the value o.
- The *Print* command is defined so $Print(v, o)$ equals v, but evaluating it causes TLC to print the values e and o.

These command are useful if TLC reports an error when evaluating an expression but does not report the precise location where the error occurred. For example, suppose the error occurs when TLC is evaluating a conjunction but the error message does not tell you in which conjunct the error occurs. TLC evaluates the conjuncts from "left to right". Inserting additional conjuncts $PrintT(\text{“a”})$, $PrintT(\text{“b”})$, etc. into the formula and seeing which ones were executed reveals where the error occurred.

PrintT conjuncts can be used this way even if there is no error, but the conjunction evaluates to FALSE and you want to find out which conjunct is false. The "conjunction" can also be a $\forall$ formula. For example, if a formula $\forall x \in S : P(x)$ is false and you want to find out for what values of x the expression $P(x)$ is false, you can rewrite that formula as

$$\forall x \in S : \text{IF } P(x) \text{ THEN TRUE ELSE } PrintT(x)$$

20.2 Having TLC Set and Read Values

TLC can set and read a special list of values while evaluating expressions. This works as follows. The *TLC* module defines two new operators:

$$TLCGet(i) \triangleq \text{CHOOSE } n : \text{TRUE}$$
$$TLCSet(i, v) \triangleq \text{TRUE}$$

When TLC evaluates $TLCSet(i, v)$, for any positive integer i and arbitrary value v, in addition to obtaining the value TRUE, it sets the i^{th} element of the list to v. When TLC evaluates $TLCGet(i)$, the value it obtains is the current value of the i^{th} element of this list. For example, when TLC evaluates the formula

$$\begin{aligned}&\land\ TLCSet(42, \langle \text{“a”}, 1\rangle)\\ &\land\ \forall i \in \{1,2,3\} : \land\ PrintT(TLCGet(42))\\ &\qquad\qquad\land\ TLCSet(42, [TLCGet(42)\ \text{EXCEPT}\ ![2] = TLCGet(42) + 1])\end{aligned}$$

it prints

```
<< "a", 1 >>
<< "a", 2 >>
<< "a", 3 >>
```

One use of this feature is to check TLC's progress during long computations. For example, suppose TLC is evaluating a formula $\forall x \in S : P$ where S is a large set, so it evaluates P many times. You can use *TLCGet*, *TLCSet*, and *Print* to print something after every 1000^{th} time TLC evaluates P.

Another use of *TLCGet* and *TLCSet* is to measure the length of time it takes TLC to evaluate an expression. The *TLC* module defines the operator *JavaTime* to be an unspecified integer. TLC evaluates it to equal approximately the number of milliseconds between 0:00 UTC on 1 January 1970 and the current time. Using the fact that TLC evaluates tuples from left to right, you can measure the approximate time in milliseconds taken to evaluate an expression e by replacing e with:

$$\langle TLCSet(1, JavaTime), e, PrintT(JavaTime - TLCGet(1))\rangle[2]$$

As explained in Section 20.4 below, you may also want to use *TLCSet* and *TLCGet* to count how many times TLC is evaluating an expression e. To use value number i as the counter, just replace e by

$$\text{IF}\ TLCSet(i,\ TLCGet(i) + 1)\ \text{THEN}\ e\ \text{ELSE}\ 42$$

(The ELSE expression is never evaluated.) Have TLC execute $TLCSet(i, 0)$ before execution and $PrintT(TLCGet(i))$ afterwards to print the number of executions of e.

For reasons of efficiency, *TLCGet* and *TLCSet* behave somewhat strangely when TLC is run with multiple worker threads (using the `-workers` option). Each worker thread maintains its own individual copy of the list of values on which it evaluates *TLCGet* and *TLCSet*. The worker threads are activated only after the computation and invariance checking of the initial states. Before then, evaluating $TLCSet(i, v)$ sets the element i of the list maintained by all threads. Thus, the lists of all the worker threads can be initialized by putting the appropriate *TLCSet* expression in an ASSUME expression or in the initial predicate.

20.3 Using LET

If multiple instances of the same subexpression occur in an expression e, TLC will evaluate that subexpression multiple times when evaluating e. This multiple evaluation can be avoided by using a LET to replace those instances by a single symbol. For most specifications, the evaluation of an individual expression is responsible for only a small part of the execution time. It's therefore generally best to use a LET only to make the specification easier to read, not to optimize execution speed. However, occasionally the use of a LET can significantly reduce execution time. This is particularly true for the evaluation of recursively defined operators.

As an example, consider this definition of the transitive closure of a relation from Section 9.6.2.

$$
\begin{array}{l}
SimpleTC(R) \triangleq \\
\quad \text{LET RECURSIVE } STC(\_) \\
\qquad STC(n) \triangleq \text{IF } n = 1 \text{ THEN } R \\
\qquad\qquad\qquad\qquad \text{ELSE } STC(n-1) \cup STC(n-1) \mathbin{**} R \\
\quad \text{IN} \quad \text{IF } R = \{\} \text{ THEN } \{\} \text{ ELSE } STC(Cardinality(NodesOf(R)))
\end{array}
$$

To evaluate $STC(n)$, TLC evaluates $STC(n-1)$ twice. If $n-1 > 1$, then each of those evaluations evaluates $STC(n-2)$ twice. This doubling of effort continues down to the evaluation of $STC(1)$, which requires evaluating the argument R. Hence, to evaluate $STC(n)$, TLC evaluates R about 2^n times. To evaluate $SimpleTC(R)$, TLC evaluates $STC(n)$ for $n = Cardinality(NodesOf(R))$. Depending upon R, this evaluation starts taking a few seconds for n between 10 and 15. For the particular application for which the definition was written, that's probably good enough. But if it isn't, we can replace the subexpression $STC(n-1) \cup STC(n-1) \mathbin{**} R$ with

$$
\begin{array}{l}
\text{LET } STCN \triangleq STC(n-1) \\
\text{IN} \quad STCN \cup STCN \mathbin{**} R
\end{array}
$$

Suppose we had written the definition of $SimpleTC$ with a recursively defined function instead of an operator—like this:

$$
\begin{array}{l}
SimpleTC(R) \triangleq \\
\quad \text{LET } FTC[n \in Nat] \triangleq \text{IF } n = 1 \text{ THEN } R \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } FTC[n-1] \cup FTC[n-1] \mathbin{**} R \\
\quad \text{IN} \quad \text{IF } R = \{\} \text{ THEN } \{\} \text{ ELSE } FTC[Cardinality(NodesOf(R))]
\end{array}
$$

Now, rather than computing the n completely separate values $STC(n)$, $STC(n-1)$, . . . , TLC is computing a single value—the function FTC. A naive evaluation would yield the same exponential blow-up, evaluating $FTC[n]$ once, $FTC[n-1]$

twice, $FTC[n-2]$ four times, ... and evaluating $FTC[1]$ about 2^n times. However, when TLC computes the value $FTC[i]$ for some i, it caches that value. Hence, there is no exponential blow-up. Using a LET expression to eliminate the two instances of $FTC[n-1]$ in the definition would accomplish nothing.

This example suggests that recursive function definitions should be preferred to recursive operator definitions. That's true if the domain of the function can be made simple (e.g., *Nat*) and you are writing the spec for readers who understand the TLA$^+$ syntax for recursive function definitions. However, I expect that a recursive operator definition will be easier to understand for a reader who is unfamiliar with TLA$^+$.

Note: TLAPS cannot yet handle recursive operator definitions. Therefore, if you want to write TLAPS-checked proofs, you can use only recursive function definitions.

20.4 The Perils of Lazy Evaluation

The obvious way to compute the value of an expression like $F(a, b)$ is to first compute a and b. TLC does not always do this. If $F(a, b)$ were defined to equal $a \in b$, then TLC can evaluate $F(42, \{x \in Nat : x > 37\})$ even though it can't compute the (infinite) set $\{x \in Nat : x > 37\}$. To accomplish this, TLC sometimes does what is known as *lazy evaluation*: not completely evaluating an expression until it has to.

TLC uses heuristics to determine whether it should completely evaluate an expression. Its heuristics work well most of the time. However, sometimes lazy evaluation can result in the expression ultimately being evaluated multiple times instead of just once. This can especially be a problem when evaluating a recursively defined operator. For example, consider this definition of the transitive closure of a relation from Section 20.3 above.

$$
\begin{array}{l}
\mathit{TransitiveClosure}(R) \triangleq \\
\quad \text{LET RECURSIVE } \mathit{STC}(\_) \\
\qquad \mathit{STC}(n) \triangleq \text{IF } n = 1 \text{ THEN } R \\
\qquad\qquad\qquad\qquad\qquad \text{ELSE LET } \mathit{STCN} \triangleq \mathit{STC}(n-1) \\
\qquad\qquad\qquad\qquad\qquad\quad \text{IN } \mathit{STCN} \cup \mathit{STCN} \mathbin{**} R \\
\quad \text{IN IF } R = \{\} \text{ THEN } \{\} \text{ ELSE } \mathit{STC}(\mathit{Cardinality}(\mathit{NodesOf}(R)))
\end{array}
$$

We would expect that, to evaluate $STC(n)$, TLC would evaluate the expression $STCN \cup STCN \mathbin{**} R$ only n times, leading to $n-1$ evaluations of $\cup$ and $**$. That is indeed what happens with the definition in Section 9.6.2$^{\square}$. However, if $**$ is defined as in Question 9.7$^{\square}$, then TLC lazily evaluates this expression. To see the effect of this, consider how TLC evaluates $STC(4)$. Because of lazy evaluation, it obtains the value

$$STC(3) \cup STC(3) \mathbin{**} R$$

using the value it computed for $STC(3)$. What value did it compute for $STC(3)$? Again, because of lazy evalution, TLC evaluated $STC(3)$ to be

$$STC(2) \cup STC(2) \mathbin{**} R$$

so the value of $STC(3)$ it obtained is actually

$$(STC(2) \cup STC(2) \mathbin{**} R) \cup (STC(2) \cup STC(2) \mathbin{**} R) \mathbin{**} R)$$

Similarly, it computed $STC(2)$ to equal

$$STC(1) \cup STC(1) \mathbin{**} R$$

which equals

$$R \cup R \mathbin{**} R$$

So the actual value of $STC(4)$ it obtained is

$$((R \cup R \mathbin{**} R) \cup (R \cup R \mathbin{**} R) \mathbin{**} R) \cup ((R \cup R \mathbin{**} R) \cup (R \cup R \mathbin{**} R) \mathbin{**} R) \mathbin{**} R)$$

Instead of 3 evaluations of $\cup$ and $**$, it performed 7. It's not hard to see that to evaluate $STC(n)$, TLC evalutes $\cup$ and $**$ 2^{n-1} times instead of $n-1$ times.

To allow you to solve this problem, the *TLC* module provides the *TLCEval* operator. It defines *TLCEval* by

$$TLCEval(x) \triangleq x$$

However, TLC evaluates the expression $TLCEval(e)$ by completely evaluating e. For the definition of transitive closure above, TLC's lazy evaluation can be prevented by using *TLCEval* as follows.

```
TransitiveClosure(R) ≜
  LET RECURSIVE STC(_)
      STC(n) ≜ IF n = 1 THEN R
                        ELSE LET STCN ≜ STC(n − 1)
                                 IN   TLCEval(STCN ∪ STCN ** R)
  IN   IF R = {} THEN {} ELSE  STC(Cardinality(NodesOf(R)))
```

If TLC is taking a long time to evaluate something, you can check if lazy evaluation is the source of the problem by using the *TLC* module's *TLCSet* and *TLCGet* operators to count how many times expressions are being evaluated, as described above.