An Automated Verification of Property TP2 for Concurrent Collaborative Text Buffers (Research Report CS-2012-25.)

Brad Lushman Adam Roegiest
University of Waterloo
{bmlushma, aroegies}@uwaterloo.ca

December 17, 2012

Abstract

Using the Coq proof assistant, we present a mechanically verified proof that the operation transforms defined on text buffer insertions and deletions, as outlined in Ressel et al's adOPTed algorithm do not satisfy Ressel's transformation property TP2. We then apply similar techniques to Cormack's Calculus for Concurrent Update (CCU) and show that the CCU transformations also do not satisfy TP2.

1 Introduction and Notation

Distributed collaborative editing of shared data, in which each participant maintains a local copy of the data, and in which there is no global reference copy, presents interesting challenges when it comes to ensuring that local copies remain consistent with one another. Consider, for example, a shared data item x, and sites A and B. Suppose sites A and B, generate, respectively, operations f and g, apply these operations to their local copy of x, and then broadcast the update to other sites. Site A applies f to x, and then receives update g from site B, so that it computes state $g(f(x))$. Site B applies g to x, and then receives update f from site A, so that site B computes state $f(g(x))$. Since $f(g(x)$ and $g(f(x))$ are not necessarily equal, it is clear from this example that concurrently-generated operations on a common state can easily lead to divergent local copies.

Operation transforms provide a potential means of mitigating this difficulty. Rather than apply operation g to $f(x)$, site A transforms g to take into account the fact that f has already been applied to x. Using CCU notation, site A computes the operation g/f, which represents a transformation of g, taking f into account. Site B computes a dual operation $f\backslash g$, which takes into account the fact that g has already been applied at B. Site A thus computes $(g/f)(f(x))$, while site B computes $(f\backslash g)(g(x))$. It is a defining property of / and \ that for all f, g, x, $(g/f)(f(x)) = (f\backslash g)(g(x))$.

Every pair of sites (S_i, S_j) has the property that S_i always uses / when transforming operations from S_j, while S_j always transforms operations from S_i using \ — or vice versa. Put otherwise, when any two sites exchange updates, one site uses /, while the other uses \. This requirement is easily satisfied if we impose some arbitrary global total order $<_S$ on sites, and then declare that a site uses / when transforming updates from higher (according to $<_S$) sites, and \ when transforming updates from lower (according to $<_S$) sites. To streamline the notation somewhat, we use the notation ˆ to mean either / or \, as appropriate. Using ˆ-notation, we say that site A computes $(g\hat{}f)(f(x))$, while B computes $(f\hat{}g)(g(x))$. The consistency property stated above, known as TP1, then becomes $(g\hat{}f)(f(x)) = (f\hat{}g)(g(x))$.

Unfortunately, when more than two participants are involved, TP1 alone is not sufficient to guarantee eventual convergence of state. A second condition from Ressel, known as TP2, has been shown to be sufficient to guarantee convergence, and is believed to be a necessary condition as well. Using ˆ-notation, TP2 requires that, for all updates f, g, and h, $(f\hat{}g)\hat{}(h\hat{}g) = (f\hat{}h)\hat{}(g\hat{}h)$.

The problem with TP2 is that it is a condition on triples of updates; verifying TP2 by hand therefore requires $O(d^3)$ effort, where d is the descriptional complexity (number of conditional cases) of the updates. Verification by hand is error-prone and takes considerable time. Nevertheless, it is essential to verify TP2 for a candidate set of updates, in order to ensure provable correctness of the groupware system.

In this paper, we explore the Coq proof assistant as a means of automatically verifying TP2 for a groupware system. The goal is twofold—to use Coq to verify or refute TP2 for two well-known sets of transformations for shared text buffer operations; and to explore the Coq tool itself, as an aid for future work. A short term goal is to have a tool for which we can quickly verify (or refute) TP2 for candidate sets of operation transforms. A longer term goal is to build this kind of verification into a programming environment or compiler. With this vision in mind, our goal is to make our verifications as general as possible, at the possible expense of efficiency. We will discuss efficiency concerns as they arise.

2 Text Buffer Transformations

The groupware system which forms the basis for our discussion is a shared text buffer—essentially a string, upon which users concurrently issue editing commands. The text buffer operations we consider are insertions and deletions. We must then define the semantics of transforming one operation against another. We consider two such formulations: the one presented as part of Ressel's adOPTed algorithm [8], and the one presented as part of Cormack's Calculus for Concurrent Update (CCU) [3].

2.1 adOPTed

Under Ressel's formulation, which is mostly based on that of Ellis and Gibbs [4], operations take the following forms:

- insert(p, c) — insert character c at position p;
- delete(p) — delete the character at position p;
- noop — do nothing.

The noop must be included as an operation because some combinations of operations, when transformed against one another, will produce noop; hence, noop is required in order that the set of operations be closed under transformation. The transformation rules for adOPTed apper in Figure 1.

2.2 CCU

Under CCU, operations take the following forms:

- insert(p, s) — insert string s at position p;
- delete(p, l) — delete l characters, starting at position p;

CCU deals with insertions and deletions of entire strings into the shared buffer, rather than individual characters. For this reason, a noop operation is unnecessary, as it may be modelled by either insert(p, “”) (for any p), or delete($p, 0$) (for any p). The transformation rules for CCU appear in Figure 2.

We can simplify the CCU operations somewhat by taking advantage of a result proved in [6] — if a set of operations can be modelled as compositions of a simpler set of operations, such that the simpler set is closed under transformation, then it suffices to check TP2 for the simpler set. Accordingly, we may restrict the CCU operations to single character insertions (as any multi-character insertion is simply a composition of single-character insertions); however, we may not restrict ourselves to single-character deletions, as the resulting operations would not then be closed under transformation (see transformation rules). The operations then become the following:

- insert(p, c) — insert character c at position p;
- delete(p, l) — delete l characters, starting at position p;

The transformations for these operations are the same as in Figure 2, except that all occurrences of $|s_1|$ or $|s_2|$ are replaced by 1.

$$\text{insert}(p_1,c_1)/\text{insert}(p_2,c_2) = \begin{cases} \text{insert}(p_1,c_1) & (p_1 < p_2) \\ \text{insert}(p_1+1,c_1) & (p_2 \le p_1) \end{cases}$$

$$\text{insert}(p_1,c_1)\backslash\text{insert}(p_2,c_2) = \begin{cases} \text{insert}(p_1,c_1) & (p_1 \le p_2) \\ \text{insert}(p_1+1,c_1) & (p_2 < p_1) \end{cases}$$

$$\text{insert}(p_1,c_1)/\text{delete}(p_2) = \begin{cases} \text{insert}(p_1,c_1) & (p_1 \le p_2) \\ \text{insert}(p_1-1,c_1) & (p_2 < p_1) \end{cases}$$

$$\text{insert}(p_1,c_1)\backslash\text{delete}(p_2) = \text{insert}(p_1,c_1)/\text{delete}(p_2)$$

$$\text{delete}(p_1)/\text{insert}(p_2,c_2) = \begin{cases} \text{delete}(p_1) & (p_1 < p_2) \\ \text{delete}(p_1+1) & (p_2 \le p_1) \end{cases}$$

$$\text{delete}(p_1)\backslash\text{insert}(p_2,c_2) = \text{delete}(p_1)/\text{insert}(p_2,c_2)$$

$$\text{delete}(p_1)/\text{delete}(p_2) = \begin{cases} \text{delete}(p_1) & (p_1 < p_2) \\ \text{noop} & (p_1 = p_2) \\ \text{delete}(p_1-1) & (p_2 < p_1) \end{cases}$$

$$\text{delete}(p_1)\backslash\text{delete}(p_2) = \text{delete}(p_1)/\text{delete}(p_2)$$

$$\text{noop}/op = \text{noop}\backslash op = \text{noop (for any } op)$$

$$op/\text{noop} = op\backslash\text{noop} = op \text{ (for any } op)$$

Figure 1: Transformation rules for adOPTed (as presented in [7], using CCU notation)

$$\text{insert}(p_1,s_1)/\text{insert}(p_2,s_2) = \begin{cases} \text{insert}(p_1,s_1) & (p_1 < p_2) \\ \text{insert}(p_1+|s_1|,s_1) & (p_2 \le p_1) \end{cases}$$

$$\text{insert}(p_1,s_1)\backslash\text{insert}(p_2,s_2) = \begin{cases} \text{insert}(p_1,s_1) & (p_1 \le p_2) \\ \text{insert}(p_1+|s_1|,s_1) & (p_2 < p_1) \end{cases}$$

$$\text{delete}(p_1,l_1)/\text{delete}(p_2,l_2) = \begin{cases} \text{delete}(p_1,l_1) & (p_1+l_1 \le p_2) \\ \text{delete}(p_1,p_2-p_1) & (p_1 \le p_2 \le p_1+l_1 \le p_2+l_2) \\ \text{delete}(p_1,l_1-l_2) & (p_1 \le p_2 \le p_2+l_2 \le p_1+l_1) \\ \text{delete}(0,0) & (p_2 \le p_1 \le p_1+l_1 \le p_2+l_2) \\ \text{delete}(p_2,p_1+l_1-p_2+l_2) & (p_2 \le p_1 \le p_1+l_1 \le p_2+l_2) \\ \text{delete}(p_1-l_2,l_1) & (p_2+l_2 \le p_1) \end{cases}$$

$$\text{delete}(p_1,l_1)\backslash\text{delete}(p_2,l_2) = \text{delete}(p_1,l_1)/\text{delete}(p_2,l_2)$$

$$\text{delete}(p_1,l_1)/\text{insert}(p_2,s_2) = \begin{cases} \text{delete}(p_1,l_1) & (p_1+l_1 \le p_2) \\ \text{delete}(p_1,l_1+|s_2|) & (p_1 \le p_2 < p_1+l_1) \\ \text{delete}(p_1+|s_2|,l_1) & (p_2 < p_1) \end{cases}$$

$$\text{delete}(p_1,l_1)\backslash\text{insert}(p_2,s_2) = \text{delete}(p_1,l_1)/\text{insert}(p_2,s_2)$$

$$\text{insert}(p_1,s_1)/\text{delete}(p_2,l_2) = \begin{cases} \text{insert}(p_1,s_1) & (p_1 \le p_2) \\ \text{insert}(p_1,\text{""}) & (p_2 < p_1 < p_2+l_2) \\ \text{insert}(p_1-l_2,s_1) & (p_2+l_2 \le p_1) \end{cases}$$

$$\text{insert}(p_1,s_1)\backslash\text{delete}(p_2,l_2) = \text{insert}(p_1,s_1)/\text{delete}(p_2,l_2)$$

Figure 2: Transformation rules for CCU

3 Modelling in Coq

To verify or refute TP2 for these transformation systems, we employ the Coq proof assistant[1]. Coq is a well-known tool among researchers in programming languages, typically used for formalizing and verifying programming language semantics. As our longer-term goals include incorporating support for operation transforms into a programming language, this tool seemed particularly appropriate. Coq is essentially an implementation of the Curry-Howard isomorphism [5]—logical statements are actually types, and proofs of logical statements are actually programs in an ML-like language that inhabit those types. Thus the Coq type system is remarkably similar to those of ML and Haskell, and we can therefore take advantage of the type system to provide convenient representations for updates.

Our proofs were developed using Coq 8.4, combined with Arthur Charguéraud's LibTactics package[2].

3.1 Representing Operations

An operation is either an insertion or a deletion (or a noop, in the case of adOPTed), packaged with parameter values (position, character, length) as needed. We can model these easily in ML with a `datatype` declaration. The declarations for adOPTed in Standard ML are as follows:

```
datatype op = Ins of int * char | Del of int | Id
```

For CCU, the definition would be as follows:

```
datatype op = Ins of int * char | Del of int * int
```

A similar construction, with minor differences in syntax, is available in Coq. For adOPTed:

```
Inductive op : Type := Ins : nat -> ascii -> op | Del : nat -> op | Id : op.
```

For CCU:

```
Inductive op : Type := Ins : nat -> ascii -> op | Del : nat -> nat -> op.
```

Note that these declarations specify only the structure of updates; no particular semantics are implied, nor indeed is the nature (i.e. type) of the shared state even mentioned. Semantics would be provided via a separate `apply` function of type `op -> state -> state`.

However, it is interesting to note that, in order to verify TP2, only the structure of the operations and the transformations themselves are needed. Indeed, TP2 is a property of operations that does not depend on their semantics, but only on their transformations. On the other hand, the property TP1 (which we do not verify in this work) does depend on the semantics of the operations.

3.2 Modelling / and \

The transformations / and \ are functions in Coq that take two `op` values and return a new `op` value. For adOPTed:

```
Definition slash(f : op) (g : op) : op :=
  match f, g with
    Ins a b, Ins c d => if leb a c then Ins a b else Ins (a + 1) b
  | Ins a b, Del c => if leb a c then Ins a b else Ins (a - 1) b
  | Ins a b, Id => Ins a b
  | Del a, Ins c d => if leb c a then Del (a + 1) else Del a
  | Del a, Del c => if beq_nat a c then Id else if leb a c then Del a else Del (a - 1)
  | Del a, Id => Del a
  | Id, _ => Id
  end.
```

[1]http://coq.inria.fr

[2]http://www.chargueraud.org/softs/tlc/

```
Definition backslash(f : op) (g : op) : op :=
  match f, g with
    Ins a b, Ins c d => if leb c a then Ins (a + 1) b else Ins a b
  | Ins a b, Del c => if leb a c then Ins a b else Ins (a - 1) b
  | Ins a b, Id => Ins a b
  | Del a, Ins c d => if leb c a then Del (a + 1) else Del a
  | Del a, Del c => if beq_nat a c then Id else if leb a c then Del a else Del (a - 1)
  | Del a, Id => Del a
  | Id, _ => Id
  end.
```

The functions `leb` and `beq_nat` above denote, respectively, the $\leq$ and equality predicates on natural numbers. The more familiar `<=` and `=` are used at the level of propositions, and therefore their types are incompatible with program code (they return type `Prop`, rather than type `bool`). Also note that Coq does not have a full suite of natural number comparison predicates for use in functions; only `leb` and `beq_nat` are provided. Other comparisons must be phrased in terms of these, and any deviation of our forumulation of `slash` and `backslash` from the originally stated formulation has been done for this reason. Note, however, that it is straightforward to automatically translate terms from a language with a complete set of comparison primitives to this core language.

For CCU, the transformations are defined as follows:

```
Definition slash (f : op) (g : op) : op :=
  match f, g with
      Ins a b, Ins c d => if ltb a c then Ins a b else Ins (a + 1) b
    | Ins a b, Del c d  =>
        if leb a c then Ins a b
        else if ltb a (c + d) then Del 0 0 else Ins (a - d) b
    | Del a b, Ins c d =>
        if leb (a + b) c then Del a b
        else if leb a c then Del a (b + 1) else Del (a + 1) b
    | Del a b, Del c d =>
        if leb (a + b) c then Del a b
        else if andb (leb a c) (leb (a + b) (c + d)) then Del a (c - a)
        else if andb (leb a c) (leb (c + d) (a + b)) then Del a (d - c)
        else if andb (leb c a) (leb (a + b) (c + d)) then Del 0 0
        else if andb (leb c a) (andb (leb a (c + d)) (leb (c + d) (a + b)))
          then Del c (a + b - c + d)
        else Del (a - d) b
 end.

Definition backslash (f : op) (g : op) : op :=
  match f, g with
      Ins a b, Ins c d => if leb a c then Ins a b else Ins (a + 1) b
    | Ins a b, Del c d => slash (Ins a b) (Del c d)
    | Del a b, Ins c d => slash (Del a b) (Ins c d)
    | Del a b, Del c d => slash (Del a b) (Del c d)
 end.
```

The function `andb` implements a boolean "and" operation (see previous comments about `leb` and `beq_nat`). We took a somewhat different approach to modelling / and \ for CCU—rather than rephrase the conditions to work exclusively with `leb` and `beq_nat`, we created a boolean < operator:

```
Definition ltb (a : nat) (b : nat) := if leb b a then false else true.
```

We then stated and proved the necessary properties of `ltb` that would allow us to translate it into < when constructing proofs:

```
Theorem ltb_iff: forall m n: nat, ltb m n = true <-> m < n.
  . . .
Qed.

Theorem ltb_iff_conv: forall m n: nat, ltb m n = false <-> n <= m.
  . . .
Qed.
```

The proofs essentially appeal to the built-in analogous results for `leb`.

Because of the `ltb` operator, our implementation of `slash` and `backslash` closely mirrors the original formulation. The only significant difference between our formulation and the original is that we eliminated some of the redundant conditions in the rule for `slash` over two `Del` operations.

3.3 Modelling TP2

The succinct formulation of TP2 (using ˆ-notation) is as follows:

$$(f\hat{}g)\hat{}(h\hat{}g) = (f\hat{}h)\hat{}(g\hat{}h)\ .$$

Recall that ˆ is simply shorthand for whichever of / and \ is appropriate, given the total ordering of the sites from which the updates originated. Hence, this notation is actually shorthand for *six* conditions, one for each possible total ordering of the sites from which f, g, and h originated. These conditions differ from one another in terms of how each occurrence of ˆ is replaced with either / or \. We observe, however, that the TP2 condition (as stated above) is symmetric in g and h; in other words, our choice of which update occurs first (i.e., plays the role of f above) completely determines the rest of the condition. Therefore, there are actually only *three* combinations of / and \ that together make up TP2:

$$(f/g)/(h\backslash g) = (f/h)/(g/h)$$

$$(f\backslash g)\backslash(h\backslash g) = (f\backslash h)\backslash(g/h)$$

$$(f\backslash g)/(h\backslash g) = (f/h)\backslash(g/h)\ .$$

In the first case, the first update (the one playing the role of f) applies / over all other updates. In the second case, it applies \ over all other updates. In the last case, it applies / over one and \ over the other.

In Coq, we model TP2 as a function that transforms operations into the relevant conditions, and then as a theorem that quantifies over inputs to the function:

```
Definition tp2 (f : op) (g : op) (h : op) : Prop :=
  slash (slash f g) (backslash h g) = slash (slash f h) (slash g h)
 /\ backslash (backslash f g) (backslash h g) = backslash (backslash f h) (slash g h)
 /\ slash (backslash f g) (backslash h g) = backslash (slash f h) (slash g h).

Theorem tp: forall f g h: op, tp2 f g h.
```

The various formulations of TP2 with respect to / and \ are only necessary if / and \ behave differently. In the case of the text buffer operations, the only time these transformations behave differently is when applied to two insertions. Hence, the fully general TP2 condition need only be checked when at least two of f, g, and h are insertions.

For this reason, and for the sake of efficiency, we also define a shortened TP2 condition, suitable when checking combinations of updates that feature at most one insertion:

```
Definition tp2short (f : op) (g : op) (h : op) : Prop :=
  slash (slash f g) (backslash h g) = slash (slash f h) (slash g h).
```

4 Proof in Coq

The theorem to be proved for both adOPTed and CCU is the theoerm `tp` given above. However, rather than prove theorem `tp` directly, we shall do it via six lemmas, one for each possible combination of `Ins` and `Del`, for the following reasons:

- Automated theorem-proving can be a slow process, and these lemmas serve as natural breaking points, with individual running times approximately one sixth the running time of the full theorem.
- If the theorem turns out to be false, it will produce a counterexample that fits one of the six combinations of operations. We may, however, be interested in whether the other five combinations also contain counterexamples (or indeed are provable), and splitting the six cases facilitates answering this question.

The lemmas are as follows (as formulated for adOPTed)[3]:

```
Lemma insinsins: forall (A B C: nat) (a b c: ascii), tp2 (Ins A a) (Ins B b) (Ins C c).

Lemma insdeldel: forall (A B C: nat) (a: ascii), tp2 (Ins A a) (Del B) (Del C).

Lemma insinsdel: forall (A B C: nat) (a b: ascii), tp2 (Ins A a) (Ins B b) (Del C).

Lemma deldeldel: forall A B C: nat, tp2 (Del A) (Del B) (Del C).

Lemma deldelins: forall A B C: nat (c: ascii), tp2 (Del A) (Del B) (Ins C c).

Lemma delinsins: forall (A B C: nat) (b c: ascii), tp2 (Del A) (Ins B b) (Ins C c).
```

The remaining combinations (i.e., `delinsdel` and `insdelins`) are not needed, due to symmetry in the second and third arguments of `tp2`.

Coq is not, at its heart, an automated theorem prover; it is a proof assistant, whose major function is to prevent the user from making errors in proofs. Nevertheless, it does include a few automated proof tactics—most notably `intuition`, which can decide intuitionistic propositional formulas. However, there is no single proof tactic that one can provide to Coq that will prove all six lemmas unaided.

Our approach was therefore to fully prove one of the lemmas by hand, executing at each step of the way the tactic that would seem most useful given the form of the subgoal at hand. We would then step back, try to perceive patterns, and see whether a more general tactic selection policy might apply to several of the lemmas, thereby reducing the amount of human intervention to a minimum, in keeping with our longer term goal of attaching this verification engine to a compiler.

4.1 adOPTed

We outline in this section the development that led to our general proof strategy for adOPTed, beginning with a proof of `insinsins`.

When lemma `insinsins` is first entered into Coq, Coq responds as follows (we use `tp2short` for brevity):

```
1 subgoal

  ============================
   forall (A B C : nat) (a b c : ascii),
   tp2short (Ins A a) (Ins B b) (Ins C c)
```

The tactic `intros` turns quantified variables and preconditions into premises:

[3]Strictly speaking, we should be including lemmas that involve `Id`; but as these cases are trivial to prove, we omit them. Proof of the full `tp` theorem includes cases involving the `Id` operation.

```
insinsins < intros.
1 subgoal

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : ascii
  ============================
   tp2short (Ins A a) (Ins B b) (Ins C c)
```

The tactic `hnf` ("head normal form") substitutes the three operations into the definition of tp2:

```
insinsins < hnf.
1 subgoal

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : ascii
  ============================
   slash (slash (Ins A a) (Ins B b)) (backslash (Ins C c) (Ins B b)) =
   slash (slash (Ins A a) (Ins C c)) (slash (Ins B b) (Ins C c))
```

The tactic `simpl` ("simplify") substitutes the fully-defined operations into the bodies of `slash` and `backslash`:

```
insinsins < simpl.
1 subgoal

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : ascii
  ============================
   slash (if leb A B then Ins A a else Ins (A + 1) a)
     (if leb B C then Ins (C + 1) c else Ins C c) =
   slash (if leb A C then Ins A a else Ins (A + 1) a)
     (if leb B C then Ins B b else Ins (B + 1) b)
```

From here, the values of the equated expressions depend on the outcomes of the `if` statements. Thus, the proof must branch, based on each possible outcome of every `if` statement. Coq has a `case` tactic that accomplishes this task:

```
insinsins < case (leb A B).
2 subgoals

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : ascii
  ============================
   slash (Ins A a) (if leb B C then Ins (C + 1) c else Ins C c) =
   slash (if leb A C then Ins A a else Ins (A + 1) a)
     (if leb B C then Ins B b else Ins (B + 1) b)

subgoal 2 is:
 slash (Ins (A + 1) a) (if leb B C then Ins (C + 1) c else Ins C c) =
 slash (if leb A C then Ins A a else Ins (A + 1) a)
   (if leb B C then Ins B b else Ins (B + 1) b)
insinsins < Focus 2.
1 focused subgoal (unfocused: 1)

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : ascii
  ============================
   slash (Ins (A + 1) a) (if leb B C then Ins (C + 1) c else Ins C c) =
   slash (if leb A C then Ins A a else Ins (A + 1) a)
     (if leb B C then Ins B b else Ins (B + 1) b)
```

The `case` tactic has two serious drawbacks:

- The user must explicitly specify the condition upon which the case is split. Although not a serious burden, this requirement does make automation somewhat more difficult.
- When the `if` statement is split on a condition, the condition itself is not maintained as a premise of the subgoal (and its negation as a premise of the sibling subgoal). This deficiency makes it impossible to detect mutually contradictory `if`-statements (i.e., unreachable branch combinations), which leads to false counterexamples.

For both of these reasons, we prefer the `cases_if` tactic, which is part of the `LibTactics` package (since issuing the previous `case` tactic, we have executed an `Undo` command):

```
insinsins < cases_if.
2 subgoals

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : ascii
  H : leb A B = true
  ============================
   slash (Ins A a) (if leb B C then Ins (C + 1) c else Ins C c) =
   slash (if leb A C then Ins A a else Ins (A + 1) a)
     (if leb B C then Ins B b else Ins (B + 1) b)

subgoal 2 is:
 slash (Ins (A + 1) a) (if leb B C then Ins (C + 1) c else Ins C c) =
 slash (if leb A C then Ins A a else Ins (A + 1) a)
   (if leb B C then Ins B b else Ins (B + 1) b)
```

```
insinsins < Focus 2.
1 focused subgoal (unfocused: 1)

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : ascii
  H : leb A B = false
  ============================
   slash (Ins (A + 1) a) (if leb B C then Ins (C + 1) c else Ins C c) =
   slash (if leb A C then Ins A a else Ins (A + 1) a)
     (if leb B C then Ins B b else Ins (B + 1) b)
```

Notice how, in these two subgoals, the condition of the `cases_if` tactic was not specified (the first candidate occurrence of an `if`-condition was used), and the condition and its negation now occur respectively as premises of the two subgoals.

We continue splitting `if` statements until none remain (below again begins right after the `simpl` tactic:

```
insinsins < cases_if; cases_if; cases_if.
8 subgoals

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : ascii
  H : leb A B = true
  H0 : leb B C = true
  H1 : leb A C = true
  ============================
   slash (Ins A a) (Ins (C + 1) c) = slash (Ins A a) (Ins B b)

subgoal 2 is:
 slash (Ins A a) (Ins (C + 1) c) = slash (Ins (A + 1) a) (Ins B b)
subgoal 3 is:
 slash (Ins A a) (Ins C c) = slash (Ins A a) (Ins (B + 1) b)
subgoal 4 is:
 slash (Ins A a) (Ins C c) = slash (Ins (A + 1) a) (Ins (B + 1) b)
subgoal 5 is:
 slash (Ins (A + 1) a) (Ins (C + 1) c) = slash (Ins A a) (Ins B b)
subgoal 6 is:
 slash (Ins (A + 1) a) (Ins (C + 1) c) = slash (Ins (A + 1) a) (Ins B b)
subgoal 7 is:
 slash (Ins (A + 1) a) (Ins C c) = slash (Ins A a) (Ins (B + 1) b)
subgoal 8 is:
 slash (Ins (A + 1) a) (Ins C c) = slash (Ins (A + 1) a) (Ins (B + 1) b)
```

The semantics of a sequence of tactics `tac1; tac2` is that `tac2` is applied to each subgoal generated by `tac1`. We now substitute the simplified arguments into the definition of `slash` (once again starting from after the initial `simpl`):

```
insinsins < cases_if; cases_if; cases_if; simpl.
8 subgoals
```

```
  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : ascii
  H : leb A B = true
  H0 : leb B C = true
  H1 : leb A C = true
  ============================
   (if leb A (C + 1) then Ins A a else Ins (A + 1) a) =
   (if leb A B then Ins A a else Ins (A + 1) a)

subgoal 2 is:
 (if leb A (C + 1) then Ins A a else Ins (A + 1) a) =
 (if leb (A + 1) B then Ins (A + 1) a else Ins (A + 1 + 1) a)
subgoal 3 is:
 (if leb A C then Ins A a else Ins (A + 1) a) =
 (if leb A (B + 1) then Ins A a else Ins (A + 1) a)
subgoal 4 is:
 (if leb A C then Ins A a else Ins (A + 1) a) =
 (if leb (A + 1) (B + 1) then Ins (A + 1) a else Ins (A + 1 + 1) a)
subgoal 5 is:
 (if leb (A + 1) (C + 1) then Ins (A + 1) a else Ins (A + 1 + 1) a) =
 (if leb A B then Ins A a else Ins (A + 1) a)
subgoal 6 is:
 (if leb (A + 1) (C + 1) then Ins (A + 1) a else Ins (A + 1 + 1) a) =
 (if leb (A + 1) B then Ins (A + 1) a else Ins (A + 1 + 1) a)
subgoal 7 is:
 (if leb (A + 1) C then Ins (A + 1) a else Ins (A + 1 + 1) a) =
 (if leb A (B + 1) then Ins A a else Ins (A + 1) a)
subgoal 8 is:
 (if leb (A + 1) C then Ins (A + 1) a else Ins (A + 1 + 1) a) =
 (if leb (A + 1) (B + 1) then Ins (A + 1) a else Ins (A + 1 + 1) a)
```

All occurrences of `slash` have now been eliminated, but additional `if` statements have appeared, which must now be split (again, starting after the initial `simpl`):

```
24 subgoals

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : ascii
  H : true = true
  H0 : leb B C = true
  H1 : leb A C = true
  H2 : leb A (C + 1) = true
  H3 : leb A B = true
  ============================
   Ins A a = Ins A a
```

```
subgoal 2 is:
 Ins (A + 1) a = Ins A a
subgoal 3 is:
 Ins A a = Ins (A + 1) a
subgoal 4 is:
 Ins A a = Ins (A + 1 + 1) a
subgoal 5 is:
 Ins (A + 1) a = Ins (A + 1) a
subgoal 6 is:
 Ins (A + 1) a = Ins (A + 1 + 1) a
subgoal 7 is:
 Ins A a = Ins A a
subgoal 8 is:
 Ins A a = Ins (A + 1) a
subgoal 9 is:
 Ins (A + 1) a = Ins (A + 1) a
subgoal 10 is:
 Ins (A + 1) a = Ins (A + 1 + 1) a
subgoal 11 is:
 Ins (A + 1) a = Ins (A + 1) a
subgoal 12 is:
 Ins (A + 1 + 1) a = Ins (A + 1) a
  . . .
```

We obtain 24 subgoals, all of which equate two operations. For those subgoals in which the two equated operations are, in fact, identical, the `reflexivity` tactic applies directly:

```
insinsins < cases_if; cases_if; cases_if; simpl; cases_if; cases_if; try reflexivity.
14 subgoals

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : ascii
  H : true = true
  H0 : leb B C = true
  H1 : leb A C = true
  H2 : leb A (C + 1) = false
  H3 : leb A B = true
  ============================
   Ins (A + 1) a = Ins A a

subgoal 2 is:
 Ins A a = Ins (A + 1) a
subgoal 3 is:
 Ins A a = Ins (A + 1 + 1) a
subgoal 4 is:
 Ins (A + 1) a = Ins (A + 1 + 1) a
subgoal 5 is:
 Ins A a = Ins (A + 1) a
subgoal 6 is:
 Ins (A + 1) a = Ins (A + 1 + 1) a
subgoal 7 is:
 Ins (A + 1 + 1) a = Ins (A + 1) a
subgoal 8 is:
 Ins (A + 1) a = Ins (A + 1 + 1) a
subgoal 9 is:
 Ins (A + 1 + 1) a = Ins (A + 1) a
subgoal 10 is:
 Ins (A + 1) a = Ins A a
subgoal 11 is:
 Ins (A + 1 + 1) a = Ins A a
subgoal 12 is:
 Ins (A + 1 + 1) a = Ins (A + 1) a
subgoal 13 is:
 Ins (A + 1) a = Ins (A + 1 + 1) a
subgoal 14 is:
 Ins (A + 1 + 1) a = Ins (A + 1) a
```

The metatactic `try` attempts to apply a tactic (in this case, `reflexivity`) but suppresses errors if the tactic is not applicable. Had we used `reflexivity` alone, the proof process would have halted at the first pair of non-identical updates, and any remaining subsequent subgoals containing equated identical updates would not have been resolved.

The `reflexivity` tactic was able to resolve ten of the 24 subgoals. The remaining 14 are equations between non-identical updates. These could represent counterexamples that refute the lemma, if the corresponding premises are satisfiable (i.e., non-contradictory), or simply impossible cases, if the corresponding premises are unsatisfiable.

Consider the current first goal: `Ins (A + 1) a = Ins A a`. The relevant premises are as follows:

```
  H0 : leb B C = true
  H1 : leb A C = true
  H2 : leb A (C + 1) = false
  H3 : leb A B = true
```

From `H1`, we get $A \leq C$, and from `H2`, we get $A > C + 1$. As these statements are contradictory, any conclusion follows, and we can, in principle, resolve this subgoal.

Unfortunately, Coq cannot see the contradiction because it does not natively understand the semantics of `leb`. We need to first use a rewrite rule to translate the `leb` statements into statements involving `<=`:

```
insinsins < rewrite leb_iff in *.
14 subgoals

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : ascii
  H : true = true
  H0 : B <= C
  H1 : A <= C
  H2 : leb A (C + 1) = false
  H3 : A <= B
  ============================
   Ins (A + 1) a = Ins A a

subgoal 2 is:
 Ins A a = Ins (A + 1) a
  . . .
```

To translate the false occurrence of `leb`, an additional rewrite is needed:

```
insinsins < rewrite leb_iff_conv in *.
14 subgoals

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : ascii
  H : true = true
  H0 : B <= C
  H1 : A <= C
  H2 : C + 1 < A
  H3 : A <= B
  ============================
   Ins (A + 1) a = Ins A a

subgoal 2 is:
 Ins A a = Ins (A + 1) a
  . . .
```

We are now well-equipped to search for a contradiction. The tactic `exfalso` replaces the current goal with `False`, thereby translating the goal into a search for a contradiction (from which anything follows, including the current goal).

```
insinsins < exfalso.
14 subgoals

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : ascii
  H : true = true
  H0 : B <= C
  H1 : A <= C
  H2 : C + 1 < A
  H3 : A <= B
  ============================
   False

subgoal 2 is:
 Ins A a = Ins (A + 1) a
  . . .
insinsins < intuition.
13 subgoals

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : ascii
  H : leb A B = true
  H0 : leb B C = true
  H1 : leb A C = false
  H2 : leb A (C + 1) = true
  H3 : leb (A + 1) B = true
  ============================
   Ins A a = Ins (A + 1) a

subgoal 2 is:
 Ins A a = Ins (A + 1 + 1) a
```

The final call to `intuition` finds the contradiction, and Coq moves on to the next subgoal.

Once all of the subgoals have been reduced to equations between two operations, three outcomes are possible for each subgoal:

- The items are identical—then `reflexivity` should resolve the subgoal in one step.
- The items are different, with contradictory premises—then `rewrite leb_iff, leb_iff_conv in *; exfalso; intuition` should resolve the subgoal in one step.
- The items are different, with consistent premises—then both of the above approaches should fail, and we have found a counterexample.

Thus, we can filter out the identities and contradictions by applying both tactics to each subgoal. What remains will be the counterexamples. To keep the command succinct, we define a solution-finding tactic:

```
Ltac tpsolve :=
  rewrite ?leb_iff, ?leb_iff_conv, ?beq_nat_true_iff, ?beq_nat_false_iff in *;
  try (solve [reflexivity | exfalso; intuition]).
```

The `?` operator in the `rewrite` tactic prevents the tactic from failing if the corresponding rewrite rule is not applicable. The metatactic `solve` only succeeds if one of the following tactics completely solves the goal; otherwise it fails, the failure is caught by the `try` metatactic, and the subgoal is reverted to its previous form. We then filter out the identities and contradictions as follows (starting after the first `simpl`):

```
insinsins < cases_if; cases_if; cases_if; simpl; cases_if; cases_if; tpsolve.
No more subgoals.
```

As all of the subgoals have been resolved, we have completed the proof of the lemma. We complete the proof with `Qed`:

```
insinsins < Qed.
intros.
hnf.
simpl.
cases_if; cases_if; cases_if; simpl; cases_if; cases_if; tpsolve.

insinsins is defined
```

Having established the lemma, we consider next how to make the proof more generally applicable. We must keep in mind that not all updates will have exactly the same number of `if` statements, with the same number of conditions. Hence the various branches of the tree of subgoals will be of different heights. In some branches, `cases_if` will be the correct next tactic to use; in other branches, we should use `simpl`. In still others, the equation may be fully reduced and it may be time to attempt to solve.

Coq provides an operator `||`, whose semantics are "the first among these tactics that does not fail". We can combine this with the `repeat` tactic (whose semantics are "keep applying this tactic while it is still applicable") to obtain the following strategy:

```
intros; hnf; simpl; repeat cases_if || simpl || tpsolve.
```

Essentially, after the first simplification, we keep applying `cases_if` while we can, and if not, then we apply `simpl`. If neither of these applies, we attempt to solve.

This single line of proof suffices to prove `insinsins`. Moreover, since it reflects none of the actual structure of the transformation rules on insertions, it should be applicable unchanged to other lemmas as well [4]. Indeed, this proof approach should apply to any transformation made up of a sequence of conditions, each of whose outcomes is another update (in particular, recursively defined transformations would not necessarily be covered by this strategy, as the proof would then likely require induction; however, we know of no such transformations). In order to completely decide TP2 for a particular transformation, we must also be guaranteed that the `intuition` tactic will be able to find contradictions. The conditions in the text buffer transformations are all inequalities on linear expressions in natural numbers (in particular, no multiplication). Hence the conditions all lie within Presburger arithmetic, which is known to be decidable. We know of no proposed set of transformations for which the conditions do not lie within Presburger arithmetic. Hence, `intuition` should be able to find contradictions in most, if not all, cases in which they exist.

Thus, we have some confidence that our proof strategy will work for most, if not all, known transformations. We executed this proof strategy on all of the remaining lemmas for adOPTed, and were able to prove all of them without additional assistance, except for one. The following is the result of attempting to prove the lemma `insinsdel`:

```
Coq < Lemma insinsdel: forall (A B C: nat) (a b: ascii), tp2 (Ins A a) (Ins B b) (Del C).
1 subgoal

  ============================
   forall (A B C : nat) (a b : ascii), tp2 (Ins A a) (Ins B b) (Del C)

insinsdel < intros; hnf; splits; simpl; repeat cases_if || simpl || tpsolve.
9 subgoals

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  H : leb A B = false
  H0 : leb B C = true
  H1 : leb A C = false
  H2 : leb (A + 1) (C + 1) = false
  H3 : leb (A - 1) B = true
  ============================
   Ins (A + 1 - 1) a = Ins (A - 1) a

subgoal 2 is:
 Ins (A + 1 - 1) a = Ins (A - 1 + 1) a
subgoal 3 is:
 Ins (A + 1 - 1) a = Ins (A - 1 + 1) a
subgoal 4 is:
 Ins (A + 1 - 1) a = Ins (A - 1 + 1) a
subgoal 5 is:
 Ins (A + 1 - 1) a = Ins (A - 1 + 1) a
subgoal 6 is:
 Ins A a = Ins (A + 1) a
subgoal 7 is:
```

[4]For the full (i.e., not the short) version of `tp2`, we include a call to the `splits` tactic (from the `LibTactics` package) before the first `simpl` tactic, to split the conjunction into separate subgoals. The remainder of the proof is as stated.

```
 Ins (A + 1 - 1) a = Ins (A - 1 + 1) a
subgoal 8 is:
 Ins (A + 1 - 1) a = Ins (A - 1 + 1) a
subgoal 9 is:
 Ins A a = Ins (A + 1) a
```

To make the premises more readable, we execute the `rewrite` tactic:

```
insinsdel < rewrite leb_iff, leb_iff_conv in *.
9 subgoals

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  H : B < A
  H0 : B <= C
  H1 : C < A
  H2 : C + 1 < A + 1
  H3 : A - 1 <= B
  ============================
   Ins (A + 1 - 1) a = Ins (A - 1) a

subgoal 2 is:
 Ins (A + 1 - 1) a = Ins (A - 1 + 1) a
subgoal 3 is:
 Ins (A + 1 - 1) a = Ins (A - 1 + 1) a
subgoal 4 is:
 Ins (A + 1 - 1) a = Ins (A - 1 + 1) a
subgoal 5 is:
 Ins (A + 1 - 1) a = Ins (A - 1 + 1) a
subgoal 6 is:
 Ins A a = Ins (A + 1) a
subgoal 7 is:
 Ins (A + 1 - 1) a = Ins (A - 1 + 1) a
subgoal 8 is:
 Ins (A + 1 - 1) a = Ins (A - 1 + 1) a
subgoal 9 is:
 Ins A a = Ins (A + 1) a
```

This set of premises is satisfied whenever $A = B+1$ and $B = C$. For example, `(Ins 2 "a") (Ins 1 "b") (Del 1)` satisfies the premises, and a quick check shows that indeed, TP2 is not satisifed for these updates (the left side is `Ins 2 "a"`, while the right side is `Ins 1 "a"`). This counterexample to adOPTed agrees with a known counterexample [7]. The set of premises associated with this counterexample actually give a complete characterization of an entire set of counterexamples. Examining the remaining goals may reveal further classes of counterexamples, or may simply indicate contradictory premises.

If we examine several of the remaining goals, we see conclusions of the form

```
 Ins (A + 1 - 1) a = Ins (A - 1 + 1) a
```

which, although not syntactically identical, are nonetheless equivalent. We can use the `f_equal` tactic to replace this goal with one that equates the individual arguments; `intuition` will then be able to solve the goal. This additional tactic is easily added to `tpsolve`:

```
Ltac tpsolve :=
  rewrite ?leb_iff, ?leb_iff_conv, ?beq_nat_true_iff, ?beq_nat_false_iff in *;
  solve [reflexivity | f_equal; intuition | exfalso; intuition].
```

We can then eliminate more legitimately provable goals:

```
Coq <  Lemma insinsdel: forall (A B C: nat) (a b: ascii), tp2 (Ins A a) (Ins B b) (Del C).
1 subgoal

  ============================
   forall (A B C : nat) (a b : ascii), tp2 (Ins A a) (Ins B b) (Del C)

insinsdel < intros; hnf; splits; simpl; repeat cases_if || simpl || tpsolve.
3 subgoals

  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  H : leb A B = false
  H0 : leb B C = true
  H1 : leb A C = false
  H2 : leb (A + 1) (C + 1) = false
  H3 : leb (A - 1) B = true
  ============================
   Ins (A + 1 - 1) a = Ins (A - 1) a

subgoal 2 is:
 Ins A a = Ins (A + 1) a
subgoal 3 is:
 Ins A a = Ins (A + 1) a
```

We are left with the same counterexample, and only two unexplored goals, which may be further counterexamples, or may simply have contradictory premises, which were not discovered by the time the counterexample surfaced.

To conclude this section, we remark that we have a fully general proof strategy that works unaided on all of the lemmas related to TP2 for adOPTed. Our proof strategy was able to find a counterexample for adOPTed that agrees with known results.

4.2 CCU

In this subsection we attempt to carry out the same automated verification process on the text buffer operation transforms that were defined as part of CCU. Although these transforms were once hand-verified [6], hand-verification is error-prone, and we know of no previous attempt to automatically verify TP2 for the CCU operations.

The biggest difference between the CCU operations and the adOPTed operations is that CCU considers multi-character insertions and deletions. Therefore it makes sense, in the context of CCU, to speak of Alice inserting a string into chapter 4 of a book, while Bob concurrently deletes chapter 4. The end result is that Alice's sentence should disappear, since the context in which it should appear has also disappeared. Without multi-character delete, we cannot sensibly speak of the surrounding context of an insertion, and so the adOPTed transforms cannot capture these semantics.

The CCU transformations themselves, however, follow much the same format as the adOPTed transformations: each transformation is specified by a sequence of conditions, each accompanied with an associated transformed operation. The only difference is in the exact nature (and number) of the conditions, and the associated outputs. Therefore, in principle, the framework we developed for adOPTed should work for CCU as well, without modification. Indeed, our framework for adOPTed does, in fact, verify TP2 for the CCU version of lemma `insinsins`.

The reality, however, is that we have a problem of performance. The insertion rules for both calculi, and indeed all of the transformation rules for adOPTed, include no more than three clauses each (sometimes only two). In contrast, the transformation rule for two deletions under CCU contains six clauses, which dramatically increases the number of branches in the proof search tree, when checking combinations of updates that feature at least two deletions. Indeed, Coq runs out of memory when trying to verify lemma `insdeldel`. Because of this difficulty, some discussion on making our Coq verification more efficient would be worthwhile.

4.2.1 Improving efficiency

Our current approach to verifying TP2 is to break every `if` statement into subgoals, and then either verify or refute each one. For `insinsins` the number of generated subgoals is relatively small, and the verification process is quite manageable.

When checking `insdeldel`, the total number of generated subgoals is approximately 1500. For each of these, our proof strategy is to carry out our `rewrite` tactics, and then either use `reflexivity` to check for identities, or attempt to find contradictions using `intuition`. As a result of this strategy, all processing of premises takes place at the leaves of the search tree. By this time, we will have encountered several `if` statements, whose conditions have been split and new subgoals generated. As more and more subgoals are split, these original conditions are duplicated over and over again. Thus, the leaves all contain true or false instances of almost the same set of premises throughout.

We can therefore eliminate some effort by being more proactive in our rewriting. By rewriting premises as they occur, rather than at the leaves of the tree, we can duplicate the rewritten premises, rather than the original ones, and thereby save duplicated rewriting effort.

Moreover, we can save additional effort by recognizing the way in which the `rewrite` tactic carries out its purpose. When we issue a tactic like `rewrite t1, t2, t3`, our observations indicate that `rewrite` attempts to rewrite by all of `t1`, `t2`, `t3`; and then, if these rewrites generate further instances of `t1`, `t2`, or `t3`, then the whole process resumes until no further opportunities to use these tactics arises. For example, our `rewrite` tactic for CCU is as follows:

```
rewrite ?Bool.andb_true_iff, ?Bool.andb_false_iff, ?leb_iff, ?leb_iff_conv,
 ?ltb_iff, ?ltb_iff_conv, ?beq_nat_true_iff, ?beq_nat_false_iff in *.
```

The additional rewrites come from our use of the operators `ltb` and `andb` in our CCU verification. This simultaneous normalization with respect to several rewrite rules comes at some expense. However, if we know that certain rewrites will not create instances of other rewrites, or at least the order in which some rewrites will create instances of others, then we can order the rewrites appropriately and execute them serially:

```
rewrite ?Bool.andb_true_iff, ?Bool.andb_false_iff in *;
 rewrite ?leb_iff in *; rewrite ?leb_iff_conv in *;
 rewrite ?ltb_iff in *; rewrite ?ltb_iff_conv in *;
 rewrite ?beq_nat_true_iff in *; rewrite ?beq_nat_false_iff in *.
```

In our experience, the performance gain from serializing rewrites has been dramatic.

Our strategy of aggressively rewriting premises as they appear applies equally well to attempting to solve subgoals. If we can identify contradictory premises as soon as they appear, we can eliminate a subgoal immediately, rather than allow it to be further split on other `if` statements, only to have all of its children be rejected later due to contradictory premises. Aggressively searching for contradictions allows us to elminate entire subtrees of the proof search tree in one fell swoop. The potential cost savings is considerable.

Our proof strategy then becomes roughly the following: at each level of the search tree, we alternate trying tactic `simpl`; with tactic `cases-if`. Since the latter creates additional subgoals with additional premises, every time we try `cases_if`, we also try to solve the subgoals by `reflexivity` or `f_equal`; failing that, we try to refute the subgoals using `exfalso; intuition`. Our current (rough) strategy follows:

```
Ltac rw := rewrite ?Bool.andb_true_iff, ?Bool.andb_false_iff in *;
```

```
  rewrite ?leb_iff in *; rewrite ?leb_iff_conv in *;
  rewrite ?ltb_iff in *; rewrite ?ltb_iff_conv in *;
  rewrite ?beq_nat_true_iff in *; rewrite ?beq_nat_false_iff in *.

Ltac crw := cases_if; rw.

Lemma insdeldel: forall (A B C: nat) (a: ascii) (b c: nat), tp2short (Ins A a) (Del B b) (Del C c).
Proof.
  intros.
  hnf.
  simpl; crw; crw; crw; crw; try (solve [exfalso; intuition]);
  simpl; crw; try (solve [exfalso; intuition]);
  try simpl; try crw; try reflexivity; try (solve [exfalso; intuition]);
  try simpl; try crw; try reflexivity; try (solve [exfalso; intuition]);
  try simpl; try crw; try reflexivity; try (solve [exfalso; intuition]);
  try crw; try reflexivity; try (solve [exfalso; intuition]);
  try simpl; try crw; try (solve [reflexivity | f_equal; intuition | exfalso; intuition]);
  try simpl; try crw; try (solve [reflexivity| exfalso; intuition]);
  try simpl; try crw; try (solve [reflexivity | f_equal; intuition | exfalso; intuition]);
  try crw; try (solve [reflexivity | exfalso; intuition]).
Qed.
```

In general, the further down the tree we go, the more tactics we try, as goals to which they apply are less likely to occur at the very top of the tree. Also, there are likely more calls to tactic `simpl` than necessary, as these are needed simply for substituting into the definitions of `slash` and `backslash`—and the number of occurences of such calls is only three on either side of the equality.

Using this strategy, we have been able to consistently keep the number of active subgoals under 120 at each stage. Nevertheless, we have not yet succeeded at completing the verification of this lemma. Research into this problem is ongoing.

4.2.2 Results for CCU

Although we have not completed the full TP2 verification for lemma `insdeldel` (and have not yet attempted lemma `deldeldel`, which promises to be much worse), we have been able to comlete the verification for lemma `insinsins` (as stated above), and lemma `insinsdel`. The former has been verified; the output from Coq for the latter is presented below (other subgoals suppressed):

```
  A : nat
  B : nat
  C : nat
  a : ascii
  b : ascii
  c : nat
  H : A < B
  H0 : C + c <= B
  H1 : true = true
  H2 : C < B
  H3 : C + c <= B
  H4 : A <= C
  H5 : B - c <= A
  ============================
   Ins A a = Ins (A + 1) a
```

If we add hypotheses `H3` and `H5`, we obtain $C + B \leq B + A$, which reduces to $C \leq A$. This, combined with hypothesis `H4`, gives us $A = C$. Thus, Coq reports a counterexample when $A = C, A < B$. For example, `(Ins 2 "a")`, `(Ins 5 "b")`, and `(Del 2 3)` fit these criteria, and a quick hand-verification confirms that TP2 fails for these updates.

We had previously thought that CCU succeeded where adOPTed failed, and had even hand-verified that the CCU was not vulnerable to the reported adOPTed counterexample. Thanks to our Coq verifier, we now know that we were wrong, and in fact, the CCU transformations (which for 10 years we thought satisfied TP2) do not satisfy TP2.

5 Related Work

This report is not the first work to apply automated techniques to TP2 verification. Boucheneb et al [1, 2] use a model checking approach for TP2. A notable theorem-proving approach is the work of Oster et al [7], who use a different tool (the SPIKE theorem prover) instead of Coq, and who attempt to verify TP1 as well as TP2.. Verifying TP1 carries with it its own challenges, which Oster et al expose very well. We have not attempted to verify TP1, as it tends to be easier to hand-verify. Oster et al do not provide many details on their TP2 verifier, but based on what they say about their TP1 verifier, it appears that their verifier is based primarily on pure automated reasoning, whereas our approach is at least partially guided by syntactic considerations of the transformations themselves.

We know of no work, aside from the present one, that attempts to verify TP2 for the CCU transforms via theorem-proving. We believe the counterexample to CCU presented herein to be novel.

6 Conclusions and Future Work

This work serves numerous purposes. It is an attempt to reproduce a previous theorem-proving result for TP2, and independently generate the same or another counterexample for the adOPTed algorithm. It represents the beginnings of an effort to produce a useable tool to facilitate further research on operation transforms. It provides the first known automated refutation of the CCU transformations. Finally, it serves as a domain-specific Coq tutorial for future researchers on this project.

Opportunities for future and ongoing work abound. It remains to find the exact combination of tactics to allow Coq to fully check the rest of CCU within the resources available. Once we have the tool running efficiently, we can proceed along at least two lines. We can work on integrating it into a compiler framework, so as to provide language-level support for operation transforms; as part of this work, we may also explore whether a compiler might be able to optimize the proof strategy based on the form of the input. In addition, the tool will be useful as we discover more kinds of shared objects, and need to verify TP2 for new sets of operations.

References

[1] Hanifa Boucheneb and Abdessamad Imine. On model-checking optimistic replication algorithms. In *Formal Techniques for Distributed Systems*, volume 5522 of *LNCS*, pages 74–88, 2009.

[2] Hanifa Boucheneb, Abdessamad Imine, and Manal Najem. Symbolic model-checking of optimistic replication algorithms. In *Integrated Formal Methods*, volume 6396 of *LNCS*, pages 89–104, 2010.

[3] Gordon V. Cormack. A calculus for concurrent update. *Research Report CS-95-06, Dept. of Computer Science, University of Waterloo*, 1995.

[4] C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD Record*, 18(2):399–407, 1989.

[5] William A. Howard. The formulae-as-types notion of construction. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, 1980.

[6] Bradley M. Lushman. Transformation-based Concurrency Control in Groupware Systems. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, 2002.

[7] Gerald Oster, Pascal urso, Pascal Molli, and Abdessamad Imine. Proving correctness of transformation functions in collaborative editing systems. *Research Report 5795, INRIA*, 2005.

[8] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 288–297, November 1996.