Verified operational transformation for trees

Sergey Sinchuk[1], Pavel Chuprikov[1], and Konstantin Solomatov[2]

[1] JetBrains, St. Petersburg, Russia
sinchukss@gmail.com, pschuprikov@gmail.com
[2] JetBrains, Boston, USA
konstantin.solomatov@gmail.com

Abstract Operational transformation (OT) is an approach to concurrency control in groupware editors first proposed by C. Ellis and S. Gibbs in 1989. Google Wave and Google Docs are examples of better known OT-based systems and there are many other experimental ones described in the literature. In their recent articles A. Imine et al. have shown that many OT implementations contain mistakes and do not possess claimed consistency properties.
The present work describes an experimental library which is based on SSReflect/Coq and contains several operational transformation algorithms and proofs of their correctness.[3]

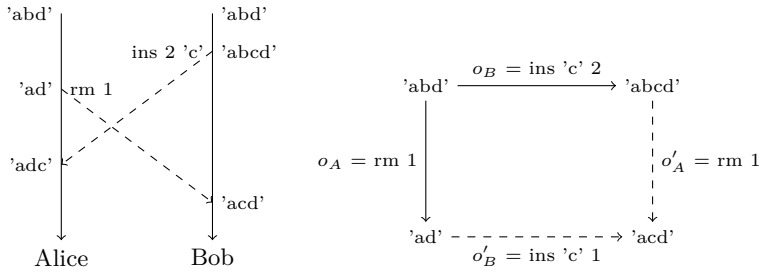
1 Introduction

A collaborative groupware editor is an application that allows multiple users to edit shared data objects (e.g., a text document or a spreadsheet). We will be mainly concerned with *synchronous* groupware editors (or real-time collaborative editors), i.e. editors which allow simultaneous editing of the shared data and provide automatic real-time synchronization between users. Moreover, such editors usually do not use locks in the implementation of their synchronization algorithm. Instead, every user is provided with his own replica of the data and is allowed to modify it freely.

Due to the network latency and the lock-free nature of the editor, a naive synchronization algorithm applying remote operations to a local replica *unchanged* will not be consistent. The replicas' states may diverge significantly from each other and a remote operation may not have its intended effect when applied to the local replica. Let's go through the simplest scenario in which this problem occurs. Alice removes symbol "b" and Bob inserts character "c" at the second position. After these commands are processed the state of the network becomes invalid (see Fig. 1a).

Operational transformation was conceived to overcome this problem. In the simplest case of two communicating clients its idea can be roughly stated as

[3] The final authenticated publication is available online at https://doi.org/10.1007/978-3-319-43144-4_22



(a) Without OT (b) With OT

Figure 1: An example illustrating the need of an OT. On (a) unmodified operations lead to the invalid state. On (b) the state remains consistent if we use an OT

follows. Instead of applying Bob's operation o_B directly, Alice first "transforms" o_B through the history of her recent local changes o_A thereby calculating the operation o'_B which is applicable to the actual version of Alice's data and has the same effect as o_B. Similarly, Bob computes o'_A from known o_A and o_B. Then, after the operations o'_A and o'_B are executed, the replicas are again in the same state. Fig. 1b illustrates the relationship between o_A, o_B, o'_A and o'_B.

A typical implementation of an OT algorithm consists of two separated components: a *transformation function* which carries out transformation of operations (i.e. computes o'_B from specified o_B and o_A) and an *integration algorithm* which is responsible for storing local histories and maintaining communication between the clients. Only the former component typically depends on the semantics of the data.

An OT algorithm is said to possess the *convergence property* if the replicas of a shared document become identical at all sites after all user operations have been executed. This property is essential for the correct operation of an OT algorithm because the existence of a counterexample to it necessarily means that the loss of user data is possible. To guarantee that an OT algorithm satisfies this convergence property one needs to ensure that the integration algorithm is correct and, moreover, that the transformation function satisfies correctness properties C_1 and C_2 (see, section 2.1 for more details).

– Roughly speaking, the property C_1 requires that the effect of executing operations $o_B \circ o_{A'}$ and $o_A \circ o_{B'}$ is the same (compare with Fig. 1b). The property C_1 suffices to ensure the convergence of an OT algorithm in the case where the network has a tree topology (e.g., in the case of a network with a central server, see [2]),
– The more complicated property C_2 becomes necessary in a more general setting when one considers less restrictive network topologies which may have loops (e.g., peer-to-peer networks).

Several generic integration algorithms together with proofs of their correctness have been proposed in the literature (e.g., in [8], [11]). On the other hand, for each datatype the transformation function must be implemented separately and such an implementation is known to be a hard and error-prone task even in the simplest case when the shared data object in question is a string buffer. Indeed, in the recent works of Imine and others it was shown that many implementations of transformation functions for strings do not possess the above correctness properties (see [7], [9], [10]).

Furthermore, even less is known about the correctness of OT algorithms for more general data types such as trees, despite the fact that such datatypes are useful in practice. Indeed, strings can be used only as a data model for a text editor, while trees can represent a variety of different structured documents (e.g., a spreadsheet or an XML document). Besides that, earlier efforts mainly concentrated on algorithms with a very small set of user operations (e.g., consisting only of operations of insertion and deletion of a single character). While formal verfications of such algorithms is easier, their behavior is less satisfactory from the semantic viewpoint as compared to algorithms with a more complicated set of operations. For example, if the last-mentioned reduced set of operations is used then it becomes impossible to take into account any higher-level semantic entities such as words or sentences in the implementation of the transformation function and, as a result, the loss of character order in the document may occur under a certain scenario, see [6, p. 325].

The main goal of this paper is to describe our attempt to formalize several transformation functions for two different kinds of a tree datatype, namely the datatype of ordered trees (which may represent, e.g., an XML-like document) and the datatype of unordered trees (which may represent the directory structure of a filesystem). The choice of an OT for trees as a subject for verification was motivated by the fact that Jetpad platform [3] stores shared data in a tree-like structure.

Our definition of the transformation function for ordered trees is a generalized and corrected variant of algorithms of Ressel and Sun (cf. [10]). Using Coq we verify that our transformation functions satisfy convergence property C_1 and inversion property IP_1 (in the terminology of [12]). The choice of Coq as a verification tool was made due to the following considerations:

- many complex algorithms including compilers and static analyzers have been verified in Coq (e.g., CompCert, Verasco, etc.);
- Coq includes a comprehensive standard library and allows tactic-based proofs, whose power is greatly increased by the SSReflect library (cf. [5]);

The rest of the article is organized as follows. In Section 2 we formalize the basic terminology related to the operational transformation approach. Then, in Section 3 we present the main results of the paper, namely the precise definitions of the transformation functions whose correctness has been verified in Coq. The library source code in Coq can be found at `github.com/JetBrains/ot-coq/`.

[3] Jetpad is a closed-source proprietary collaborative platform of JetBrains upon which several products such as CoachingSpaces or CensusAnalyzer are based.

2 Formalization of OT algorithms and their correctness properties

2.1 Operation model

In this section we formally define the notion of a transformation function and formulate its correctness properties. Our definitions generally follow [7], however, there are certain differences which are explained in detail below.

First of all, we need to give names to variables corresponding to a set of possible states of a shared data object and a set of atomic user operations which can modify it. Let us denote these two variables by X and cmd, respectively.

Now, we define the type class abstracting the minimal functionality of an OT algorithm. This class should encapsulate the following three entities:

1. an *interpretation* (or *transition*) function `interp` specifying how user operations are applied to the data;
2. a *transformation* function `it` which performs the transformation of operations;
3. a formula expressing the convergence property C_1 of the function `it`.

First of all, notice that we do not require the function `interp` to be total, i.e. we allow certain operations to be inapplicable to certain states of the data. For example, an operation of file deletion is only applicable if the file exists. Thus, we choose the following signature for `interp`:

$$\texttt{interp}\colon cmd \to X \to \texttt{option}\ X.$$

The equality $\texttt{interp}\ op\ x = \texttt{None}$, thus, should be interpreted as "op is inapplicable to x".

Now we are going to specify the signature of the transformation function. In the literature (e.g., in [4,10,7]) it is typically defined as $\texttt{it}\colon cmd \to cmd \to cmd$. The operation $op_1' = \texttt{it}\ op_1\ op_2$ is interpreted as the result of a transformation of op_1 through op_2.

In this context, convergence property C_1 can be stated as follows: any pair of atomic operations op_1, op_2 applicable to s can be completed to a square by means of operations $\texttt{it}\ op_1\ op_2$, $\texttt{it}\ op_2\ op_1$ (see Fig. 2a).

For the sake of completeness, we also give the precise statement of the convergence property C_2. The property C_2 requires that for any op_1, op_2, op_3 one has the following equality of operations (cf. [7, Definition 2.12]):

$$\texttt{it}\ (\texttt{it}\ op_1\ op_2)\ (\texttt{it}\ op_3\ op_2) = \texttt{it}\ (\texttt{it}\ op_1\ op_3)\ (\texttt{it}\ op_2\ op_3).$$

This property is rather restrictive and difficult to prove in practice. Also, it has been suggested in [10] that it is not possible to implement a transformation function for text buffers satisfying C_2. Furthermore, as we said before, the property C_2 is not necessary to achieve convergence for networks with dedicated servers which is the main case of interest for us. For these reasons, we do

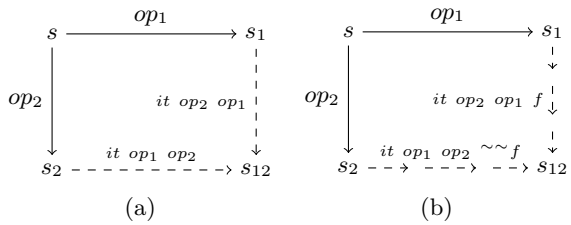


Figure 2: Diagrams expressing different variants of convergence property C_1

not include the statement of C_2 into our formal definition of a convergent OT algorithm presented below.

There are two things that we change in the signature of the function `it`. The first is that we make the definition of `it` asymmetric by adding a special boolean flag which allows `it` to take into account possible difference in the priority of clients. We give an example illustrating why such a prioritization may be necessary.

Consider the following situation: Alice and Bob simultaneously insert two different characters into the same position of a shared text (say, op_1 = ins "a" 0, op_2 = ins "b" 0). The problem with the signature $\texttt{it}\colon \texttt{cmd} \to \texttt{cmd} \to \texttt{cmd}$ is that there appears to be no way to implement `it` so that the resulting conflict is resolved in a semantically satisfactory manner and the property C_1 is also preserved.

A possible way to resolve this problem is to assign different priorities to different clients and then take them into account in the implementation of the function `it`. Thus, in the last example, we could agree to always insert the string typed by Alice before Bob's if both are inserted into the same position.

Thus, we are either forced to store the information about client priorities inside *cmd*, despite the fact that it is not used in the implementation of an interpretation function, or we should make the signature of `it` asymmetric. In our algorithm we use the latter approach, while the former was used, for example, in the algorithms of Ellis–Gibbs and Ressel (see [10, § 2.3]). Notice that in these algorithms the priority of operations is encoded with natural numbers since they assume the presence of multiple clients with different priorities.

In contrast, in our integration algorithm we do not assign different priorities to clients. Instead, all users are connected to the central server and each client-server connection is considered as a network with two collaborating users of which the central server has the higher priority. Notice that the server does not modify the shared data by itself and only propagates user-made changes. With this approach a boolean flag is sufficient to distinguish between the client and the server on each client-server connection.

The second modification is that we allow the result of a transformation of two atomic operations to be a composite operation, i.e. a list of atomic operations.

This not only makes our definition more flexible and general, but also simplifies the definition of the operation type *cmd* in practice. For example, it is very common that the result of a transformation of two nonempty primitive operations is empty (e.g., if two users simultaneously delete the same file). The fact that we allow `it` to return composite operations eliminates the need to define a dedicated empty operation constructor for *cmd*. Instead, `it` can simply return an empty composite operation. We conclude with the following two definitions which will be used throughout the rest of the article.

Definition 1 *A transformation function is a function with the following signature:*

$$it : cmd \to cmd \to bool \to list\ cmd.$$

Definition 2 *We say that the transformation function* `it` *satisfies the property* C_1 *if for any boolean flag* f*, any pair of primitive operations* op_1*,* op_2*, applicable to a state* s*, can be completed to a square Fig. 2b.*

We can put together all our formal definitions stated above into the following Coq class.

```
Class OTBase (X cmd: Type) := {
interp:cmd → X → option X;
it    :cmd → cmd → bool → list cmd;
it_c1 :forall (op1 op2: cmd)(f: bool) m (s1 s2: X),
interp op1 s = Some s1 → interp op2 s = Some s2 →
let s21:= exec_all interp (Some s2) (it op1 op2 f) in
let s12:= exec_all interp (Some s1) (it op2 op1 ~~f)in
 s21 = s12 /\ s21 <> None}.
```

In the above code example `exec_all interp` is the function extending the interpretation function to composite operations in an obvious way.

In our model we also want to have a special type class formalizing the notion of an OT algorithm with invertible user operations. We can define it as a descendant class of `OTBase` by adding the following two members: the inversion function `inv` and the formula expressing the property IP_1 (see [12]). The latter asserts that the effect of any operation *op* applicable to a state s can be undone by means of `inv` *op*.

```
Class OTInv (X cmd : Type) (M : OTBase X cmd) := {
 inv : cmd → cmd;
 ip1 : forall op s s1, interp op s = Some s1 → interp (inv op) s1
     = Some s}.
```

Other, more subtle inversion properties have also been described in the literature (e.g., properties IP_2, IP_3, see [12]). However, these properties are not satisfied by the transformation algorithms described in Section 3. For this reason we do not include them into the definition of `OTInv`.

Apart from the property C_1, we do not impose any semantical constraints on the behavior of the transformation function. In particular, C_1 is satisfied by the trivial transformation function, which always cancels operations of both

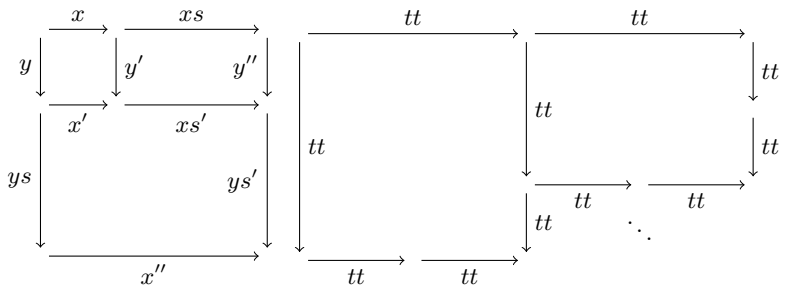
clients (e.g., `it` op_1 op_2 f = [:: `inv` op_1].) Another trivial example of the function satisfying C_1 is the function that always rolls back an operation of the client with a lower priority:

$$\texttt{it}\ op_1\ op_2\ true = [::],\ \texttt{it}\ op_1\ op_2\ false = [::\ \texttt{inv}\ op_2; op_1].$$

2.2 Transformation of composite operations (file `Comp.v`)

In the previous subsection we defined the signature of a transformation function in such a way that the result of transforming two atomic operations could be a composite operation. While this approach has multiple advantages mentioned above it also creates difficulties associated with the transformation of composite operations.

Imagine that we want to write a function transforming a composite operation op_1 through another composite operation op_2 provided that we already know how atomic operations are transformed. Of course, there is only one way to do this: first cut atomic pieces off op_1 and op_2, then transform these pieces using the transformation function `it` for atomic operations and, finally, run the transformation recursively on the remaining chunks. The following piece of code implements this behavior (cf. Fig. 3a):



(a) `mtrans` (b) Infinite looping of `mtrans`

Figure 3: Illustration of the transformation of a composite operation

```
Fixpoint mtrans (it: cmd → cmd → bool → list cmd) (op1 op2:
    list cmd) (nSteps: nat): option ((list cmd) * (list cmd)):=
match nSteps with
| 0 ⇒ None
| S nSteps' ⇒
match op1, op2 with
| nil, _ | _, nil ⇒ Some (op2, op1)
| x :: xs, y :: ys ⇒
 match mtrans it xs (it y x false) nSteps' with
```

```
 | Some (y'', xs') ⇒
  match mtrans it ((it x y true) ++ xs') ys nSteps' with
  | Some (ys', x'') ⇒ Some (y'' ++ ys', x'')
  | _ ⇒ None
  end
 | _ ⇒ None
 end
end
end.
```

Notice that we had to add a parameter *nSteps* to the definition of `mtrans` to limit the recursion depth, otherwise, Coq would reject the definition as potentially nonterminating. It is easy to see that such nontermination, indeed, may occur. For example, consider the following trivial implementation of OT (with the property C_1 also trivially satisfied).

```
Definition bad_it := (fun (_ _ : unit) (_ : bool) ⇒ [::tt;
    tt]).
Instance nonterm : OTBase unit unit :=
 {interp := (fun _ _ ⇒ Some tt); it := bad_it}.
```

Although this function works well for elementary operations, we will get an infinite loop if we try to transform composite operations. Indeed, `mtrans` loops on the following simple example after the first two iterations and the result of transformation can not be computed (cf. Fig. 3b).

```
Eval compute in mtrans bad_it [::tt] [::tt] 2.
     = Some ([:: tt; tt], [:: tt; tt])
Eval compute in mtrans bad_it [::tt] [::tt; tt] 100.
     = None
```

Although the above example looks somewhat artificial it is, in fact, similar to the following situation often encountered in practice. Imagine that two concurrent user operations are semantically inconsistent with each other, i.e. there is no sensible way to transform them so that the effect of both operations is preserved. For example, this may happen if the operation model of OT becomes sufficiently complex. In this case the only way to enforce property C_1 is to rollback the operation of one of the users and execute the operation of the other. Such implementation forces the transformation function to return a composite operation when invoked on a pair of elementary ones.

In the remainder of this section we describe a condition sufficient for practical purposes which guarantees that the result of transformation of two composite operations can be computed in a finite number of steps. We will further refer to this condition as the *computability property*.

The idea is to define two natural-valued functionals which should be interpreted as assignments of "size" and "cost" to an atomic operation:

$$\texttt{sz0}\colon cmd \to nat,\ \texttt{si0}\colon cmd \to nat.$$

The rationale here is the following: if the total "size" of a composite operation increases in the process of transformation then the value of the "cost" functional

should at the same time decrease. Since the latter is a natural number, it is guaranteed that the total "size" of the operation will stop increasing at some point and that the transformation function will terminate.

Formally speaking, we extend `sz0` and `si0` by additivity to composite operations, denoting them by `sz` and `si`. Every atomic operation is required to have nonzero "size" and both functionals `sz` and `si` are required not to increase on each transformation square from Definition 1 (i.e. the sum of values on the "transformed" right and bottom arrows should not exceed the sum of values on the "original" left and top arrows). We also assume that for each transformation square at least one of the following two statements holds:

– The operations are transformed without rollbacks. In this situation "size" functional `sz` does not increase for both operations, i.e. $\texttt{sz}\ o'_i \leq \texttt{sz}\ o_i$, $i = 1, 2$.
– If one of the operations is rolled back, the "size" of one of the transformed arrows may increase. In this situation we require that the "cost" functional strictly decreases on such transformation square.

It is easy to see that under these assumptions the result of transforming two composite operations o_1 and o_2 can be computed in less than $(\texttt{sz}\ o_1 + \texttt{sz}\ o_2)^2 + \texttt{si}\ o_1 + \texttt{si}\ o_2$ steps (i.e. atomic transformations). We formulate this as a theorem in Coq.

```
Context {X cmd: Type} (ot: OTBase X cmd) (comp: OTComp X cmd
    ot).
Theorem ot_computable: forall (op1 op2: list cmd),
 exists nSteps, mtrans it op1 op2 nSteps <> None.
```

3 Examples of verified transformation functions

In the current section we present two concrete implementations of abstract classes defined in Section 2. In each of the two cases we describe how exactly the abstract classes and signatures are instantiated and also outline the idea of the proof of corresponding convergence and computability properties. The algorithms described below are contained in the following modules of our library: `TreeOt.v`, `Fs.v`, `RichText.v`.

3.1 The case of ordered trees with labels

In this subsection we describe an OT algorithm for concurrent editing of ordered trees. The algorithm in question is a modification of the OT algorithm of Ressel for text buffers (cf. [10, 2.3.2]).

We are working with ordered trees labeled by elements of some fixed type T (i.e. a label of type T is assigned to every vertex of a tree). Of course, we will need the two most basic operations of tree editing: insertion and removal of a tree branch. Similarly to Sun's algorithm for strings (see [10, 2.3.3], [13]) we allow sequential insertion and deletion of multiple tree branches in a single

operation. We also include one more operation `EditLabel` which modifies the label of a single tree node via some user-defined set of commands TC and, otherwise, leaves the tree structure unchanged.

Since we want our OT algorithm for trees to be C_1-consistent we should first assume the OT algorithm for labels to be C_1-consistent (see Section 2). We implement this in Coq by adding several parameter variables.

```
Context {T: eqType} (TC: Type) {otT: OTBase T TC}.
```

Now, we can give the following definition for the type of elementary operations:

```
Inductive tree_cmd : Type :=
 | EditLabel of TC
 | TreeInsert of nat & list (tree T)
 | TreeRemove of nat & list (tree T)
 | OpenRoot of nat & tree_cmd.
```

The first three constructors of this type correspond to the operations modifying the root node of a tree, while a sequence of `OpenRoot` constructors can be used to specify a position in the tree to which the first three operations are to be applied.

The semantics of the interpretation function `interp` for this operation set is as follows.

- Case `EditLabel` tc. The operation executes command tc on the label of the tree's root node using the function `interp` specified in the "parameter" class otT.
- Case `TreeInsert` n l. The operation inserts the list l into n-th position of the children list of the root node. `None` is returned if a range check error occurs during this process.
- Case `TreeRemove` n l. The operation compares the list l with the sublist of branches of the root node starting at n-th position. If these lists are the same then the corresponding sublist of children is removed from the root node. Otherwise, or if a range check error occurs, the function returns `None`.
- Case `OpenRoot` n c. The operation applies operation c to the n-th child of the root. The operation returns `None` if there is no child with such index.

The behavior of the interpretation function is illustrated in Fig. 4.

It is easy to see that the above set of operations satisfies property IP_1 provided so does the algorithm for labels. The inversion function can be defined by swapping `TreeInsert` and `TreeRemove` constructors:

```
Context (ipT : OTInv _ _ otT).
Fixpoint tree_inv (c : tree_cmd) :=
match c with
| EditLabel c'    ⇒ EditLabel (@inv _ _ _ ipT c')
| TreeInsert n l ⇒ TreeRemove n l
| TreeRemove n l ⇒
 if l is [::] then TreeInsert 0 [::] else TreeInsert n l
| OpenRoot n c'   ⇒ OpenRoot n (tree_inv c')
```

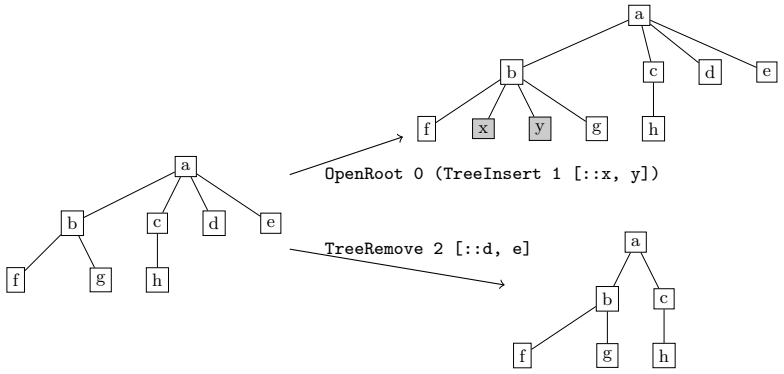


Figure 4: An example illustrating the behavior of operations `TreeInsert` and `TreeRemove`

```
end.
Instance treeInv: (OTInv (tree T) tree_cmd treeOT) := {inv :=
    tree_inv}.
```

Notice that the fact that we compare the list of nodes in the actual model with the list of nodes specified in `TreeRemove` command before executing the latter is essential for checking property IP_1.

The next step is to define the transformation function for tree operations. First, we define some auxiliary functions.

```
Definition tr_ins (len: nat) (n1 n2: nat): nat :=
 if (n1 < n2) then n1 else n1 + len.
Definition tr_rem (len: nat) (n1 n2: nat): option nat :=
 if (n1 < n2) then Some n1 else
  (if (n1 >= n2 + len) then Some (n1 - len) else None).
Fixpoint cut {X} (l : list X) (sc rc : nat) :=
match sc, rc, l with
| S sc', _, x :: xs ⇒ x :: (cut xs sc' rc)
| 0, S rc', x :: xs ⇒ cut xs sc rc'
| _, _, _ ⇒ l
end.
```

The function `cut` l n m removes from l the sublist of length m starting from n-th position. Now we are all set to define the transformation function for trees. The idea behind its definition is that we attempt to preserve the intuitive "effect" of both operations op_1 and op_2. To accomplish this we have to perform a large case-by-case analysis.

```
Fixpoint tree_it (op1 op2: tree_cmd) (f: bool): list tree_cmd :=
let triv := [:: op1] in
match op1, op2 with
| EditLabel c1, EditLabel c2 ⇒
 map EditLabel (@it _ _ otT c1 c2 f)
```

```
| EditLabel _, _ | _, EditLabel _ ⇒ triv
| OpenRoot n1 tc1, OpenRoot n2 tc2 ⇒
 if n1==n2 then map(OpenRoot n1) (tree_it tc1 tc2 f) else triv
| TreeRemove n1 l1, OpenRoot n2 tc2 ⇒
 match tr_rem (size l1) n2 n1 with
 | None ⇒ let i := n2 - n1 in
  match replace i (tree_interp tc2 (nth i l1)) l1 with
  | Some l1' ⇒ [:: TreeRemove n1 l1']
  | _ ⇒ triv
  end
 | _ ⇒ triv
 end
| _, OpenRoot _ _ ⇒ triv
| OpenRoot n1 tc1, TreeInsert n2 l2 ⇒
 [:: OpenRoot (tr_ins (size l2) n1 n2) tc1]
| OpenRoot n1 tc1, TreeRemove n2 l2 ⇒
 match tr_rem (size l2) n1 n2 with
 | Some n1' ⇒ [:: OpenRoot n1' tc1]
 | None ⇒ nil
 end
| TreeInsert n1 l1, TreeInsert n2 l2 ⇒
 if (n1==n2) then
 (if f then triv else [::TreeInsert (n1+size l2) l1])
 else [:: TreeInsert (tr_ins (size l2) n1 n2) l1]
| TreeInsert n1 l1, TreeRemove n2 l2 ⇒
 let len := size l2 in
 if n1≤n2 then triv else
 if n1≥n2+len then[::TreeInsert (n1-len) l1] else nil
| TreeRemove n1 l1, TreeRemove n2 l2 ⇒
 let (len1, len2) := (size l1, size l2) in
 (if n2 + len2≤n1 then [::TreeRemove (n1-len2) l1] else
  (if n2 ≤ n1 then
   (if n2 + len2 < n1 + len1
    then [:: TreeRemove n2 (cut l1 0 (len2+n2-n1))]
    else nil)
   else [:: TreeRemove n1 (cut l1 (n2-n1) len2)]))
| TreeRemove n1 l1, TreeInsert n2 l2 ⇒
 let (len1, len2) := (size l1, size l2) in
 if n1 + len1 ≤ n2 then triv else
 (if n2 ≤ n1 then [:: TreeRemove (n1+len2) l1] else
  match insert (n2 - n1) l2 l1  with
  | Some l1' ⇒ [:: TreeRemove n1 l1']
  | None ⇒ triv
  end)
end.

Instance treeOT: (OTBase (tree T) tree_cmd) :=
 {interp := tree_interp; it := tree_it}.
```

The above transformation function always cancels `TreeInsert` whenever it conflicts with a concurrent `TreeRemove`. In order to transform two concurrent `EditLabel` operations the transformation function for the label type is invoked. Notice that the priority flag f is used in this piece of code to resolve the conflicting situation when two lists of trees are concurrently inserted into the same position.

The proof of the fact that `tree_it` satisfies property C_1 is rather bulky and technical. After applying induction over op_1 it essentially boils down to proving a number of commutation lemmas about list operations. The proof of C_1 itself takes 250 lines of Coq code, in addition, further 700 lines are occupied by commutation lemmata for list operations.

Notice that the problem of transforming composite operations mentioned in Section 2.2 does not arise for `tree_it` provided it does not arise for the "parameter" algorithm otT. The reason for this is that `tree_it` can only return a proper composite operation as a result of transformation of two `EditLabel` operations (which is clear from the examination of the definition).

3.2 The case of unordered trees

Now we describe a transformation algorithm for unordered trees i.e., trees for which the order among siblings is irrelevant. Informally speaking, the difference between the datatypes of ordered and unordered trees is the same as between the datatypes of ordered lists and sets. The directory structure of a filesystem may serve as an example of an unordered tree.

From the implementation viewpoint it will be convenient for us to consider unordered trees as usual ordered trees whose branches are sorted with respect to some total ordering defined on the type of labels. In particular, such implementation simplifies the test for equality and also allows us to implement unordered trees in Coq as a *subset type*, i.e. as an ordered pair consisting of a tree and the evidence (proof object) of its sortedness (see, e.g., [1, § 6]).

The main difference between the operation set for unordered trees and the operation set from the previous subsection is that now we refer to nodes of an unordered tree using their labels rather than indices. We support 3 different atomic tree operations: modification of the node's label (file renaming), insertion and deletion of a subtree.

```
Inductive raw_fs_cmd T :=
 | Edit of T & T
 | Create of tree T
 | Remove of tree T
 | Open of T & raw_fs_cmd.
```

More formally, semantics of these operations can be described as follows.

– Case `Edit` l_1 l_2. The operation seeks a child with label l_1 among the children of the root node. If such a child is found, its label is changed to l_2. `None` is returned if there is no such child, or if there is another child with label l_2 among the children of the root node.

– Case `Create` t. The operation adds t to the set of children of the root node. `None` is returned if there is already another node with the same label.
– Case `Remove` t. The operation looks for a child of the root node which coincides with t and then removes it. `None` is returned if such a child is not found.
– Case `Open` l c. The operation looks for a child with label l and applies operation c to it. As before, `None` is returned if there is no such child.

The transformation function presented below is even simpler as compared to the function `tree_it` from Section 3.1 because in the context of unordered trees there is no need to compare indices of operations and only node labels are to be taken into account (`value` t denotes the label of the root node of t).

```
Fixpoint fs_it (op1 op2 : raw_fs_cmd) (f : bool) : seq fs_cmd :=
match op1, op2 with
| Edit l1 l1', Edit l2 l2' ⇒
 match l1 == l2, l1' == l2' with
 |false, false ⇒ [:: op1]
 |true, true ⇒ [::]
 |true, false ⇒ (if f then [::] else [:: Edit l2' l1'])
 |false, true ⇒ [:: Edit l2' l2]
 end
| Edit l1 l1', Create t2 ⇒
 if l1' == value t2 then [:: Remove t2; op1] else [:: op1]
| Edit l1 _, Remove t2 ⇒
 if l1 == value t2 then [::] else [:: op1]
| Create t1, Edit l2 l2' ⇒
 if value t1 == l2' then [::] else [:: op1]
| Create t1, Create t2 ⇒
  if value t1 == value t2 then merge_trees t1 t2 else [:: op1]
| Remove t1, Edit l2 l2' ⇒
  if l2 == value t1 then [:: Remove (Node l2' (children t1)) ]
      else [:: op1]
| Remove t1, Remove t2 ⇒
 if value t1 == value t2 then [::] else [:: op1]
| Remove t1, Open l2 yc ⇒
 if value t1 == l2 then
 (if fs_interp yc t1 is Some t then [:: Remove t]
  else [::])
 else [:: op1]
| Open l1 c1, Edit l2 l2' ⇒
 if l1 == l2 then [:: Open l2' c1 ] else [:: op1]
| Open l1 _, Remove t2 ⇒
 if value t2 == l1 then [::] else [:: op1]
| Open l1 c1, Open l2 c2 ⇒
  if l1 == l2 then map (Open l1) (fs_it c1 c2 f)
  else [:: op1]
| _, _ ⇒ [:: op1]
end.
```

In the above piece of code the priority flag f is used to resolve the conflicting situation when two users concurrently attempt to replace a node's label with

two different labels. There is a notable conflicting situation which did not occur in the previous section, namely when two different trees with identical labels are concurrently inserted into the same node. We handle such a conflict by recursively merging the contents of these trees (`merge_trees` t_1 t_2 generates a sequence of create operations for all descendants of t_1 that are not descendants of t_2).

Notice that the above transformation function can return a composite operation as a result of transforming two atomic `Create` operations. In order to verify that the result of transformation can always be computed in a finite number of steps we use the sufficient condition for computability i.e. we define certains functionals `fs_sz0`, `fs_si0`: `raw_fs_cmd` $\rightarrow$ `nat` and check that they satisfy the inequalities of Section 2.2.

4 Related work

In [3] A. H. Davis et al. have proposed an OT algorithm for editing structured data (e.g., trees). However, their protocol has not been formally verified and, moreover, assumed the presence of a garbage collector.

In [7], [10] SPIKE theorem prover and UPPAAL TIGA model checker were used to find counterexamples violating the correctness of several published OT algorithms for strings. As a result, it was shown that all tested algorithms violate property C_2 while some of them violate even C_1. In [9] Coq was applied to the same problem of finding counterexamples to property C_2.

However, to the best of our knowledge, our work is the first to obtain a formal proof of property C_1 by means of a proof assistant.

5 Conclusions and future work

In our work we attempted to formalize OT algorithms for two different datatypes commonly met in practice and obtained the following results:

- a library containing commutation lemmas for lists and trees has been developed (file `ListTools.v`);
- the correctness of several inclusive transformation functions has been verified by means of Coq (Sections 3.1, 3.2).

Our algorithms may be easily extracted from the library (either automatically or manually) and be used in real world applications.

Along the way, we propose a generalized signature of the transformation function `it` which allows the transformation result to be a composite operation. The latter allows the implementation of `it` to be more flexible and, moreover, makes possible to define semantically correct transformation functions without polluting the set of commands with domain-irrelevant entities such as *nop* operation. We also present a sufficient condition ensuring computability of transformed operations in this setting, which is an essential requirement for any practical

implementation. We believe that the ideas behind our approach are general enough to be successfully applied to other data models as well.

There is a number of different directions one could take for future research within this topic. For example, one could try to figure out whether property C_2 holds for the algorithm formulated in Section 3.2. One could also attempt to formalize OT algorithms for different datatypes and try different sets of operations for the datatypes used in this article. For example, our library already contains an attempt to extend the operation set of Section 3.1 with two more operations `TreeUnite` and `TreeFlatten` which may be useful for the implementation of rich-text editors (see `RichText.v`).

References

1. Chlipala, A.: Certified programming with dependent types. The MIT press (2013)
2. Cormack, G.: A counterexample to the distributed operational transform and a corrected algorithm for point-to-point communication. Tech. Rep. CS-95-06, Univ. Waterloo (1995)
3. Davis, A.H., Sun, C., Lu, J.: Generalizing operational transformation to the standard general markup language. In: Proc. Conf. Comp. Supported Coop. Work. pp. 58–67 (2002)
4. Ellis, C., Gibbs, S.: Concurrency control in groupware systems. In: Proc. 1989 ACM SIGMOD Int. Conf. on Management of data. vol. 18, pp. 399–407 (1989)
5. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Tech. Rep. RR-6455, Inria Saclay Île de France (2015)
6. Ignat, C.L., Norrie, M.C.: Customizable collaborative editor relying on treeOPT algorithm. In: Proc. 8th European Conf. Comp. Supported Coop. Work. pp. 315–334. ECSCW'03 (2003)
7. Imine, A., Rusinowitch, M., Molli, P., Oster, G.: Formal design and verification of operational transformation for copies convergence. Theor. Comput. Sci. 351(2), 167–183 (2006)
8. Lushman, B., Cormack, G.V.: Proof of correctness of Ressel's adOPTed algorithm. Inform. Process. Lett. 86(6), 303–310 (2003)
9. Lushman, B., Roegist, A.: An automated verication of property tp2 for concurrent collaborative text buffers. Tech. Rep. CS-2012-25, Univ. Waterloo (December 2012)
10. Randolph, A., Boucheneb, H., Imine, A., Quintero, A.: On consistency of operational transformation approach. Elec. Proc. Theor. Comput. Sci., vol. 107, pp. 45–59 (2013)
11. Suleiman, M., Cart, M., Ferrié, J.: Concurrent operations in a distributed and mobile collaborative environment. In: Proc. 14th Int. Conf. Data Eng., Feb. 23–27, Orlando, FL, USA. pp. 36–45 (1998)
12. Sun, C.: Operational transformation frequently asked questions and answers (2010)
13. Sun, C., Jia, X., Zhang, Y., Yang, Y., Chen, D.: Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. ACM Trans. Comput.-Hum. Interact 5(1), 63–108 (1998)