VERIFIED OPERATIONAL TRANSFORMATIONS FOR TREES

Sergey Sinchuk, *Pavel Chuprkov*, Konstantin Solomatov

Interactive Theorem Proving 2016

INTRODUCTION

REAL-TIME COLLABORATIVE EDITOR

A collaborative editor allows multiple users to edit a shared object (e.g., *Google Wave, Overleaf, Google Docs, …*).

The following properties are required:

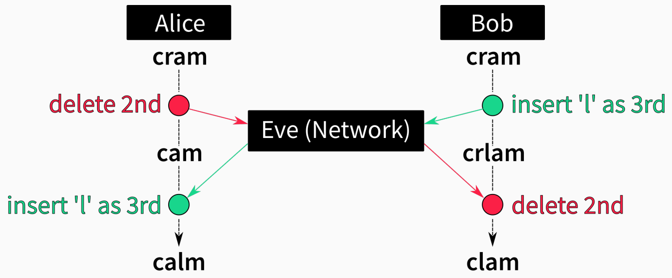
- Editing operations are interactive.
- The shared object is eventually consistent.
- Inter-user update latency is minimized.

Solution (almost):

- per-user replicas;
- remote execution.

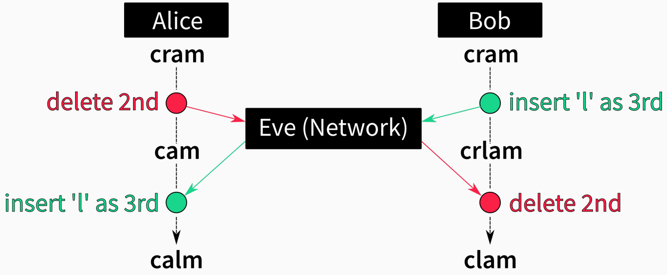
EVENTUALLY INCONSISTENT

But consider the following concurrent interaction:



EVENTUALLY INCONSISTENT

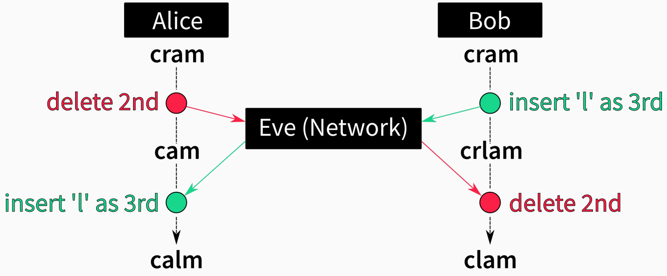
But consider the following concurrent interaction:



Problem: remote operations apply to a modified state.

EVENTUALLY INCONSISTENT

But consider the following concurrent interaction:



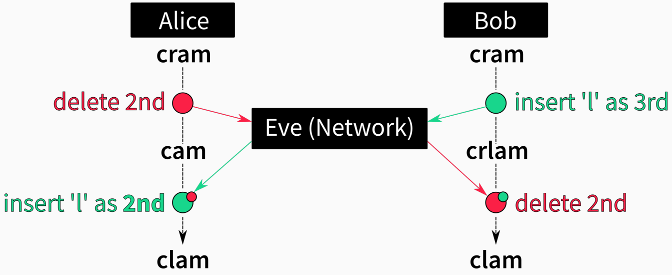
Problem: remote operations apply to a modified state.

Solution: transform remote operations to respect the change.

OPERATIONAL TRANSFORMATION — EXAMPLE

Consider the same interaction, but:

- Instead of applying ● Alice applies ●, which is a version of the former that has been *transformed through* ● to respect its changes.
- Bob does the same for ●.



Now, final states are the same.

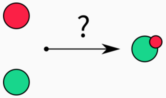
OPERATIONAL TRANSFORMATION — STRUCTURE

To use an operational transformation we must understand:

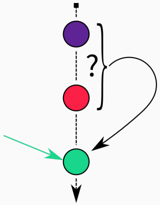
- how two elementary operations are transformed;
- the order in which operations are transformed.

Operational transformation

Transformation function



Integration algorithm

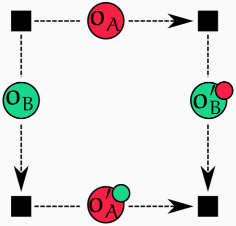


OPERATIONAL TRANSFORMATION — PROPERTIES

In the literature certain properties of the transformation function have been found that guarantee eventual consistency of data for any sequence of operations and any network behavior.

Definition (Convergence property C_1)

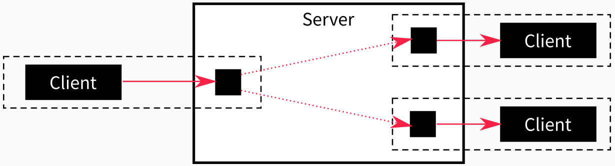
Given two operations issued by two different users o_A and o_B, and they corresponding transformed versions o'_A and o'_B, the results of executing $o_A \circ o'_B$ and $o_B \circ o'_A$ are the same.



OPERATIONAL TRANSFORMATION — MULTIUSER

- The property C_1 guarantees convergence only for 2 users.
- A stronger property C_2 works in the general case but is hard to meet.

For the client-server architecture C_1 is enough:



1-to-1 OT	⬚
Virtual server data objects	■
Virtual execution	⋯→

OT FORMALIZATION

OVERVIEW

The formalization of an OT for a particular data model consists of:

- formalization of the data model and the operations set;
- an interpretation function `interp` that defines operation semantics;
- a transformation function `it` that performs transformation;
- proof of the formula expressing property C_1 of `it`;

Formalization toolkit:

- The Coq Proof Assistant (Coq)
- A Small Scale Reflection Extension (SSReflect)

INTERPRETATION FUNCTION

Domains:

- X — the set of data object states ■
- $\mathtt{cmd}$ — the set of operations ●

There could be certain circumstances under which a particular operation is inapplicable to the given data object state:

- *Text Editor*: Remove/insert a symbol at an non-existent position
- *Filesystem*: Remove/edit a file that does not exist

Thus, we arrive to the following signature:

$$\mathtt{interp} \colon \mathtt{cmd} \to \mathrm{X} \to \mathtt{option}\,\mathrm{X}.$$

TRANSFORMATION FUNCTION — CLASSIC

There is a straightforward signature for transformation function `it`:

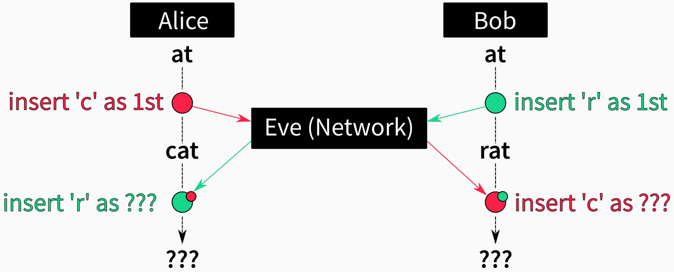
$$\mathtt{it}_0 : \mathbf{cmd} \to \mathbf{cmd} \to \mathbf{cmd}.$$

In terms of the circled notation we used so far: `it`(🟢, 🔴) = 🟢.

Although this signature served well in the literature, we are going to introduce two modifications aiming to simplify implementation of `it` .

TRANSFORMATION FUNCTION — PRIORITIES

Consider the following conflicting situation:



Both transformation functions are executed under almost the same transformation context. Extra care must be taken to ensure C_1.

TRANSFORMATION FUNCTION — PRIORITIES

There are many ways to solve the conflict that can be found in the literature:

- Cancel both operations.

TRANSFORMATION FUNCTION — PRIORITIES

There are many ways to solve the conflict that can be found in the literature:

- Cancel both operations. *Semantics and UX are broken.*

TRANSFORMATION FUNCTION — PRIORITIES

There are many ways to solve the conflict that can be found in the literature:

- Cancel both operations. *Semantics and UX are broken.*
- Use model-specific information (e.g., a letter that has a lower Ascii code goes first).

TRANSFORMATION FUNCTION — PRIORITIES

There are many ways to solve the conflict that can be found in the literature:

- Cancel both operations. *Semantics and UX are broken.*
- Use model-specific information (e.g., a letter that has a lower Ascii code goes first). *The definition of* `it` *becomes unnecessary complex.*

TRANSFORMATION FUNCTION — PRIORITIES

There are many ways to solve the conflict that can be found in the literature:

- Cancel both operations. *Semantics and UX are broken.*
- Use model-specific information (e.g., a letter that has a lower Ascii code goes first). *The definition of* `it` *becomes unnecessary complex.*
- Embed user IDs (or priorities) into the operation.

TRANSFORMATION FUNCTION — PRIORITIES

There are many ways to solve the conflict that can be found in the literature:

- Cancel both operations. *Semantics and UX are broken.*
- Use model-specific information (e.g., a letter that has a lower Ascii code goes first). *The definition of* `it` *becomes unnecessary complex.*
- Embed user IDs (or priorities) into the operation. *This information is irrelevant to operation's main purpose — data modification.*

TRANSFORMATION FUNCTION — PRIORITIES

There are many ways to solve the conflict that can be found in the literature:

- Cancel both operations. *Semantics and UX are broken.*
- Use model-specific information (e.g., a letter that has a lower Ascii code goes first). *The definition of* `it` *becomes unnecessary complex.*
- Embed user IDs (or priorities) into the operation. *This information is irrelevant to operation's main purpose — data modification.*
- Inform a transformation function externally about operation priorities. *The consistency condition* C_1 *must now quantify over these priorities.*

We choose the last option since it has better logical consistency and ease of implementation. For client-server architecture boolean flag is enough:

$$\mathtt{it}_1 : \mathrm{cmd} \to \mathrm{cmd} \to \mathtt{bool} \to \mathrm{cmd}.$$

TRANSFORMATION FUNCTION — RESULT

Here are few operation transformation “patterns” that we encountered during the course of OT implementation:

- *do nothing* (e.g., editing of a deleted word);
- *cancel one operation and apply another* (e.g., contradicting operations);
- *split operation* (e.g., words removal crosses text formatting boundaries).

In all these cases we do not use any new kinds of operations, but rather we use a combination of existing operations *(compound operation)*:

$$\mathtt{it}_1 : \mathtt{cmd} \to \mathtt{cmd} \to \mathtt{bool} \to \mathtt{list}\ \mathtt{cmd}.$$

COMPLETE DEFINITION

Everything that we have considered so far can be captured in the Coq class:

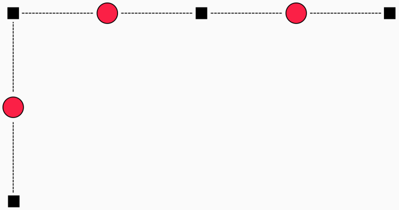
```
Class OTBase (X cmd: Type) := {
  interp:cmd → X → option X;
  it    :cmd → cmd → bool → list cmd;
  it_c1 :forall (op₁ op₂: cmd)(f: bool)(s s₁ s₂: X),
    interp op₁ s = Some s₁ → interp op₂ s = Some s₂ →
      let s₂₁:= exec_all interp (Some s₂) (it op₁ op₂ f) in
      let s₁₂:= exec_all interp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

Where `exec_all` executes a list of operations by the sequential application of `interp`.

ONE CAVEAT

The introduction of composite operations has an unpleasant effect:

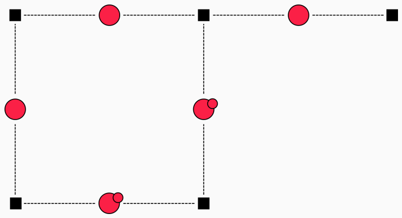
- consider $\mathtt{cmd} = \{$●$\}$;
- let the transformation be $\mathtt{it}($●, ●$) = [$●, ●$]$;
- assume that Alice has executed ● only once, but Bob has done it twice.



ONE CAVEAT

The introduction of composite operations has an unpleasant effect:

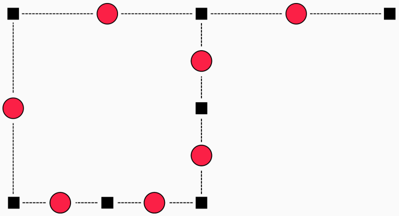
- consider $\mathtt{cmd} = \{●\}$;
- let the transformation be $\mathtt{it}(●, ●) = [●, ●]$;
- assume that Alice has executed ● only once, but Bob has done it twice.



ONE CAVEAT

The introduction of composite operations has an unpleasant effect:

- consider $\mathbf{cmd} = \{●\}$;
- let the transformation be $\mathtt{it}(●, ●) = [●, ●]$;
- assume that Alice has executed ● only once, but Bob has done it twice.



Oops!

TERMINATION CONDITION

To overcome the problem we use a sufficient termination condition.

Formally, we define two measures: `size` and `cost` : $\mathtt{cmd} \to \mathbb{N}$, and `size` must be greater than zero. Finally, we extend those measures to compound operations by additivity.

Now consider any transformation that starts with some 🔴, 🟢 and results in 🔴 and 🟢, where the latter operations are compound. It must hold that:

- the total `size` does not increase;
- the total `cost` does not increase;
- at least one of the following is true:
    - `size` of neither 🔴 or 🟢 decreases
    - the total `cost` must decrease.

Intuitively, composite operations do not occur while `size` does not change, but `cost` can not decrease forever.

APPLICATIONS

SOFTWARE ENGINEERING CLASSICS

Classic software engineer “correctness proof” techniques:

- Extensive (automated) unit testing *does not cover all cases.*
- Proof by hand *is error prone if too bulky*.

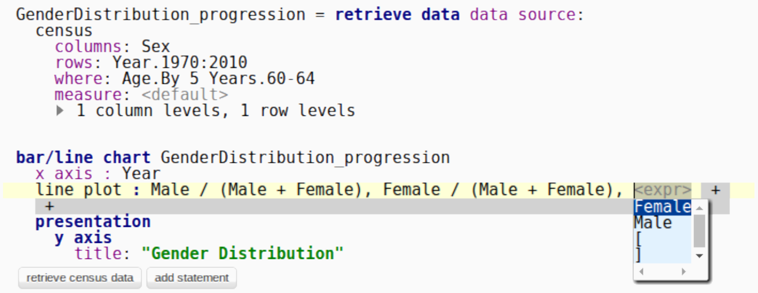
Those tools are industry standards and are time-proven, but OT has a few specificities that complicates correctness check:

- OT lies at the very core of the system and, thus, is a critical component.
- The number of cases in a proof is enormous.

On the way of a *JetPad* platform development we decided that an ultimate tool is required — the formal verification.

TEXT EDITOR — PROBLEM STATEMENT

The first component of the *JetPad* platform is a *projectional* text editor.



```
GenderDistribution_progression = retrieve data data source:
  census
    columns: Sex
    rows: Year.1970:2010
    where: Age.By 5 Years.60-64
    measure: <default>
    ▸ 1 column levels, 1 row levels

bar/line chart GenderDistribution_progression
  x axis : Year
  line plot : Male / (Male + Female), Female / (Male + Female), <expr>  +
   +                                                            Female
  presentation                                                  Male
    y axis                                                      [
      title: "Gender Distribution"                              ]
```

To support modularity and projectional nature of the editor, the data model has to fulfil the following requirements:

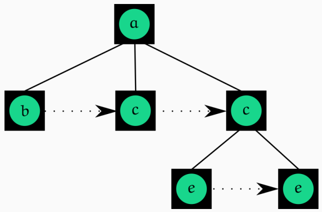
- a hierarchical tree-like structure;
- the specific data content should be abstracted away.

TEXT EDITOR — MODEL DESCRIPTION

We will use as a model an ordered rooted tree where each internal node has a label, which is itself an instance of `OTBase`:

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

Model



Operations

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

`OpenRoot 2 (TreeRemove 0 [::e])` removes the first e node.

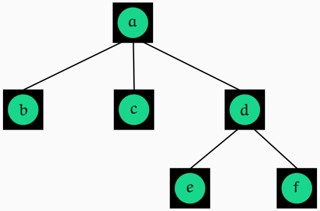
C_1 has been proven; computability is trivial.

FILE SYSTEM

To collaboratively store and manage documents created with the text editor, *JetPad* uses an internal file system, which is also naturally a tree, but is different from the text editor:

- the tree is unordered;
- operations do not aggregate (affect only a single file);
- the `Edit` operation has a simple replacing semantics.

Model



Operations

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

C_1 and computability have been proven.

EXPERIMENTAL RICH TEXT EDITOR OPERATIONS

There is a tradeoff between operations complexity and semantic accuracy. Consider the following scenario:

1. Alice and Bob start with “I love IP”.
2. Alice decides to insert “T” between “I” and “P”.
3. Bob decides to make “IP” italic.

If OT supports only letter by letter operations then they will get “I love *I*T*P*”.

To remedy the situation we introduced two more operations for text editors:

```
| TreeUnite : nat → T → list (tree T) → tree_cmd
| TreeFlatten : nat → tree T → tree_cmd
```

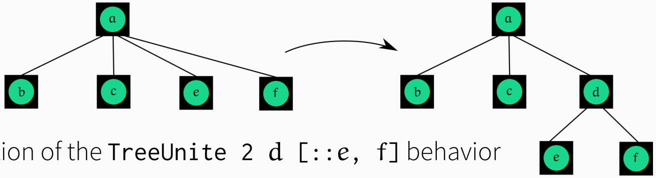


Illustration of the `TreeUnite 2 d [::e, f]` behavior

CONCLUSION

CONCLUSION

From our perspective the most notable implications from our work are:

- ITP makes formal OT verification feasible even for complex data models such as hierarchically structured data;
- tools are relatively easy to master by an average software engineer;
- encountered contradictions are easily convertible to definition errors.

Our contribution to the ITP/OT:

- modular library of OT definitions (`github.com/JetBrains/ot-coq`);
- compound operations and their OT computability property;
- Coq correctness proof of text editor and FS OT implementations.

ITP/OT's contribution to us:

- Several implementation errors unnoticed during testing were fixed.

Thank you for your attention and for the great work you do!