



南京大学

本科生毕业论文 (申请学士学位)

论文题目 面向 **Redis List** 的操作转换函数的设计与验证

作者姓名 纪业

学科、专业方向 计算机应用

指导教师 魏恒峰 讲师

研究方向 分布式算法

2018 年 6 月 2 日

学 号：141220044

论文答辩日期：2018 年 6 月 7 日

指 导 教 师： (签字)

Design and Verification of Operational Transformation Functions for Redis List

by

Ji Ye

Supervised by

Wei Hengfeng (Research Assistant)

A dissertation submitted to
the undergraduate school of Nanjing University
in partial fulfilment of the requirements for the degree of

BACHELOR

in

Computer Application



Department of Computer Science and Technology
Nanjing University

June 2, 2018

南京大学本科生毕业论文中文摘要首页用纸

毕业论文题目： 面向 Redis List 的操作转换函数的设计与验证

计算机应用 专业 2014 级学士姓名： 纪业
指导教师（姓名、职称）： 魏恒峰 讲师

摘 要

协同编辑系统允许处于不同地点的多用户同时编辑同一份文档。为了获得较快的响应和较高的可用性，该类系统通常采用副本复制技术，即将文档副本存放在不同的节点上。用户可以在某个本地（或就近的）副本节点上编辑文档，并将做出的修改异步地传递给其他副本节点。与此同时，协同编辑系统必须保证文档的一致性。

复制列表 (Replicated List) 经常被用以建模协同编辑应用的核心功能，如插入字符、删除字符等。收敛性 (Convergence) 是复制列表的一种常用规约，是对文档一致性要求的形式化描述。基于操作转换 (Operational Transformation; 简称 OT) 的思想，研究者提出了多种实现收敛性的协议。一个基于操作转换的协同编辑系统通常包含两个关键部分：上层的并发控制算法以及底层的操作转换函数。控制算法负责决定何时对哪些操作进行转换，而操作转换函数则决定如何对两个操作进行转换。

要保证收敛性，底层的操作转换函数需要满足一定的性质，如本文所关心的 CP1 (Convergence Property 1) 性质。现有的针对复制列表的操作转换函数通常仅支持简单的插入、删除等基本操作。这对于实际应用来说是不够的。本文参考 Redis 开源系统中的列表数据结构，实现了 14 种非阻塞列表操作的操作转换函数，并使用 TLA+ 工具对其正确性进行了验证。

本文的主要贡献包括：

- 我们基于操作所涉及的列表范围，将 Redis 列表数据结构所支持的 14 种非阻塞操作分为三类，并针对每一类（尤其是最具广泛性的第三类操作）设计了相应的操作转换函数。
- 我们使用 TLA+ 工具验证（三类操作的）操作转换函数的正确性，即是否满足 CP1 性质。我们对验证的性能也作了统计与分析。

关键词： 协同编辑系统；操作转换；Redis 列表数据结构；TLA+

南京大学本科生毕业论文英文摘要首页用纸

DESIGN and Verification of Operational Transformation Functions for Redis List

SPECIALIZATION: Computer Application

POSTGRADUATE: Ji Ye

MENTOR: Wei Hengfeng (Research Assistant)

Abstract

Collaborative editing systems allow multiple users to concurrently edit the same document. For low latency and high availability, such systems often replicate the document at several replicas. A user can edit the document at a local or nearby replica, and the updates are propagated to other replicas asynchronously. Meanwhile, the documents at different replicas should be kept consistent.

The replicated list object has been frequently used to model the core functionality (e.g., insertion and deletion) of collaborative editing systems. Convergence is a common specification of a replicated list, a formal requirement of document consistency. Based on the idea of operational transformation (OT), the researchers have proposed several protocols to achieve convergence. A typical OT-based system consists of two layers: the concurrency control algorithm at the top layer and the OT functions at the bottom layer. The concurrency control algorithm decides the time to transform and the operations to be transformed, while the OT functions decide the way how a pair of operations are transformed.

To achieve convergence, the OT functions are required to satisfy some properties, such as CP1 (Convergence Property 1) we are concerned about in this paper. Existing OT functions for replicated lists are often limited to the character-wise operations, i.e., insertion and deletion. In this paper, we have designed a complete set of OT functions for 14 non-blocking list operations supported by Redis, a well-known open-source distributed key-value store. We have also verified the correctness (with respect to cP1) of these OT functions using TLA+.

We make two main contributions in this paper:

- We divide all the 14 non-blocking operations of Redis list into 3 categories, according to the positions involved. Then we design the OT functions for each category. We focus on the third category, consisting of the most general operations.
- We verify the correctness of the OT functions of three categories with respect to CP1 using TLA+. The performance of verification is also given.

keywords: collaborative editing system, operational transformation, Redis list, TLA+

目 次

目 次	v
1 前言	1
1.1 应用背景：协同编辑应用	1
1.2 技术背景	1
1.2.1 复制列表的规约	1
1.2.2 操作转换思想	2
1.2.3 基于“操作转换”的协议	3
1.3 Redis 系统	4
1.3.1 Redis 简介	4
1.3.2 Redis 特点	4
1.4 本文研究工作：面向 Redis List 的 OT 函数的设计与验证	5
1.5 论文组织	5
2 相关工作	6
2.1 OT 函数的性质	6
2.2 OT 函数的设计	6
2.2.1 第一类 OT 函数的设计	7
2.2.2 第二类 OT 函数设计	8
2.3 OT 函数的验证	8
3 Redis List OT 函数设计	10
3.1 Redis List API 分类	10
3.1.1 Redis List API 简介	10
3.1.2 Redis List API 分类	11
3.2 第三类 OT 函数设计	12
3.3 其它 OT 函数设计	16

目 次	vi
4 基于 TLA+ 的 OT 函数验证	17
4.1 TLA+ 简介	17
4.1.1 TLA+ Modules	18
4.2 使用 TLA+ 描述 OT 函数	18
4.2.1 List 相关操作表示	18
4.2.2 命令执行的表示	21
4.2.3 OT 函数的描述	21
4.3 正确性验证	24
4.3.1 TLA+ Model Checker 设置及实验环境	25
4.3.2 实验结果	26
5 结论与未来工作	29
5.1 工作总结	29
5.2 研究展望	29
致 谢	31
A 附录	32
A.1 OT 函数的设计	32
A.1.1 第一类 OT 函数的设计	32
A.1.2 第二类 OT 函数设计	33

第一章 前言

本文基于协同编辑的应用背景，出于实际需求，参考 Redis 开源系统中的列表数据结构和 Jupiter 协议，完成了 OT 函数的设计和验证。

可再详细一点，重点在于为什么做。可以将摘要缩写一下。注意：像 OT 这种缩写词，第一次出现之前要有介绍。

1.1 应用背景：协同编辑应用

协同编辑系统，如 Google Docs^①、Apache Wave^②、wikis^[7] 等允许不同地点的用户同时编辑同一份文档。为了获得较快的响应和较高的可用性，系统会在不同的地点或设备进行文档的复制。用户可以在某个副本上进行文档的编辑，并将做出的修改异步地传递给其他副本。本地操作可以立即执行，不必等待服务器处理完再响应用户操作。系统必须保证编辑的一致性，即在所有用户完成文档的编辑后，所有的副本内容一致。

1.2 技术背景

复制列表 (Replicated List) 经常被用以建模协同编辑应用的核心功能，如插入字符、删除字符等^{[7][8]}。收敛性 (Convergence) 是复制列表的一种常用规约^[7]。基于操作转换 (Operational Transformation; 简称 OT) 的思想，研究者提出了多种实现收敛性的协议^{[7][9][10][11][12][13]}。

1.2.1 复制列表的规约

复制列表至少需要支持三类常见操作: (1) Insert: 在某指定位置插入给定字符; (2) Delete: 删除某指定位置上的字符; (3) Read: 读取整个列表。Ellis 等人在

^①<https://docs.google.com>

^②<https://incubator.apache.org/wave/>

1989 年提出了复制列表的收敛性规约^[2]。收敛性规约是对“系统静止状态”^[2]的一种约束。

定义 1-1 对于一个复制系统而言，如果所有操作都已在各个副本节点上得以处理——换句话说，没有操作仍在传输之中或者等待处理——则称系统处于静止状态。

定义 1-2 如果一个复制系统处于静止状态时，所有副本节点的状态（即列表内容）相同，则称该系统满足收敛性。

在收敛性的基础上，研究人员提出了另一种稍强的复制列表规约，称为强最终一致性 (Strong Eventual Consistency)^[2]。

定义 1-3 如果两个副本处理了相同的操作集合，那么它们的状态（即列表内容）必须相同。

由于大多数的复制列表协议都满足强最终一致性，所以下文所提到的收敛性指的都是强最终一致性。

1.2.2 操作转换思想

操作转换 Operational Transformation (OT) 是一种为了支持协作功能，在协作软件系统中所采用的技术。OT 最早在 1989 年被提出，是为了在纯文本文档的协同编辑中实现一致性和并发控制所发明的。经过二十余年的研究，OT 的能力已经得到了拓展，在 2009 年 OT 做为一种核心技术被 Apache Wave 和 Google Docs 所采用来实现协同编辑。OT 的基本思想是根据先前执行的并发操作的影响，对正在编辑的操作的参数进行转换或调整，使转换后的操作能够保持文档的一致性。考虑如下例子：

如图所示，在某个状态 S 时，list 内容为”ABCDE”。此时 client 端执行了一个删除第 4 个字符的操作，server 端执行了一个删除第 2 个字符的操作，如图所示，client 端 list 的内容变为“ABCE”，server 端 list 的内容变为“ACDE”。如果不进行转换，直接将操作相互传递，则 client 端接受到的操作是删除第 2 个字符，server 端接受到的操作是删除第 4 个字符，完成操作后 client 端 list 的内容为“ACE”，server 端 list 的内容为“ACD”，此时 list 的内容不同，不满足一致性。其原因在于 client 端的删除第 4 个字符的操作应该是删除

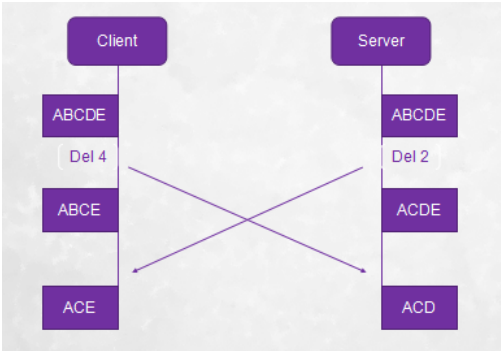


图 1-1: 不使用 OT

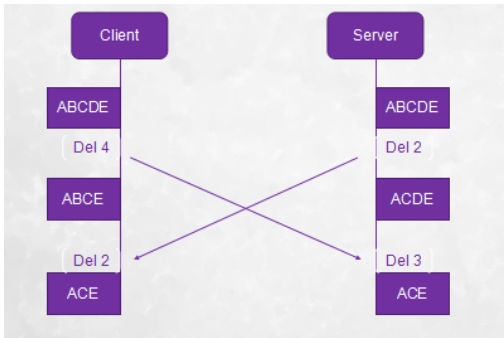


图 1-2: 使用 OT

D，但是传递到 server 端时，由于没有进行操作的转换，此时删除第 4 个字符即删除 E，造成了删除内容的不一致。

使用 OT 技术将 *Del 4* 转换为 *Del 3*, *Del 2* 转换为 *Del 2*, 这样操作的传递完成后，文档的内容仍然是一致的。

1.2.3 基于“操作转换”的协议

一个 OT 系统通常包含 2 个关键的部分：上层的控制算法和底层的 OT 函数。

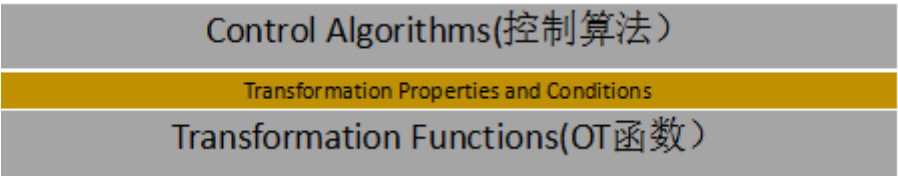


图 1-3: OT 系统结构

控制算法负责决定哪些操作以何种顺序进行转换，而具体的 OT 函数则实施具体的两个操作之间的变化。OT 函数的数量由 OT 系统的模型所支持的数据和操作类型所决定。这两者由一系列转换条件和属性结合在一起，所以整个 OT 系统的正确性就是由控制算法和 OT 函数的正确性以及协议的正确性所共同决定的。

由于这种分层结构，我们可以单独考虑 OT 函数的设计，而不必关心控制算法。

本文中 OT 函数的设计与验证基于 Jupiter 协议^[?]]。

Jupiter 是一个多用户协同软件，支持长期的远程协作，支持共享文档、共享工具以及在线的音频视频通信。

Jupiter 采用 client-server 结构，即中央服务器存储虚拟对象的状态和执行相关的程序代码，客户端进行输入输出等工作，Jupiter 算法也是基于主从结构来设计和实现的。并且 Jupiter 采用积极的并发控制算法，即不必要等待服务器处理完再响应用户操作，本地操作可以立即执行，然后再将执行的结构异步地进行传递，如果同时有两个或以上的参与者进行修改，就需要采取冲突解决算法来进行相应的操作调整。与其他 groupware system 不同，Jupiter 不使用直接建立在客户端之间的同步协议，而是使每个客户端与服务器同步，服务器将某个客户端所做的修改经过转换分享传递给其他用户。

Jupiter 算法依赖于 OT 来修复冲突的消息，保证一致性的实现。可以用状态图来表示 server 和 client 的状态，若 client 处于 (x,y)，表示此时 client 端已经产生并执行 x 两条自己的操作，并且从 server 接受并执行了 y 条操作，server 端会为每个 client 端维护一个这样的二维空间，如果执行顺序一致，则 server 和 client 在状态图上行进的路径是一直，所在位置也相同，但如果发生冲突，行进路径就会发生分歧，此时可以接收来自对方的经过 OT 转换的操作，使两者达到相同的状态，并且算法可以保证，不管 server 端和 client 端在状态图上的分歧有多远，只要接收来自对方的经过 OT 转换的操作，达到相同的状态，那么两者数据的值就一定是相同的，满足一致性的要求。

1.3 Redis 系统

1.3.1 Redis 简介

Redis 是一个开源的、可基于内存也可持久化的数据结构存储系统，可以用作数据库、缓存和消息中间件，并且提供多种语言的 API。Redis 支持多种类型的数据结构，如字符串 (string)，散列 (hash)，列表（即本文中所关心的 lists），集合 (set)，有序集合 (sorted set) 等。

1.3.2 Redis 特点

与其他 key-value 缓存产品相比，Redis 具有以下几个特点：

- Redis 支持数据的持久化，可以在磁盘中保存内存里的数据
- 除了简单的 key-value 型数据，Redis 还提供多种数据结构的存储（包括 List）

- Redis 支持主从模式的数据备份
- 可以对 Redis 中的各种数据结构执行原子操作

1.4 本文研究工作: 面向 Redis List 的 OT 函数的设计与验证

本次毕业设计的目标是实现 Redis List 所支持的 14 种非阻塞操作的 OT(Operational Transformation) 函数, 并且对实现函数的正确性进行验证。阿里云和 RedisLab 的团队目前都在对 Redis List 的操作进行开发, Redis List 操作的 OT 函数实现具有应用前景和商业前途。

在 OT 函数的设计方面, 本文使用数学公式表示出所有 OT 函数的基本形式, 并以图片和表格进行说明。

在 OT 函数的验证方面, 使用 TLA+ 完成了对所设计 OT 函数的验证, 证明其满足 CP1 正确性, 并且对验证代码的复杂度进行了相应分析。

1.5 论文组织

本文后续内容组织如下:

第2章介绍本文的相关工作, 包括系统模型和已有的相关 OT 函数的设计。

第3章介绍了 Redis 列表相关的基本命令, 并对其进行了分类, 然后进行了对应 OT 函数的设计。

第4章介绍了 TLA+, 并使用 TLA+ 完成了对上述设计好的 OT 函数的验证, 对实验结果进行了分析。

第5章是本论文的结论和以后工作的相关展望。

第二章 相关工作

2.1 OT 函数的性质

过去的 OT 研究已经建立了一套完整的 OT 算法正确性的性质标准^[2]，违反任何转换性质会导致错误的转换，从而可能使得文档不一致。我们关注的性质：

- Convergence Property 1 (CP1): 这是 Jupiter 协议正确性的必要条件。对于定义在文档状态 S 上的给定操作 $O1$ 和 $O2$ ，满足 CP1 等式： $S \circ O1 \circ OT(O2, O1) = S \circ O2 \circ OT(O1, O2)$ ，也就是说在 S 上按顺序实施 $O1, OT(O2, O1)$ 操作和实施 $O2, OT(O1, O2)$ 操作效果相同。
- Convergence Property 2 (CP2): 对于定义在文档状态 S 上的给定操作 $O1$ 和 $O2$ ，满足 CP2 等式： $OT(OT(O1, O2), OT(O3, O2)) = OT(OT(O1, O3), OT(O2, O3))$
- Inverse Property 1 (IP1)、Inverse Property 2 (IP2)、Inverse Property 3 (IP3)

关注这五个特性的原因是 CP1, CP2 是收敛性的变换性质，IP1、IP2 和 IP3 是与撤销相关的变换性质。CP1 和 CP2 被发现是在 OT 系统中实现收敛性的充分必要条件，允许操作以任何顺序进行转换，不过并不是所有 OT 系统都需要满足这两个条件。CP1 在某些 OT 系统中不是必须的，例如 GOT^[2]，这些 OT 系统从不以不同的顺序变换两个操作；CP2 同样也不是必须的，比如 Jupiter 和 Google Docs 等 OT 系统，这些 OT 系统会在操作中施加特殊的转换命令。由于是基于 Jupiter 协议所作的 OT 函数设计，所以本文中只验证所设计函数的 CP1 正确性。

2.2 OT 函数的设计

可以设计 OT 函数，并通过控制算法的调用来保证最终结果的一致性。本文中将 Ins（插入）、Del（删除）、Set（设置）单个元素的简单操作称为第一类操作，将 Ins, Del 单个区间的操作称为第二类操作，作用于多个区间的操作称为

第三类操作（本文中只讨论删除多个区间的操作）。假设 list 当前状态为 s，那么具体操作的定义如下：

操作分类	操作类型	操作效果
第一类操作	$Del(i)$	删除第 i 个元素
	$Ins(i, x)$	在位置 i 插入元素 x
	$Set(i, x)$	将位置 i 上的元素设为 x
第二类操作	$Ins(p, s)$	在位置 i 插入串 s
	$Del(p, l)$	从位置 p 开始删除长度为 l 的串
第三类操作	$Del(p_1, l_1; p_2, l_2; \dots; p_k, l_k)$	删除区间 $[p_1, p_1 + l_1], \dots, [p_k, p_k + l_k]$ 内的元素

现在 Ins,Del,Set 等简单操作相互之间的 OT 函数已经设计好,Ins,Del 单个区间的 OT 函数也已经基本上设计完成。

2.2.1 第一类 OT 函数的设计

我们定义第一类函数为第一类操作之间的 OT 函数，即 Ins,Del,Set 三种操作之间的 OT 函数，共有 $3*3=9$ 个。这里列出了 Del 操作对应于 Ins,Del,Set 的 OT 函数，全部 9 个函数的数学公式详见附录。[?]]

$$Del \left\{ \begin{array}{l} OT(Del(i), Set(j, x)) = Del(i) \\ OT(Del(i), Ins(j, x)) = \begin{cases} Del(i+1) & i \geq j \\ Del(i) & i < j \end{cases} \\ OT(Del(i), Del(j)) = \begin{cases} Del(i-1) & i > j \\ Del(i) & i < j \\ no-op & i = j \end{cases} \end{array} \right. \quad (2-1)$$

套用公式解释前言中举例的 OT 变换， $OT(Del(4), Del(2)) = Del(3)$, $OT(Del(2), Del(4)) = Del(2)$ ，与例子相符。

2.2.2 第二类 OT 函数设计

我们定义第二类函数为第二类操作之间的 OT 函数，即 Ins,Del 两种区间操作之间的 OT 函数，共 $2*2=4$ 个。这里列出了 Ins 操作对应于 Ins,Del 的 OT 函数，全部 4 个函数的数学公式详见附录。^[2]

$$OT(Ins(p1, s1), Ins(p2, s2)) = \begin{cases} Ins(p1, s1) & p1 < p2 \\ Ins(p1 + |s2|, s1) & p1 > p2 \\ Ins(p1 + |s2|, s1) & p1 = p2 \quad pr1 < pr2 \\ Ins(p1, s1) & p1 = p2 \quad pr1 > pr2 \end{cases} \quad (2-2)$$

Ins 自身的

$$OT(Ins(p1, s1), Del(p2, l1)) = \begin{cases} Ins(p1, s1) & p1 \leq p2 \\ no - op & p2 < p1 < p2 + l1 \\ Ins(p1 - l1, s1) & p1 \geq p2 + l1 \end{cases} \quad (2-3)$$

2.3 OT 函数的验证

CP1 与 CP2 性质要求 OT 函数的设计者考虑所有可能的操作组合情况，并且，对一种特定的操作组合，还要考虑所有可能的访问位置的组合情况。OT 函数的设计易于出错。因此，有一系列工作致力于如何验证 OT 函数的正确性。Imine 等人^[2] 基于代数规约框架开发了 OT 函数自动验证工具 VOTE。该工具可以自动验证给定的 OT 函数的正确性或者给出反例。Imine 等人使用该工具检测了 GROVE^[2], Joint Emacs^[2], REDUCE^[2], CoWord, SAMS^[2] 和 So6^[2] 协同编辑器。结果发现，这六个协同编辑器所使用的 OT 函数都不满足 CP2 性质，而且 GROVE 和 Joint Emacs 的 OT 函数也不满足 CP1 性质。Sun 等人^[2] 开发了一套验证框架和一个验证工具，能够穷举所有可能的操作转换情况。基于此，作者不仅重现了之前已有工作中指出的 OT 函数设计的错误之处，而且揭示了所有可能的 OT 函数的设计错误。在其后续工作中，Sun 等人^[2] 推广了该方法，用于验证支持字符串插入与字符串删除的列表 OT 函数的正确性。在这两份工作中，作者不仅考虑了 CP1 和 CP2 性质，还考虑了 OT 函数可能需要满足的其它性质。Randolph 等人^[2] 从理论上考察满足 CP1 和 CP2 性质的 OT 函

数的存在性问题。作者将该问题转化为 (Controller Synthesis) 合成问题，然后尝试借助形式化验证工具 Uppaal-Tiga^[2] 自动合成 OT 函数。结果表明，在不使用位置 (position) 之外的信息的情况下，不存在满足 CP2 性质的 OT 函数。Liu 等人^[2] 利用模型检验技术验证了某些 OT 函数满足 CP1 性质，也重现了违反 CP2 性质的 OT 函数的设计错误。此外，作者还第一次验证了一类基于操作上下文 (Operation Context) 概念^[2] 的 OT 控制算法的正确性。

第三章 Redis List OT 函数设计

3.1 Redis List API 分类

3.1.1 Redis List API 简介

本文针对 Redis 系统的 List 命令进行的，首先对 Redis 中的列表 (List) 及其相关命令进行一个简要的介绍。Redis 中的列表是字符串列表，按照插入的顺序进行排序，一个列表可以包含的最多元素为 $2^{32} - 1$ (4294967295) 个。

列表相关的基本命令共有 17 个，其中 3 个为阻塞性操作（即没有元素时会阻塞列表直至等待到有元素可以执行该命令），即其余 14 种为非阻塞性操作，即下面列出的 14 个命令。

- LINDEX key index: 通过索引 index 来获取列表 key 中的元素
- LINSERT key BEFORE(AFTER) pivot value: 在列表 key 某个元素 pivot 的前面 (BEFORE) 或者后面 (AFTER) 插入元素 value，若元素不在列表中或者列表不存在则不执行任何操作
- LLEN key: 获取列表 key 的长度, 若 key 不存在则返回 0，如果 key 不为列表类型则出错
- LPOP key: 用于移除并返回列表的第一个元素
- LPUSH key value1 [value2]: 将一个值 (value1) 或者多个值 (value1,value2..) 插入到列表 key 头部，若 key 不存在则创建一个新列表并执行操作
- LPUSHX key value: 将一个值 (value) 插入到列表 key 头部，若列表 key 不存在则操作无效
- LRANGE key start stop: 返回列表 key 中指定区间 [start,stop] 内的元素
- LREM key count value: 根据 count 的值，移除列表 key 中与 value 值相等的元素（即删除 count 个）若 $count > 0$ 从表头开始搜索，若 $count < 0$ 从表尾开始搜索，若 $count = 0$ 则移除 key 中所有与 value 相等的元素
- LSET key index value: 通过索引 index 来设置列表 key 中元素的值为 value

- LTRIM key start stop: 对列表 key 进行修剪, 只保留 [start,stop] 之间的元素, 不在该区间的元素全部删除
- RPOP key: 移除并获取列表 key 中的最后一个元素
- RPOPLPUSH source destination: 移除并获取列表 source 中的最后一个元素, 并添加到另一个列表 destination 中, 返回该列表
- RPUSH key value1 [value2]: 将一个值 (value1) 或者多个值 (value1,value2..) 插入到列表 key 尾部, 若 key 不存在则创建一个新列表并执行操作
- RPUSHX key value: 将一个值 (value) 插入到列表 key 尾部, 若列表 key 不存在则操作无效

而其中 LRange, LLen 和 LIndex 这三个命令与 list 的内容修改无关, 因此在本文中不考虑这两个命令的相关 OT 操作。本文中只考虑剩余这 11 个命令的 OT 函数的设计和验证。

3.1.2 Redis List API 分类

经过分析, 这 11 个命令可以根据操作类型和作用范围分为以下这三类。

- 单个元素的删除、修改、插入
- 单个区间的删除、插入
- 多个区间的删除

也就是在前一章中提到的三类操作, 可以用这三类操作来表示这所有的 11 个命令, 如下表所示:

操作分类	LIST 命令	具体操作
第一类操作	<i>LPUSHX</i>	<i>Ins(0, ele)</i>
	<i>RPUSHX</i>	<i>Ins(len, ele)</i>
	<i>LINSERT</i>	<i>Ins(pos, ele)</i>
	<i>LPOP</i>	<i>Del(0)</i>
	<i>RPOP</i>	<i>Del(len - 1)</i>
	<i>RPOPLPUSH</i>	<i>Del(len - 1)</i>
	<i>LS ET</i>	<i>Set(pos, ele)</i>
第二类操作	<i>LPUSH</i>	<i>Ins(0, str)</i>
	<i>RPUSH</i>	<i>Ins(len, str)</i>
第三类操作	<i>LTRIM</i>	<i>Del(0, pos1 - 1; pos2 + 1, len - pos2 - 1)</i>
	<i>LREM</i>	<i>Del(pos1, len1; pos2, len2; ...; posk, lenk)</i>

注：*len* 表示列表的长度这样，由于 Redis 中 list 相关命令都可以用这三类操作来表示，所以我们只要设计完成这三类操作之间的所有 OT 函数，即可完成 Redis 中 list 相关命令的 OT 函数设计。由于第一类函数和第二类函数已经设计完成，接下来只要完成剩余的 OT 函数设计即可。

3.2 第三类 OT 函数设计

我们定义第三类函数为第三类的 Del 命令与第二类命令中的 Ins 操作之间的 OT 函数，以及第三类 Del 命令自身的 OT 函数，共 $2+1=3$ 个。

首先考虑第三类的 Del 命令与第二类命令中的 Ins 操作之间的 OT 函数，显然，我们需要按 Ins 操作的插入位置和 Del 操作的删除区间之间的关系进行分类，进行函数的设计。

Ins 操作对于 Del 操作的转换，如果是插入位置位于删除区间中，则 Ins 操作转换为 NOP，否则插入的位置要减去删除操作在该位置之前删除区间的总长度。具体的数学公式如下：

$$OT(Ins(p_{k+1}, s_{k+1}), Del(p_1, l_1; p_2, l_2; \dots; p_k, l_k))$$

$$= \begin{cases} Ins(p_{k+1}, s_{k+1}) & p_{k+1} \leq p_1 \\ no-op & p_i < p_{k+1} < p_i + l_i \\ Ins(p_{k+1} - l_1 - l_2 - \dots - l_i, s_{k+1}) & p_i + l_i \leq p_{k+1} \leq p_{i+1} \\ Ins(p_{k+1} - l_1 - l_2 - \dots - l_k, s_{k+1}) & p_{k+1} \geq p_k + l_k \end{cases} \quad (3-1)$$

Del 操作对于 Del 操作的转换，如果是插入位置位于删除区间中，则该区间删除长度增加 $|s|$ ，之前的区间不变，之后的区间都向后移动 $|s|$ 单位长度

如果插入位置不在删除区间中，那么在插入位置之前的区间不变，之后的区间都向后移动 $|s|$ 单位长度具体的数学公式如下：

$$OT(Del(p_1, l_1; p_2, l_2; \dots; p_k, l_k), Ins(p_{k+1}, s_{k+1}))$$

$$= \begin{cases} Del(p_1 + |s_{k+1}|, l_1; p_2 + |s_{k+1}|, l_2; \dots; p_i, l_i; p_{i+1} + |s_{k+1}|, l_{i+1}; \dots; p_k + |s_{k+1}|, l_k) & p_{k+1} \leq p_1 \\ Del(p_1, l_1; p_2, l_2; \dots; p_{i-1}, l_{i-1}; p_i, l_i + |s_{k+1}|; p_{i+1} + |s_{k+1}|, l_{i+1}; \dots; p_k + |s_{k+1}|, l_k) & p_i < p_{k+1} < p_i + l_i \\ Del(p_1, l_1; p_2, l_2; \dots; p_i, l_i; p_{i+1} + |s_{k+1}|, l_{i+1}; \dots; p_k + |s_{k+1}|, l_k) & p_i + l_i \leq p_{k+1} \leq p_{i+1} \\ Del(p_1, l_1; p_2, l_2; \dots; p_k, l_k) & p_k + l_k \leq p_{k+1} \end{cases} \quad (3-2)$$

Del 操作自身的转换时最为复杂的，一开始想要将要转换的 Del 操作中所有的区间一起转换，发现这样不仅做起来难度很大，而且写公式和用代码表达都很容易出错，如果是将每个区间都用某个公式来转换，然后将转换后的新区间合并为新的删除操作，就可以方便的实现第三类 Del 操作自身的转换了。显然，每个区间经转换后的新区间长度肯定是减少的（删除未被删除的内容），若该区间中所有内容都已经被删除则执行 NOP。需要考虑的问题是如果将每个区间单独进行转换，则区间之间会不会相互影响，由于原有区间都是互不重复的，即删除的内容互不相干，因此每个区间可以单独进行转换，转换完成后简单地将所有区间合并成新的 Del 操作即可。

转变思路后，第三类 Del 自身的转换就可以用第二类 Del 操作对于第三类

Del 操作的转换来代替了，减少了公式的复杂度，同时也增加了可读性。

与前面的设计类似，由删除区间与删除区间之间的位置关系进行函数的设计。我们不妨假设目前待转换的删除区间端点为 pos ，区间长度为 len ，转换完成后的新区间端点为 $newpos$ ，新区间长度为 $newlen$ ，对于操作 $op2$ 进行转换。定性地进行分析，新区间即为待转换区间中没被 $op2$ 删除的部分；定量地进行分析， $newpos$ 即为 $op2$ 在 pos 之前的删除总长度， $newlen$ 为待转换区间中没有被 $op2$ 删除的长度，这样新区间 $[newpos, newpos+newlen]$ 即为未被 $op2$ 删除的部分了。

下面两种表格中列出了经过转换后的新区间开始端点和新区间的长度，本文第 4 章中部分代码的编写便是参照这两张表格进行的。

$OT(Del(p_{k+1}, l_{k+1}), Del(p_1, l_1; p_2, l_2; \dots; p_k, l_k))$		
p_{k+1} 位置	$p_{k+1} + l_{k+1}$ 位置	转换后的 p_{k+1}
$p_{k+1} < p_1$	任意位置	p_{k+1}
$p_i \leq p_{k+1} < p_i + l_i$	任意位置	$p_i - l_1 - l_2 - \dots - l_{i-1}$
$p_i + l_i \leq p_{k+1} < p_{i+1}$	任意位置	$p_{k+1} - l_1 - l_2 - \dots - l_{i-1}$
$p_{k+1} \geq p_k + l_k$	任意位置	$p_{k+1} - l_1 - l_2 \dots - l_k$

$OT(Del(p_{k+1}, l_{k+1}), Del(p_1, l_1; p_2, l_2; \dots; p_k, l_k))$		
p_{k+1} 位置	$p_{k+1} + l_{k+1}$ 位置	转换后的 l_{k+1}
$p_{k+1} < p_1$	$p_{k+1} + l_{k+1} \leq p_1$	l_{k+1}
	$p_j < p_{k+1} + l_{k+1} \leq p_j + l_j$	$p_j - l_1 - l_2 - \dots - l_{j-1} - p_{k+1}$
	$p_j + l_j < p_{k+1} + l_{k+1} \leq p_{j+1}$	$l_{k+1} - l_1 - l_2 - \dots - l_j$
	$p_{k+1} + l_{k+1} > p_k + l_k$	$l_{k+1} - l_1 - l_2 - \dots - l_k$
$p_i \leq p_{k+1} < p_i + l_i$	$p_j < p_{k+1} + l_{k+1} \leq p_j + l_j$	$p_j - p_i - l_i - l_{i+1} \dots - l_{j-1}$
	$p_j + l_j < p_{k+1} + l_{k+1} \leq p_{j+1}$	$p_{k+1} + l_{k+1} - p_i - l_i - l_{i+1} - \dots - l_j$
	$p_{k+1} + l_{k+1} > p_k + l_k$	$p_{k+1} + l_{k+1} - p_i - l_i - l_{i+1} - \dots - l_k$
$p_i + l_i \leq p_{k+1} < p_{i+1}$	$p_j < p_{k+1} + l_{k+1} \leq p_j + l_j$	$p_j - p_{k+1} - l_{i+1} - l_{i+2} - \dots - l_{j-1}$
	$p_j + l_j < p_{k+1} + l_{k+1} \leq p_{j+1}$	$l_{k+1} - l_{i+1} - l_{i+2} - \dots - l_j$
	$p_{k+1} + l_{k+1} > p_k + l_k$	$l_{k+1} - l_{i+1} - l_{i+2} - \dots - l_k$
$p_{k+1} \geq p_k + l_k$	任意位置	l_{k+1}

从表格中不难发现，区间进行转换后新区间的开始端点与原区间的长度有关，而新区间的长度则与原区间的开始端点和长度都有关系。具体的数学公式如下：

$$\begin{aligned}
 & OT(Del(p_{k+1}, l_{k+1}), Del(p_1, l_1; p_2, l_2; \dots; p_k, l_k)) \\
 = & \left\{ \begin{array}{ll}
 Del(p_{k+1}, l_{k+1}) & p_{k+1} < p_1 \\
 & p_{k+1} + l_{k+1} \leq p_1 \\
 Del(p_{k+1}, p_j - l_1 - l_2 - \dots - l_{j-1} - p_{k+1}) & p_{k+1} < p_1 \\
 & p_j < p_{k+1} + l_{k+1} \leq p_j + l_j \\
 Del(p_{k+1}, l_{k+1} - l_1 - l_2 - \dots - l_j) & p_{k+1} < p_1 \\
 & p_j + l_j < p_{k+1} + l_{k+1} \leq p_{j+1} \\
 Del(p_{k+1}, l_{k+1} - l_1 - l_2 - \dots - l_k) & p_{k+1} < p_1 \\
 & p_{k+1} + l_{k+1} > P_k + l_k \\
 Del(p_i - l_1 - l_2 - \dots - l_{i-1}, p_j - p_i - l_i - l_{i+1} \dots - l_{j-1}) & p_i \leq p_{k+1} < p_i + l_i \\
 & p_j < p_{k+1} + l_{k+1} \leq p_j + l_j \\
 Del(p_i - l_1 - l_2 - \dots - l_{i-1}, p_{k+1} + l_{k+1} - p_i - l_i - l_{i+1} - \dots - l_j) & p_i \leq p_{k+1} < p_i + l_i \\
 & p_j + l_j < p_{k+1} + l_{k+1} \leq p_{j+1} \\
 Del(p_i - l_1 - l_2 - \dots - l_{i-1}, p_{k+1} + l_{k+1} - p_i - l_i - l_{i+1} - \dots - l_k) & p_i \leq p_{k+1} < p_i + l_i \\
 & p_{k+1} + l_{k+1} > P_k + l_k \\
 Del(p_{k+1} - l_1 - l_2 - \dots - l_{i-1}, p_j - p_{k+1} - l_{i+1} - l_{i+2} - \dots - l_{j-1}) & p_i + l_i \leq p_{k+1} < p_{i+1} \\
 & p_j < p_{k+1} + l_{k+1} \leq p_j + l_j \\
 Del(p_{k+1} - l_1 - l_2 - \dots - l_{i-1}, l_{k+1} - l_{i+1} - l_{i+2} - \dots - l_j) & p_i + l_i \leq p_{k+1} < p_{i+1} \\
 & p_j + l_j < p_{k+1} + l_{k+1} \leq p_{j+1} \\
 Del(p_{k+1} - l_1 - l_2 - \dots - l_{i-1}, l_{k+1} - l_{i+1} - l_{i+2} - \dots - l_k) & p_i + l_i \leq p_{k+1} < p_{i+1} \\
 & p_{k+1} + l_{k+1} > P_k + l_k \\
 Del(p_{k+1} - l_1 - l_2 \dots - l_k, l_{k+1}) & p_{k+1} \geq p_k + l_k \\
 & (i \geq j)
 \end{array} \right. \quad (3-3)
 \end{aligned}$$

3.3 其它 OT 函数设计

显然，第三类的 del 命令可以涵盖第一类和第二类的 del 命令，第二类的 ins 命令可以涵盖第一类的 ins 命令，因此我们最后只要考虑 set(pos,ele) 命令和第二类的 ins 操作、第三类的 del 操作之间的 OT 函数关系，即可覆盖完全所有的 OT 函数设计。共有 $2+2=4$ 个函数。

Ins 操作对于 Set 操作的转换就是其本身。

$$OT(Ins(i, s), Set(j, x)) = Ins(i, s) \quad (3-4)$$

Set 操作对于 Ins 操作的转换, 如果插入位置在 Set 位置之前则 Set 位置要增加 $|s|$ 单位长度, 否则 Set 操作不变。

$$OT(Set(i, x), Ins(j, s)) = \begin{cases} Set(i, x) & i < j \\ Set(i + |s|, x) & i \geq j \end{cases} \quad (3-5)$$

Del 操作对于 Set 操作的转换就是其本身。

$$OT(Del(p_1, l_1; p_2, l_2; \dots; p_k, l_k), Set(p_{k+1}, x)) = Del(p_1, l_1; p_2, l_2; \dots; p_k, l_k) \quad (3-6)$$

Set 操作对于 Del 操作的转换, 如果是 Set 位置位于删除区间中, 则 Ins 操作转换为 NOP, 否则插入的位置要减去删除操作在该位置之前删除区间的总长度。 $OT(Set(p_{k+1}, x), Del(p_1, l_1; p_2, l_2; \dots; p_k, l_k))$

$$= \begin{cases} Ins(p_{k+1}, s_{k+1}) & p_{k+1} < p_1 \\ no-op & p_i \leq p_{k+1} < p_i + l_i \\ Set(p_{k+1} - l_1 - l_2 - \dots - l_i, x) & p_i + l_i \leq p_{k+1} < p_{i+1} \\ Set(p_{k+1} - l_1 - l_2 - \dots - l_k, x) & p_{k+1} \geq p_k + l_k \end{cases} \quad (3-7)$$

第四章 基于 TLA+ 的 OT 函数验证

4.1 TLA+ 简介

TLA+ 是一种形式化的规约语言。它是一种设计系统和算法的工具，并且用来验证这些系统有没有关键错误。

正确性，是一个系统最为重要的性质，同时，正确性是比较难以证明的，特别是并发系统的正确性，因为存在着数目众多的状态变化，而 TLA+ 可以将系统的行为或者状态抽象为时态逻辑，即系统的行为或者状态会随着时间反生变化，然后通过一些数学分析的方法，来判断系统是否正确。

TLA+ 并不同于一般传统意义上的编程语言，更类似于一种数学语言，因为其语法大部分来自于实际的数理逻辑。

TLA+ 提供了工具集 TLAToolbox/TLC，同时还可以使用 TLA+ 的语法糖 PLUSCAL 来完成代码的编写，由于本文中并未涉及，在此不做展开。TLA+ model checker 与经典的模型检验工具类似，通过遍历系统模型的所有可能的行为，验证其正确性。

```
----- MODULE M -----  
EXTENDS M1,...Mn \* modules  
  
CONSTANTS C1,...,Cn \* constants  
  
VARIABLES x1,...,xn \*variables  
  
\*Definitions()  
OP(x1,...xn) == exp  
  
\*state  
Init,Next  
=====
```

图 4-1: TLA+ 编码模板

4.1.1 TLA+ Modules

TLA+ 提供了多种数据结构的实现，包括在本次实验中使用的 `sequence` 和 `record` 等。

`record` 数据结构类似于 C 语言中的 `struct` 结构，即一个 `record` 包含若干个 `field`，每个 `field` 可以在给定的集合中选取元素填充 `field`。`sequence` 表示一个序列，如果引入 `Sequences` 模块，也可以将数组当作序列来进行处理，并且使用模块里的相应函数在本次实验中，我们引入了 TLA+ 中的 `Sequences` 模块，将 `List` 表示为一个 `Sequence`，并且使用了如下的 API：

operator	operation	example
Head	First element	$\text{Head}(\langle\langle 1, 2 \rangle\rangle) = 1$
Tail	Sequence aside from head	$\text{Tail}(\langle\langle 1, 2 \rangle\rangle) = \langle\langle 2 \rangle\rangle$
Append	Add element to end of sequence	$\text{Append}(\langle\langle 1 \rangle\rangle, 2) = \langle\langle 1, 2 \rangle\rangle$
Len	Length of sequence	$\text{Len}(\langle\langle 1, 2 \rangle\rangle) = 2$
$\backslash o$	Connect two sequences	$\langle\langle 1 \rangle\rangle \backslash o \langle\langle 2 \rangle\rangle = \langle\langle 1, 2 \rangle\rangle$

以上的操作都不会使原有的操作序列发生变化。

4.2 使用 TLA+ 描述 OT 函数

解释同时描述并验证 OT 函数。

4.2.1 List 相关操作表示

4.2.1.1 第一、二类操作的表示

采用 `record` 的数据结构来表示具体的操作。第一类操作的 `record` 分为两种，`set` 和 `ins` 有三个域：`pos` 位置，`ch` 操作字符，`pr` 优先级。而 `del` 操作少了 `ch` 字符域，这样定义可以减少无效重复的操作数量，使验证代码的效率得到

提高。

$$OP_1_set \triangleq [type : \{ "set" \}, pos : POS, ch : CH, pr : PR]$$

$$OP_1_ins \triangleq [type : \{ "ins" \}, pos : POS, ch : CH, pr : PR]$$

$$OP_1_del \triangleq [type : \{ "del" \}, pos : POS, pr : PR]$$

$$OP_1 \triangleq OP_1_set \cup OP_1_ins \cup OP_1_del$$

第二类操作的 record 也分为两种,ins 操作有三个域:pos 插入位置,pr 优先级, str 要插入的字符串。del 操作也有三个域:pos 插入位置,pr 优先级, len 删除区间的长度

$$OP_2_ins \triangleq [type : \{ "ins_r" \}, pos : POS, pr : PR, str : STR]$$

$$OP_2_del \triangleq [type : \{ "del_r" \}, pos : POS, pr : PR, len : LEN]$$

$$OP_2 \triangleq OP_2_ins \cup OP_2_del$$

4.2.1.2 第三类操作的表示

第三类操作同样也是使用 record 来表示, 包含两个 filed, 即 type 类型和 ints 删除区间的集合。

$$OP_3 \triangleq [type : \{ "del_m" \}, ints : NoncoSeq]$$

在表示第三类 OT 函数时, 我们首先遇到的问题便是如何表示第三类 List 命令, 由于第三类 List 命令为删除多个不相交的集合区间, 因此我们需要使用 TLA+ 遍历在 List 长度上所有可以表示出来的区间, 一个命令的删除区间即为若干个不相交的符合条件的区间。在 TLA+ 中支持两个集合的笛卡尔积操作, 因此所有符合条件的区间的集合 Intervals 可以如下定义:

$$Intervals \triangleq \{ \langle a, b \rangle \in POS \times POS : a + b \leq Maxnum(POS) + 1 \}$$

a 为区间的起始断点,b 为区间长度, 因为一个命令的删除区间为若干个不相交的符合条件的区间, 定义区间后, 便可以生成操作的删除区间。将一个删除操作的删除区间称为一个区间组合, 那么有两种方式表示这种区间组合, 一种是区间二元组的集合, 另一种是二维数组, 由于在接下来的 OT 函数中, 需要较为

方便地对操作进行转换，所以在本次实验中采用第二种表示方法，先生成第一种表示方法，即二元数组的集合。

$NoncoIntervals \triangleq$

$$\{ints \in SUBSET Intervals : \forall i, j \in ints : i[2] + i[1] \leq j[1] \vee j[2] + j[1] \leq i[1] \vee i = j\} \setminus \{\}$$

这样， $NoncoIntervals$ 中每个元素都是一个区间组合，下面我们需要将这个集合变成一个二维数组，由于在 TLA+ 中没有执行某个指令固定次数的相应代码，所以在本次实验中需要多次使用递归来表示这种循环操作。一个符号在使用前必须先声明或者定义，所以使用递归要在函数或定义之前加上 `recursive` 关键字。那么比如在这里我们需要将区间组合的集合转换为区间组合的数组，就需要使用两个递归的定义。首先定义了将集合转化为序列的 $SetTOSeq()$:

RECURSIVE SetTOSeq($_$)

$SetTOSeq(T) \triangleq IF T = \{\} THEN <<>>$

$ELSE LET t \triangleq CHOOSE x \in T : TRUE$

$IN <t> \ \backslash o \ SetTOSeq(T \setminus t)$

接下来便可以定义 $Seqset(T)$, 将二元组的集合的集合 T 转化为二维数组的集合:

RECURSIVE Seqset($_$)

$Seqset(T) \triangleq$

$IF T = \{\} THEN \{\}$

$ELSE LET t \triangleq CHOOSE x \in T : TRUE$

$IN Seqset(T \setminus t) \cup SetTOSeq(t)$

$NoncoSeq \triangleq Seqset(NoncoIntervals)$

得到的集合 $NoncoSeq$ 便为 $ints$ 域的取值范围，这样遍历时得到的具体操作的 $ints$ 域便为一个二维数组，该数组中横坐标表示第几个删除区间，纵坐标 1 的值表示该区间开始端带点，纵坐标为 2 的值表示区间长度。

4.2.2 命令执行的表示

命令的执行与 LIST 和具体操作相关，所以定义中使用的参数便是 LIST 和具体操作中的参数，使用 Sequences 模块中的 API 进行代码的编写。以第二类的 del 操作为例，调用了 $\backslash o$ 和 $SubSeq$ 定义：

$$del_ran_op(list, pos, len) \triangleq SubSeq(list, 1, pos - 1) \backslash o SubSeq(list, pos + len, Len(list))$$

唯一不同的是第三类操作的执行函数，同样是由于需要采用循环操作，所以还是要使用递归的定义完成命令的执行。

RECURSIVE $del_mulran_op(_, _, _)$

$$del_mulran_op(list, ints, num) \triangleq$$

$$IF\ num = 0\ THEN\ list$$

$$ELSE\ del_mulran_op(SubSeq(list, 1, ints[num][1] - 1) \backslash o SubSeq(list, ints[num][2] + ints[num][1], Len(list)), ints, num - 1)$$

num 是 ints 的区间个数，从后往前进行删除，这样不会影响下面要操作的区间，每次递归删除一个区间，并且值就是删除后的 list，当 num 为 0 时就表示删除完成了，直接返回当前的参数中的 list 即可。

4.2.3 OT 函数的描述

使用定义 Xform 表示 OT 函数，即 $Xform(lop, rop)$ 为 lop 对于 rop 转换后的新操作。以第一类函数为例：

$$Xform(lop, rop) \triangleq$$

$$CASE\ lop.type = "ins" \wedge rop.type = "ins" \rightarrow Xform_ins_ins(lop, rop)$$

$$\square\ lop.type = "ins" \wedge rop.type = "del" \rightarrow Xform_ins_del(lop, rop)$$

$$\square\ lop.type = "ins" \wedge rop.type = "set" \rightarrow Xform_ins_set(lop, rop)$$

4.2.3.1 第一类函数的表示

第一类函数的表示较为简单，不需要使用递归结构。以 *ins* 操作对于 *ins* 操作的转换来举例说明，使用定义 $Xform_ins_ins$ 表示 OT 函数，即 $Xform_ins_ins(lins, rins)$ 为 *ins* 操作对于 *ins* 操作转换后的新操作。

$$\begin{aligned}
 Xform_ins_ins(lins, rins) &\triangleq \\
 &IF \quad lins.pos < rins.pos \\
 &THEN \quad lins \\
 &ELSE \quad IF \quad lins.pos > rins.pos \\
 &\quad THEN \quad [lins \quad EXCEPT!.pos = @ + 1] \\
 &\quad ELSE \quad IF \quad lins.pr < rins.pr \\
 &\quad \quad THEN \quad [lins \quad EXCEPT!.pos = @ + 1] \\
 &\quad \quad ELSE \quad lins
 \end{aligned}$$

其中 EXCEPT ! 的意思为除了特定的域发生变化，其他域保持不变。这样参照着之前第一类函数的数学公式，便可以类似地将其他第一类函数用同样的方式表示出来。

4.2.3.2 第二类函数的表示

第二类函数的表示也比较简单，同样不需要使用递归结构。以 *ins* 操作对于 *del* 操作的转换来举例说明，使用定义 $Xform_ins_del_r$ 表示 OT 函数，即 $Xform_ins_del_r(ins, del)$ 为 *ins* 操作对于 *del* 操作转换后的新操作。

$$\begin{aligned}
 Xform_ins_del_r(ins, del) &\triangleq \\
 &CASEins.pos \leq del.pos \rightarrow ins \\
 &\quad [ins.pos > del.pos \wedge ins.pos < del.pos + del.len \rightarrow NOP \\
 &\quad [ins.pos \geq del.pos + del.len \rightarrow [ins \quad EXCEPT!.pos = @ - del.len]
 \end{aligned}$$

EXCEPT ! 的意思为除了特定的域发生变化，其他域保持不变。这样参照着之前第二类函数的数学公式，便可以类似地将其他第二类函数用同样的方式表示出来。

4.2.3.3 第三类函数的表示

第三类函数为第三类的 Del 命令与第二类命令中的 Ins 操作之间的 OT 函数，以及第三类 Del 命令自身的 OT 函数，共 $2+1=3$ 个。这类函数的设计代码量占到了整个设计的一半，大量使用了递归定义，这里拿最为复杂的 Del 命令自身的 OT 函数来举例。

RECURSIVE $Xform_del_del_m(_, _, _)$

$Xform_del_del_m(l_del, r_del, i) \triangleq$

IF $i > Len(l_del.ints)$ *THEN* l_del

ELSE $Xform_del_del_m([l_del EXCEPT !.ints[i] = transdel4(@, r_del.ints)], r_del, i + 1)$

使用定义 $Xform_del_del_m$ 表示 OT 函数, 这里设置 i 为递归变量, i 为 l_del 中当前正在进行转换的区间, 转换函数为 $transdel4$, 按照公式 (3-10) 进行编写, 即进行一个删除区间相对于 r_del 的转换, 这样当 i 的值大于 l_del 中删除区间的数量时, 所有删除区间都完成了转换, 将转换后的新区间合并起来, 便是 l_del 转换后的新操作。

$transdel4(int, ints) \triangleq$

$\langle\langle newpos(int[1], ints, 1), newlen(int[1], int[2], ints, 1, 0) \rangle\rangle$

$newpos$ 和 $newlen$ 便是这个删除区间对于 r_del 的转换后的新区间端点和区间长度。 $newpos$ 和 $newlen$ 两个定义同样使用递归结构来表示。 $newpos$:

RECURSIVE $newpos(_, _, _)$

$snewpos(pos, ints, i) \triangleq$

IF $pos < ints[1][1]$ *THEN* pos

ELSE IF $i = Len(ints) \wedge pos \geq ints[i][1] + ints[i][2]$ *THEN* $pos - Dlen(ints, i, 0)$

ELSE IF $pos \geq ints[i][1] \wedge pos < ints[i][1] + ints[i][2]$ *THEN* $ints[i][1] - Dlen(ints, i - 1, 0)$

ELSE IF $ints[i][1] + ints[i][2] \leq pos \wedge pos < ints[i + 1][1]$ *THEN* $pos - Dlen(ints, i, 0)$

ELSE $snewpos(pos, ints, i + 1)$

递归变量为 i ，代表当前在对 $rdel$ 中第几个区间进行 $newpos$ 的计算，每递归一次， i 的值加一，如果满足条件要求则返回相应 $newpos$ 的值。其中 $Dlen$ 的作用为统计 $rdel$ 中前 i 个删除区间的长度之和。 $newlen$:

RECURSIVE $newlen(_, _, _, _, _)$

$newlen(pos, len, ints, i, sum) \triangleq$

IF $i > Len(ints)$ *THEN* $len - sum$

ELSE IF $pos + len < ints[i][1]$ *THEN* $newlen(pos, len, ints, i + 1, sum)$

ELSE IF $pos < ints[i][1] \wedge ints[i][1] < pos + len \wedge pos + len \leq ints[i][1] + ints[i][2]$

THEN $newlen(pos, len, ints, i + 1, sum + pos + len - ints[i][1])$

ELSE IF $pos < ints[i][1] \wedge pos + len > ints[i][1] + ints[i][2]$

THEN $newlen(pos, len, ints, i + 1, sum + ints[i][2])$

ELSE IF $ints[i][1] \leq pos \wedge pos < ints[i][1] + ints[i][2] \wedge pos + len \leq ints[i][1] + ints[i][2]$

THEN 0

ELSE IF $ints[i][1] \leq pos \wedge pos < ints[i][1] + ints[i][2] \wedge pos + len > ints[i][1] + ints[i][2]$

THEN $newlen(pos, len, ints, i + 1, sum + ints[i][1] + ints[i][2] - pos)$

ELSE $newlen(pos, len, ints, i + 1, sum)$

递归变量为 i 和 sum , i 表示当前在对 $rdel$ 中第几个区间进行 $newlen$ 的计算, sum 表示在当前计算完成的所有区间中 len 转换为 $newlen$ 需要减去的长度。这样便根据公式 (3-10) 完成了 Del 命令自身的 OT 函数，第三类的 Del 命令与第二类命令中的 Ins 操作之间的 OT 函数可以用类似地方式表示出来，同时，剩余的 OT 函数也可以用递归或非递归的形式根据公式进行表示，在此不再赘述。

4.3 正确性验证

上节中已经完成了 OT 函数的 TLA+ 描述，剩下的就是决定验证的目标和方式。验证的目标: CP1 如何用 TLA+ 表示。

我们验证 CP1 性质的正确性。即同一个 List 经过 $OT(OP2, OP1), OP1$ 或者 $OT(OP1, OP2), OP1$ 这两种操作序列后，最终的结果是一致的。结合以上给出的

定义，用 TLA+ 来描述所设计 OT 函数的 CP1 正确性：

$$\text{apply}(\text{apply}(\text{list}, \text{op1}), \text{Xform}(\text{op2}, \text{op1})) = \text{apply}(\text{apply}(\text{list}, \text{op2}), \text{Xform}(\text{op1}, \text{op2}))$$

只要对于任意的两个操作，这个等式都成立的话，那么 CP1 正确性即可得到验证。

因为在本次实验中不涉及系统状态的变化，只需证明等式对于所有操作都成立即可，因此没有 behavior spec，即没有初始状态 Init 和状态关系 Next。不过，在 TLC 中提供了 Evaluate Constant Expression 的功能，即可以计算常量表达式的值。以第一类函数的 CP1 验证为例，我们给出如下正确性定义：

$$\text{correctness}_1(\text{list}) \triangleq$$

$$\forall \text{op1}, \text{op2} \in \text{OP}_1 :$$

$$\vee \text{op1.pr} = \text{op2.pr}$$

$$\vee \text{apply}(\text{apply}(\text{list}, \text{op1}), \text{Xform}(\text{op2}, \text{op1})) = \text{apply}(\text{apply}(\text{list}, \text{op2}), \text{Xform}(\text{op1}, \text{op2}))$$

这样的话， $\text{op1}, \text{op2}$ 就实现了对 OP_1 中所有操作的遍历，并且对于满足要求的任意 op1 和 op2 ，只要优先级满足要求（两个操作的优先级不相等），则其一定满足 CP1 等式。在 Evaluate Constant Expression 内填写这个表达式，我们便可以 check model 并根据 value 的值来检验函数的正确性了。

4.3.1 TLA+ Model Checker 设置及实验环境

在本次实验中，我们定义 Model Checker 中常量的参数值如下：

```
STR <- {"like", "enjoy", "fond", "love", "fantasy"}
```

```
CH <- {"a", "b", "c", "d", "e"}
```

```
PR <- {1, 2}
```

STR 为第二类插入操作中 str 域的可选择范围，CH 为第二类插入、设置操作中 ch 域的可选择范围，PR 为操作的优先级，在此处我们设定两个操作的优先级不相同，否则为违规操作变换。通过调整 LIST 常量的长度，实验在不同长度的 LIST 下完成各类函数的验证所需要的时间。

实验环境：

Number of worker threads: 12

Fraction of physical memory allocated to TLC: 16173 mb

4.3.2 实验结果

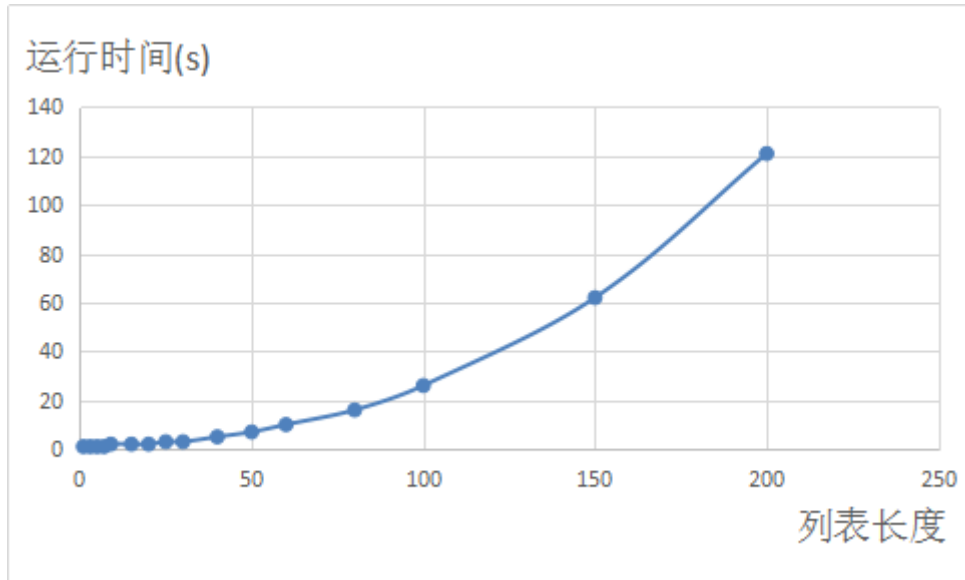


图 4-2: 验证第一类函数正确性的运行时间与列表长度（横坐标）的关系

结果分析：若列表长度为 n ，则第一类操作的个数： $5 * 2 * n(ins) + 5 * 2 * n(set) + 2 * n(del) = 22n$ 任意选取其中两个操作验证 CP1 正确性，所以验证算法的时间复杂度为 $O(n^2)$ 从实验结果看，与复杂度的分析相一致。

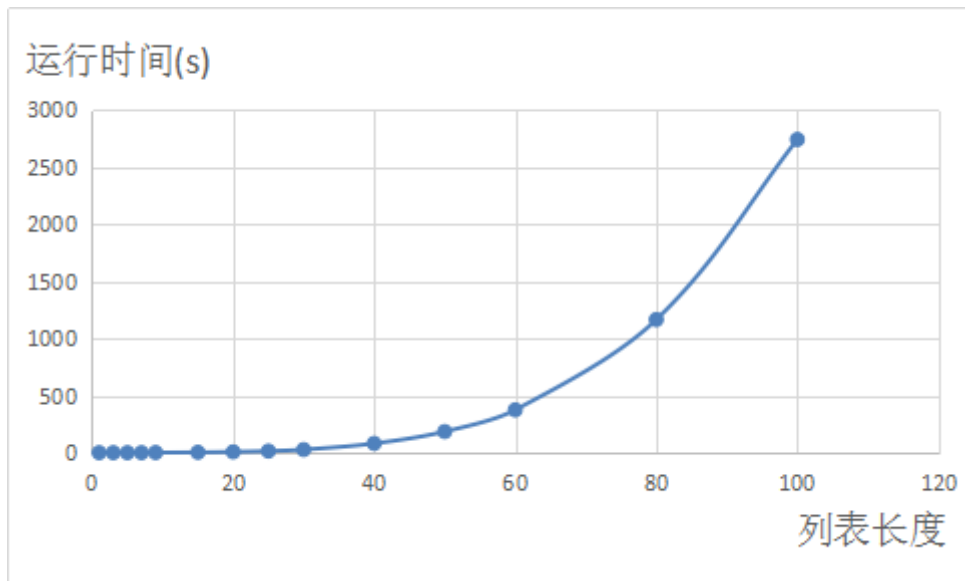


图 4-3: 验证第二类函数正确性的运行时间与列表长度（横坐标）的关系

结果分析：若列表长度为 n ，则第二类操作的个数： $5 * 2 * n(ins) + n * 2 * n(del) = 2n^2 + 10n$ 任意选取其中两个操作验证 CP1 正确性，所以验证算法的时间复杂度为 $O(n^4)$ 从实验结果看，与复杂度的分析相一致。

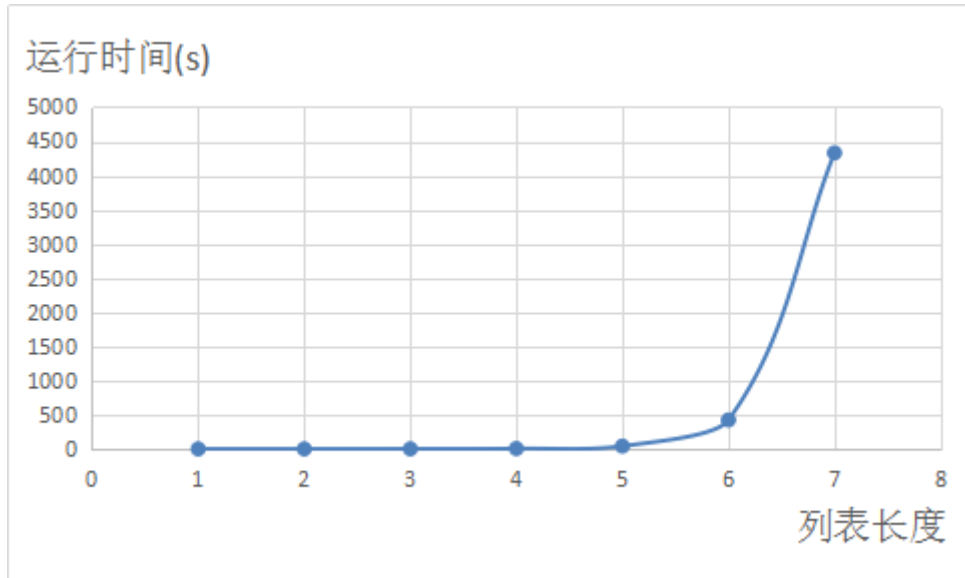


图 4-4: 验证第三类函数正确性的运行时间与列表长度（横坐标）的关系

结果分析：目前第三类函数的验证结果只能到 LIST 长度为 7。因为第三类操作的个数为指数级别 $O(3^n)$ ，并且在生成操作和操作的转换过程中多次使用了递归，更大大增加了验证算法的复杂程度。

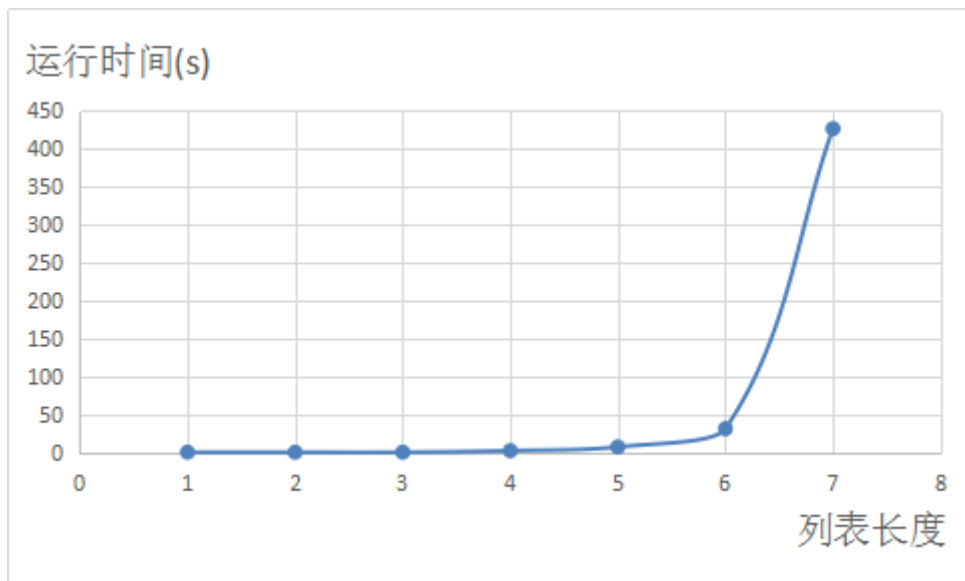


图 4-5: 验证其余函数正确性的运行时间与列表长度（横坐标）的关系

结果分析：目前其余函数的验证结果只能到 LIST 长度为 7。与第三类函数的验证相同，在其余函数的验证算法中也需要生成 $O(3^n)$ 级别的第三类操作和使用递归代码实现操作的转换，因此复杂度与第三类函数相仿。

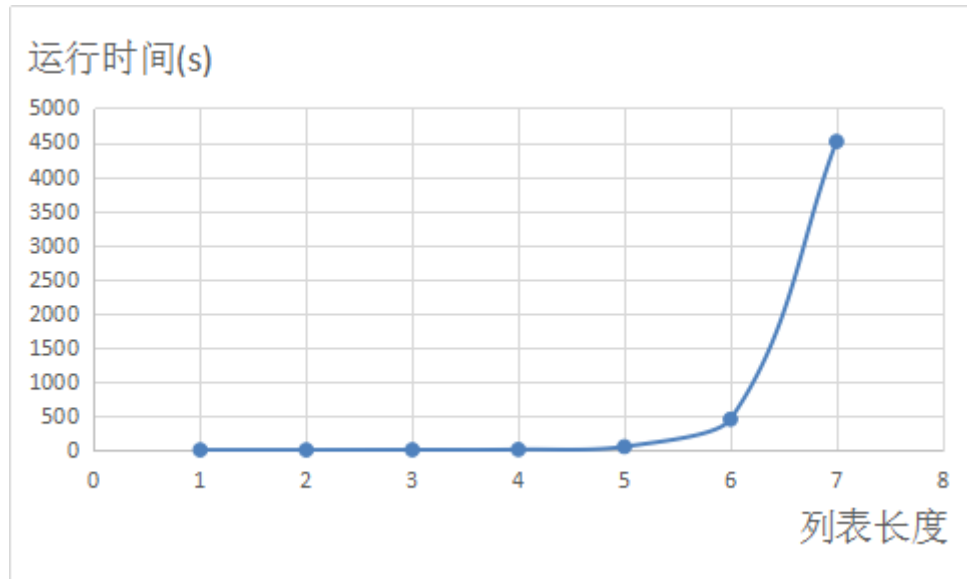


图 4-6: 验证所有 OT 函数正确性与列表长度（横坐标）的关系

结果分析：时间复杂度与第三类函数的复杂度相同，目前可以验证所设计的 OT 函数在 LIST 长度为 7 之内时的正确性。

Redis List 命令的 OT 函数设计经过验证是成功的，其满足 CP1 正确性。

第五章 结论与未来工作

5.1 工作总结

本文首先基于协同编辑的应用背景，以及 Replicated List 规约，引出了 OT 的定义和 OT 系统的结构，以及 Redis 系统中 list 相关命令的应用需求。接着，本文回顾了相关工作，分析了他们的不足。针对这些不足，本文研究了 Redis 系统中 List 相关命令，实现了其 OT 函数的设计与验证，主要贡献包括：

首先将 Redis List 的相关命令进行基于坐标和操作方式的分类，分成三大类操作，能够更方便地实现 OT 函数的设计。

其次对于所有简化后的命令进行了 OT 函数的数学公式设计，补全了目前的研究中所欠缺的部分。

最后使用一种新的证明方式，即通过 TLA+ 实现了对于所设计 OT 函数的证明，并且对实验结果进行相应的分析。对于第一类函数和第二类函数的验证都很顺利，对于第三类函数，由于在代码中大量使用了递归的结构，导致运行的效率不是很高，验证成功的列表长度有限，不过已经可以大大增强所设计数学公式的可信度了。

5.2 研究展望

验证所设计的 OT 函数正确性只是一个起点，要真正能够在 Redis 系统中实现 List 的 OT 函数，还有许多工作亟待完成。

由于 Redis 系统的主从结构和我们的要求不符合，要想在 Redis 中实现 OT 函数，还需要修改 Redis 的系统通信框架，使其变为 P2P 的通信结构。

此外，由于在本次实验的 TLA+ 代码中大量使用了递归的函数或者定义，使得对于第三类函数以及全部函数正确性验证的过程时间较长，效率较低，代码还存在着改进的空间。之所以采用 TLA+ 来完成 OT 函数的验证，正是因为其可以遍历所有操作的特性，改进验证代码目前有两个想法：一是使用语法糖

Pluascal 从而避免对于递归定义的依赖，二是使用其他编程语言来编写验证代码。

致 谢

本文的工作是在魏恒峰老师的悉心指导下完成的。魏老师严谨的治学态度和认真的工作态度给了我极大的影响。在相关知识的学习、实验过程以及论文的写作过程中，魏老师也给了我莫大的帮助。在此衷心感谢魏老师这半年的关心和指导。

感谢黄宇老师和马晓星老师，他们在我的学习和实验过程中也给出了很多宝贵的指导和建议。

感谢我的同学们，很幸运能与易星辰、王芷芙、唐瑞泽等同学在 TLA+ 讨论班中共同学习和进步。感谢黄奕学长，张宇奇学长在我实验过程中给与的帮助和指导。感谢缪娅，谢谢你的理解和陪伴。

感谢我的父母，谢谢你们一直关心着我，一直给我无私的爱。你们的支持是我顺利完成学业的最大动力。

附录 A 附录

A.1 OT 函数的设计

A.1.1 第一类 OT 函数的设计

第一类函数为第一类命令之间的 OT 函数，即 Ins,Del,Set 三种操作之间的 OT 函数，共有 3*3=9 个。

$$Set \left\{ \begin{array}{l} OT(set(i, x), set(j, y)) = \begin{cases} no - op & pr1 > pr2 \quad i = j \\ set(i, x) & else \end{cases} \\ OT(Set(i, x), Ins(j, y)) = \begin{cases} Set(i, x) & i < j \\ Set(i + 1, x) & i \geq j \end{cases} \\ OT(Set(i, x), Del(j)) = \begin{cases} Set(i, x) & i < j \\ no - op & i = j \\ Set(i - 1, x) & i > j \end{cases} \end{array} \right. \quad (A-1)$$

$$Ins \left\{ \begin{array}{l} OT(Ins(i, x), set(j, y)) = Ins(i, x) \\ OT(ins(i, x), ins(j, y)) = \begin{cases} ins(i + 1, x) & i > j \\ ins(i, x) & i < j \\ ins(i + 1, x) & i = j \quad pr1 < pr2 \\ ins(i, x) & i = j \quad pr1 > pr2 \end{cases} \\ OT(Ins(i, x), Del(j)) = \begin{cases} Ins(i, x) & i \leq j \\ Ins(i - 1, x) & i > j \end{cases} \end{array} \right. \quad (A-2)$$

$$Del \left\{ \begin{array}{l} OT(Del(i), Set(j, x)) = Del(i) \\ OT(Del(i), Ins(j, x)) = \begin{cases} Del(i+1) & i \geq j \\ Del(i) & i < j \end{cases} \\ OT(Del(i), Del(j)) = \begin{cases} Del(i-1) & i > j \\ Del(i) & i < j \\ no-op & i = j \end{cases} \end{array} \right. \quad (A-3)$$

A.1.2 第二类 OT 函数设计

第二类函数为第二类命令之间的 OT 函数，即 Ins, Del 两种命令之间的 OT 函数，共 $2 \times 2 = 4$ 个。

$$OT(Ins(p1, s1), Ins(p1, s2)) = \begin{cases} Ins(p1, s1) & p1 < p2 \\ Ins(p1 + |s2|, s1) & p1 > p2 \\ Ins(p1 + |s2|, s1) & p1 = p2 \quad pr1 < pr2 \\ Ins(p1, s1) & p1 = p2 \quad pr1 > pr2 \end{cases} \quad (A-4)$$

$$OT(Ins(p1, s1), Del(p2, l1)) = \begin{cases} Ins(p1, s1) & p1 \leq p2 \\ no-op & p2 < p1 < p2 + l1 \\ Ins(p1 - l1, s1) & p1 \geq p2 + l1 \end{cases} \quad (A-5)$$

$$OT(Del(p1, l1), Ins(p2, s1)) = \begin{cases} Del(p1, l1) & p1 + l1 \leq p2 \\ Del(p1, l1 + |s1|) & p1 < p2 < p1 + l1 \\ Ins(p1 + |s1|, l1) & p1 \geq p2 \end{cases} \quad (A-6)$$

$$OT(Del(p1, l1), Del(p2, l2)) = \begin{cases} Del(p1, l1) & p1 < p2 \quad p1 + l1 \leq p2 \\ Del(p1, p2 - p1) & p1 < p2 \quad p2 < p1 + l1 \leq p2 + l2 \\ Del(p1, l1 - l2) & p1 < p2 \quad p2 + l2 < p1 + l1 \\ no - op & p2 \leq p1 < p2 + l2 \quad p1 + l1 \leq p2 + l2 \\ Del(p2, p1 + l1 - p2 - l2) & p2 \leq p1 < p2 + l2 \quad p1 + l1 > p2 + l2 \\ Del(p1 - l2, l1) & p1 \geq p2 + l2 \end{cases} \quad (A-7)$$