Lab Report

COP290: Rust Lab Report

Vani Gupta (2023CS10126)[a], Arnav Panjla (2023EE10978)[a] and Basil Labib (2021TT11175)[a]

[a]*Indian Institute of Technology, Delhi*

Professor Abhilash Jindal, COP290 TAs

Abstract—We added three extensions to the base Rust TUI spreadsheet program, 1) extended functionality such as undo/redo, autofill, etc. to TUI 2) web application to ease usage 3) multi-user, state-consistent CRDT-based WebSocket spreadsheet programme. We will discuss our design decisions, the system architecture for each extension, and the challenges and learnings from this project in this report.

Keywords—*lab report, rust, spreadsheet, CRDT, webapp, WebSocket*

1. Introduction

We proposed a decentralised, state-consistent spreadsheet program. In order to design and build such a complex system, we decomposed it into three checkpoints. We were able to achieve two out of these three checkpoints and one additional checkpoint not included in our proposal.

- ✓Browser-based GUI app using client-server architecture
- ✓Centralised state-consistent real-time GUI app using WebSockets and CRDTs
- ✗Decentralised state-consistent real-time GUI app using WebSockets and CRDTs

In addition, we also extended the base TUI which was not included in our initial proposal.

- ✓Added undo/redo up to 5 previous steps
- ✓Predict cell values and autofill ranges
- ✓Support for non-integer datatypes like string, float and consistent operations on strings and floats

2. TUI Extension

2.1. Command-Line Interface and Mode Selection

Our Rust-based spreadsheet system supports two runtime-selectable modes: **Standard** and **Extended**. This design enables clean separation of concerns and feature modularity.

- **Standard Mode:**

  ```
  make
  ./target/release/spreadsheet 10 10
  ```

  Modules Used: `parser.rs`, `function.rs`, `graph.rs`, `display.rs`
  Core Features:

  - Integer-only cell support
  - Basic arithmetic operations (+, -, *, /)
  - Range functions: SUM, AVG, MIN, MAX, STDEV
  - Sleep functionality to delay cell evaluation

- **Extended Mode:**

  ```
  make
  ./target/release/spreadsheet -extended 10 10
  ```

  Modules Used: `parser_ext.rs`, `function_ext.rs`, `graph_ext.rs`, `display_ext.rs`, `util_ext.rs`
  Additional Features:

  - Support for float and string cell types
  - String concatenation using '+'
  - Pattern-based Autofill: Arithmetic (AP), Geometric (GP), Fibonacci, Constant
  - Undo/Redo via `StateSnapshot`

2.2. Software Architecture

The system follows a layered, modular architecture:

Core Modules:

- `parser.rs` / `parser_ext.rs` – expression parsing
- `function.rs` / `function_ext.rs` – function evaluations
- `graph.rs` / `graph_ext.rs` – dependency tracking and cycle detection
- `display.rs` / `display_ext.rs` – terminal UI rendering
- `util_ext.rs` – shared utilities (e.g., polymorphic arithmetic handling)

Entry Point: `rustlab/cli/main.rs` dynamically selects between standard and extended based on CLI flag `-extended`.

2.3. Key Extensions Implemented

- **Typed Cells:** Cells can now hold `Int`, `Float`, or `String` using an enum-based `CellValue`.
- **String Operations:** Support for string assignment and '+'-based concatenation.
- **Mixed-Type Arithmetic:** Fully supports operations like 'Float + Int', 'Int / Int', and errors on invalid ops (e.g., 'String - Float').
- **Autofill Feature:** From a 4-cell seed, detects AP/GP/Fibonacci/Constant and populates the column up to a given length.
- **Undo/Redo:** Snapshots captured as `StateSnapshot`, used for user-directed rollback.

2.4. Primary Data Structures

- `CellValue (enum): Int(i32) | Float(f64) | String(String)`
- `Cell`: Holds `CellValue` and a validity flag
- `Formula`: Stores op_type, op_info1, op_info2 to describe each cell's formula
- `Graph / GraphExt`: Adjacency list and range dependency representation
- `State / StateSnapshot`: Captures pre-modification state for undo/redo

2.5. Module Interfaces

- `parser → graph`: Adds/removes edges and ranges
- `parser → function`: Invokes range or arithmetic evaluation
- `graph → recalc`: Handles topological sort and propagation
- `display → core`: Renders spreadsheet with ERR/value handling
- `parser_ext → util_ext`: Performs typed arithmetic via `arithmetic_eval`

2.6. Encapsulation Strategies

- Separate file namespace for '_ext' modules; standard and extended logic never mix
- Use of stateless pure functions where possible
- Unsafe usage (`static mut`) restricted to well-documented parser.rs
- Public APIs shield internals of evaluation and recalculation logic

2.7. Design Justification

- Easy switching between standard and extended logic based on mode
- Clear path for future extensions (e.g., Date type, Graph plots)

- Minimized duplication through shared utilities like `util_ext.rs`
- Debug-friendly separation: extended features can be tested without affecting standard ones

2.8. Design Modifications During Development

- Introduced `static mut HAS_CYCLE` and `INVALID_RANGE` for backtracking in parser
- Moved from shared modules to a clean `_ext.rs` hierarchy
- Added pattern recognition + generation to support autofill logic
- Modified formula structure to accommodate multi-typed cells

2.9. Demonstration of Extensions

The following session demonstrates the extended features including string operations, float handling, mixed-type formula evaluation, and autofill.

Running the Spreadsheet in Extended Mode

```
$ ./target/release/spreadsheet -extended 10 10
```

This command launches a 10x10 spreadsheet grid in extended mode.

Testing String Cell Support

We input string values and concatenate them:

```
> A1="hi"
> A2="hello"
> A3=A1+A2
```

Result:

- A1 = "hi"
- A2 = "hello"
- A3 = "hihello" (concatenation successful)

If A1 is changed to "hello", A3 will be changed to "hellohello"

Testing Float Support and Arithmetic

We input float and integer values, then add them:

```
> B1=1.09
> B2=90
> B3=B1+B2
```

Result:

- B1 = 1.09
- B2 = 90
- B3 = 91.09

If B2 is changed to 0, B3 will be changed to 1.09

Autofill Feature Demonstration

We manually seed a pattern and autofill a column:

```
> C1=1
> C2=2
> C3=3
> C4=4
> =autofill C 10
```

Result:

- Column C is autofilled as 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

Similarly, patterns like AP, GP, fibonacci and constants can be autofilled.

Undo/Redo Functionality

Demonstrates snapshot-based state restoration:

```
> A1=90
> A2=8
> undo        % Reverts A2=8
> undo        % Reverts A1=90
> redo        % Redoes A1=90
> redo        % Redoes A2=8
> redo        % No effect (nothing to redo)
```

Result:

- After first undo: A1 = 90, A2 = 0
- After second undo: A1 = 0, A2 = 0
- After two redos: A1 = 90, A2 = 8

3. Web Application GUI

3.1. Frontend Application Architecture

The frontend is implemented in Rust and compiled to WebAssembly using the Yew framework, offering a dynamic and reactive interface for interacting with the spreadsheet application.

3.1.1. Frontend Structure

- **App Component** - The root component responsible for initializing and structuring the application.
- **TableComponent** - Renders the spreadsheet grid including rows, columns, and populated cell data.
- **CellComponent** - Represents each editable spreadsheet cell with live interaction support.
- **RequestForm** - An input area for command and formula execution.

3.1.2. State Management

- **AppContext** - Global context for managing application-wide state via reducer pattern.
- **AppState** - Contains shared state data, such as the sheet and UI refresh triggers.
- **AppAction** - Defines state-changing actions such as refreshing the view or updating sheet content.
- **Sheet Model** - The internal data structure representing the spreadsheet grid.

3.1.3. Component Features

TableComponent

- Fetches data dynamically from the backend API.
- Renders an Excel-style grid with labeled row and column headers.
- Refreshes the view when a state trigger is activated.

CellComponent

- Allows direct in-place editing of individual cells.
- Submits values on Enter key press.
- Manages focus and selection for a seamless editing experience.
- Communicates cell updates to the backend via API.

RequestForm

- Provides a dedicated field for command or formula input.
- Sends asynchronous requests to the server.
- Displays responses or errors as formatted feedback.

3.1.4. API Communication

- Uses `gloo_net` for performing HTTP requests in the browser.
- Serializes and deserializes data using JSON for communication with the server.
- Handles different content types and responses from multiple API endpoints.
- Provides robust error handling for failed requests and malformed responses.

3.1.5. UI Features

- Clean and modern responsive design.
- Clear feedback for loading states and errors.
- Well-structured headers for intuitive navigation.
- A command center panel for formula input and result display.

3.1.6. Technical Implementation

- Compiled to WebAssembly for high-performance execution in modern browsers.
- Uses Yew hooks to manage component lifecycles and reactivity.
- Implements event-driven interactions including form submission and input events.
- Employs node references for accessing and manipulating DOM nodes.
- Provides in-browser logging and error diagnostics through console output.

3.2. Challenges Faced

Throughout the development of the spreadsheet application, several technical and architectural challenges were encountered:

- **Formula Parsing Complexity** - Implementing a parser capable of handling nested formulas, precedence rules, and cell references required custom logic and rigorous testing.
- **Dependency Tracking** - Maintaining a real-time graph for dependent cells and updating them in the correct order posed consistency and performance challenges.
- **Asynchronous Communication** - Synchronizing frontend updates with backend state through async HTTP requests introduced potential race conditions and refresh issues.
- **Undo/Redo System** - Designing a history stack that captures full application state snapshots while maintaining performance proved non-trivial.
- **WebAssembly Limitations** - Debugging and error tracking in Rust-compiled WebAssembly environments required special attention due to limited debugging tools in browsers.
- **Frontend-Backend Type Matching** - Ensuring accurate serialization and deserialization of complex types like formulas and expressions required strict JSON schema adherence.
- **Thread Safety** - Managing concurrent access to shared state (‘Arc<RwLock<...»‘) without introducing deadlocks or inconsistencies needed careful architectural design.

3.3. Future Improvements

Several areas have been identified where the application can be further enhanced:

- **UI Enhancements** - Add features such as cell coloring, borders, copy-paste, and multi-cell selection for a richer user experience.
- **Live Formula Suggestions** - Integrate formula autocomplete and inline documentation similar to Excel for improved usability.
- **Performance Optimization** - Introduce diff-based updates instead of full sheet refreshes to reduce backend load and network traffic.
- **Cell Format Types** - Enable formatting for numbers, currencies, dates, and percentage representations.
- **Persistent Storage** - Integrate a database (e.g., SQLite or PostgreSQL) for saving and loading spreadsheet sessions across sessions or users.
- **User Authentication** - Add login/signup support to allow users to manage private spreadsheets and sync across devices.
- **Testing Coverage** - Expand unit and integration test coverage, particularly around formula evaluation and graph updates.
- **Real-time Collaboration** - Implement WebSocket-based multi-user editing support for real-time collaboration.

This frontend architecture combines the performance of WebAssembly with the ergonomics of Rust and the reactivity of Yew, resulting in a fast, maintainable, and user-friendly spreadsheet interface.

4. WebSocket and CRDT-based Web Application

4.1. System Design

The `WebSocket` based approach implements a `WebSocket` enabled server which handles incoming requests from clients and promotes them to a `ws` connection which is handled by a different thread using `tokio::spawn()`.

- **crdt/** - Contains `structs` and datatypes for client-server communication, storing the sheet data model, and CRDT (Conflict-free replicating data types)-based `structs`.
- **server/** - Contains implementation of a simple WebSocket based server which handles `on_connection`, `grid_update`, and `on_close` requests from the client. Uses `tokio` and `tokio::tungstenite`.
- **ws_client/** - A WebSocket-enabled client using `Leptos` and uses timestamps to update and make appropriate requests to the server.

4.2. Structs and Interfaces

4.2.1. crdt/

- **Client** - represents a Client with a unique name at the server.
- **Event** - represents a generic event with some data and a typef
- **InitEvent** - represents an `Init` event when client wants to initiate a connection. The `InitEvent` contains a name string.
- **GridUpdateEvent** - represents a grid update event when a client makes a change to the sheet. It contains the name of the change-maker and the entire grid data.
- **ClientListEvent** - represents the list of clients currently connected to the server. The server broadcasts this information for every client to update its own client list.
- **Column** - represents a single cell. It contains the name of the last client who changed it, the timestamp, the index, and the value.
- **Row** - represents a `Row` in the sheet. Each `Row` contains a list of `Column` structs.

4.2.2. server/

The `server` object is a simple WebSocket-based server which listens on port 3030 for any client connection requests. It spawns a thread for each client and goes into an infinite loop serving that client until the client disconnects. There are separate handlers for each client action:

- `accept_connection(TcpStream, Clients)` - using `tokio::tungstenite`, we promote the connection to a WS connection and match the client request to event type and dispatch the corresponding handler.
- `handle_init(InitEvent, Clients)` - adds this client to the `ClientList` struct and broadcasts the list so that other clients update their local lists.
- `handle_grid_update(GridUpdateEvent, Clients)` - simply read, decode the Event object and broadcast to everyone on the client list.
- `handle_close(client_id, Clients)` - remove this client from the `ClientList` and broadcast the updated `ClientList` to everyone.

4.2.3. client/

The `ws_client` module uses `Leptos` to build a simple frontend for the application. The markup is stored in `index.html` which contains an input field for the username, a Client list to show in the DOM and a table component for rendering the grid.

The App component in `ws_client/lib.rs` uses two effects to handle local state from incoming server messages and another to propagate its own local state to the server using client requests.

The rest of the code builds the following components:

- **App** - main component which renders the page and handles server communication. It also handles changes to the local sheet data if required.
- **Connect** - a simple `FormInput` component which sends an `InitEvent` request to the server.
- **Clients** - a list of actively connected clients as broadcasted by the server in its `ClientListEvent` message.
- **Grid** - a table component which renders the spreadsheet and sends a `GridUpdateEvent` to the server if any of the cell is updated by the user.

4.3. Challenges faced

We faced multiple challenges during the implementation of this app. Some of them are listed below:

- **Lack of accessible literature on CRDT** - This made it difficult to understand CRDTs as a beginner from reading academic papers. Eventually, we found few blogposts which explained the concept well.
- **Understanding tokio library's async/await model** - The async/await model used by `tokio` to implement its server was not clear to us and took time to implement properly to handle multiple client requests.
- **Outdated Leptos packages** - There were few outdated `leptos-use` package which we had to use for constraints with `leptos`. This made us rewrite the `ws_client` using more primitive `web_sys::WebSocket`, which Leptos depends on.

5. Demo images

5.1. WebSocket based GUI Application

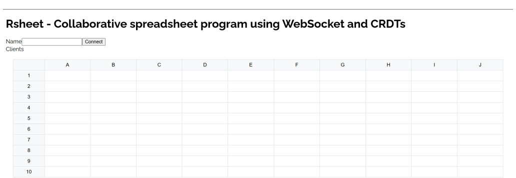

Figure 1. Frontend of the CRDT-enabled websocket application.

Figure 2. Screenshot of the terminal showing server and client processes.

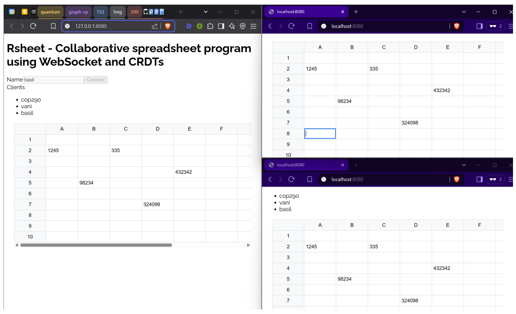

Figure 3. Screenshot showing multiple users interacting with the same spreadsheet.

5.2. Client-Server based GUI Application

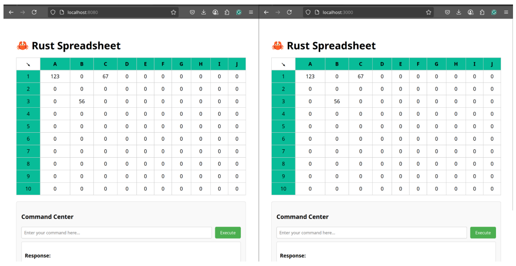

Figure 4. Screenshot of multiple clients.

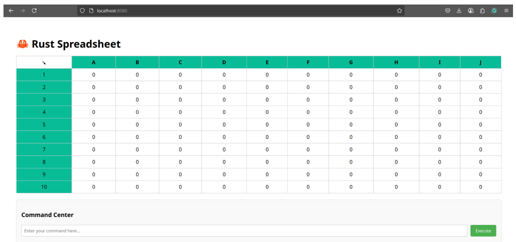

Figure 5. Showcasing Web GUI

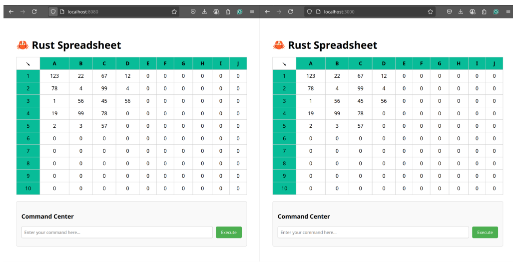

Figure 6. Synchronizing data between different clients

5.3. Extended TUI Application

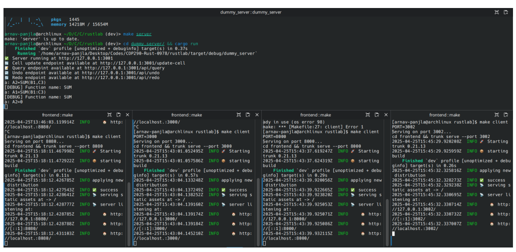

Figure 7. Running multiple clients

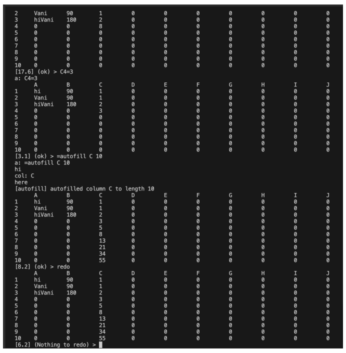

Figure 8. Performing string concatenation

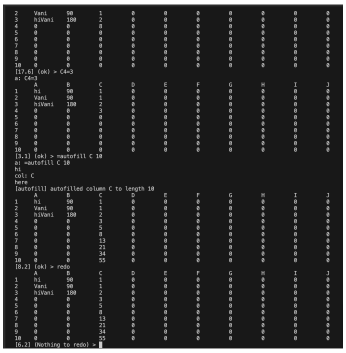

Figure 9. Performing recalculation and float-int addition

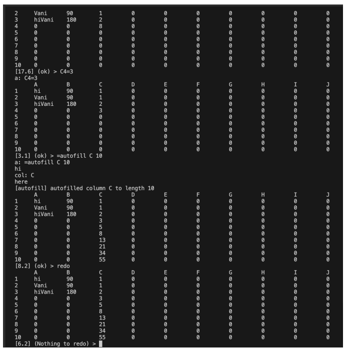

Figure 10. Performing recalculation

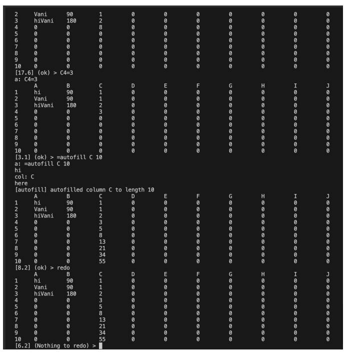

Figure 11. Performing autofill for AP series, and undo

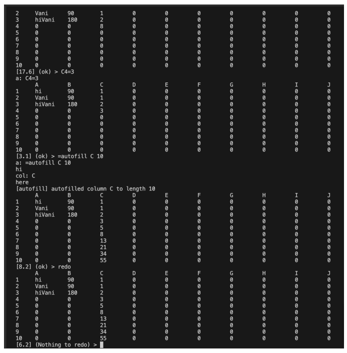

Figure 12. Performing undo

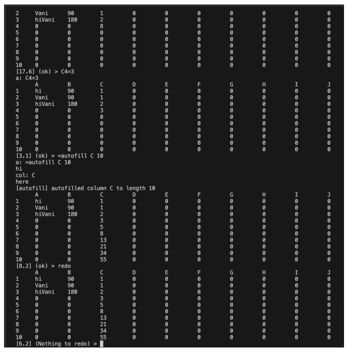

Figure 13. Performing redo

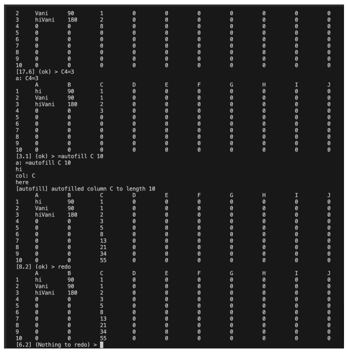

Figure 14. Performing autofill for GP

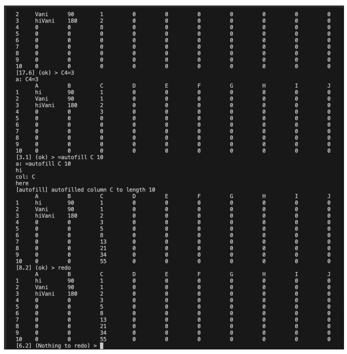

Figure 15. Performing autofill for fibonacci

