COP290 C lab: Spreadsheet Implementation

Shivankur Gupta 2023CS10809
Vanshika 2023CS10746
Samyak Sanghvi 2023CS10807

April 21, 2025

Contents

1 Introduction

We have migrated the existing spreadsheet project from C to Rust, enhanced its functionality, and integrated a browser-based GUI. The frontend is built using HTML and CSS and the backend is built using Rust. The application provides an intuitive interface with navigation, support for complex formulas, graph plotting similar to excel and enhanced editing features such as copy and paste, and formula management.

2 Design and Software Architecture

2.1 System Overview

The architecture adopts a client-server model with a browser-based frontend and a Rust backend. The system encompasses core spreadsheet functionality while supporting modern features like formula management, undo/redo capabilities, and file operations.

2.2 Architecture Components

2.2.1 Frontend Layer

- **Technologies**: HTML, CSS, WebAssembly
- **Frameworks**: eframe for GUI rendering, wasm for browser execution
- **Key Components**:
  - Cell Grid Renderer - Manages the viewport and displays the active 10×10 grid
  - Formula Bar Component - Facilitates formula entry and editing
  - Navigation - Handles keyboard/mouse/touchpad navigation
  - Event Handler - Processes user actions and communicates with backend

2.2.2 Backend Layer

- **Technologies**: Rust, actix-web
- **Core Components**:
  - Cell Storage Engine - Maintains the spreadsheet data structure (up to 999 rows × 18,278 columns)
  - Formula Evaluator - Processes formulas and functions
  - Dependency Graph - Tracks cell relationships for efficient recalculation
  - Recalculation Engine - Updates dependent cells when values change
  - Error Handler - Manages exceptional conditions and circular references
  - Command Processor - Interprets user commands and delegates actions

2.2.3 Data Persistence Layer

- **File Storage**: Gzip compression with bincode serialization
- **Formats**: Custom binary (.bin.gz) and CSV export/import
- **Version Tracking**: Sequential state storage for undo/redo operations

2.3 Communication Flow

1. User actions in the browser trigger WebAssembly functions
2. WebAssembly code communicates with Rust backend via actix-web
3. Backend processes commands and updates the spreadsheet state
4. Dependency resolution triggers recalculation of affected cells
5. Updated state is returned to frontend and rendered
6. Changes are persisted to storage in compressed format

2.4 API Endpoints

The system exposes RESTful endpoints:

- `/load/{filename}` - Retrieves spreadsheet data
- `/save/{filename}` - Persists current state
- `/edit/{cell}` - Modifies cell content
- `/copy/{range}`, `/cut/{range}`, `/paste/{cell}` - Clipboard operations
- `/formula/{string}` - Processes formula input
- `/release/{range}` - Reverts cell modifications

2.5 Performance Considerations

The architecture optimizes for:

- Minimal memory footprint through sparse cell storage
- Efficient recalculation via dependency tracking
- Reduced network traffic with compressed data formats
- Responsive UI through WebAssembly compilation

2.6 Security Model

- IP-based access limitations for private files
- Metadata tracking for file ownership
- Input validation to prevent formula injection

3 Implementation Details

3.1 Modules

We have the following modules:

1. **Cell** : This module contains the cell struct and its functions like initialization, dependencies insert , dependencies delete, dependencies contains etc.
2. **Spreadsheet** : This module is the main module which contains the main function and all the functions related to the spreadsheet like initialization, formula evaluation, dependencies management, error handling etc. It also contains the main loop of the program which takes the input from the user and calls the respective functions based on the input.
3. **login** : Handles user registration, login, logout, and password management (reset/change). Manages session creation and validation (using cookies stored in a global session map). Reads and writes user data from a JSON file (users.json) and also handles per-user file management (list.json for spreadsheets).
4. **mailer** : Responsible for generating and sending password reset emails. Used by the login flow when a user requests to reset their password.
5. **saving** : Manages the persistence of spreadsheets to disk. Serializes spreadsheets with compression (using gzip) and writes to files with a ".bin.gz" extension.
6. **downloader** : Converts spreadsheet data into downloadable formats such as CSV and XLSX. Used by API endpoints for exporting user data.
7. **graph** : Generates graphs (such as line, bar, scatter, and area charts) based on spreadsheet data. Uses parameters such as X and Y ranges along with options (title, labels, dimensions) to produce image data.

8. **app (middleware)** : Defines the overall routing structure of the website. Separates public routes (login, signup, read-only sheet access) from protected routes (editing spreadsheets, updates, downloads, etc.). Attaches shared state (such as the current spreadsheet and user file lists) to the router. The middleware function (require_auth) checks if a valid session exists, and if not, either permits public API access (if the file is marked public) or redirects to the login page.

4 Webserver Management

This section provides a brief overview of the webserver's endpoints and its management logic.

4.1 Public Endpoints

- **Authentication Pages:** `/login`, `/signup`, `/logout`, `/forgot-password`, `/reset-password`, `/change-password`. These endpoints serve pages and handle form submissions for user registration, login, and password recovery.
- **Static Assets:** Served from the `/static` directory.
- **Public Sheet Access:** Endpoints such as `/:username/:sheet_name` that allow users to view spreadsheets if marked public, and API endpoints (`/api/sheet`, `/api/cell/:cell_name`, `/api/sheet_info`) which return read-only data.

4.2 Protected Endpoints

- **Spreadsheet Operations:** Endpoints like `/sheet` which serve the editing UI, along with APIs for updating cells (`/api/update_cell`), saving (`/api/save`), exporting (`/api/export`), loading (`/api/load`), and graph generation (`/api/graph`).
- **Download Endpoints:** `/api/download/csv` and `/api/download/xlsx` for converting spreadsheet data into downloadable formats.
- **File Management:** Endpoints for listing user files (`/:username`), creating (`/:username/create`), updating status (`/:username/:sheet_name/status`), and deleting spreadsheets (`/:username/:sheet_name/delete`).

4.3 Management Logic

- **Authentication Middleware:** The middleware intercepts protected requests to ensure that a valid session cookie exists.
  - If a valid session is found (via `validate_session`), the associated username is added to the request context and the request proceeds.
  - If no valid session is detected, for API endpoints the middleware checks the corresponding user's `list.json` to determine whether the requested spreadsheet is public.
  - If neither condition is met, the user is redirected to the login page.
- **Session Management:** User sessions are created during login, stored in a global session map with an expiration time, and validated by the middleware. This ensures that only authenticated users can access protected endpoints.
- **Module Overview:**
  - *login:* Handles user registration, login, logout, password resets, and session management.
  - *mailer:* Responsible for sending password reset emails.
  - *saving:* Manages serialization and persistence of spreadsheets using compression.
  - *downloader:* Converts spreadsheet data into CSV and XLSX formats.
  - *graph:* Generates graphs (such as line, bar, scatter, area) from spreadsheet data.

5 Design Decisions and why this is a good design

1. We have used the crate of plotters to plot the graph of the given data in different types like line graph, bar graph, area graph, scatter graph etc. This is a good design decision because it provides flexibility in data visualization and supports multiple formats which can enhance user experience and understanding of the data.

2. We have used the crate of actix-web to create the server and handle the requests from the client. This is a good design decision because actix-web is a powerful, performant, and asynchronous framework that helps build scalable and responsive web servers easily.

3. We have used the crate of bincode to serialize and deserialize the data for saving and loading the file. This is a good design decision because bincode is highly efficient in binary serialization, leading to faster save/load times and reduced file sizes, which improves performance.

4. We have used the crate of gzip to compress and decompress the data for saving and loading the file. This is a good design decision because compressing the serialized data reduces storage space and speeds up file transfer or I/O operations.

5. We have used the crate of serde to serialize and deserialize the data for saving and loading the file. This is a good design decision because serde is a highly flexible and reliable serialization framework that works seamlessly with many data formats and types.

6. - We have Spreadsheet as an object oriented approach. The spreadsheet is an object which contains the cells and the functions to manipulate the cells. This allows us to easily add new features in the future without changing the existing code. This is a good design decision because encapsulating functionality within objects makes the code modular, reusable, and easier to test or extend.
   - We have used the crate of regex to parse the formula and validate the input in a single function. This abstraction allows us to easily modify the format of the formulas as input from user in future without affecting the rest of the codebase. This is a good design decision because regex provides powerful pattern matching, allowing robust validation and parsing with minimal code complexity.
   - We ensure that the parsing is done only once for each formula and the formula is then passed as arguments to other functions. This also saves time of parsing the formula again and again. This is a good design decision because it minimizes redundant computations, leading to more efficient runtime performance, especially as the spreadsheet grows.
   - Formula as Enum : Since there can be different types of formulas for various operations, we have used an enum to represent the different types of formulas. This allows us to easily add new types of formulas in the future without changing the existing code. This is a good design decision because enums offer a type-safe way to handle distinct variants and allow pattern matching, improving code clarity and reducing bugs.
   - Dependencies : We store the cells that depend on the current cell in a vector and then convert it into an ordered set when the number of dependencies increases. This allows us to efficiently manage the dependencies and also saves space for the first few commands. This is a good design decision because it balances memory efficiency with performance, adapting to both small and large dependency graphs dynamically.
   - The workflow inside the spreadsheet is as follows:
     - The user enters a command in the command line.
     - The command is parsed.
     - The dependencies are checked and only then the evaluation proceeds. This helps in avoiding unnecessary calculations and also helps in avoiding circular dependencies else we will have to revert the changes made if cycle is detected. This is a good design decision because it ensures correctness of spreadsheet evaluations and protects against logical errors like cycles.
     - The dependencies are now updated, this is required to be done before evaluation so that evaluating current cell also re-evaluates it's correct dependents. This is a good design decision because it guarantees that the computation order respects dependency relationships, ensuring consistent and accurate outputs.

- To avoid unnecessary re-calculations, topo sort is must, we must update the dependents in the order of their dependencies. This is a good design decision because topological sorting ensures correct update ordering in dependency graphs, minimizing computational overhead and preventing stale values.
- The updated spreadsheet is displayed to the user. This is a good design decision because immediate feedback helps the user understand the result of their command and makes the application feel responsive.

6 Changes made to Proposed Implementation

- We did not add the CUT implementation in our final software since it involved multicellular formulas which takes updating several cells in one command. If we had to do so then we have to check for cycles without changing original dependencies since the command may be cycle involving, which would require creating a clone or copy of dependencies which was a huge overhead.
- For UNDO REDO and RELEASE, we have simply implemented a command UNDO which just takes us to the previous state.
- While not implemented in the current version, extending the system to support floating-point numbers would require a minimal change from `i32` to `f32` in our core data structures. This modification was deemed a straightforward extension that could be addressed in future iterations without significant architectural changes.

7 Extra extension ideas

1. We could support multicellular formulas, which we skipped here due to time constraints. This would allow users to enter formulas that span multiple cells, enhancing the spreadsheet's functionality.
2. We could also support more complex functions like VLOOKUP, HLOOKUP, and other advanced Excel functions.
3. In our GUI we could have supported drag and drop functionality for cells, which would allow users to easily move cells around the spreadsheet.
4. In out GUI we could have added the functionality of selecting multiple cells which is right now restricted to selecting only single cells.

8 Primary Data Structures

1. **Spreadsheet**: The spreadsheet struct is the main data structure that represents the entire spreadsheet. It contains a 2D array of cells, each cell being represented by a cell struct. It also contains the functions to manipulate the cells and the spreadsheet.
2. **Cell**: The cell struct is the core data structure that represents each cell in the spreadsheet. It contains the value, formula, location, and dependents of a cell.
3. **Formula**: The formula is represented as an enum which contains the different types of formulas that can be used in the spreadsheet. This allows us to easily add new types of formulas in the future without changing the existing code.
4. **Vector**: The vector is used to store the dependencies of a cell and is converted into an ordered set when the number of dependencies increases.
5. **B Tree**: It is used to store the dependents of a cell. It is implemented as an AVL tree which allows us to efficiently manage the dependencies and also saves space for the first few commands. Also this is used while implementing recursive functions.
6. **Stack**: The stack is used to implement the UNDO functionality. It stores the previous state of the spreadsheet and allows us to revert to the previous state when required. Also, it is used while implementing recursive functions.

9 Interface between software modules

1. Spreadsheet Cell Interface : The Spreadsheet module interacts with Cell module using the methods such as get_dependents, remove dependents and update dependencies which calls for cell_dep_insert, cell_dep_remove. For setting the value of cell it directly accesses the value field of Cell struct which is public.
2. Main file interacts with Spreadsheet by calling for the functions such as display and execute command using set_cell_value or undo function.
3. Event-Command Interface : User actions from frontend (or terminal) are captured and sent to the backend via WebAssembly. The backend processes these commands and updates the spreadsheet state accordingly. The frontend then receives the updated state and renders it for the user.
4. Graph-Spreadsheet Interface : The graph.rs logic in the backend interacts using the intermediate backend app.rs which takes as input the response from frontend and sends it to the graph module. The graph module then processes the data and returns the graph data to the frontend for rendering.
5. Downloader-Spreadsheet Interface : The downloader module interacts with the spreadsheet module using the methods such as download and upload which calls for the functions in the spreadsheet module to get the data and save it in the required format.
6. Login Interface :

10 Approaches for Encapsulation

The software is designed to encapsulate the core functionality of a spreadsheet application, providing a clear interface for users to interact with. The encapsulation is achieved through the use of modules and functions that abstract away the underlying complexity of the implementation.

1. We have a main.rs file which imports the spreadsheet module. The main.rs file contains the main function which is the entry point of the program. The main function initializes the spreadsheet and starts the event loop.
2. We have defined a struct spreadsheet which contains fields for storing the cells, the number of rows and columns, undo stack and visible section for display.

11 Whether we had to modify the design

1. We encountered performance issues during execution, particularly related to time efficiency. To address this, we leveraged flamegraph profiling to identify bottlenecks in our implementation. Based on the insights gained, we made several optimizations. For example, we ensured that formula parsing is performed only once per formula to avoid redundant computations. Additionally, we used `lazy_static` to precompile regular expressions, reducing the overhead of repeated regex compilation during runtime.
2. We ran into storage limitations, especially as the spreadsheet grew larger. To optimize memory usage, we changed the data types of row and column indices from default integer types to `u16`, as it was the smallest type capable of representing all possible row and column values. This change resulted in noticeable memory savings.
3. Through flamegraph analysis, we also discovered that frequent allocations and deallocations of `String` objects in the dependencies list were contributing to performance degradation. To resolve this, we modified the dependency tracking system to store row and column indices as tuples of integers instead of strings. This significantly reduced memory overhead and improved runtime efficiency.

12 Data Structures and Uses

The following Datastrutures have been used in this implementation

- **Ordered Set (AVL Tree)**: To store a large list of dependencies of a cell.

- **Cell Structure**: To store the value, formula, location and dependents of a cell.
- **Directed Acyclic Graph**: To model and capture formula dependencies.
- **Linked List**: To store the order of operations after Topo sort.
- **Stack**: To replace recursion, allowing larger recursion depths.
- **Vector**: To initially store the list of dependencies of a cell.

13 Covered Edge Cases

In our testing suite, we have done unit testing as well as Blackbox testing. The following edge cases were covered:

- **Cyclic Dependency**:
  - We tested if the system can detect cyclic dependencies between cells. The system correctly identifies circular dependencies and raises an error message while rejecting the formula. (tested by `input_cycle.txt` and `spreadsheet_test.c`).
  - $A1 = 0 * B1$ $B1 = A1$ instead of A1 being a constant, it is recognised as dependent on B1, hence it shows cyclic dependency.
- **Invalid Formula**:
  - In addition to the well-defined signatures, we have given support for categorizing numerals with + and − signs, so inputs like $1 + +1$ and $1 + -1$ are considered valid.
  - All other inputs and out-of-bound inputs are raised as invalid.
  - We do not support case invariance in any function or arbitrary '0's in the cell names.
  - We also consider range functions with empty inputs (eg. MAX(), SLEEP()) to be a invalid input. (tested by `input_arbit.txt`, `input_unrec_cmds.txt` and `spreadsheet_test.c`).
- **Sleep Cells**:
  - We tested if the normal sleep function works as expected. We tested for cascaded sleep cells and their behavior upon changing the base of the cascade.
  - We also tested for SLEEP(num), where num is a negative number. (tested by `input_sleep.txt` and `spreadsheet_test.c`).
- **Invalid Arithmeic**: We tested if logically invalid arithmetic operations, such as division by zero, and its cascades are flagged correctly by showing ERR on display. (tested by `input_checks_err.txt`)
- **Scrolling**: We unit-tested the scrolling nature to match the specifications of the assignment. (tested by `scroll_test.c`)
- **Datastructures Correctness**: We unit-tested for the correctness and robustness Cell struct and Spreadsheet. (tested by `spreadsheet_test.c` and `cell_test.c`)
- Empty input : Empty input i.e. just '\n' from the user is recognised as invalid command.

14 Links

GitHub Repository of the project