Documentation

Dhruv Pawar Himanshi Bhandari Eshita Zjigyasu

25th April 2025

Why were the proposed extensions not implemented

The primary limitation we encountered was that our backend logic, developed during the C Lab, was already highly robust and tightly coupled. Implementing any major extensions would have required significant modifications to this backend, making the process labor-intensive and potentially necessitating a complete overhaul of the existing logic.

As a result, we decided not to implement certain proposed extensions, including:

- Database integration
- Filtering functionality
- Conditional statements
- Support for floating-point numbers

Could we implement extra extensions over and above the proposal?

Yes, we were able to implement a few additional features beyond the original proposal. These include:

- Light and dark theme support for improved user interface customization
- Color-based cell dependency tracking, which visually distinguishes parent and child cell relationships

Primary Data Structures Used

1. `VecDeque`
    - **Usage**: `undo_stack` and `redo_stack`

- **Application**: Acts as stacks to store snapshots of the grid for undo/redo functionality.
  - `undo_stack`: Stores previous states of the grid.
  - `redo_stack`: Stores states undone by the user for redoing.
- **Reason for `VecDeque`**: Efficient push and pop operations from both ends.

2. `Vec`
   - **Usage**:
     - `grid`: The 2D spreadsheet grid.
     - `formula_strings`: Stores user-entered formulas.
     - `copy_stack`: Temporary storage during copy/cut operations.
     - `dependents` (within `CellData`): Tracks dependent cells.
     - `dirty_stack` and `process_stack`: Used for topological sorting and dependency resolution.
     - `csv_data`: Temporary storage for CSV import/export.
   - **Application**: Provides dynamic resizing and efficient indexing.
3. **Stack** (Implemented using `Vec`)
   - **Usage**:
     - `stack` in `reset_found`
     - `dirty_stack` in `set_dirty_parents`
     - `process_stack` in `update_dependents`
   - **Application**: Used for depth-first traversal, topological sorting, and cycle detection.
4. `UnsafeCell`
   - **Usage**: `grid: UnsafeCell<Vec<Vec<CellData>>>`
   - **Application**: Enables interior mutability for the 2D grid of cells, allowing mutable access even when the `Backend` instance is immutable. This facilitates efficient updates to cell values and their dependencies.
5. `String`
   - **Usage**:
     - `filename`: Stores the filename for CSV import/export.
     - `formula_strings`: Contains user-entered formulas.
   - **Application**: Manages textual data such as formulas and CSV content.
6. `Tuple`

- **Usage**:
  - `(usize, usize)`: Represents cell coordinates.
  - `(i32, CellError, Vec<(i32, i32)>)`: Captures a snapshot of a cell's state.
- **Application**: Efficiently groups related data.

7. `Range`
   - **Usage**: Used in loops to iterate over rectangular ranges of cells.
   - **Application**: Facilitates implementation of range functions like `SUM`, `AVG`, `MIN`, and `MAX`.

Summary

The `Backend` struct employs a variety of data structures to manage the spreadsheet's grid, dependencies, and operations effectively. Key design highlights include:

- **Undo/Redo**: Enabled through `VecDeque` for efficient history management.
- **Dependency Graph**: Managed using `Vec` and stack-based processing.
- **CSV Import/Export**: Relies on `Vec` and `String` for temporary data storage.

Interfaces Between Software Modules

An interface in software defines how different modules (components, services, or layers) communicate and interact. It specifies:

- What data can be exchanged
- What operations can be performed
- How modules should interact

In our spreadsheet application, interfaces ensure that both the terminal and GUI versions work seamlessly by maintaining a consistent structure and communication between frontend and backend logic.

1. Application Workflow Overview

We maintained the original design of the terminal version and extended it by adding GUI-specific modules, ensuring minimal disruption. This modularity enabled reusability of core logic (frontend, parser, backend) across both interfaces.

Limitations faced:

- We did not support float values (decimal precision) as originally proposed.
- We realized that implementing `if-else` would require an abstract syntax tree (AST) for conditionals, so it was excluded.

Module Workflow:

```
main.rs
 cli.rs → frontend.rs → parser.rs → backend.rs   (Terminal mode)
 main_gui/
     app.rs → frontend.rs → parser.rs → backend.rs   (GUI mode)
     index.html
     styles.css
```

Workflow Explained:

- **main.rs:** Entry point. Based on CLI/GUI flag, calls either `cli.rs` or `main_gui.rs`.
- **cli.rs:** Handles terminal input/output. Sends expressions to `frontend.rs`.
- **frontend.rs:** Shared logic. Parses input and sends it to `backend.rs`.
- **parser.rs:** Converts text into structured AST.
- **backend.rs:** Evaluates AST using spreadsheet data.
- **main_gui/:** Web interface using Yew/WASM. `app.rs` manages UI, calls shared `frontend.rs`.

2. Frontend-Backend Interface

Purpose: Connects the UI (Yew) with the Rust backend to handle user interactions and updates.

Components:

- `GridProps`, `FormulaBarProps`, `CommandBarProps`: Define communication between UI components.
- `UseStateHandle<Rc<RefCell<Frontend>>>`: Shared state for backend access.

Example Workflow:

- User selects a cell → state updates in frontend.
- Backend recalculates dependent cells.
- Grid refreshes with updated values.

3. File I/O Interface

Purpose: Allows saving and loading spreadsheets.
Components:

- `to_csv_string()`: Backend to CSV string.
- `load_csv_from_str()`: CSV to backend.
- `download_csv()`: Triggers browser download.

Example Workflow:

- User clicks Save → `to_csv_string()` → `download_csv()`.
- User loads file → `load_csv_from_str()` updates backend.

4. Cell Data Interface

Purpose: Synchronizes visual and logical cell data.
Components:

- `Cell` struct: stores cell state and formulas.
- `get_cell_value()` / `set_cell_value()`: Safe accessor methods.

Example Workflow:

- User edits a cell → `set_cell_value()` → recalculation.
- UI updated via `get_cell_value()`.

Why Are Interfaces Important?

- **Separation of Concerns:** UI and logic stay decoupled.
- **Maintainability:** Modules can evolve independently.
- **Testability:** Each module is independently testable.
- **Scalability:** Future features (e.g., undo/redo) are easy to integrate.

Approaches for Encapsulation

Encapsulation is achieved in our project by organizing code into modular components and exposing only the necessary interfaces between them. While certain modules like `structs.rs` are openly shared, most logic-heavy components follow encapsulation principles to ensure maintainability, testability, and clean separation of concerns.

1. Central Data Structs: `structs.rs`

All data structures such as `Cell`, `Spreadsheet`, and `Function` are defined in `structs.rs`, and **all fields are public**.

- **Impact**: This allows for easy sharing across modules but reduces strict encapsulation. Any module can directly modify internal state, which may lead to tight coupling or hard-to-trace bugs.
- **Opportunity**: In the future, we can make fields private and expose only required getters/setters.

2. Logic Encapsulation: `backend.rs`

The backend is the core engine that evaluates expressions, tracks dependencies, and updates cell values.

- The backend provides **getter and setter methods**, ensuring other modules (like `app.rs`) interact with it through a **defined interface** rather than direct manipulation.
- Internally, `backend.rs` handles all vector and matrix operations (from `vec.rs`) and maintains dependency logic, keeping this complexity hidden from the frontend or parser.

3. Parsing as a Standalone Service: `parser.rs`

The parser is fully decoupled from the backend and GUI. It processes expressions like `=SUM(A1:B3)` and returns them as `Function` AST objects.

- The returned `Function` object is then passed to the backend for evaluation.
- This separation of concerns follows good encapsulation: **parsing and execution are isolated**.

4. Web Frontend Encapsulation: `app.rs` (Yew)

The Yew-based frontend is implemented entirely in `app.rs`. It handles UI rendering, user input, and calls backend methods in response to events.

- The frontend does **not directly access backend data structures**, but uses **function calls** to update or retrieve state.
- This interface acts as a layer of encapsulation between the user interaction layer and the core logic.

Summary of Encapsulation Approaches Used

Module	Encapsulation Approach
`structs.rs`	Shared types across modules; fields currently public
`backend.rs`	Getter/setter interface; internal logic hidden
`parser.rs`	Stateless; returns parsed ASTs to backend
`app.rs`	Uses function calls to backend; encapsulates UI logic
`vec.rs`	Internal helper module; used only within backend

Why This Is a Good Design

Our design choices reflect a deliberate balance between performance, clarity, modularity, and future extensibility. Below, we justify why the current structure serves the project well.

1. Clear Abstraction of Spreadsheet Concepts

We introduced custom types like `Cell`, `CellData`, `Function`, and `Operand` to model spreadsheet concepts directly in code. These abstractions:

- Make the system easier to reason about and debug.
- Prevent logical errors by enforcing type safety.
- Allow extension (e.g., new function types) without affecting unrelated modules.

2. Efficient and Appropriate Data Structures

We selected standard data structures based on their time complexity and use case:

- `VecDeque` enables efficient undo/redo with O(1) operations.
- `HashMap` and `HashSet` offer fast lookup for cell values and dependencies.
- Sparse storage of cells avoids unnecessary memory usage for unused grid locations.

3. Modular and Testable Components

Each file in the codebase has a clearly defined role:

- The parser is fully stateless and testable independently.
- The backend provides a clean API with getter/setter methods, hiding internal complexity.
- The frontend does not manipulate data directly but interacts through exposed interfaces.

This separation allows for easier debugging, testing, and parallel development.

4. Encapsulation and Future Safety

Although fields in `structs.rs` are currently public for convenience, the overall structure respects encapsulation principles:

- Logic-heavy modules hide internal state behind functions.
- Functionality is exposed through well-defined methods, not raw field access.
- The design allows for gradual hardening by making fields private in future iterations.

5. Handles Complexity Gracefully

The use of enums like `FunctionData`, `OperandData`, and `CellError` enables:

- Compact and expressive pattern matching for evaluation.
- Centralized error handling and validation.
- Easier implementation of future features (e.g., ternary operations or new error types).

6. Extensibility and Maintainability

The design makes it easy to add features such as:

- New functions like `MEDIAN` or `IF`.
- New data types or formatting options.
- Cell reference tracing, already supported by our dependency system.

Because responsibilities are cleanly divided, such changes do not require deep rewrites or risk introducing regressions in unrelated parts of the code.

Conclusion

Overall, our design strikes a solid balance between practical implementation and long-term software engineering principles. It is modular, expressive, and robust—ideal for a spreadsheet system with growing complexity.

Whether we had to modify our design

- While converting from terminal to extensions, we did not change the design of the terminal part and kept them separate by just adding functions to the old files and then creating new files for the GUI.

- Hence, we could not support float, which we had proposed in the original proposal (i.e., decimal point precision).
- We also realized that implementing `if-else` would require building an AST to handle complex conditions, and therefore, we did not implement it.