Rust Spreadsheet Lab — Design & Architecture

Ayush Singh (20203CS10322)
Aayush Prasad (2023CS51132)
Prithvi Raj (2023CS50077)

Contents

1 Introduction

This document articulates the design and software architecture of our **Rust Spreadsheet Lab**. It complements inline API documentation by presenting a holistic view of module interactions, data structures, and the rationale behind key decisions. Our objectives were:

- Reimplement the C lab spreadsheet in safe, idiomatic Rust, leveraging language features for memory and thread safety.
- Support all original CLI features (cell formulas, viewport scrolling, built-in functions, output control, incremental recalculation) with identical behavior and autograder compatibility.
- Extend functionality with optional features: cell history logging, multi-level undo/redo, and an interactive GUI charting interface.
- Maintain clean module boundaries to facilitate testing, documentation, and future extension.

2 High-Level Architecture

We adopt a layered, modular design ensuring clear separation of concerns and minimizing inter-module coupling:

```
+-----------------------+       +------------------+
|      main.rs          |<----->| cli_app / gui_app|
+-----------------------+       +------------------+
             ^                            |
             |                            v
     +-----------------+      +-------------------+
     |   sheet module  |<----->|   parser module   |
     +-----------------+      +-------------------+
             ^                            |
             |                            v
     +-----------------+      +-------------------+
     |  Data Storage   |      |  Thread-local     |
     |  (HashMap, Vec) |      |  RANGE_CACHE      |
     +-----------------+      +-------------------+
```

3 Core Modules & Interfaces

3.1 Parser Module

- **Responsibilities**: Lexical analysis of formula strings, recursive-descent parsing into sub-expressions, evaluation with error handling and range caching.
- **Public API**:

```
pub fn evaluate_formula(
    sheet: &CloneableSheet<'_>,
    formula: &str,
    row: i32,
    col: i32,
    error: &mut i32,
    status_msg: &mut String
) -> i32;
```

```
pub fn clear_range_cache();
pub fn invalidate_cache_for_cell(row: i32, col: i32);
```

- **Range Cache**: Thread-local `RefCell<HashMap<String, (i32, HashSet<(i32,i32)>)»` tracking computed range results and their dependencies, enabling O(1) lookups and selective invalidation.
- **Advanced Formulas**: Enabled via the `advanced_formulas` feature, supports IF, COUNTIF, SUMIF, ROUND; CONCATENATE planned.

3.2 Sheet Module

- **Responsibilities**: Manage spreadsheet metadata, cell storage, formula assignment, dependency maintenance, and trigger recalculation workflows.
- **Key Structures**:
  - `Cell`: value, formula index, status, dependency sets, optional history buffer.
  - `Spreadsheet`: grid dimensions, sparse cell map, formula storage, dirty set, cache, undo/redo stacks.
- **Public API**: Construction, formula updates, value and status retrieval, viewport control, history queries, undo/redo.
- **Recalculation**: On cell update, marks dependents as dirty, then performs a topological sort (Kahn's algorithm) in batches for efficiency.
- **Error Propagation**: Division-by-zero or invalid references mark cells and transitive dependents as `Error`, showing "ERR" consistently.

3.3 CLI
GUI Applications

- **CLI App**: A lightweight, autograder-compatible terminal interface that delegates user commands (scrolling, output control, formula updates) to `sheet::process_command`, and uses the display functions (e.g., `display_grid`, `display_grid_from`) to render a dynamic 10×10 viewport.
- **GUI App**: An optional graphical frontend (enabled via the `gui_app` feature) using `eframe/egui` to provide interactive data visualization, chart configuration dialogs, and real-time plot updates (bar, line, scatter with trendlines).
- Both frontends share the same underlying `Spreadsheet` data model, ensuring consistent behavior, data integrity, and unified error handling across modes.

4 Primary Data Structures

4.1 Cell

```
pub struct Cell {
    pub value: i32,
    pub formula_idx: Option<usize>,
    pub status: CellStatus,
    pub dependencies: HashSet<(i32,i32)>,
```

```
    pub dependents: HashSet<(i32,i32)>,
    #[cfg(feature = "cell_history")]
    pub history: VecDeque<i32>, // Recent values log
}
```

4.2 Spreadsheet

```
pub struct Spreadsheet {
    total_rows: i32,
    total_cols: i32,
    cells: HashMap<(i32,i32), Cell>,
    formula_storage: Vec<String>,
    dirty_cells: HashSet<(i32,i32)>,
    cache: HashMap<String, CachedRange>,
    #[cfg(feature = "undo_state")] undo_stack: Vec<PreviousCellState>,
    #[cfg(feature = "undo_state")] redo_stack: Vec<PreviousCellState>,
    // Additional fields: viewport indices, output flags
}
```

4.3 CloneableSheet

A thin wrapper exposing read-only cell views to the parser, avoiding borrow conflicts.

5 Interfaces Between Software Modules

- **main.rs**: Bootstraps the application, selects `cli`$_{a}pp :: main() or$`gui`$_{a}pp :: main() based on Cargo features.$`cli`$_{ap}$

6 Approaches for Encapsulation

- **Module Privacy**: Only essential APIs are `pub`, internal helpers remain private.
- **Feature Flags**: Optional capabilities isolated behind Cargo features, reducing binary size for CLI-only builds.
- **Rust Ownership**: Leverage borrowing and lifetimes to ensure safety and avoid global mutability.

7 Justification of Design Quality

Our design delivers on robustness and performance by:

- Achieving full C-lab feature parity with identical correctness under autograder tests.
- Ensuring safety and preventing runtime panics via Rust's static checks and exhaustive error handling.
- Optimizing memory with sparse storage and reducing allocations via central formula storage.
- Delivering performant incremental recalculation using batched topological sorting.
- Facilitating future growth through modular, well-documented code and feature flags.

8 Design Modifications

Significant improvements over a naive dense-array model:

- Adopted sparse `HashMap` for cell storage, minimizing memory footprint on large, sparse grids.
- Centralized formula deduplication in `formula_storage` to avoid repetitive string cloning.
- Introduced `dirty_cells` and efficient recalculation routines for performance.

9 Limitations & Unimplemented Proposals

Despite prototyping several advanced features, some were deferred to focus on core performance and stability:

- **Vim-like UI**: While a raw-mode prototype using `crossterm` was created, full integration was postponed to prioritize memory/time optimizations and avoid unexpected panics under heavy use. The architecture remains ready for a `vim_ui` feature in future work.
- **CONCATENATE**: Requires extending the value model to fully support text, plus Unicode handling; deferred due to increased complexity and testing overhead.

10 Future Work & Extra Extensions

Beyond the original rubric, our design can support and partially implements:

- **Cell History Table**: Persistent log of the last 10 values per cell, accessible via `history <CELL>`.
- **Multi-Level Undo/Redo**: Captures full cell state and dependency changes, enabling robust rollback across complex edit sequences.
- **Advanced GUI Charts**: Grouped bar, line, and scatter plots with trendlines, interactive tooltips, and zoom controls via the `gui_app` frontend.
- **Performance Optimization**: Plans for custom allocators or specialized hashing to further reduce latency on large datasets.
- **UI Enhancements**: Prospective support for configurable keybindings, theming, and minimaps for rapid navigation in CLI mode.

11 Quality Assurance & Tooling

This project incorporates rigorous quality controls to ensure reliability and maintainability:

- **Linting**: Zero extttclippy warnings enforced; configured with extttclippy.toml for project-specific rules.
- **Formatting**: Consistent code style via extttrustfmt –check; automated in pre-commit hooks.
- **Unit Testing**: Coverage exceeding 80
- **CI Pipeline**: GitHub Actions runs linting, formatting checks, test suite, and doc generation on every pull request, gating merges on all passing checks.
- **Documentation**: Inline Rustdoc on all public APIs; generated docs reviewed for clarity. Dedicated design PDF (this document) provides architectural context.

12 Testing Strategy

Comprehensive tests ensure correctness and performance:

- extbfUnit Tests: Validate parser logic, cell updates, dependency graph behavior, cache invalidation, and undo/redo mechanics.
- extbfIntegration Tests: Simulate user sessions with scripted CLI inputs to verify viewport scrolling, output suppression, and autograder compatibility.
- extbfPerformance Benchmarks: Custom benchmarking harness measures recalculation throughput on large (10,000×1) sparse and dense sheets, guiding optimizations.
- extbfRegression Monitoring: Baseline benchmarks and memory usage snapshots stored in CI artifacts to detect unintended slowdowns or bloat.

13 Conclusion

Our Rust-based spreadsheet faithfully replicates and extends the C lab's functionality with idiomatic safety, modularity, and performance. Feature flags and sparse data structures ensure a lean, maintainable codebase primed for future enhancements.

Our Rust-based spreadsheet faithfully replicates and extends the C lab's functionality with idiomatic safety, modularity, and performance. Feature flags and sparse data structures ensure a lean, maintainable codebase primed for future enhancements.