Rust Spreadsheet Application Documentation

Sourav Kumar Patel (2023CS10751)
Vishal Kumar (2023CS10816)
Vegad Aditya (2023CS10819)

April 25, 2025

1 Design and Software Architecture

The Rust Spreadsheet Application is designed as a modular, multi-crate workspace that follows modern Rust architecture principles. The system is structured around a core spreadsheet engine with separate frontend interfaces for different use cases.

1.1 Workspace Structure

The application is organized in a Rust workspace with multiple crates:

- **cores** - The central spreadsheet engine containing all the core logic
- **cli** - Command-line interface for text-based interaction
- **gui** - Graphical user interface built with Dioxus
- **app** - Main application entry point

1.2 Architectural Patterns

We applied several architectural patterns and principles:

1.2.1 Core-Frontend Separation

The core spreadsheet engine is completely separated from the UI components, allowing:

- Multiple interfaces (CLI and GUI) sharing the same core logic
- Easy testing of core functionality without UI dependencies
- Better maintainability through clear separation of concerns

1.2.2 Component-Based UI Architecture

The GUI is built using Dioxus, a React-like framework for Rust:

- Component hierarchy for reusability and separation of concerns
- Stateful components with reactive updates
- Context-based state management for sharing data across components

1.3 Core Module Design

The `cores` crate implements the following key modules:

- `sheet.rs` - Core spreadsheet data structure and computation engine
- `parse.rs` - Formula parsing and expression evaluation
- `make_graphs.rs` - Data visualization functionality
- `read_csv.rs`/`write_csv.rs` - File I/O operations

1.4 Data Flow Architecture

The spreadsheet implements a reactive data flow architecture:

1. User inputs formulas through CLI or GUI
2. Formulas are parsed into structured commands
3. Dependencies between cells are tracked
4. Changes trigger recalculation of dependent cells using topological sorting
5. Results propagate through the dependency graph
6. UI updates to reflect new values

2 Why Proposed Extensions Could Not Be Implemented

2.1 Row/Column Delete Operation

Implementing a delete operation for rows or columns presented significant challenges:

- **Complex Dependency Management**: Cell dependencies form a directed graph. Deleting a row/column could break the dependency chain, requiring extensive graph restructuring.
- **Vector Limitations**: Our implementation uses Rust's `Vec<Vec<Cell>>` for the grid. Removing elements from the middle of vectors would require shifting all subsequent elements, which is an O(n) operation.

- **Formula Reference Updates**: All cell references in formulas (e.g., A1, B2) would need to be updated to account for the shifted row/column indices.
- **Cycle Detection Complexity**: If dependency cycles emerged after deletion, we would need to restore the entire spreadsheet state, requiring a complex rollback mechanism.

```
// Current dependency tracking in set_dependicies_cell function
let t = row * ENCODE_SHIFT + col;
let depend_vec = &mut self.grid[i][j].depend;
if !depend_vec.contains(&t) {
    depend_vec.push(t);
}
```

Listing 1: Current dependency tracking mechanism

2.2 Multi-Range Selection in GUI

The multi-range selection feature was challenging due to:

- **Complex State Management**: Tracking multiple non-contiguous selections requires a more sophisticated state management system.
- **Event Handling Complexity**: Implementing keyboard modifiers (Ctrl/Shift+click) for multi-selection posed challenges in the Dioxus event handling system.
- **Rendering Overhead**: Efficiently rendering multiple selection highlights across a large grid without performance degradation.

2.3 Heatmap in Graph Section

With four graph types already implemented (line, bar, pie, scatter):

- **Time Constraints**: Prioritization of core functionality over additional visualization types.

2.4 Multiple Sheets

Multiple sheet support would require:

- **Significant Architecture Changes**: Adding a new layer of abstraction to manage multiple Sheet instances.
- **Inter-Sheet Dependencies**: Supporting formulas that reference cells across different sheets.
- **UI Complexity**: Adding sheet navigation and management controls.
- **File Format Expansion**: Extending the file format to store multiple sheets.

Doing all of this in the given time constraint was difficult.

3 Possibility of Additional Extensions

3.1 Sequence Generation Based on Pattern

Implementing pattern-based sequence generation would be feasible by:

- Adding new range function types in the `CommandFlag` structure
- Implementing pattern detection and generation algorithms
- Extending the formula parser to recognize sequence patterns

```
// Extend the range function types
let cmd = match func_name {
    "MIN" => 0,
    "MAX" => 1,
    "SUM" => 2,
    "AVG" => 3,
    "STDEV" => 4,
    "AP" => 6,  // Arithmetic progression
    "GP" => 7,  // Geometric progression
    "FACT" => 8, // Factorial sequence
    _ => {
        container.flag.set_error(1);
        return;
    }
};
```

Listing 2: Potential implementation in parse.rs

3.2 Supporting Floating Point Numbers

Converting to floating-point arithmetic would be straightforward:

- Change the `value` field in the `Cell` struct from `i32` to `f64`
- Update arithmetic operations to handle floating-point values
- Modify the formatter for cell display to handle decimal places
- Update parsing to handle decimal notation in formulas

```
pub struct Cell {
    /// The current calculated value of the cell (changed from i32 to f64
    )
    pub value: f64,
    /// The formula assigned to the cell
```

```
    pub formula: CommandCall,
    /// List of cells that depend on this cell's value
    pub depend: Vec<usize>,
}
```

Listing 3: Cell structure with floating point support

4 Primary Data Structures

The application is built around several key data structures that enable efficient formula processing, dependency tracking, and value calculation.

4.1 Command and Formula Representation

```
pub struct CommandFlag {
    /// Command type: 0 = value/cell, 1 = arithmetic, 2 = range function
    pub type_: B2, // 2 bits
    /// Operation code (depends on type_):
    /// - For arithmetic: 0 = add, 1 = subtract, 2 = multiply, 3 = divide
    /// - For range functions: 0 = MIN, 1 = MAX, 2 = SUM, 3 = AVG, 4 =
   STDEV, 5 = SLEEP
    pub cmd: B3, // 3 bits
    /// Parameter 1 type: 0 = value, 1 = cell reference
    pub type1: B1, // 1 bit
    /// Parameter 2 type: 0 = value, 1 = cell reference
    pub type2: B1, // 1 bit
    /// Error code: 0 = no error, 1 = invalid input, 2 = cycle detected
    pub error: B2, // 2 bits
    /// Division by zero flag: 1 = division by zero occurred
    pub is_div_by_zero: B1, // 1 bit
    /// Reserved bits for future use
    pub is_any: B6,
}

/// A structure representing a parsed formula command.
#[derive(Clone, serde::Serialize, Debug)]
pub struct CommandCall {
    /// Flag bits indicating the command type and attributes
    pub flag: CommandFlag, // 16 bits
    /// First parameter - either a direct value or an encoded cell
   reference
    pub param1: i32, // 4 bytes
    /// Second parameter - either a direct value or an encoded cell
   reference
```

```
    pub param2: i32, // 4 bytes
}
```

Listing 4: Core data structures for formulas

4.2 Efficient Memory Usage with Bitfields

The `CommandFlag` structure uses bitfields to efficiently store multiple flags in a single 16-bit value:

- Saves memory by compressing 16 bits of information into a single field
- Enables fast bit manipulation operations for flag checking
- Allows zero-cost abstraction for accessing individual flags

4.3 Cell and Sheet Structures

```
pub struct Cell {
    /// The current calculated value of the cell
    pub value: i32,
    /// The formula assigned to the cell
    pub formula: CommandCall,
    /// List of cells that depend on this cell's value
    pub depend: Vec<usize>,
}

pub struct Sheet {
    /// The grid of cells in the spreadsheet, represented as a 2D vector.
    pub grid: Vec<Vec<Cell>>,
    /// Number of rows in the spreadsheet.
    pub row: usize,
    /// Number of columns in the spreadsheet.
    pub col: usize,
}
```

Listing 5: Cell and Sheet structures

4.4 Dependency Tracking

Each cell maintains a list of dependent cells, enabling:

- Efficient propagation of value changes through the spreadsheet
- Cycle detection in formula references
- Topological sorting for update order determination

4.5 Cell Reference Encoding

Cell references are encoded into integer values for efficient storage and manipulation:

```
pub fn encode_cell(cell_ref: String) -> i32 {
    // Convert A1 notation to encoded integer format
    let (row, col) = convert_to_index(cell_ref);
    if row == 0 || col == 0 {
        return 0;
    }
    (row as i32) * ENCODE_SHIFT + (col as i32)
}
```

Listing 6: Cell reference encoding

5 Interfaces Between Software Modules

5.1 Core-GUI Interface

The GUI interfaces with the core spreadsheet engine through:

- **Arc¡Mutex¡Sheet¿¿**: Thread-safe reference to the sheet, allowing concurrent access
- **Context Providers**: Using Dioxus contexts to share sheet state across components
- **Signal Handlers**: Reactive state updates when the sheet changes

```
let sheet: SheetContext = use_signal(|| Arc::new(Mutex::new(Sheet::new
   (100, 100))));
provide_context(sheet);

// Usage in child components
let sheet = use_context::<SheetContext>();
if let Ok(sheet_locked) = sheet.cloned().lock() {
    let formula = sheet_locked.get_formula(row, col);
    // Update UI state
}
```

Listing 7: Core-GUI interface in spreadsheet component

5.2 Formula Processing Pipeline

The formula processing pipeline connects multiple modules:

1. **User Input** → String formula from GUI/CLI
2. **Parser** → Converts text to `CommandCall` structure

3. **Dependency Tracker** → Identifies and records cell dependencies
4. **Evaluator** → Calculates results and handles errors
5. **Updater** → Propagates changes to dependent cells

```
pub fn update_cell_data(&mut self, row: usize, col: usize, new_formula:
    String) -> CallResult {
    // Stage 1: Parse formula
    let mut command = parse_formula(&new_formula);

    // Stage 2: Save old command and set dependencies
    let old_command = self.grid[row][col].formula.clone();
    self.remove_old_dependicies(row, col, command.clone());

    // Stage 3: Topological sort
    let topo_vec = self.toposort(row * ENCODE_SHIFT + col);

    // Stage 4: Update cells
    if topo_vec.is_empty() {
        self.grid[row][col].formula.flag.set_error(2);
    } else {
        self.update_cell(topo_vec);
    }

    // Return result with timing information
    CallResult {
        time: start_total.elapsed().as_millis() as f64,
        error: Error::None,
    }
}
```

Listing 8: Formula processing interface

5.3 Visualization Interface

The graph generation system provides a clean interface between data and visualization:

- Methods like `line_graph`, `bar_graph`, etc. convert cell ranges to JSON
- Consistent parameter patterns across different graph types
- Error handling for invalid ranges or data

```
// Interface in Sheet implementation
pub fn line_graph(
    &self,
    range: &str,
    x_labels: &str,
    y_lable: &str,
    title: &str,
) -> Result<String, String> {
    // Convert cell range to chart JSON
}

// Usage in UI component
if let Ok(sheet_locked) = sheet.cloned().lock() {
    let x = sheet_locked.line_graph(
        &range.cloned(),
        &x_label.cloned(),
        &y_label.cloned(),
        &title.cloned(),
    );
    if let Ok(json) = x {
        chart_json.set(json);
    }
}
```

Listing 9: Graph interface example

6 Approaches for Encapsulation

6.1 Structure-Based Encapsulation

We used Rust's module system and struct-based encapsulation:

- **Data Structures**: Defined with clear responsibilities and minimal public fields
- **Implementation Methods**: Functionality encapsulated within impl blocks
- **Public API**: Carefully controlled through exported functions

```
impl Sheet {
    // Public API
    pub fn new(row: usize, col: usize) -> Self {
        // Implementation details hidden
    }
```

```
    pub fn get_value(&self, row: i32, col: i32) -> i32 {
        // Public interface for retrieving cell values
    }

    // Private implementation details
    fn toposort(&self, node: usize) -> Vec<usize> {
        // Internal algorithm not exposed to users
    }
}
```

Listing 10: Encapsulation through method implementation

6.2 Module-Level Privacy

Rust's module system was used to control visibility:

- **Public Exports**: Only necessary types and functions exposed
- **Private Functions**: Helper functions kept module-private
- **Controlled Dependencies**: Each module has minimal dependencies on others

6.3 Component Isolation in GUI

The GUI encapsulates functionality within isolated components:

- Each component manages its own state and rendering
- Communication happens through explicit props and contexts
- Components have single responsibilities (e.g., grid, cell, formula bar)

7 Justification of the Design

7.1 Memory Efficiency

Several design decisions were made to optimize memory usage:

- **BitFields for Flags**: Using the `CommandFlag` bitfield structure to pack 16 flags into 2 bytes
- **Integer Encoding**: Encoding cell references as integers instead of strings
- **Dependency Vectors**: Using `Vec<usize>` for dependencies instead of more complex structures

7.2 Performance Considerations

Performance was a key factor in our design:

- **Topological Sorting**: Ensures efficient update propagation with minimal recalculations
- **Cell Reference Encoding**: Fast lookups through integer-based cell references
- **Dependency Tracking**: Direct tracking of forward dependencies for quick propagation

```
fn toposort(&self, node: usize) -> Vec<usize> {
    let mut visited = vec![false; (self.row + 1) * (self.col + 1)];
    let mut temp = vec![false; (self.row + 1) * (self.col + 1)];
    let mut result: Vec<usize> = Vec::new();

    if self.is_cyclic_util(node, &mut visited, &mut temp, &mut result) {
        return Vec::new(); // Detected cycle
    }

    result.reverse();
    result
}
```

Listing 11: Topological sorting for efficient updates

7.3 Separation of Concerns

The clear separation between core engine and UI provides several benefits:

- **Testability**: Core logic can be tested independently of UI
- **Maintainability**: Changes to UI don't affect core logic and vice versa
- **Flexibility**: Support for both CLI and GUI without code duplication

7.4 Error Handling Strategy

Our error handling design provides good user experience:

- **Error Types**: Different error types (DivByZero, CycleDetected, InvalidInput)
- **Error Propagation**: Errors properly propagate through cell dependencies
- **UI Feedback**: Clear error messages in both GUI and CLI

8 Design Modifications

8.1 Dependency Tracking Optimization

We made significant improvements in dependency tracking:

- **Vector vs. HashSet**: Changed from HashSet to Vec for dependency storage
- **Memory Layout**: Improved cache locality by using contiguous memory in Vec
- **Performance Impact**: Faster initialization and iteration over dependencies

```
// Before: Using HashSet
// depend_vec.insert(row * ENCODE_SHIFT + col);

// After: Using Vec with containment check
let t = row * ENCODE_SHIFT + col;
if !depend_vec.contains(&t) {
    depend_vec.push(t);
}
```

Listing 12: Optimized dependency tracking with Vec

8.2 Graph Generation Extensions

The graphing capabilities were extended to support:

- Multiple chart types (line, bar, pie, scatter)
- Custom labels and titles
- Different data range selections

8.3 User Interface Enhancements

The UI underwent several improvements:

- Responsive grid that adjusts to window size
- Navigation controls for large spreadsheets
- Error display
- Formula bar with interactive editing