Graphsheet and The Minicell System

Nima Johartizadeh

November 23, 2020

We introduce a general framework to embed domain-specific languages inside the spreadsheet grid. Our approach relies on supporting new data-types inside cells, which provides the opportunity to extend the spreadsheet interpreter with domain-specific formulas. As one concrete example, we apply our framework to design a domain specific language for graph modeling. Our framework is applicable to a wide range of domains and amplifies the favorable properties of spreadsheets.

Summary of Key Accomplishments

- We developed a general conceptual framework ("The Cell-Centric Augmentation of The Spreadsheet Grid") that characterizes a family of domain-specific languages based on rich types inside spreadsheet cells.
- We implemented Minicell, a spreadsheet system with rich types inside cells,
- We implemented Graphsheet, a cell-centric domain-specific language for graph querying and network optimization problems
- We extended Graphsheet with GPU accelerated graph primitive for subgraph matching. In doing so we
  - Added Python support to Minicell's calc engine
  - Implemented a Python wrapper for Gunrock's subgraph matching
  - Defined a tabular syntax for sheet-defined query graphs
  - Implemented a basic support for spilling in Minicell

1 The Cell-Centric Augmentation

The VisiCalc (1983) system is famously known as the killer app for early Personal Computing. VisiCalc introduced the classic notion of spreadsheets, which is consisted of

- a tabular layout that makes it easy to *store raw data* on the grid,

- a simple programming model makes it easy to *express calculations* that operate on the raw data, and
- a calculation engine that ensures *changes to the grid are immediately visible* when a value is modified.

Spreadsheets are in wide use today. Existing systems such as Excel or Google Sheets both provide easy to use primitives to work with numerical values, dates, strings, and conditionals.

Cell-Centric Augmentation refers to an approach to extend the functionality of major spreadsheets while retaining their favorable properties. The key idea behind cell-centric augmentation is to enrich spreadsheet cells with new kind of values, and subsequently extend the set of formulas with new primitives.

A cell-centric domain-specific language enriches the spreadsheet cell with a new datatype, and extends the formulas with a set of formulas centered around constructing, splitting, combining and transforming the newly introduced type.

1.1 Prior work on rich types inside cells

Smalltalk objects inside cells

Only 3 years after invention of spreadsheets, a system [15] emerging from Xerox (1986) leveraged the Smalltalk environment and allowed arbitrary Smalltalk objects inside cells. This system is distinct because

- It is the first system to suggest arbitrarily rich types inside cells.
- It is the first system to provide a mechanism for user-defined types. (through Smalltalk)
- It leveraged the underlying abstractions of Smalltalk in its implementation which reduced the complexity of the system. According to the author, it took only 6 person-months to deliver a functioning system with a polished UI and and extensible calc engine.
- To provide extensibility, it used general purpose language with a vibrant ecosystem as a "escape hatch".

Since then, multiple works have focused on designing domain-specific tools based on rich types inside cells.

Images inside cells

Spreadsheet for Images (1999) [11] is an interactive domain-specific language to combine and manipulate images stored in spreadsheet cells. With respect to this work, an interesting direction to explore is to extend Minicell with GPU-accelerated image manipulation primitives to leverage capabilities of modern hardware in real-time image processing.

Although existing major systems like Excel and Google Sheets display image values inside cells, the formula languages are deprived from image manipulation primitives.

Vectors, Matrices and Spreadsheets inside cells

Spreadsheets can accommodate one and two dimensional arrays in two ways. A *collapsed* value refers to storing and referencing the entire value using one cell. In contrast, *spilling* refers to the case where individual elements of the array spill out of the original cell and occupy adjacent cells.

The most ergonomic way to work with vectors seems to be spilling arrays. Surprisingly, support for collapsed version of these values is not eagerly demanded.

In the literature, one of the earliest mentions of collapsed matrices inside cells is sheet-defined functions [9].

Gridlets (higher-order spreadsheets) [8, 17] are centered around the idea of having a spreadsheet inside a cell of another spreadsheet, and spilling a portion of the source grid into the target grid. Gridlets provide a reuse mechanism for spreadsheets that is more robust that copy/paste.

Functions inside cells

Perhaps one of the most exciting developments in spreadsheets is the idea of having formulas that turn an ordinary cell into a callable value. This idea has been suggested in the literature for as early as 1997 [2]. Currently support for such capability is under careful consideration from Microsoft.

Record types inside cells

In Excel, json-like structures are now supported inside cells, mainly for the purpose of storing encyclopedia-style information about concepts. Examples include the nutritious facts about fruits, information about chemical molecules (molecular mass, formula, boiling point), cities (area, local time, weather, population) or biographical information about the current president of the United States. In 2020, Microsoft announced that this information is provided by Wolfram, therefore it is expected that the structure of the data is similar to the kind of information that is returned by Wolfram Alpha.

The problem of manipulating structured information in functional languages has been extensively studied, however providing an ergonomic way to manipulate structured data in typed functional languages remains to be a difficult problem. [1]

[1] For example, here is a quote from a functional programmer who is interested in learning lenses– the preferred way to work with structured data in Haskell:

> It is obviously a powerful tool, but this library deviates from many ideas that I learned to appreciate when I started learning Haskell: The types become more cryptic, it uses language features that are outside of native Haskell. It seems like overkill for something that was supposed to make it easier to work with

Although this problem is easier to tackle in dynamically typed programming languages, neither Excel nor Google Sheets attempt to ship formulas to manipulate structured data.

Streaming values inside cells

The current price for a stock symbol is an example of a value that is updated asynchronously. The number of seconds passed from the Unix epoch is another value that flows in time. Major spreadsheet environments are starting to support asynchronous values that automatically update the grid without human intervention, however for major environments, support is at its early stages. In the literature, the Gneiss system describes a spreadsheet model for handling streaming data [1]. Another work provides a semantics for streaming data in spreadsheets based on the notion of time windows [7]. The client/server architecture of Minicell accommodates a limited degree of streaming values as is described later.

User Defined Types

As of today, there is no ergonomic solution for user-defined types in major spreadsheets. Minicell does not provide a mechanism for dynamically defined user defined types.

Audio and Video inside cells

No existing system provides support for neither recording, playback nor manipulation of video and audio files. The current architecture of Minicell is not suitable for manipulating video and audio either. We discuss how architectural changes to Minicell can accommodate certain degree of interactive manipulation of video and audio content.

1.2 Graphsheet: A Cell-centric Domain Specific Language

In this section, we describe the design of Graphsheet, a cell-centric domain specific language for graph processing.

Existing spreadsheets provide poor support for tasks that require domain-specific support beyond simple date and number arithmetic. For example, in order to work with network models, an end-user programmer must *cast* graph problems into an integer linear formulation and apply Excel's "solver" functionality to find optimal solutions. Working with network models in this manner has major drawbacks:

- First, casting a graph problem to Integer Linear Programming (ILP) manually is nontrivial, prone to human error and is difficult to modify and debug.

records. It's also seems like learning a whole new language just to make sense of the library ecosystem.

- Second, the immediate visual feedback that makes what-if analysis possible is missing from Excel's solver feature.
- Finally, endless varieties of models are instances of graph problems. Not all such models have a straightforward ILP formulation.

In summary Excel's "solver" feature is general purpose, but brittle. Consequently current spreadsheet software poorly handles graph models.

Formula	Description
`=ORDER(G1)`	Number of nodes
`=SIZE(G1)`	Number of edges
`=OUTDEGREE(G1, "davis")`	Computes the out-degree of a vertex
`=SP(G1, "davis", "berkeley")`	Calculates the shortest path between two nodes
`=MF(G1, "s", "t")`	Calculates the maximum flow between source "s" and sink "t"
`=REACHABLE(G1, "davis", "zurich")`	Is `"Oakland"` reachable from `"Portland"`?
`=ISOMORPHIC(G1, G2)`	Are `G1` and `G2` isomorphic?
`=SUBGRAPH(G1, G2)`	Is `G1` a subgraph of `G2`?
$<=$	Alias of `SUBGRAPH`
$<$	Is `G1` a strict subgraph of `G2`?
$>$	Is `G1` a strict supergraph of `G2`?
$>=$	Is `G1` a supergraph of `G2`?
$<>$	Is `G1` different than `G2`?
`G1` = `G2`	Is `G1` equal to `G2`?

Table 1: Interaction of graphs with existing types

A shortest path problem

Consider the spreadsheet in Table 2, which encodes distances between 5 cities in Northern California in `A2 : F6`. We are interested in finding the shortest path between two cities specified in `C1` and `D1`.

	A	B	C	D	E	F
1	`=X(A2:F6)` ⇓ ☆	`=SP(A1,C1,D1)` ⇓ 470	portland	oakland		
2		sacramento	berkeley	portland	oakland	davis
3	Davis	20	90			
4	Berkeley				20	
5	Portland					360
6	Sacramento					

Table 2: Spreadsheet data for distance between 5 cities, along with solution to the shortest path problem in row 1, where `A1=X(A2:F6)`. contains the graph corresponding to `A2:F6`, and `B1=SP(A1,C1,D1)`.

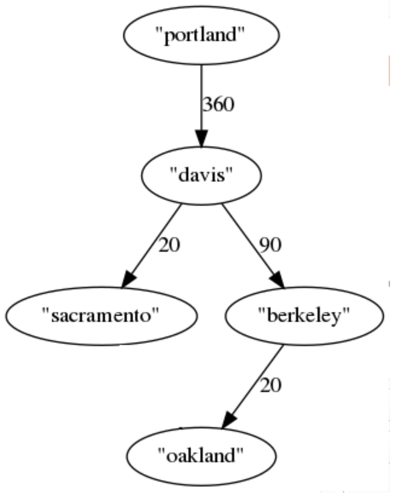


Figure 1: Graph from table 2 stored in `A1` (☆).

Cells on the rectangular region `A2` : `F6` store distance information between cities encoded as incidence matrix of a directed graph.

In this table, the vertical range `D3` : `D6` and the horizontal range `B2` : `F2` store nodes of our graph.

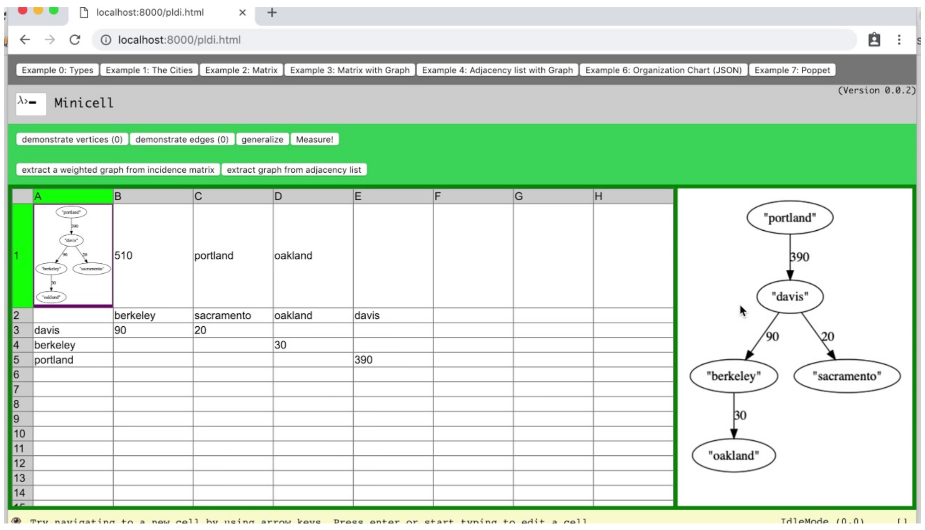
Each number inside `B3` : `F6` represents an edge going out from the node in the header column into the node in the header row.

For example `C2` means it takes 20 minutes to get from `Davis` to `Sacramento`. To calculate the shortest path between `Portland` and `Oakland` we need to

- Interpret `A2` : `F6` as a graph value, then
- Run Dijkstra's shortest path algorithm on the wrangled graph

To accomplish this, we

- Enter `=X(A2 : F6)` in `A1` to *construct* a graph value,
- Enter `=SP(A1, C1, D1)` in `B1` to calculate the length of the shortest path.



	A	B	C	D	E	F	G	H
1		510	portland	oakland				
2		berkeley	sacramento	oakland	davis			
3	davis	90	20					
4	berkeley			30				
5	portland				390			
6								
7								
8								
9								
10								
11								
12								
13								
14								

Try navigating to a new cell by using arrow keys. Press enter or start typing to edit a cell. IdleMode (0,0) []

Figure 2: Screenshot of Graphsheet system

2 Implementation of Minicell and Design Decisions

This section describes the philosophy and implementation of Minicell, a spreadsheet system with new kind of values inside cells. By enriching cells with new kind of values, Minicell hosts a family of domain-specific languages on the spreadsheet grid.

We have made the following design decisions in implementing Minicell:

1. **A client/server architecture** Minicell grew out of an Elm [3] implementation executing purely inside the web browser. However relying exclusively on the web platform is limiting, specially for many kinds of IO intensive applications.

   As opposed to Desktop spreadsheet applications, Minicell relies on a central server for computation. In this aspect, Minicell is closer to Google Sheets as opposed to Desktop Excel.

   The client and the server encode and decode Minicell values from and to JSON. The Minicell client can render values from any server as long as the JSON protocol is implemented.

2. **Caching of IO intensive operations** A `Data.Map EExpr EExpr` assists the evaluator to short-circuit long-waiting IO operations like HTTP fetch.

3. **Single-threaded blocking evaluator**

4. **Pull-based instead of push-based interface updates** The Minicell frontend sits on top of the Elm architecture and is able to re-render the sheet quite efficiently. However, the frontend uses a pulling model to periodically fetch values from the server.

5. **Non-minimal recalculation of values** The Haskell backend does not construct a calculation chain and does not implicitly memoize result of vales or subexpressions.

A Haskell backend stores the spreadsheet in memory and serves the Elm client via a JSON API. Both the frontend and Backend are able to serialize and deserialize Minicell values to and form JSON.

For example, to display shapes inside cells, Minicell translates spreadsheet formulas and uses Haskell's `diagrams` package to write files on disk. Then it serves the file through an HTTP endpoint, and shares the URI with the Elm frontend.

This client/server architecture enables Minicell to compute and cache expensive computations on the server, while making it easy to display a lightweight representation of those values in the frontend.

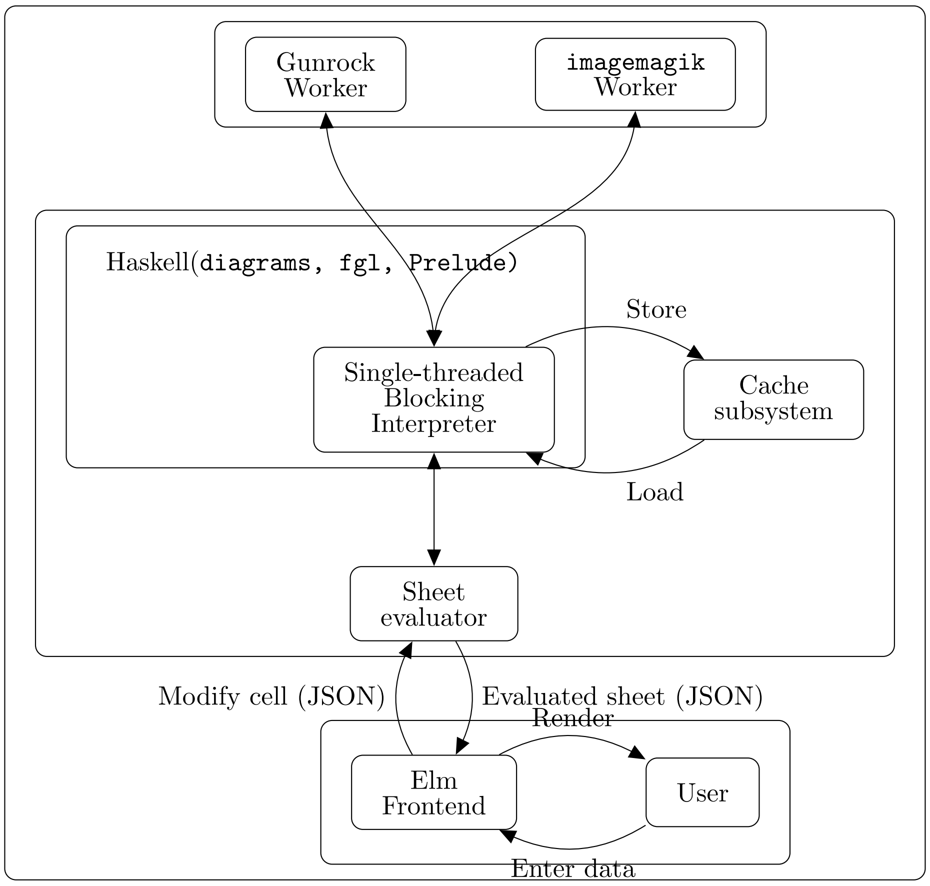
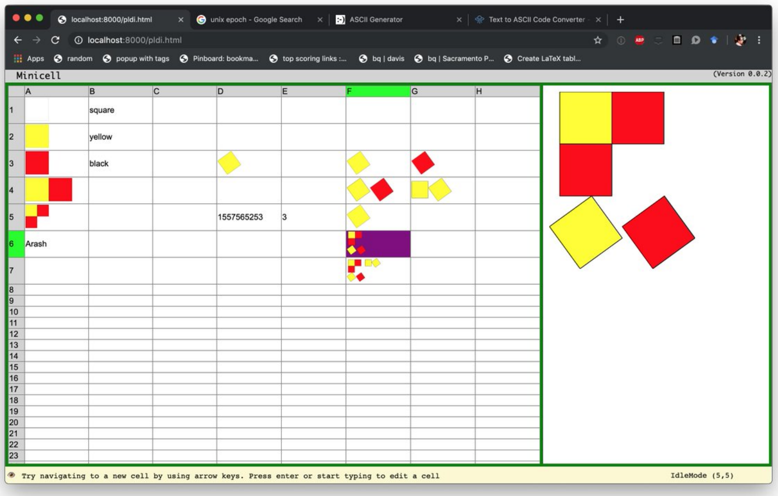


Figure 3: Architectural diagram of Minicell

2.1 Values that flow in time

Functional Reactive Programming (FRP) [5] suggests a calculus for combining and manipulating temporal values. The key idea behind FRP is to simply represent a temporal α-value as a function f of type $t \to \alpha$, where t represents the time domain, and α is an arbitrary type. The rich calculus provided by FRP was originally used for creating animations.



Try navigating to a new cell by using arrow keys. Press enter or start typing to edit a cell

IdleMode (5,5)

Figure 4: Example of animated graphics generated via Minicell

Temporal values in Minicell are not FRP. In particular, the notion of time in Minicell is discrete (not continuous) and Minicell combinators for temporal values do not follow a discipline of precise and simple denotation. However, we have borrowed the idea of explicitly taking time as an input parameter. Through this mechanism we construct a temporal value and provide immediate visual feedback to the programmer.

Because of the pull-based model, providing support for temporal values in the server is trivial. We simply get the current time inside `evalIO`:

```
evalIO :: Model-> EExpr -> IO EExpr
evalIO model expr = case expr of
    --- ...
    EApp "UNIXEPOCH" _ -> do
        t <- getPOSIXTime
        return $ EILit (round t)
    --- ...
```

At the frontend, the Elm architecture greatly simplifies pulling and rendering values from the backend. In short, the pulling works by adding a timer in the *subscriptions* of our Elm application. This will dispatch values such as `Tick 1605318162` and `Tick 1605318173` to the `update` method.

In Elm, the `update : Cmd.msg -> Model -> (Model, Cmd.msg)` method is the main mechanism to receive messages from external triggers (like time). The `update` method may also ask the runtime to start new IO actions.

In `(Model, Cmd.msg)`, the first element contains the new model and the second element are further IO actions that the application may want to start.

If the `update` call demands no further IO action, it sets the second element in the pair as `Cmd.none`.

```
type Msg
    = Tick Time.Posix
    | NextEvalCycle (Result Http.Error E.Value)

update msg model =
    case msg of
        -- ...
        Tick t ->
            case model.mode of
                EditMode _ -> (model, Cmd.none)
                _ -> ( { model | currentTime = t }, pullHttp model)
        -- ...
        NextEvalCycle res ->
            -- If res is successful,
            -- update the local hashmap with new values from res
        -- ...

pullHttp : Cmd Msg
pullHttp =
  Http.get
    { url = "http://.../sheet0/read.json"
    , expect = Http.expectJson (NextEvalCycle) (D.value)
    }
```

Finally, we instruct Elm to dispatch the `Tick` action twice a second:

```
subscriptions : Model -> Sub Msg
subscriptions model = Sub.batch [ Time.every 500 Tick ]
```

3 Future directions

3.1 Limitations of a single-threaded blocking interpreter

Minicell differs from Excel and Google Sheets mainly because it considers to extend the calc engine with IO-bonud operations. The aim of the interface is to perform independent IO operations in parallel, and communicate the progress of operations via the user interface. Our current architecture is not suitable for this class of operations. For example basic operations on images, audio and video are computationally demanding. Current Minicell architecture is not suitable for interactive manipulation of such data types, and the interpreter evaluates cells sequentially.

Switching away from a single-threaded blocking interpreter can accommodate long running compute-bound (image manipulation) or IO-bound operations (HTTP retrieval) with a reasonable interactive experience on a Desktop machine.

Here, the choice for the next step is between

- A multithreaded interpreter with blocking semantics
- A single-threaded interpreter with non-blocking semantics
- A multithreaded interpreter with nonblocking semantics

For example, consider a simple sheet that applies two different color effects on the same image. For example, let

```
A1=IMAGE("teapot.jpg");
B1=BW(A1);
B2=ROTATE(A1, 90);
```

To reduce latency, it is desirable to evaluate `B1` and `B2` in parallel and on different threads.

Furthermore, the blocking IO model is not suitable for rapid reevaluation of the grid. In a highly interactive application like a spreadsheet, it is desirable to inform the user with intermediate values as soon as they are available.

Consider

```
A1=HTTP("https://.../large-file.csv");
B1=LEN(A1);
B2="Hello world!";
```

Here it is desirable to display `B2`, even though it might not be able to display the value of `B1` because `A1` is not finished downloading.

It would be helpful to display some intermediate information while the IO action in `A1` is in progress. For example, the interface could display the following for the contents of `A1`:

```
%36 (889.05 MB of 2.3GiB) is downloaded at 2.0Mbps.
(ETA 13.17 minutes)
```

3.2 A missing component: IO Manager

Choosing the right concurrency primitives to implement a multithreaded and/or non-blocking calc engine is not trivial. However, we suspect we can decouple the complexity of IO operations away from the complexity of the calc engine.

The IO Manager will have three main purposes:

- The IO Manager will do the bookkeeping of progress of tasks (how many bytes the HTTP request has already received, what was the output of that shell command, etc.). This also includes caching result of IO operations in a lookup table.
- The IO Manager will respond to the queries of the calc engine, and respond with an `EIncomplete "reason"` if the IO is incomplete.
- The IO Manager will orchestrate a thread pool or process pool of workers that perform blocking or non-blocking IO.

This way, the calc engine can make synchronous queries to the IO manager to *enqueue* a new IO task, to *retrieve the final value*, or *query the progress of the IO operation*, without caring too much about how to orchestrate IO operations.

It is not clear what concurrency primitives we must pick so that the interpreter and the IO manager can talk to each other as separate components residing on a single machine.

In absence of the IO manager, the most immediate way to parallelize IO operations is to carefully launch threads while considering the dependency among cells, and launch the *eval* function in each thread. Ideally we'd like to communicate the results to the frontend as immediate as possible. Therefore each separate thread needs to be able to communicate the calculated results of the portion of the grid that it is responsible for.

It seems like the IO manager will subsume the functionality of the existing caching subsystem, and will furthermore take over the role of communicating between the evaluator and the external workers.

3.2.1 Memoization, Parallel Fetch and Recomputation

The problem of augmenting an interpreter with a caching system shows itself in the literature in different domains. This issue touches multiple areas, including concurrency, language primitives, build systems and runtime system implementation.

Depending on requirements and the trade offs, one can pick different concurrency, memoization, dependency tracking and recalculation strategies. From the perspective of build systems, these choices has been elegantly studied in [13].

Haxl [12] provides an abstraction for efficient, concurrent and concise data access. A cache subsystem based on Haxl will be able to memoize IO operations. In addition, it can manage the degree in which tasks are parallelized and batched.

Naiad [14], originally implemented in C$^\sharp$, now in Rust, provides abstractions to construct dataflows and obtaining the output in an incremental way. Naiad abstractions can provide fast answers to queries on large datasets despite changes to original data.

Abstractions such as Haxl, Naiad or Rust's `salsa` library are good candidates to empower a spreadsheet's calc engine. However no spreadsheet system is directly using either of these abstractions.

Different spreadsheet systems prioritize different use cases. Since calc engine is a complex piece of software, different spreadsheet systems provide a unique set of capabilities based on the trade-offs made in the design of their calc engine.

In Minicell, we are taking a simpler approach. We are explicit and selective towards which operations would need to be cached. We make this decision in `evalIO` and we do not persist the cached values on disk. Currently, we are not able to continue with the evaluation of the sheet. Instead we must wait until the IO operation is complete.

The IO manager can cooperate with `evalIO` to schedule long-running jobs and quick response for queries on the progress of the job. This way, the evaluation will not need to freeze until the IO operation is complete.

3.3 Dynamically Extending Minicell types

Every newly introduced type needs to implement

- a way to convert to a `string`, and
- a way to convert to an image for display inside cells.

This way, types in the backend can grow indefinitely without any new implementation in the frontend.

For example, a `graph` type implements a way to convert the stored graph in memory into a rendered GraphViz diagram. Additionally, the backend is free to grow the formulas independent of the Elm frontend.

Adding a new type or a DSL to Minicell is easy, however Minicell does not have a mechanism to work with types that are defined at runtime. Extend Minicell types during runtime is highly desirable, however is not viable, and will require additional components working in conjunction with the evaluator component.

3.4 Candidate Workflows for Interactive Graph Analytics

In the literature, some existing works have pointed towards promising workflows. Several systems have addresses solutions for Interactive Anomaly Detection, Graph Pattern Matching and Ad-Hoc Querying on temporal graphs.

The following characterize the elements of an ideal workflow for interactive graph analytics:

- A large graph
- A fast graph engine that can answer to graph queries in order of milliseconds to seconds
- An *interactive workflow* in which a human analyst runs an algorithm, then interprets the results then runs another algorithm based on the previous results.

In order to build a system around these use cases, we need

1. A problem with an inherent interactive iteration loop

2. An implementation to provide rapid solutions to the spreadsheet
3. A large and labelled dataset that captures the problem

The following examples have some, but not all of the above properties.

Tegra is a system for Ad-Hoc Analytics on Time-Evolving Graphs. An example workflow from Tegra is a scenario in which a network engineer is assisted with a tool to find points in time in which a cellular network has been disrupted.

OJRank [10] is an algorithm for interactive anomaly detection. In order to re-rank anomalies, at each iteration it presents to the analyst the top anomaly. Within this loop, the analyst highlights or lowlights the top record. This information is used in the next iteration to re-rank and display the top anomaly based on previous feedback.

GraphUCB [4] is a method for Interactive Anomaly Detection on Attributed Networks that works similar to OJRank, however uses formulates the problem via a multi-armed bandit framework. The implementation of this algorithm needs careful engineering for working with million node graphs.

GraphQUBE (Querying By Example) tries to answer the following question: *"If I have an interesting pattern of behavior between some entities-of-interest, can I look through all my data to find other such examples of this type of behavior?"*

Another example is Graph Analysis for Detecting Fraud, Waste, and Abuse in Health-Care Data.

Finally, one interactive direction is querying networked and relational data. However we are only interested in tasks, queries and algorithms that exceed ordinary relational operations.

We have not explored the domain of problems that recompute answer on a dynamic graph, however these problems are of interest of systems such as Naiad [14] as well.

3.5 Gunrock

As a user of Gunrock APIs, we have examined how static graph analytics on million-node graphs via Gunrock APIs may cooperate with abstractions of an external REPL like Minicell.

Gunrock [16] provides fast and highly parallel implementation of nearly 30 graph algorithms. Combined together, these primitives can be pieces of a data pipeline for

- Link prediction (recommendation engines)
- Community Detection (graph partitioning, Louvain method)
- Anomaly detection (outlier detection, Scan Statistics)
- Graph pattern matching

Additionally, Gunrock has fast implementation for algorithms such as PageRank and HITS, various tree traversals (e.g. BFS), maximum flow and many others.

In presence of a suitable interactive workload, from the perspective of an external system, it would be desirable to interact with Gunrock APIs in the following ways:

- For subgraph matching, it would be useful to load the base graph only once inside GPU memory and dispatch many queries without unloading the base graph from the GPU memory.
- For subgraph matching, it might be worthwhile to accept string metadata for nodes and edges. This can make Gunrock suitable for workflows that need subgraph matching not just on topology, but also with conditionals on weights and labels of nodes and edges.
- Depending on the query graph, result of subgraph matching can be in the order of the size of original graph. What are some strategies to rank the results, and display a reasonable number of them to a human agent? What Gunrock algorithms can provide the ranking information? How can we customize the ranking function using Gunrock's C++ API?
- Allowing non-numeric information for subgraph matching can extend the range of queries. Similar to NetworkX's subgraph matching APIs, it is desirable to accept tests such as lambda functions that given a node (or an edge), they will return a boolean, determining whether the node (or edge) is desirable or not. The lambda should be able to access the metadata associated to the node or edge. It is not clear how Gunrock should manage this metadata in memory. Nor it is clear if we can decouple this into two phases if we utilized a filter-based preprocessing step. Providing support for such metadata will make it easy to answer queries that are convenient to express in SPARQL or Cypher [6].
- Before integrating Gunrock with an interactive interface it is important to have a concrete answer to the following:
  - How will the human analyst interact with the output computed by Gunrock? Is the interaction one-way, or is there a feedback loop with multiple cycles?
  - What real-world dataset is chosen? Is Gunrock capable of loading the dataset in the memory?
  - Is the result of the computation interpretable by a human analyst?
  - In what ways is the interaction different than spawning Gunrock's CLI, or computing the answer inside a REPL like Jupyer Notebooks?
  - Is the workflow characterized by one round of Gunrock compute, followed by interpretation by the human analyst, followed by another round of Gunrock compute, inspired by the results of the first round of calculations? In what way results from the first Gunrock call inform the analyst in running the second query?

4 Conclusion

We have implemented the Minicell system, an umbrella project that embodies the idea of cell-centric augmentations. To extend the range of tasks that can be tackled using spreadsheets, we have implemented Graphsheet on top of Minicell. Cell-centric augmentation amplifies favorable properties of spreadsheets, and is widely applicable to a variety of domains. We believe cell-centric augmentation is a promising direction towards extensibility of spreadsheets. The source code for Minicell is available on GitHub.

5 Appendix

5.1 Running Minicell

First, clone the Minicell repository from GitHub

```
git clone https://github.com/johari/minicell
```

Then run the following nix command to setup the Haskell environment with appropriate packages:

```
nix-shell -p "haskellPackages.ghcWithPackages (pkgs: with pkgs;
[fgl generic-random QuickCheck brick fgl-arbitrary hspec diagrams
palette z3 mysql-simple logict hslogger wai warp aeson
wai-websockets wai-extra wai-cors fgl-visualize graphviz dhall
wreq stache])"
```

You can run the server via

```
runhaskell src/SimpleServer.hs
```

To recompile the changes in the frontend, run

```
make build
```

The application is served at `http://localhost:6462`.

References

[1] Kerry Shih-Ping Chang and Brad A. Myers. A spreadsheet model for handling streaming data. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, page 3399–3402, New York, NY, USA, 2015. Association for Computing Machinery.

[2] Chris Clack and Lee Braine. Object-oriented functional spreadsheets. In *Proc. 10th Glasgow Workshop on Functional Programming (GlaFP'97*, 1997.

[3] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for guis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 411–422, New York, NY, USA, 2013. Association for Computing Machinery.

[4] Kaize Ding, Jundong Li, and Huan Liu. Interactive anomaly detection on attributed networks. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, WSDM '19, page 357–365, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, page 263–273, New York, NY, USA, 1997. Association for Computing Machinery.

[6] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1433–1445, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Martin Hirzel, Rodric Rabbah, Philippe Suter, Olivier Tardieu, and Mandana Vaziri. Spreadsheets for stream processing with unbounded windows and partitions. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*, DEBS '16, page 49–60, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Nima Joharizadeh, Advait Sarkar, Andrew D. Gordon, and Jack Williams. Gridlets: Reusing spreadsheet grids. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI EA '20, page 1–7, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A user-centred approach to functions in excel. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 165–176, New York, NY, USA, 2003. ACM.

[10] Hemank Lamba and Leman Akoglu. Learning on-the-job to re-rank anomalies from top-1 feedback. In *Proceedings of the 2019 SIAM International Conference on Data Mining*, pages 612–620. SIAM, 2019.

[11] Marc Levoy. Spreadsheets for images. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, page 139–146, New York, NY, USA, 1994. Association for Computing Machinery.

[12] Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. There is no fork: An abstraction for efficient, concurrent, and concise data access. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 325–337, New York, NY, USA, 2014. Association for Computing Machinery.

[13] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. Build systems à la carte. *Proc. ACM Program. Lang.*, 2(ICFP), July 2018.

[14] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[15] Kurt W. Piersol. Object-oriented spreadsheets: The analytic spreadsheet package. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '86, page 385–390, New York, NY, USA, 1986. Association for Computing Machinery.

[16] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, New York, NY, USA, 2016. Association for Computing Machinery.

[17] Jack Williams, Nima Joharizadeh, Andrew D. Gordon, and Advait Sarkar. Higher-order spreadsheets with spilled arrays. In Peter Müller, editor, *Programming Languages and Systems*, pages 743–769, Cham, 2020. Springer International Publishing.