Functional Programming in Financial Markets (Experience Report)

ATZE DIJKSTRA and JOSÉ PEDRO MAGALHÃES, Standard Chartered Bank, UK
PIERRE NÉRON, Standard Chartered Bank, SG

We present a case-study of using functional programming in the real world at a very large scale. At Standard Chartered Bank, Haskell is used in a core software library supporting the entire Markets division – a business line with 3 billion USD operating income in 2023. Typed functional programming is used across the entire tech stack, including foundational APIs and CLIs for deal valuation and risk analysis, server-side components for long-running batches or sub-second RESTful services, and end-user GUIs. Thousands of users across Markets interact with software built using functional programming, and over one hundred write functional code.

In this experience report we focus on how we leverage functional programming to orchestrate type-driven large-scale pricing workflows. The same API can be used to price one trade locally, or millions of trades across thousands of cloud nodes. Different parts of the computation can be run and inspected individually, and recomputing one part triggers recalculation of the dependent parts only. We build upon decades of research and experience in the functional programming community, relying on concepts such as monads, lenses, datatype generics, and closure serialisation. We conclude that the use of functional programming is one of the main drivers of the success of our project, and we see no significant downsides from it.

CCS Concepts: • **Software and its engineering** → **Functional languages**; • **Applied computing** → *Electronic commerce*.

Additional Key Words and Phrases: functional programming, Haskell, finance, experience report

ACM Reference Format:
Atze Dijkstra, José Pedro Magalhães, and Pierre Néron. 2024. Functional Programming in Financial Markets (Experience Report). *Proc. ACM Program. Lang.* 8, ICFP, Article 244 (August 2024), 15 pages. https://doi.org/10.1145/3674633

1 Introduction

Functional programming has a long and broad history of use in finance, even if this is not widely documented. Clancy [3] describes the use of functional programming languages such as Haskell and OCaml at places like Barclays and Jane Street for at least a decade. The seminal work of Peyton Jones et al. [9] demonstrated how financial contracts can be suitably encoded and reasoned about functionally, but arguably the main functional programming tool used in finance is *the spreadsheet*. As long as one is ready to accept that spreadsheets are programs [7] – and, moreover, functional programs [8] – there is no lack of evidence of their widespread use across finance.

The use of functional programming at Standard Chartered Bank involves spreadsheets whether one chooses to accept them as a functional programming language or not. In 2008, a proprietary functional language called *Lambda* was used as a "bridge" between existing C++ code and Microsoft Excel spreadsheets as an add-in.[1] Lambda was designed with a pure interface in mind, and supported higher-order functions and parametric polymorphism (although these were clunky to use due to

[1] https://learn.microsoft.com/en-us/office/dev/add-ins/excel/excel-add-ins-overview

Authors' Contact Information: Atze Dijkstra, Atze.Dijkstra@sc.com; José Pedro Magalhães, JosePedro.Magalhaes@sc.com, Standard Chartered Bank, London, UK; Pierre Néron, PierreJeanMichel.Neron@sc.com, Standard Chartered Bank, Singapore, SG.

lack of any special syntax in Excel or currying). At that same time, Haskell also started being used to develop auxiliary tooling. By 2009 Lambda had significant usage by end-users, also outside Excel with a standalone interpreter. Its lack of a module system and poor type-checking performance prevented wider adoption; a full-fledged programming language was needed. However, we did not want to lose the advantages of Lambda, which included easy C++ and Excel integration and the ability to serialise all values, which greatly simplified cloud computation. Cloud Haskell [5] was not available at that time, and we did not have the resources to build a custom GHC back end to target our existing run-time. Instead we decided to develop a dialect of Haskell that would mesh well with the rest of our ecosystem, and we named it *Mu*.

```
main :: IO ()
main = do
  tds <- io $ TDS.connectService Nothing
  ts  <- io Date.now

  trades <- io $ TDS.getTrades tds [ TDS.PortfolioId `TDS.Elem` ["A","B"]
                                   , TDS.Status `TDS.Elem` ["Live"] ]
  (QR.Error sErr, QR.Result res) <- QR.runShepherdQR QR.defaultShepherd ts
                                      [QR.inputTrades trades]
                                      [QR.strategy QR.defAquila
                                      ,QR.riskList Nothing [IR.Delta.risk]]
  Core.saveValue "IR.Delta.dat" res

  sendAny def { to      = ["recipient@example.com"]
              , from    = "sender@example.com"
              , subject = "IR Delta Risk"
              , body    = unlines [ "Number of trades considered: "
                                      & show (A.size trades)
                                  , "Number of errors: " & show (A.size sErr)
                                  , "IR delta attached." ]
              , attachments = ["IR.Delta.dat"] }
```

Listing 1. Computing interest rate delta and emailing results in Mu

Mu is often indistinguishable from Haskell by looking at syntax alone, but it comes loaded with a powerful financial analytics library and many useful utilities for working with data at scale. Listing 1 shows how simple it is to fetch (potentially hundreds of thousands of) live trades, compute their interest rate (IR) delta via a cloud computing provider, and email the result – all in a few lines of Mu.

The rest of this paper shows how we use Mu to simplify financial analytics workflows using functional principles at all stages, from architecture to implementation. Section 2 introduces our domain and the Mu compiler in more detail. Section 3 showcases QuickRisk, a type-driven API to a pricing library, while in Section 4 we present a few other important components briefly. We discuss interoperability with other programming languages in Section 5 before concluding in Section 6.

2 Context

In this section we provide some context needed to understand the rest of the paper.

2.1 Financial Markets

The work described in this paper is built by the Modelling and Analytics Group (MAG), which is part of the Markets division of Corporate and Investment Banking of Standard Chartered Bank. This

part of banking typically deals with large volume and/or large scale financial transactions between large corporate clients and other financial institutions [2]. The types of financial transactions vary in complexity and domain. One simple example is a foreign exchange (FX) spot transaction (exchanging one currency by another at the prevailing market rate). A complex example would be an airline wishing to hedge the cost of aircraft fuel over the next six months in a given foreign currency.

In order to be able to quote prices for these kind of transactions and to manage exposure to market risks, large banks typically employ a Quantitative Analytics function that is responsible for developing proprietary financial models. MAG operates in this space, as part of Front Office for frequent and direct interaction with trading and structuring functions. Next to the financial mathematics aspects, our group is also responsible for delivering end-to-end solutions relying on quantitative models, such as server-side components for intraday risk analysis of prospective deals, visualisation tools for complex financial data, and daily bank-wide price and sensitivity computation. This requires building robust software that can perform common tasks across varied inputs in a generic fashion, while catering for (and abstracting from) the millions of individual details in market conventions and prices.

2.2 Cortex

Cortex is the name of the entire software ecosystem developed by MAG for financial analytics in Markets. It includes:

- Various libraries for quantitative financial modelling
- The Mu compiler
- Interfaces to Excel, C++, .NET, Java, and Python
- Dozens of end-user GUI applications
- The code for several production server-side components, such as an intraday risk service

Its codebase consists of Mu code (over 7 million lines), C++ (under 2 million lines), Haskell (around 500 thousand lines), and some F# for the Windows GUI library. Figure 1 shows the evolution of the number of lines of code written in each programming language since the project inception in 2008.

Cortex is built and released automatically on a daily basis via a continuous integration system, and then used directly by end-users or by other teams who integrate it into their own solutions. We have our own build and test infrastructure, and all code changes are run against over 5000 tests (in a mix of unit and property-based testing, covering compiler correctness as well as financial analytics). We also have a custom mechanism for assigning ownership of parts of the codebase to specific groups of users, and tracking review approvals on changes outside the ownership (so anyone can propose any change, but different changes by different users may require different approvals).

2.3 Mu

Mu is the programming language of choice for all development in Cortex. At the surface, Mu looks extremely similar to Haskell; one often cannot tell by simply looking at source code, as Listing 1 reveals. However, looking deeper we find several important differences.

2.3.1 Runtime and semantics. Mu code is compiled to bytecode with a small runtime interpreter. The runtime is strict, but some constructs have lazy operational semantics. This might seem strange, but a compromise is needed, otherwise Mu would be *too different* from Haskell. Haskell programmers expect expressions like `fromMaybe undefined (Just ())` to just work, but these would fail in a fully strict setting, since `undefined` would be evaluated before `fromMaybe`. We use compiler

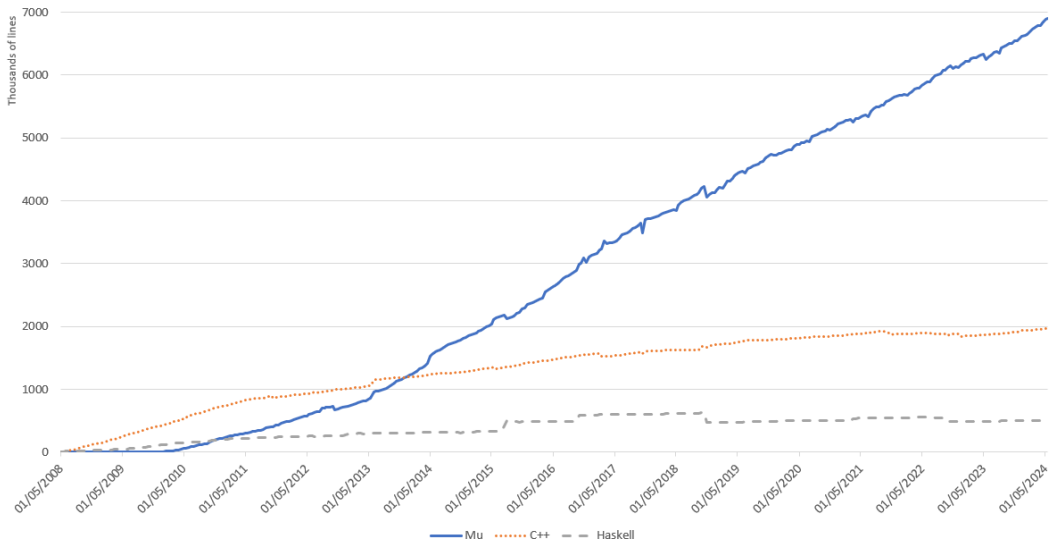

Fig. 1. Lines of code over time

pragmas to introduce laziness in crucial places so that programmers do not need to think (much) about evaluation order when writing code.

The Mu compiler also uses lazy semantics to do aggressive code motion for performance optimisation, such as let-clauses lifting, inlining, and common sub-expression identification and lifting. The worst that can happen is that errors may happen in a different order at runtime (or not at all), compared to a fully strict implementation.

In general, the choices taken in Mu semantics are informed by theory but driven by pragmatism. Mu does not have formal semantics and this is normally not a concern in practice. We have a large monorepo and we occasionally make small tweaks to semantics together with code rewrites in a single change request. The foundations of the compiler have not changed much over time, and they are not the main focus of our work as a financial institution.

2.3.2 Foreign-function interface. There is no FFI: all C++ functions are called natively, thanks to a unified C++ `IFunction` interface which both C++ and Mu functions adhere to. C++ can apply any `IFunction` natively, so we get seamless calling across languages. All C++ functions of interest are assigned a globally unique identifier which the runtime system uses for finding the compiled object code. This is essential for cross-platform and cross-version code execution.

2.3.3 Recursion. Direct recursion is discouraged (and disabled by default) due to lack of tail-call optimisation, which is not easy to implement when recursive calls can freely cross between C++ and Mu. We do have tail-recursion specialised to one operation, and use it to define other recursive primitives: `loopM :: Monad m =>(a -> m (Either a b)) -> a -> m b`.

A positive side-effect of this limitation is that it encourages programmers to use recursion combinators. We also find that the need for explicit recursion arises infrequently in our setting: it is used by under 4% of the modules.

2.3.4 Data structures. Strings are a native datatype in Mu, not a list of characters. They are implemented as UTF-8 C++ `std:string`. Mu does not have a `Char` type.

Lists are not the natural choice for collections of data; we prefer relations [1], even though our lists are implemented as arrays. Relations are (abstract) first class objects with a pure API. They can be pictured as a tabular structure with named and typed columns followed by rows of data. There is no material ordering of either columns or rows. We have an efficient implementation of the relational operators with well-defined algebraic properties, which makes it easy to reason about and helps with refactoring. We regularly work with relations containing millions of rows and over one hundred columns.

2.3.5 Memory management. We have a C++ implementation of records and use it for representing Mu datatypes. Regular C++ reference counting-based memory management applies to these. Data created via GHC libraries is managed by GHC's own garbage collector within its runtime system. The lack of a unified view on memory use and its connection to the original source code can make it sometimes challenging to track identifiers when debugging.

2.3.6 Serialisation. (De-)serialisation is supported by default for mostly everything in Mu. This makes it easy to save any partially complete computation and restart it on another machine, even across operating-system and architecture boundaries. We rely on full (de-)serialisation heavily for parallel computing.

2.3.7 Extensions. Mu does not support all of GHC's extensions. Notably, it lacks existential quantification, GADTs, and rank-2 (and higher) types – but it supports multi-parameter type classes, functional dependencies, and poly and data kinds. We find that the current set of extensions provides a reasonable balance between expressiveness and compiler complexity, although having GADTs would be useful.

2.3.8 Separate compilation. Mu is a whole-program compiler, so the entire source of all libraries is used in each compilation. This has some advantages: there is no need for runtime dictionaries for class instances, and all function calls are monomorphised at their usage sites.

It does also come with some challenges. For example, a typical program can have thousands of transitive dependencies, and compilation times of 5 minutes are frequent. Our largest applications can take 10 minutes to compile. Long compilation times, in turn, affect some program design decisions. Sometimes we analyse module dependencies and separate type declarations from functionality, to minimise the amount of code brought into scope – but this is a good practice anyway. In select cases we resort to dynamic module imports; these happen at runtime, triggering compilation of the module being imported if this wasn't done in advance, and leading to potential runtime type errors (which would have otherwise been compile-time errors).

In terms of program size, one of our largest applications consists of 850,000 lines of source code from 4,000 modules, and compiles to a core language expression with 1,000,000 nodes (after optimisation, all figures approximate).

2.3.9 `SafeIO`. Many common effectful Haskell functions live in the `SafeIO` monad in Mu, rather than `IO`. We use `SafeIO` to distinguish "read-only" effects; for example, we have the function `putStrLn :: String -> IO ()` but `readFile :: String -> SafeIO String`. Such operations can be lifted with `io :: SafeIO a -> IO a`, but we provide a `perform :: SafeIO a -> a` function with fewer caveats than the infamous `unsafePerformIO`.

Despite these differences, we are currently working on replacing the Mu front end with GHC [6]. This involves using GHC to compile Mu to GHC Core, translating Core to the current Mu compiler intermediate representation, and then using the rest of the existing pipeline.

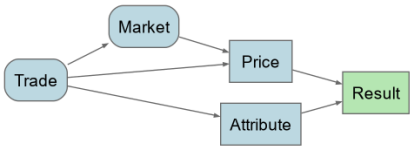

Fig. 2. Simplified QuickRisk workflow

Fig. 3. Minimal pricing workflow

3 QuickRisk

In this section we present *QuickRisk*, a type-driven API to our main pricing library, and also its underlying *Work* API and the *Shepherd* job scheduler it is typically used with. QuickRisk allows us to generate and execute type-driven workflows that depend on readily available inputs and outputs requested by the caller. Its goals are:

- To provide a single API allowing different kinds of pricing and other derived computations to be performed in a consistent sequence;
- To have a type-driven workflow generation, that is, the ability to generate the minimal workflow required to compute the requested outputs;
- To separate the concerns of workflow generation and its execution (e.g. sequentially in process, in parallel, or using a job scheduler).

In this paper we present a simplified version of the API which relies on two sources of data. One source is trade data, i.e. a representation of financial contracts as described by Peyton Jones et al. [9]. We use a variant of this representation of financial contracts, which is particularly useful for implementing "generic" operations across all trade types, but most analytics are implemented per trade type. The other source is market data, either observed or predicted values (e.g. the exchange rate between two currencies). With these two sources of data, QuickRisk allows us to compute different kinds of financial analyses in separate stages with inter-dependencies, and to extract the final results we are interested in. As we cannot make the library itself available for use outside our company, we focus on highlighting aspects where we think functional programming insights were valuable to us, and could be valuable to others as well. Code listings referred in this section can be found in the appendix.

Figure 2 illustrates the computation graph for the simplified QuickRisk workflow we present in this section; Figure 6 shows a complete QuickRisk workflow, with many details that we elide for conciseness. Nodes correspond to the different stages of computation. In the simplified QuickRisk workflows, rounded nodes use IO while the rectangular ones depict pure computations.

- `Trade` produces the portfolio (i.e. a trade set). Depending on input parameters these can be read from a file or queried from a database.
- `Market` produces the market data. It depends on `Trade` as we do not want to fetch all possible market data but only that required to price the current portfolio.
- `Price` computes the price of the trades using the trade and market data, calling trade-specific quantitative pricing algorithms.
- `Attribute` extracts some specific information about the trades, like the total notional of the trade or the currencies it uses. These are often useful for grouping or aggregating the final result.
- `Result` provides a unified view of the data computed in both the `Price` and `Attribute` stages, e.g. total value per currency.

Before going into more details about QuickRisk we first describe the *Work API* that is used to define and execute (acyclic) graphs of computations in Mu.

3.1 The Work API

The main goal of the *Work API* is to separate the concerns between the definition of a sequence of inter-dependent computations (represented as a DAG) and the execution of these computations. In this API, a node (or `Work`) represents a single unit of computation, and it can be either:

- A constant (already computed) value;
- The work of applying a function to another `Work` item, and thus creating a dependency between these;
- The lifting of an IO computation.

In the absence of existential quantification in Mu, we make use of two specific types, `Any` and `Fun`, which are used to embed values (resp. functions) of any type to represent untyped work units (Listing 2). This `Work_` ADT is not exposed to users but forms the basis of a typed interface (Listing 3), mimicking the typed ADT we could write directly if Mu had existential quantification.

In order to execute `Works` which share common dependencies, we first define `WorkPool` as a set of `Work` representing all the work nodes we are interested in running (using list as implementation for convenience). Sharing of nodes during execution is achieved using an identification mechanism on `Work` which associates each node with a unique `Key`. This avoids duplication of (sub-)work between the different elements of the `WorkPool` and allows retrieving results based on the input `Work` from the final computation result. These `WorkPool` can then be evaluated using different methods, such as:

- Sequentially, simply reducing the work graph in the current thread;
- In parallel, running each of the nodes in a different thread or process, up to some limit;
- Using Shepherd, our internal job scheduler introduced in Section 3.3.

The actual implementation of resolving a `WorkPool` into a `WorkResult` using each of the different methods is out of the scope of this paper.

3.2 QuickRisk Workflow

In this section we explain how QuickRisk uses the Work API to produce computation graphs that depend on the output type and the provided inputs (if any). For each run, QuickRisk generates a graph that only contains the work units required to generate the specified output, and uses the specified inputs to replace sub computations with constant work nodes. Figure 3 depicts an example where the `Trades` have already been fetched and are provided as input and only the `Price` output is required. This graph corresponds simply to the following QuickRisk call: `runQR [inputTrade t] :: SafeIO Price`.

Workflow construction. The first stage in building the computation graph for each call is the construction of the complete workflow. This is done by defining the different stages of computation (i.e. the nodes of the graph) and assigning a `Work` unit to each of them, which is either a constant value that is provided as input or a function to compute this node depending on other nodes or `IO`. We use a mapping from `Stage` to `Any` to represent the provided inputs and a similar mapping for `Works` (`Dict` is the type of dictionaries (HashMaps) in Mu, `D.value` is the total lookup function).

The graph construction uses the list of inputs that have been passed directly to the QuickRisk call to inject constant value nodes in the graph instead of functions or IO nodes that would normally be used to produce such stages (as in the `tradeWork` function). See Listing 4 for more details.

WorkPool Generation. The `WorkSpace` defined before allows us to map any QuickRisk stage to a corresponding unit of work where these work share dependencies to avoid duplicating computations. We now want to generate a minimal `WorkPool` containing only the nodes corresponding to specified output nodes and their dependencies. The QuickRisk API also lets users attach `IO` actions to specific

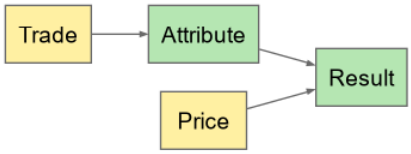

Fig. 4. QR API call

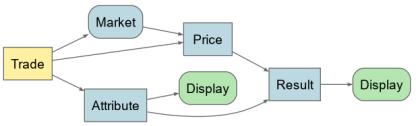

Fig. 5. QR CLI call

outputs (e.g. save to file, display, etc.) which leads to the generation of extra leaf nodes corresponding to these actions:

```
data Act = Sink | Display | Save
```

Here, `Sink` requests a node without doing any additional action. Given a list of outputs we create the minimal `WorkPool` by:

- Selecting the nodes that are part of the associated output;
- Attaching an extra `Work` to the non-`Sink` outputs to execute the requested action. This is done by the `toWorkPool` function (Listing 5).

Type-driven output. This function generates the minimal computation graph for the specified output, running all the specified output actions (e.g. `Save`) as the corresponding nodes get executed. When QuickRisk is used with a command line interface, the list of `[(Stage,Output)]` can be easily derived from the command line arguments, but when it is used as a function call in Mu this list is derived from the expected output type, and the corresponding value is extracted from the WorkPool execution (i.e. the `WorkResult`). These two steps are done using type classes. The `Collect` type class defines which Stages are required to generate the requested output types. Similarly the `Workable` class defines a function that extracts the value of the expected type from a `WorkSpace` and its execution `WorkResult` (Listing 6).

Using these two classes we can now write the `runQR` function, which is the top-level function of the QuickRisk API:

```
runQR :: forall r . (Collect r, Workable r) => Inputs -> Parameters -> SafeIO r
runQR inputs tp =
  runReader workable . (, w)
    <$> runWork (toWorkPool (map (, Sink) $ collected (Proxy :: Proxy r)) w)
  where w = workSpace inputs tp
```

Here `runWork` is one of the possible methods to run a `WorkPool` and compute its corresponding `WorkResult`. For illustration, Figure 4 represents the computation graph generated from the code

```
runQR [inputTrd trd, inputPrc prc] mempty :: SafeIO (Result, Attribute)
```

while Figure 5 is the graph generated using a CLI call

```
qr --inputTrades=myFile --display=Res,Attr
```

3.3 Shepherd

Shepherd is a workflow management system and job scheduler aware of (typed) job dependencies, implemented in Mu. Figure 6 shows a screen capture of the Shepherd GUI client (also implemented in Mu) with a graph plot of a QuickRisk run in progress. There are a variety of workflow management systems in the market, but none immediately satisfied all our needs, so we wrote our own. The design goals for Shepherd are:

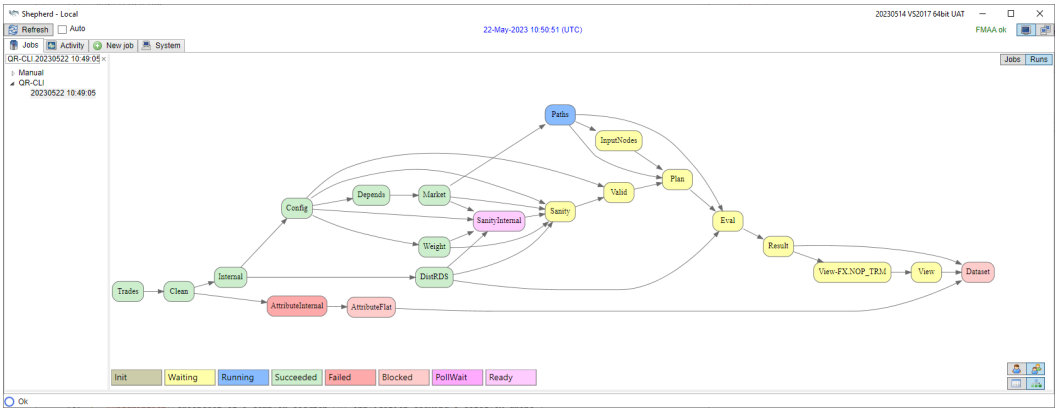

Fig. 6. The Shepherd GUI application.

- To have a robust process management strategy, in particular dealing well with jobs that may crash or otherwise misbehave;
- To be able to manage dependencies between different jobs;
- To support periodic jobs on flexible schedules (e.g. daily, hourly, every other Friday at 2pm New York time);
- To allow jobs to declare resources needed, and to schedule them based on available resources;
- To support running jobs on multiple Cortex versions;
- To be usable as a library, and as client or server;
- To not require a process constantly running, or anything to run on system boot (therefore being resilient to system reboots and being able to run as a non-privileged user).

Shepherd pairs particularly well with QuickRisk workflows as these tend to be large and long-lived, but we use it in many other settings as well.

4 Others

We have chosen to showcase QuickRisk as a prime example of a functional component at Standard Chartered Bank, but we have many other examples. We briefly describe a few of them in this section.

Lenses. We use both `fclabels`-style[2] lenses as well as profunctor optics[3] heavily, in particular in the context of selective manipulation of values of large datatypes (dozens of fields and constructor alternatives).

Monadic API for RESTful queries. We have built a monadic API to a RESTful service that automatically optimises queries by batching (collect queries and perform them all at once) and bulk (query full contents upfront and perform local queries subsequently) querying, which change the query behaviour depending on the desired style without requiring changing code (apart from the top-level invocation).

Virtual Widget. We rely on the Windows Presentation Foundation for building GUIs in Windows, and the Mu library for this purpose is called Widget. More recently we have built *Virtual Widget* on top of it, in a Model-View-Update style [4]. Virtual Widget allows us to keep GUI and business logic

[2] https://hackage.haskell.org/package/fclabels
[3] https://hackage.haskell.org/package/lens

separate, enables simple persistence of application state, and typically leads to more responsive GUI applications thanks to automatic avoidance of UI update loops.

Both libraries support an HTML + JavaScript back end as well, so the same code gives rise to an application that can run on a Windows desktop, or an HTTP server (running on Windows or Linux) that provides the same functionality over the intranet.

Market Data Framework. For the purposes of parsing and manipulating market data, we have developed a domain-specific language (DSL) at a high level of abstraction. It is intended for use by domain experts in financial data modelling. The resulting models work in real-time, publishing streaming data to an internal data bus, which is then used as a data source for financial analysis. The models are stored in a database, and can be viewed as a kind of stored procedures. At the start of each business day, the active models are loaded from the database and run on high-availability servers.

The DSL is a declarative, pure, dynamically-typed, first-order functional language comprising both standard constructs for control flow and working with structured data, and also highly specialised constructs to facilitate financial modelling. Given that it works on streaming data, it can also be seen as a (synchronous) data-flow language, but it is semantically stateless: the model is a pure function mapping only input to output for the current point in time.

Domain experts develop models through a purpose-built IDE that allows the models to be visualised as data-flow graphs with extensive support for troubleshooting and debugging, including the ability to evaluate (fragments of) models on live or canned data for instantaneous feedback.

There are actually two flavours of the DSL. One is the User-Facing Language (UFL), developed for use in the IDE to support model development and debugging. The other is the Server-Facing Language (SFL), which, while still being an interpreted language, is more of a "core language" intended for efficient evaluation. The translation from UFL to SFL includes an optimisation pass with common sub-expression elimination and arithmetic simplifications, as well as statically arranging the code for incremental evaluation for nodes depending on inputs that are expected to change infrequently, and static load balancing informed by a cost model for parallel evaluation. Our implementation makes extensive use of concepts like (bi-)functors, profunctors, folds, monads, monad transformers, and recursion schemes (catamorphisms, anamorphisms, hylomorphisms, etc).

Risk schemas. We deal with many different kinds of financial data and analytics, but we also need a way to conveniently handle all data generically. To this end, we have defined a framework for encoding structured risk data as a typed relation, with automatic derivation of an SQL representation (which is particularly useful for clients outside of Cortex, as explained in Section 5.1). This is achieved by classifying each column into one of three distinct categories:

Position A unique identifier for the position. Typically this is single column uniquely identifying a trade, but could also be a portfolio or product type, for example.

Coordinates Zero or more columns forming an n-space within which risk results are represented. The dimensionality of this space is determined by the specifics of the data. For example, a scalar is sufficient to capture fair value, so a zero-dimensional space is sufficient. For FX delta we need to distinguish between currencies so we enumerate this in a 1-space represented by one `CURRENCY` column. Following the same reasoning we arrive at a 3-space for IR delta (currency, curve, tenor).

Value The resulting sensitivity and auxiliary information (error and log data). This is a fixed set of columns common to all schemas. Values are required to be additive over positions, so arbitrary aggregation is well-formed (as long as the coordinate space is preserved).

This column categorisation is encoded in Mu at the type level for each risk scenario, and is used to generically convert relational data to and from SQL.

5 Interoperability

While the components we have described are all implemented in Mu, we often need to communicate with components built in non-functional programming languages, and we also expose direct Mu functionality to other languages. In many cases our components are only one piece in a larger technology solution, so it is essential to ensure we are both able to take inputs from as well as provide outputs to other components. There are fundamentally three ways to deal with this; we describe each of them in detail.

5.1 Avoiding the Issue

Usually the easiest approach is to have components communicate via plaintext files (e.g. CSV) or storage (e.q. SQLite with only primitive tyeps) whenever possible. This is often the case for end-of-day data analyses and reporting, and communicating via files or shared databases has the advantage of simplifying future architectural changes. For example, a replacement component can be built without changing the existing component, and it is easy for several components to consume a shared input.

5.2 Exposing Functionality as a Service

Another approach is to provide a REST-ful service for specific functionality. This is often more suitable for request/response interactions, and JSON, XML, or FpML[4] are common ways to encode the request and/or response.

5.3 Exposing Functionality as an Interface

Finally, we also provide some specific direct interfaces to certain programming languages. These require an initial effort to design and implement an adequate interface, as well as an ongoing effort to ensure the interface remains up-to-date with all Mu functionality as the language evolves, but the effort is justified when there is significant demand for ad hoc functionality from other environments.

6 Conclusion

In this report we showcased a few of the main ingredients of Cortex, with a particular focus on the Mu language and how we use it to develop quantitative financial software at large. We conclude with a discussion of the advantages and disadvantages of using functional programming in our setting; while noting that we are far from a controlled setting where different programming language techniques could be formally compared (e.g. by solving the same problem with the same budget over two separate teams and technologies), we do observe some patterns from our perspective, and we think at least some may be applicable in a general setting.

6.1 Advantages

Fewer bugs One of the main advantages from our use of Mu is that entire classes of bugs are absent thanks to a strong, static type system and memory management. A significant portion of time spent debugging our C++ code is on segmentation faults, which simply do not happen in Mu-only code. Purity by default, controlled effects, and hardly any explicit array indexation go a long way in preventing bugs.

[4]Financial products Markup Language: https://www.fpml.org/.

Types as structure We rely heavily on algebraic datatypes to structure our code, and on "types as documentation" whenever appropriate. In particular, each different kind of financial product (e.g. "FX Forward" or "IR Swap") is modelled by a different Mu datatype. This simplifies data migration and evolution as the type-checker points out all the places in the code that need to be changed (although we still have to take care of compatibility between clients on different versions or stored data).

Developer productivity While stressing that we have not compared this in a controlled setting, we generally find that our team can deliver similar end-user solutions in a shorter time-frame and/or with fewer developers than another internal team that uses Java and Scala as their technologies of choice. We find no significant difference in terms of hardware requirements.

6.2 Myths

We do not see any significant downsides from our use of functional programming, so we choose to instead address some myths often associated with it.

Interoperability We already covered in Section 5 how we communicate across the programming language barrier, and we believe this is not significantly different in a setting without functional languages, nor do we think it is something that can be avoided in general – large systems will always require interaction between components using different technologies. There are situations where such interactions may be made easier by the use of mainstream programming languages. For example, some web services and libraries may provide official APIs and example code written in Java or C, but will rarely do so for Haskell. But in our experience, the amount of time spent implementing a wrapper Mu API becomes quickly negligible over the lifetime of a project.

Performance That low-level languages are generally faster at program execution than high-level languages is not a myth, but an accepted fact. However, there is an exaggerated perception of how much and how often runtime performance must be a deciding factor in choice of programming language. In our setting (and note, in particular, that we are not involved in high-frequency trading) we find that the overhead of using a high-level language is minor; most of the end-to-end time to obtain a result is spent on database interaction (often across long distances). It is also generally clear which algorithms are best implemented in C++ (e.g. root-finding, Monte-Carlo simulation), and our seamless interaction between Mu and C++ (Section 2.3.2) makes it easy to write code involving the two languages.

Hiring Lack of developers with Haskell expertise is often cited as a concern for wider Haskell adoption in industry. In our experience, having hired over 100 developers whose main focus is to write Haskell/Mu, this has never been a limiting factor in team growth or business delivery. While we notice some trends depending on location and seniority, each of our open roles generally attracts between 10 and 30 candidates, and typically 3 to 5 qualify to go through our full interview process. We find that candidates that have demonstrated typed functional programming experience (whether in industry, academia, or by personal hobby) fare well in our interview process, which simplifies the problem of finding good candidates.

In conclusion, using functional programming proved to be a significant catalyst in our setting, and we hope our experience can motivate others to continue expanding the use of functional programming in industry.

Disclaimer

The authors are employed by Standard Chartered Bank. This paper has been created in a personal capacity and Standard Chartered Bank does not accept liability for its content. Views expressed in this paper do not necessarily represent the views of Standard Chartered Bank.

Acknowledgments

We would like to thank the reviewers for their helpful feedback, and also the many colleagues at Standard Chartered Bank (both past and present) who contributed to the various systems described in this paper or provided feedback.

A Supplementary Material

```
data Work_ = WorkPure Fun
           | WorkApp Work_ Work_
           | WorkIO (Any -> IO Any) Work_

newtype Work a = Work {unWork :: Work_}

type WorkPool = [Work_]

toKey :: Work_ -> Key

type WorkResult = Map Key Any
```

Listing 2. Work internals

```
workPure :: a -> Work a
workPure x = Work $ WorkPure $ cast $ \() -> x

workApp :: Work (a -> b) -> Work a -> Work b
workApp a b = Work $ WorkApp (unWork a) (unWork b)

workWorkIO :: Work (IO a) -> Work a
workWorkIO a = Work $ WorkIO (fmap (toAny :: a -> Any) . fromAny) $ fromWork a

getResult :: WorkResult -> Work a -> a
getResult wr w = fromAny <$> lookup (toKey w) wr
```

Listing 3. Typed Work API

```
data Stage = Trds | Mkt | Prc | Attr | Res
type Inputs = Dict Stage Any
type WorkSpace = Dict Stage (Work Any)

tradeWork :: Inputs -> TradeParams -> Work Any
tradeWork input p = maybe (workIO $ toAny <$> queryTrades p) workPure
                  $ D.value Trds inputs

workSpace :: Inputs -> Params -> WorkSpace
workSpace inputs Params{tradeParams,mktParams} =
  D.fromList [ (Trds, tradeWork inputs tradeParams)
             , (Mkt , mktWork   inputs trdWk mktParams)
             , (Prc , priceWork inputs trdWk mktWk)
```

```
            , (Attr, attrWork  inputs trdWk)
            , (Res , resWork   inputs prcWk attWk) ]
```

Listing 4. Workflow construction

```
toWorkPool :: [(Stage, Output)] -> WorkSpace -> WorkPool
toWorkPool outP ws = mconcat . map include $ [Trds .. Res]
  where include :: Stage -> WorkPool
        include s = mconcat . concat $ [ workPool (D.value s ws) : wps
                                       | not . L.null $ os ]
          where wps = L.map (outputWork w) $ L.filter nonSink os
                os = [ o | (s1,o) <- outP, s == s1 ]
```

Listing 5. Work pool

```
class Collect s r where
  collected :: Proxy r -> s

instance (Monoid s, Collect s a, Collect s b) => Collect s (a, b) where
  collected _ = collected (Proxy :: Proxy a) <> collected (Proxy :: Proxy b)

instance Collect [Stage] [Trade] where
  collected _ = [Trds]

class Workable r where
  workable :: Reader (WorkResult, WorkSpace) r

instance (Workable r1, Workable r2) => Workable f (r1, r2) where
  workable = (,) <$> workable <*> workable

instance Workable Price where
  workable = reader $ Price . castAny . g
    where g (r, w) = getResult r (D.value Prc w)
```

Listing 6. Type-driven output

References

[1] Lennart Augustsson and Mårten Ågren. 2016. Experience Report: Types for a Relational Algebra Library. In *Proceedings of the 9th International Symposium on Haskell* (Nara, Japan) *(Haskell 2016)*. Association for Computing Machinery, New York, NY, USA, 127–132. https://doi.org/10.1145/2976002.2976016

[2] Standard Chartered Bank. 2024. Standard Chartered—Annual Report 2023. https://www.sc.com/en/investors/financial-results/, 1 Basinghall Avenue, London, EC2V 5DD, UK.

[3] Luke Clancy. 2019. Functional programming reaches for stardom in finance. https://www.risk.net/6395366.

[4] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 411–422. https://doi.org/10.1145/2491956.2462161

[5] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. 2011. Towards Haskell in the Cloud. *SIGPLAN Not.* 46, 12 (Sep 2011), 118–129. https://doi.org/10.1145/2096148.2034690

[6] Gergő Érdi. 2022. Compiling Mu with GHC: Halfway Down the Rabbit Hole. Talk at the 2022 Haskell Implementors' Workshop. https://www.youtube.com/watch?v=fZ66Pz7015Q

[7] Felienne Hermans, Bas Jansen, Sohon Roy, Efthimia Aivaloglou, Alaaeddin Swidan, and David Hoepelman. 2016. Spreadsheets are Code: An Overview of Software Engineering Approaches Applied to Spreadsheets. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5. 56–65. https://doi.org/10.1109/SANER.2016.86

[8] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. 2003. A User-Centred Approach to Functions in Excel. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (Uppsala, Sweden) *(ICFP '03)*. Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/944705.944721

[9] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. 2000. Composing Contracts: An Adventure in Financial Engineering (Functional Pearl). *SIGPLAN Not.* 35, 9 (Sep 2000), 280–292. https://doi.org/10.1145/357766.351267

Received 2024-02-28; accepted 2024-06-18