Functional Programming in Financial Markets

José Pedro Magalhães
MD, Global Head of Core Strats

Joint work with
Atze Dijkstra and Pierre Néron

20 August, 2025

- Context
  - Financial Markets
  - Functional programming in Finance
- Cortex
  - Mu
- Examples
  - Widget
  - QuickRisk
  - Shepherd
- Conclusions

DISCLAIMER: The opinions expressed are the presenter's own and do not reflect the view of the current or previous employers.

Context

Context – Financial Markets

Quantitative Analytics: a function that enable banks to maintain a competitive presence in the securities markets business

Part of Front-Office; frequent and direct interaction with trading and structuring functions

Fast changing priorities and demand for quick turn-around

Heavy on financial mathematics (stochastic calculus, measure theory)

Convention-laden (millions of details of market conventions)

High-performance compute oriented (lots of linear algebra and Monte-Carlo simulation)

Product and model development mirror the corresponding business lines (e.g. FX, Rates, Credit)

Core (including compiler), CI / DevOps, applications development, project management and Model Risk are shared across the business lines

Context – Functional Programming

Programming paradigm where functions and function composition dominate

Functions are first-class citizens

No sequence of imperative statements updating state

Particularly interesting when combined with purity – no unexpected side effects

Roots in lambda calculus

Lisp, Scheme, Erlang, OCaml, Haskell, F#

Context – Functional Programming in Finance

There's a significant history of FP being used in finance, for at least 15 years (Barclays, Jane Street, Standard Chartered)

But let's not forget spreadsheets

Spreadsheets were at the core of FP use at Standard Chartered

Lambda in 2008 bridged existing C++ code and Microsoft Excel via an add-in

- Had a pure interface, supported higher-order functions and parametric polymorphism

Increased popularity of Lambda highlighted the lack of a module system

- A full-fledged programming language was needed
- Haskell was a strong candidate... but we did not want to lose C++ and Excel interoperability and the ability to serialise all values

Mu was created as a dialect of Haskell that would mesh well with our existing ecosystem

Cortex

Main pricing library in FM

Cortex

Cortex: the entire software ecosystem developed by MAG for financial analytics in FM

Various libraries for quantitative financial modelling

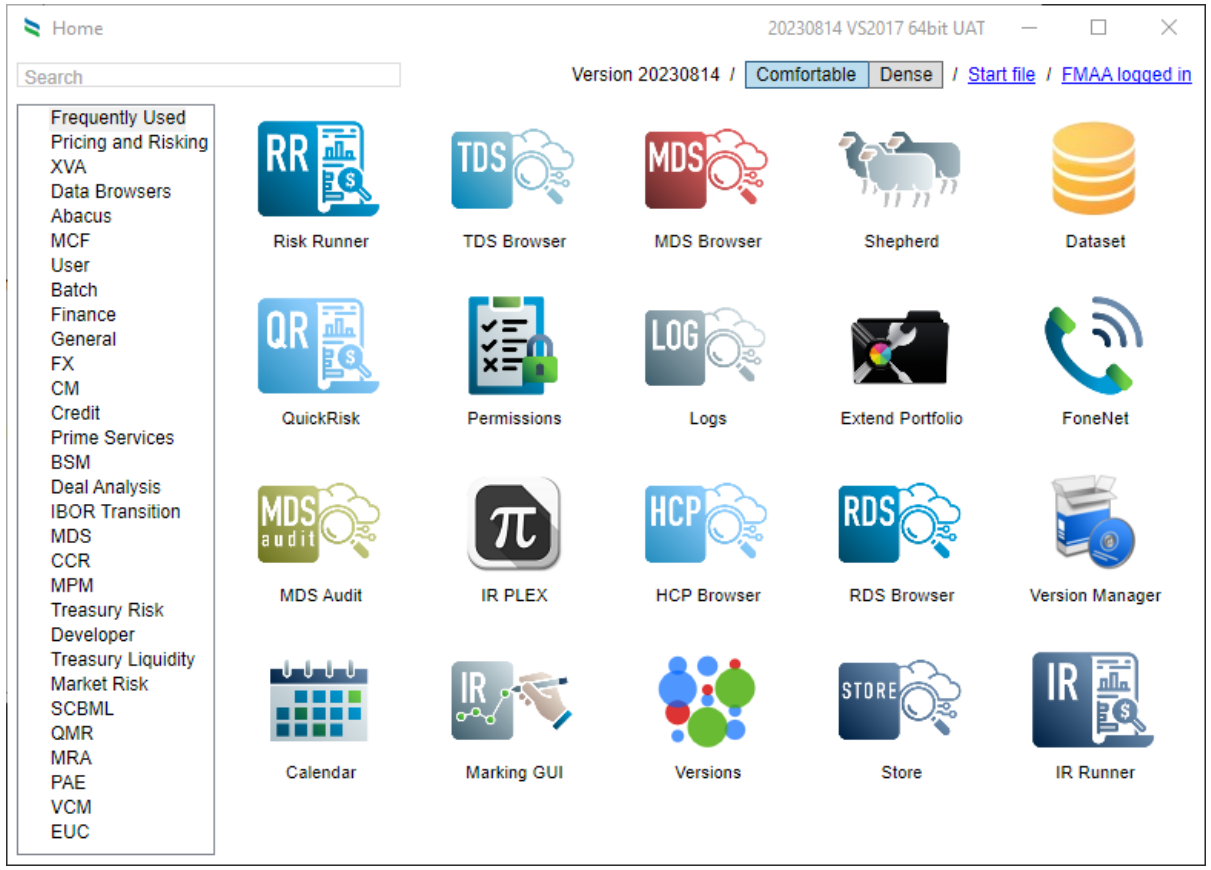
The Mu compiler

Interfaces to Excel, C++, .NET, Java, Python

End-user GUI applications

Several production server-side components, such as an intraday risk service

Built, tested, and released automatically on a daily basis



Cortex – Rate aggregation from executed trades

```
calc_DF_2 :: [FXD_Transaction HiborTrace] -> [FixedIncome_Transaction HiborTrace]
          -> Maybe Choice
calc_DF_2 ts_fxswaps ts_fixedIncome =

  let rateFxSwaps = (+ spreadAdjFX)
                  . (/ aggregateNotionalFX)
                  . sum
                  . map (\t -> (abs . getAmount . getHKDAmount $ t) * unRate (dc02_ImpliedXIBOR t))
                  $ ts_fxswaps

      rateFixedIncome = (+ spreadAdjFI)
                      . (/ aggregateNotionalFI)
                      . sum
                      . map (\t -> (abs . getAmount . dc03_Notional $ t) * rateEFBN t)
                      $ ts_fixedIncome

  in toDF2 . Rate $
     ( (if null ts_fxswaps     then 0 else rateFxSwaps     * aggregateNotionalFX)
     + (if null ts_fixedIncome then 0 else rateFixedIncome * aggregateNotionalFI)
     )
     / (aggregateNotionalFX + aggregateNotionalFI)
```

Cortex – Mu and its key differences from Haskell

- No separate compilation – Mu is a whole-program compiler
- Compiled to bytecode with a small runtime interpreter
- Strict runtime, lazy semantics!
  - We don't want Mu to be too different from Haskell
  - For example, `fromMaybe undefined (Just ())` is expected to work by Haskell programmers
- Strings are native, implemented as UTF-8 C++ `std::string`
- No foreign-function interface
  - All C++ functions are called natively
  - Unified C++ `IFunction` interface which both C++ and Mu functions adhere to
- Recursion is discouraged (lack of tail-call optimisation) and disabled by default
  - Positive side-effect of Mu runtime implementation
  - Rarely needed (< 4% of the modules)
- Extensions: no existentials, GADTs, rank-2 (and higher)
- (De-)serialisation is supported by default for mostly everything in Mu
- Relations instead of lists
- `SafeIO` and `IO`

Cortex – Mu compiler pipeline

Current

Future

Examples

Widget – GUI library for desktop applications

We rely on the Windows Presentation Foundation for building GUIs in Windows

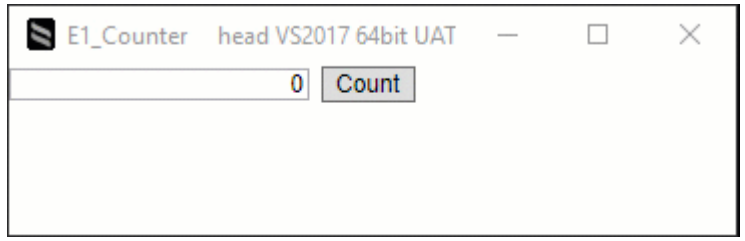
The library is called *Widget*

More recently we have built *Virtual Widget* on top of *Widget* in a Model-View-Controller style:

- Allows us to keep GUI and business logic separate
- Enables simple persistence of application state
- Typically leads to more responsive GUI applications

Widget – Native vs Virtual – counter

```
main :: IO W.Widget
main = W.widgetMain $ do
    input <- W.intBox [] 0
    btn   <- W.button [] "Count"
    W.event btn $ W.modify input succ
    return $ input W.<|> btn
```




```
type Model = Int
data Msg = Change Int | Click

update :: Msg -> Model -> Model
update (Change x) = const x
update Click      = succ

view :: V.View Model Msg
view = V.intBox (V.onChange Change) id V.<|>
    V.button (V.onClick Click) (V.label mempty "Count")

main :: IO V.Widget
main = V.widgetMain $ do
    st <- V.appState 0 mempty
            (\model msg -> (update msg model, mempty))
    V.renderView st view
```

Widget – Native vs Virtual – converter

```haskell
cToF :: Double -> Double
cToF c = c * (9/5) + 32

fToC :: Double -> Double
fToC f = (f - 32) * (5/9)

main :: IO W.Widget
main = W.widgetMain $ do
    inputC <- W.doubleBox [] 5
    inputF <- W.doubleBox []
                W.@= cToF W.-# inputC
    inputF W.+= W.put inputC . fToC
    return $ inputC W.<|> "Celsius"
               W.<|> inputF W.<|> "Fahrenheit"
```


```haskell
type Model = Double -- celsius
type Msg   = Double -- celsius

update :: Msg -> Model
update = id

view :: V.View Model Msg
view = V.horizontal
    [ V.doubleBox (V.onChange (id :: Double -> Double)) id
    , V.label mempty "Celsius"
    , V.doubleBox (V.onChange fToC) cToF
    , V.label mempty "Fahrenheit"
    ]

main :: IO V.Widget
main = V.widgetMain $ do
    st <- V.appState 5 mempty
            (\_ msg -> (update msg, mempty))
    V.renderView st view
```

QuickRisk – Type-driven API to a pricing library

Generates and executes type-driven workflows depending on given inputs and needed outputs

Typical use-case: trades as inputs, risk results as output

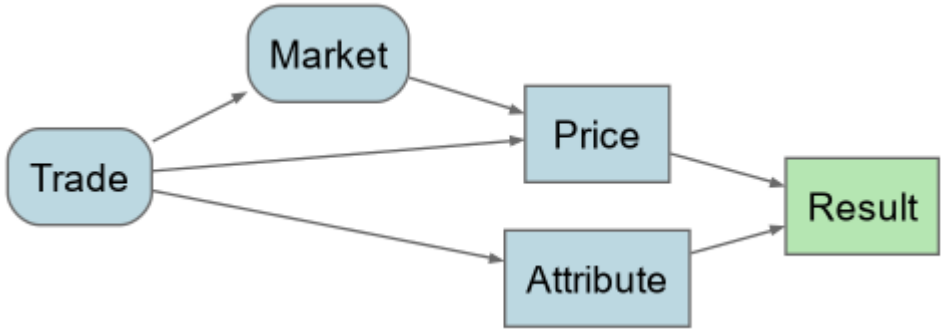
- But one could obtain prices and market as well
- Or supply trades and obtain only prices:
  `quickRisk [ inputTrade t ] :: IO Price`

Computational graph encoded via an ADT

Execution of the graph can choose different evaluation strategies (sequentially, in parallel, out of process, etc)

Sharing of nodes during execution

From input trades to output result



From input trades to output price

Shepherd – Workflow management system

Workflow management system and job scheduler

Implemented in Mu

Robust process management strategy, in particular dealing well with jobs that may crash

Manages (typed) dependencies between different jobs

Supports periodic jobs on flexible schedules (e.g. daily, hourly, every other Friday at 2pm New York time)

Allows jobs to declare resources needed, and schedules them based on resources available

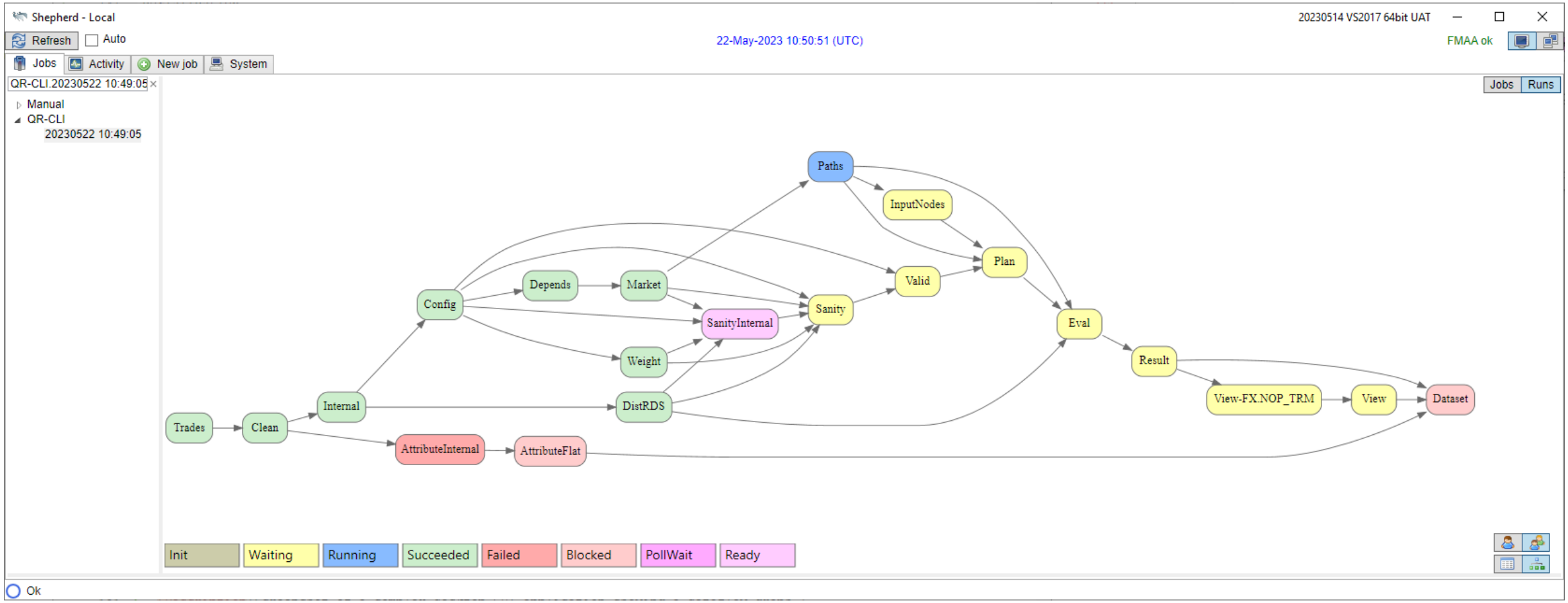
Supports running jobs on multiple Cortex versions

Usable as a library, and as client or server

Does not require a process constantly running, or anything to run on system boot

Resilient to system reboots, and able to run as a non-privileged user on Windows

Shepherd – GUI Application



Conclusions

Some (approximate) figures

Over 120 FTE in Modelling & Analytics

- About half is Core Strats

Nearly 7.5 million LoCs of Mu, 0.5 of Haskell, under 2 of C++

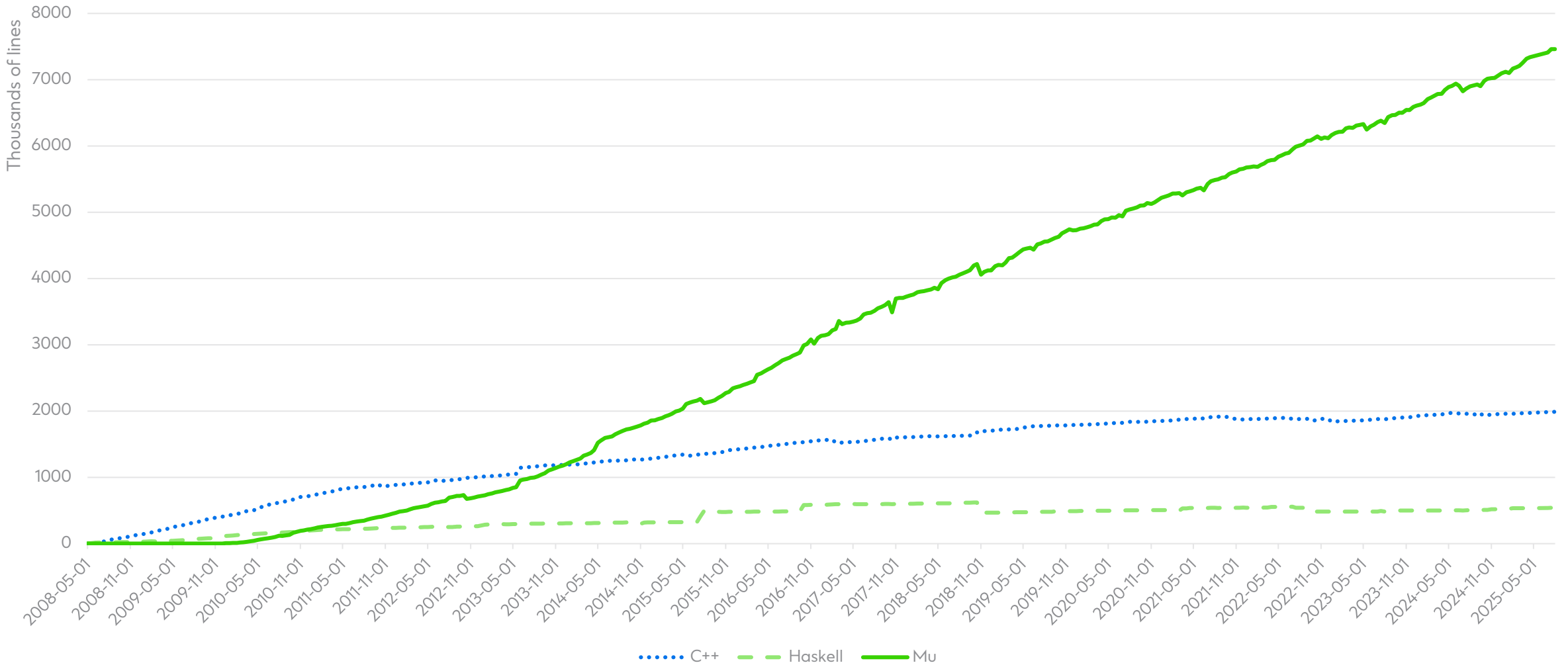
- Over 700 contributors over the past 15Y, over 300k commits
- 7,375 pull requests merged in 2024, 302 releases

Over 5,800 tests run on all code changes

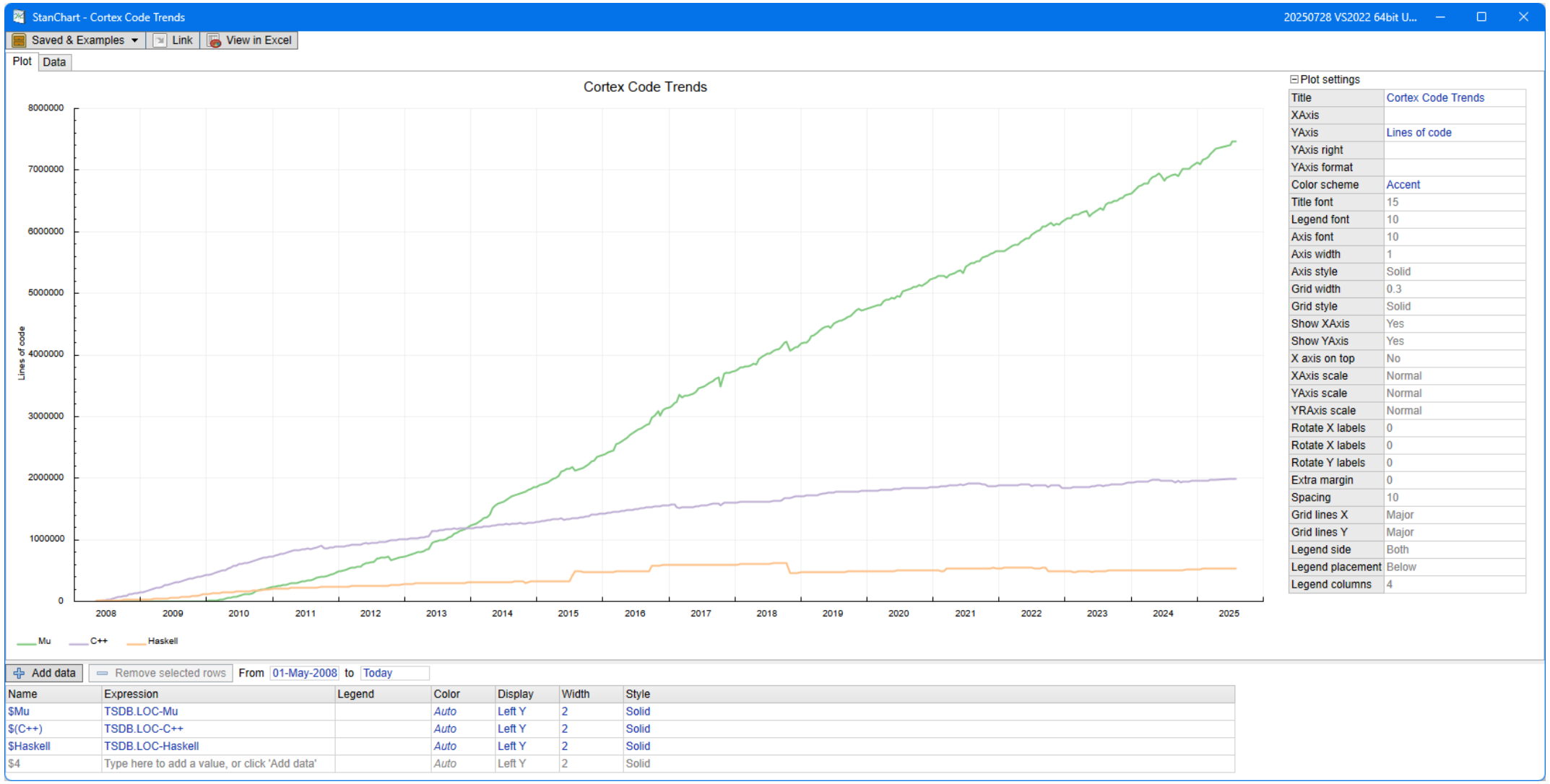
Hundreds of applications

- Popular applications have over 1k unique users per month
- One large application consists of 850k lines of source code from 4k modules

Lines of code growth over time



Lines of code growth over time, in a Cortex app



Advantages

Fewer bugs

- Entire classes of bugs 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
- Purity by default, controlled effects, and hardly any explicit array indexing go a long way in preventing bugs

Types as structure

- We rely heavily on algebraic datatypes to structure our code, and on “types as documentation” whenever appropriate
- Each different kind of financial product (e.g. “FX Forward” or “IR Swap”) is modelled by a different Mu datatype

Developer productivity

- Hard to compare in a formal way, but I believe we deliver solutions faster/cheaper

Myths

No significant downsides from our use of functional programming, so we choose to instead address some myths often associated with it

Interoperability: not significantly different in a setting without functional languages, nor something that can be avoided in general

Performance

- That low-level languages are generally faster at program execution than high-level languages is not a myth, but an accepted fact
- … but there is an exaggerated perception of how much and how often runtime performance must be a deciding factor in choice of programming language
- It is generally clear which algorithms are best implemented in C++

Hiring

- 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
- 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

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.

Read the paper: Functional Programming in Financial Markets (Experience Report),
in *Proceedings of the ACM on Programming Languages, Volume 8, Issue ICFP*

Questions?

Thank you

Contact: JosePedro.Magalhaes@sc.com