Functional Programming in Financial Markets (Experience Report)

José Pedro Magalhães
MD, Global Head of Core Strats

Joint work with
Atze Dijkstra and Pierre Néron

standard chartered

5-Sep-2024

- Context
- Cortex and Mu
- Conclusions

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

Context

Context
Functional programming in finance

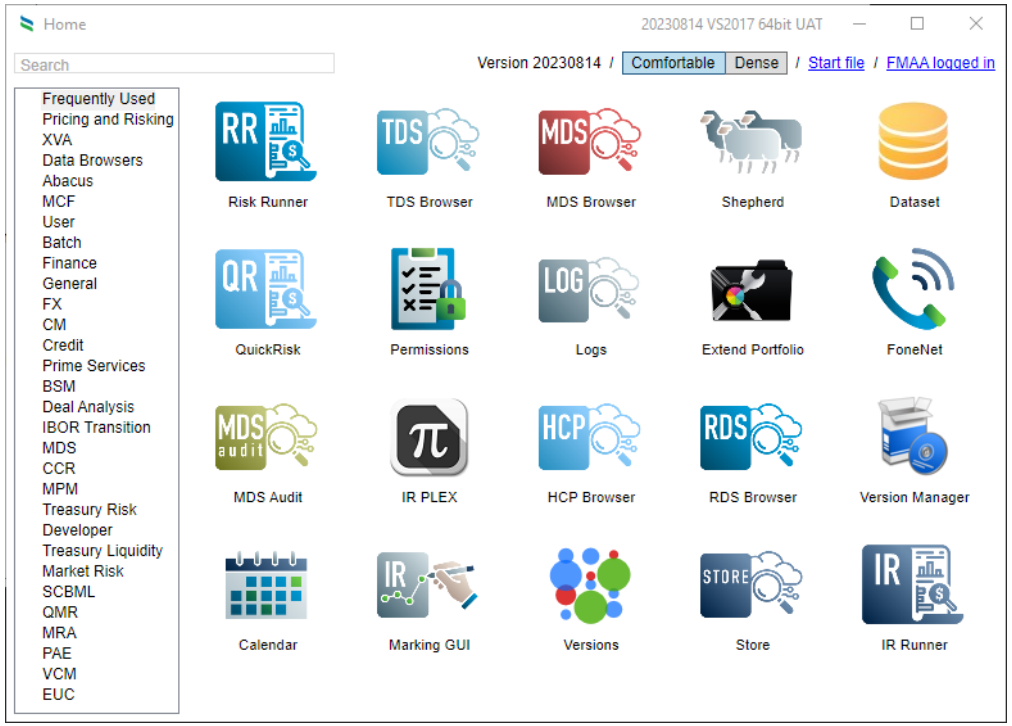
- There’s a significant history of FP being used in finance, for at least 15 years (Barclays, Jane Street, Standard Chartered)
- But let’s not forget spreadsheets
- Spreadsheets were at the core of FP use at Standard Chartered
- *Lambda* in 2008 bridged existing C++ code and Microsoft Excel via an add-in
  - 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

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

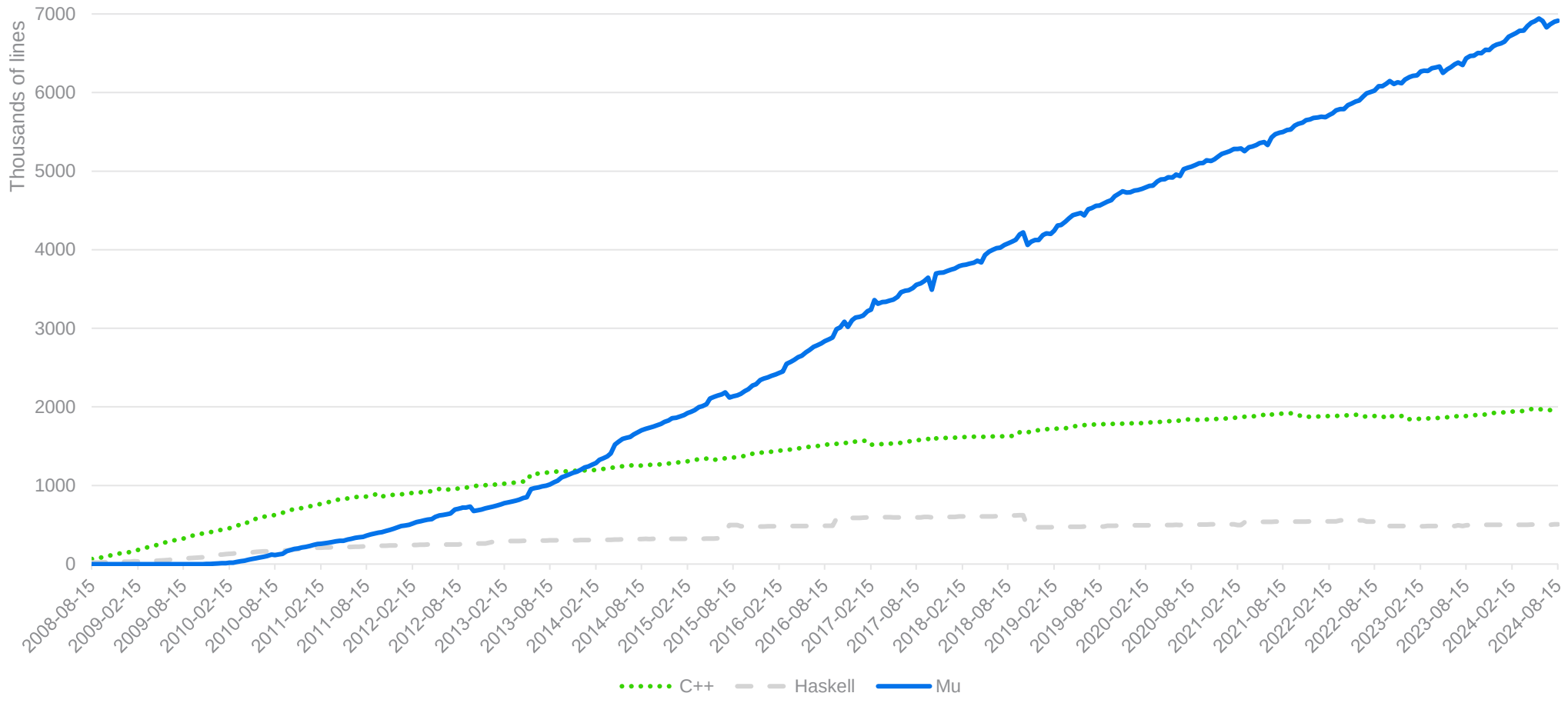
Future

Conclusions

Some (approximate) figures

- > 100 FTE in Modelling & Analytics
  - About a third is Core Strats
- > 7 million LoCs of Mu, 0.5 of Haskell, < 2 of C++
  - >700 contributors over the past 15Y, >300k commits
  - 3800 pull requests merged in H1 2023
- > 5k tests run on all code changes
- Hundreds of applications
  - Popular applications have > 1k unique users per month
  - One large application consists of 850k lines of source code from 4k modules

Lines of code growth over time



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 indexation 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
  - While stressing that we have not compared this in a controlled setting…
  - We generally find that we 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

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 in industry.

Questions?

Thank you