Findates Crate: A Rust Library for dates in finance

Guilherme Nunes Kobara

Birkbeck College, University of London

Table of Contents

Abstract

Any basic calculation for financial products references a notion of time. Multiple conventions exist so that computation of time in discrete periods can be achieved. While there are multiple resources for learning the theory of financial products pricing and the – often complex – models used, a lot of these fail to address the simple practicalities of getting the data to be used in those models. Moreover, there is no piece of data more elemental than getting the correct schedules, day counts, fractional periods for those calculations. The Rust Library Crate presented in this document aims to provide some essential functionality for these common necessities when dealing with dates in a financial products context.

Keywords: Rust, quantlib, finance, dates, holidays, calendars.

Findates Crate: A Rust Library for dates in finance

Computing human dates and time can be considered one of the most essential building blocks in Information Technology (IT) and Information Systems. Although relatively simple at the surface, the multiple time zones, daylight savings time, calendar changes, leap years, leap seconds, etc. have proved to be an immense challenge for computer scientists [1].

The financial services industry has also been dealing with the challenges of date and time arithmetic since its early days. Before there were any institutions like the modern banks, there was lending, borrowing and interest rates. Since interest can be defined as the value of money through time [2], an accurate way of measuring such "time" was essential. Although banks were some of the earliest adopters of Computers for their day-to-day operations [3], tackling calculations using date and time arithmetic had to be done in a clear manner much before that – as any calculation on how much money to pay or receive usually is subject to a great amount of scrutiny. As a way of dealing with the particularities of time mentioned above, and at the same time guarantee the accuracy and simplify calculations, market practitioners have adopted conventions on which calendars to use, how to count the days in a year, duration between dates, etc. Moreover, as is the case for many other attempts of creating a unified standard, various conventions were used for different products, such as regions and exchange rates.

After the advent of computing, this large variety of conventions had to then be modelled in software that was adopted by the financial industry. But like the issues with calendar changes and leap years, implementation was not as simple as it might first have looked. It is a common site, even nowadays, to see a phisical business day calendar, such as the one in *figure 1*, in many trading floors of banks or hedge funds. This gives away how software, historically, might have

had faced difficulties in date calculations, causing lots of traders to rely on a simple paper calendar and turning pages.

Nevertheless, the financial services industry has developed, or contracted, software solutions that deal not only with dates but with a myriad of other financial calculations and data processing needs. But those are mostly proprietary, not available to the public as open source. *Findates* aims to deliver just that: a library of functions and abstractions that can efficiently and reliably deal with date and time calculations and that can be incorporated into a larger piece of software catering to the financial industry. This library should deal with any of the most common current conventions, make it easy to implement any new ones and be implemented in a programming language that allows for fast computation while not compromising safety: Rust.

Literary Review

The Rust Programming Language

Although the focus of the current work is not the development and enhancements of programming languages, it is well worth to present Rust, its history, and main features. This is because not only its unique characteristics guided so much of the development, but also because the choice of the project itself is due to the fact that, even with the increasing popularity of Rust, there is a lack of an open-source library for dealing with date operations in financial services. Other popular languages have some established libraries for dealing with this subject that will be presented later, but as of the time of this project, there was nothing fully developed as such libraries in Rust.

Rust is a compiled, strongly, and statically typed programming language with strict evaluation. It was originally developed within Mozilla for systems programming, serving as a safer and more modern alternative to lower-level languages such as C and C++ [4]. These languages empower programmers with more control over memory usage, allowing them to

tailor it to the specific needs of the software being developed and to the kind of hardware it is supposed run. This made them the preferred choice for programming controllers with limited resources and operating systems kernels for example. But control of memory allocation has its downsides, mainly in the form of bugs – around 70% of security bugs are related to memory safety [5] [6]. Rust features over ownership and lifetimes aim to provide the same level of control over memory as C and C++, but without compromising safety.

Rust would also provide the higher-level abstractions of languages like Haskell, Python and Java without sacrificing performance or incurring overheads in memory usage. Iterators, Algebraic Data Types, Closures, Enumerations, Type Traits etc. are all packed within the language, available for use without imports of multiple libraries or interpreter and virtual machine "magic" – essentially contrived modifications of the language behavior opaque to the programmer.

The lack of a Garbage Collector is also what differentiates Rust from these languages and allows for better performance, as there are no additional memory management processes running in parallel with the main program. Memory safety is instead guaranteed by the *borrow checker* [7]. In essence, Rust has a strict set of rules over how data can be accessed and modified, enforcing at compile-time that there will be no possibilities for dangling pointers, data races or overflows during runtime.

There is also some trade-offs made for all these advantages. This kind of safety without compromises in performance can only be achieved by a large language with a big type system, various keywords, configurable attributes and concepts that simply don't exist in any other language. The Rust compiler also carries a lot of the burden, with extensive compile time checks and optimizations, with compilation for larger crates taking a longer time. These disadvantages sometimes present as a barrier for new programmers, as dealing with all these

new concepts and checks can be a struggle, leaving them with the feeling of "fighting" with the compiler to write even basic programs.

Despite these downsides, Rust's popularity seems to only keep growing, topping Stack Overflow's most "loved" programming language for 7 years in a row now [8]. The tooling developed around the language is already one its strongest points, with *Cargo* for package management, *Rustdocs* for documentation and language servers available on most Integrated Development Environments. Automated testing facilities can also be run via *Cargo* [9], with a command line utility that allows for integration and unit testing, specified tests only, on separate threads and examples with "assert" statements.

Although there is no agreed standard on what *idiomatic Rust* is, there are some key principles that the community adhere to, among others it can be cited: safety through *ownership* and *borrowing*; pattern matching; error handling using *Result* and *Option;* and higher order functions and closures. Rust also adopts a multi-paradigm approach to programming, bringing elements from Object-Oriented Programming and Functional programming together. It is important to note how this flexibility in design can be incredibly useful when integrating Rust into a larger IT ecosystem. Software doesn't exist in a vacuum [10] and *Findates* takes note of that by using the language's features to enable future implementations with concurrency, integration with Excel, serialization, etc.

Relevant Rust Libraries

Chrono and Time crates. There are two popular and well-maintained crates that deal with time and dates: *time* [9] and *chrono* [10] [11]. These can be found in the central repository crates.io [12], where most of the libraries made freely available by developers can be downloaded. Both crates seem to be actively maintained and widely used – as of the time of this writing, *chrono* had around 98.2 millions of downloads and *time* had 155.9 millions. Both

crates provide abstractions for a date, a datetime, durations and simple parsing, but time aims to provide only the most basic functionality while chrono gives more tools and it is time zone aware. For *Findates,* given the choice of dependency between two libraries that seem virtually equivalent in terms of its basic quality, the one with an additional feature that might be desirable in the future was chosen: *chrono*.

It is worth then expanding on some of the internals of chrono as some of the design choices made there guided and influenced the work on *Findates*. Actual dates there are represented via structs: Date and NaiveDate, with the latter representing an ISO 8601 [13] calendar date and the former representing the same date but with a time zone component. Since you can easily construct a Date from a NaiveDate by specifying the time zone, *Findates* will mostly use NaiveDate as the abstraction for a date and implement most of its functions and further abstractions using it.

The NaiveDate module in chrono also contains trait implementations for it, such as Add, Sub, Copy, Eq, Ord, etc. aside from many convenience functions for construction, transformation, and parsing. It makes sensible use of the *option* module [14], in line with the type-driven design principle of pushing unsafe or undefined behavior to the borders of your program [15]. *Findates* leverages most of this work and does not duplicate any functionality already present in chrono. It also provides *enums* for days of the week and months [10], which are required functionalities for dates operations in the *Findates* context.

Rustquant and Quantmath. Specialized quantitative finance libraries in Rust are, at the time of writing, very incipient in its development. *Rustquant* [16] and *Quantmath* [17] are perhaps the two biggest ones that can be found in *crates.io* and, even so, they have relatively small amount of downloads. The two libraries aim to provide quantitative finance functionalities for pricing, risking, etc. of financial products.

Of the two, *Rustquant* seems to be the one with the most comprehensive scope, aiming to emulate the *Quantlib* C++ library – that will be presented later. It was created fairly recent, with the initial commit on August 2022, and development seems active, with commits being done as this report is being written. For now, it seems that work is more focused on pricing and risking models, with the modules that deal with dates calculations still under development. They use the *Time* library, so there is no time zone awareness and implementations for only a few of the day count conventions seem to be available.

Work on *Quantmath* on the other hand, seems to have stopped. The last commit to its repository was done over 5 years ago at the time of this writing [17]. But more than dealing only with the more "glamorous" parts of financial calculations involving stochastic calculus and Monte Carlo simulations, but it also states that the library aims to "introduce a level of real-world messiness that is often missing from academia." [17]. This is of interest since *Findates* objective is to deal with a specific part of real-world "messiness": date operations. *Quantmath* has a separate module for date calculations, where it defines its own struct for a date object. This creates no dependencies to *Chrono* or *Time*, but it also means that a lot of the work that those libraries did with regards to the more difficult parts of date handling such as parsing different formats, handling leap years, time-zones etc. is absent.

A more important point when comparing these two libraries with *Findates* is simply that *Findates* does not aim to be a complete library for financial products and all of its calculations, models, etc. Date operations are required not only by Options, Swaptions or any other exotic derivative with complicated pricing models. A simple mortgage calculation, savings account or credit card interest payment also requires a simple but robust set of tools for date handling. This is what *Findates* is focused on and why it has been created as a separate library.

Quantlib C++ Library

Quantlib can be seem as the *de-facto* standard open source library for financial calculations. It was originally developed within a company, Statpro [19], but later made available as a free and open-source library. Multiple books, papers, blog posts and talks have been produced about it [20]. It is a very mature and stable project, with multiple contributors, including former market practitioners, and, different to the Rust Libraries, actual use in production [21]. It is written in C++, which still dominates the financial services industry[1], specially in any part of the software infrastructure that demands performance through huge amounts of calculations, such as real time risking and pricing of traders' positions.

It provides a full suite of tools for risk analysis, pricing models, optionality, mark-to-market calculations, etc. These models comprise the bulk of the work and the focus of *Quantlib*, allowing researchers, regulators, financial institutions, etc. to have a framework for the best practices of financial institutions in regards to its quantitative finance solutions.

Quantlib also has a date and time module that is incredibly comprehensive. It follows an Object-Oriented Model (OOM), with date and time objects defined within the library itself. No external dependencies can increase robustness as dependency management in C++ has notoriously been at odds with furthering the progress of the language [22]. It has abstractions for day count conventions, calendars, frequencies and schedules but also provides convenience objects that more steeped market practitioners use such as IMM dates [23] and Futures contracts maturity codes [24]. These are also modelled as objects with internal states and methods used to transform and query them.

Given how well stablished *Quantlib* is, it would seem that a simple *translation* of the date and times module into Rust would suffice, maintaining its OOM design and making the

[1] Although there is no official source for which language is more used and in which parts of the financial sector, this can be gauged by observing the amount of jobs postings requiring C++. Anecdotal evidence from *quants* and IT staff in the financial industries also confirms that.

necessary changes only when strictly required due to the particularities of Rust. It will be clear through the methodology and implementation section how these particularities often "guide" the design in a different direction than what a classic OOM would suggest, and that is a good thing. *Quantlib* recognizes the difficulties that such model might have when put inside a diverse IT ecosystem such as one that might exist in a Bank [21]. Namely, designing with Rust's algebraic data types, generic types, traits and lifetimes is perhaps the biggest improvement that could be made over *Quantlib*, or any other software written within an OOM paradigm. Nonetheless, *Findates* takes the most inspiration from *Quantlib* out of any other existing library, and mirror some of its most efficient algorithms on date algebra for example.

Java and Python Libraries

The main choice for a Python developer looking for a quantitative finance library would be *Quantlib*. It provides Python bindings generated via SWIG [25], allowing for the C++ code to be called as it would any other API. Python is popular for its ease of development, being a garbage collected, dynamically typed and interpreted language. But all of these conveniences come with a cost on performance and memory usage.

The most popular and comprehensive open-source quantitative finance library in Java is *Strata* [26]. It was developed by Open Gamma [27] and it aims to deliver a full set of tools for market risk analytics. Java is also a garbage collected language, that adheres to the OOM principles quite strictly. The *Findates* implementation in Rust will differ the same way it differs from *Quantlib*, but the libraries documentation [28] has also been used for reference of some of the calculations and conventions treatments.

Aims and Objectives

The main aim of the project is to develop a library in Rust for dealing with date operations that are typical for financial products valuation, risk measurement, cash flows generation, etc. This library will only deal with date and time calculations, but it could be incorporated and used in a more general quantitative finance library or in any other software that deals with financial products. It should be relatively lightweight, reliable, and performant in terms of speed and memory usage.

To reach that aim, we lay out the following tasks and objectives:

1. Create an enumeration for the more commonly used day count conventions.
2. Functions for day count fractions using such conventions.
3. Create an abstraction for calendars and set operations between them.
4. Business day count and arithmetic.
5. Schedule Rule abstraction and subsequent schedule generation using this rule.
6. A simple parser for the creation of Schedule rules from text.

Methodology

As previously mentioned, the choice of Rust as the programming language guides a lot of the design decisions concerning the development of *Findates*. The language features allow for a multi-paradigm approach that is particular to and enabled by Rust. These features will be explained in more detail throughout this section as the data structures and modelling are explicated.

Cargo provides a convenient `new` command for creating a folder structure for any Rust project, passing a "*—lib*" argument with it generates the folder structure for a Library Crate, i.e. without a *main.rs* file – this would be called a binary crate, as it contains a program that can be compiled into binary. *Figure 2* shows the folder structure generated by Cargo for *Findates*.

The *"src"* folder contains the source code for the library and, instead of a *main.rs* file, *Cargo* creates a *lib.rs* file which is called the *crate root.* It is where compilation starts from and what makes up the root module of the crate. The *examples* folder contains files with *main* functions that can be run using the `cargo run –example` command. The *tests* folder has integration tests – by convention, unit tests are implemented within the same files that they refer to in a separate "*tests"* module. The *target* directory contains the outputs of the compilations processes: libraries, binaries, and generated documentation.

Findates is divided in 4 main modules: *Algebra, Calendar, Conventions,* and *Schedule.* Modules in Rust are used to organize code and control scope. Therefore, all the internal implementations directly related to each of the abstractions are contained within its own module.

Conventions Module

The conventions module is where all the enumerations regarding day counts, adjustment rules, frequencies and tenors are implemented. These were implemented using Rust's *enum* type, so variables of that type can take the form of only one of the *enum variants* at a time. The compiler assures that only those conventions that are inside the *enums* are allowed, and that implementations for all of these *variants* are present whenever they are used. This ties back into the Rust's principle of handling any possible errors in programming at compile time and not at run time.

DayCount Enum. Typically – and sometimes by law [31]– interest rates are expressed in annualized terms, so market practitioners have adopted conventions on how to count the number of days in a year and days in a month so rates can be converted in different fractional periods. A simple string parser was implemented as the *from_str* function, which is a required function for the *FromStr trait.* The inverse operation, i.e., the *Display trait,* was also implemented manually. The *enum* derives the default implementations for the *PartialEq, Eq, Copy, Clone, Debug* and *Hash traits,* providing convenient functions for cloning, printing to output, sorting, etc.

Each of the variants for the different conventions implemented in *findates* is described below:

1. *Act360:* The number of actual days in the period over a 360 days year.
2. *Act365:* The number of actual days in the period over a 365 days year.
3. *Bd252:* The number business days in the period over a year with 252 business days – This convention is, of course, dependent on a holiday calendar.
4. *ActActISDA:* The number of actual days in the period over: 366 days year the parts of the period that fall in a leap year; 365 days year for the parts of the period that fall in a regular year.
5. *D30360Euro:* The number of actual days in a period taking into account that if either the start or end date of that period falls on a 31st of the month, it will be adjusted to the 30th. This over a 360 days year.
6. *D30365:* The number of actual days, but with months between the start date and end date having exactly 30 days and years with 365 days.

AdjustRule Enum. Different financial products and markets have different conventions about how to adjust a scheduled date e.g., the maturity of a bond, in case

it falls on a non-business day. This enumeration lists the implemented adjustment conventions and, again, any useful *traits*. The *variants* are listed out below:

1. *Following:* The next business date immediately after the original date.
2. *Preceding:* The business date immediately before the original date.
3. *ModFollowing:* Referred to as "Modified Following", the same as *Following* but, if the next business date falls in the next month, use *Preceding*.
4. *ModPreceding:* Referred to as "Modified Preceding", the same as *Preceding* but, if the previous business date falls in the previous month, use *Following*.
5. *Unadjusted:* Keep the original date, regardless of it falling in a holiday or weekend.
6. *HalfMonthModFollowing:* Referred to as "Half month modified following", same as *ModFollowing*, but if the adjustment crosses the 15th of the month, use *Preceding.*
7. *Nearest:* Simply adjust to the nearest business day and, if they are equidistant, use *Following.*

Frequency Enum. Interest payments for various financial products are usually done in a periodic basis instead of a lump sum at maturity. The common different frequencies are covered by this enumeration. Again, a simple parser has been implemented as the *from_str* function, and the most common use *traits* have been derived. The *variants* for this *enum* are: *Daily, Weekly, Biweekly, EveryFourthWeek, Monthly, Bimonthly, Quarterly, EveryFourthMonth, SemiAnnual, Annual,* and *Once.* The names of the *variants* already provide enough information as to how they should be implemented.

Calendar Module

As mentioned previously, different products trade in different markets with different conventions. Those markets can also be in different countries – or sometimes different states – that have different Holiday calendars or even weekdays that are workdays e.g., some middle-

eastern countries have Fridays and Saturdays as their weekends. The *Calendar* module contains all of the functions and abstractions needed for representing these different holiday calendars. In its basic form, they are modelled as a *struct* containing two fields: *weekend* and *holidays.* The code in *figure 3* shows the declaration of the *Calendar struct*. Both fields are made public using the *pub* keyword, allowing for other functions to alter the "state" of the *Calendar.* This is in contrast with what traditional OOP would dictate regarding encapsulation, but as previously mentioned, Rust does not aim to provide safety via the OOP model. Scope is controlled by modules and there is no inheritance. The ownership model and borrow checker will assure that no unsafe behavior is permitted regarding the state of the *Calendar* fields regardless of how they're being accessed and modified.

The *weekend* field is a wrapper over a *Hashset* of *Weekdays*. A *HashSet* in Rust is one of the *Collections* allocated in the heap, permitting it to grow and diminish in size as needed, its size is not known at compile time. *Weekday* is an *enum* representing weekdays from *chrono,* so no additional implementation was required in *Findates* in regards to representing weekdays. Although the most common work week would only require a stack allocated *list* with a length of two, the implementation allows for the possibility of a four days work week, three days work week, etc. in case this ever gets adopted in the future. It might also serve on impact analysis before such change ever gets implemented.

Holidays is also a wrapper, but this time over a *Hashset* of *Naivedate's*, which is simply the representation of a date in *chrono*, but without any time zone information hence "naive". The *Hashset* collection was chosen again as a means of storing the dates that are holidays for a given calendar. This data structure allows for fast look up [32], which is essential given that the most basic operation over *Calendar* would be to check if a date is present inside *holidays*. But it also allows another very common requirement within the financial industry: Set operations, e.g. the union of two calendars need to be used to adjust settlement dates that of

cross-currency products. It is also the most efficient choice given that there would be no sense in storing repeated dates inside *holidays*.

Methods were implemented for: the creation of a *Calendar*; modifiers for the two fields allowing inclusion of holidays and weekends; retrievers of the data in both fields; and relevant set operations like union and intersection. Free functions were also provided for convenient creation of a "basic" calendar with only Saturdays and Sundays as weekends and for the union of multiple calendars using an iterator and a closure for efficiency [33].

Schedule Module

The main data structure inside this module is the *Schedule struct* and its *iterator* format the *ScheduleIterator struct.* Conceptually, these abstractions are used to represent a list of dates that follow a particular set of rules, i.e.: every 10th of the month, every month for the next 2 years from a starting date, adjusting for weekends and holidays according to one of the *AdjustmentRule* conventions. The algorithms for generating such schedules are constructed in parts, enforcing the functional principle of composability.

The *struct* itself has three public fields: *frequency, calendar* and *adjust_rule. figure 4* shows the code snippet with the implementation. Both the *calendar* and the *adjust_rule* fields are wrapped with the *Option enum*, which was inspired by a feature commonly used in Haskell: the *Maybe monad* [34]. It allows for the generation of schedules that don't take a holiday calendar or an adjustment rule into consideration, or both options.

The set of rules of which a *Schedule* is composed of, can easily be seem as an *iterator* [35] and its implementation and use in *Findates* takes full advantage of the features of such structures. The *Iterator trait* was implemented for the *ScheduleIterator struct,* which wraps *Schedule* and adds one more field so that the iteration can actually occur: *anchor.* This *NaiveDate* provides a starting point for schedule generation and the list of dates being output

by any function or method that handles *Schedule* will include this date as its first entry. It is worth noting that iterators in Rust are *lazy* [35], which means that the actual dates dictated by a *Schedule* won't be generated until they are needed by the program. Allowing *Schedule* to be an infinite structure – only an initial date needs to be provided and the *next* method, could be called indefinitely. This allows for great flexibility as, ideally, range of dates don't have to be stored and can be generated for as large of a period as needed but also, only as needed.

The initial function in this module that provides the bulk of the functionality is *schedule_next* public function. This same function is used for the *Iterator trait* required *next* method – the only required method for this *trait*, which is in turn used in the *generate* method of *Schedule*, that outputs a finite *Vector* of dates that conform to its rules including the start and end dates provided. The function itself takes an anchor date, a frequency, an optional holiday calendar, and an optional adjustment rule, which coincide exactly with the fields required for a *Schedule* and *ScheduleIterator.* Its implementation considers all of the possible *variants* in the *Frequency enum* and in the *AdjustmentRule enum* and the different combinations, returning the next date dictated by a schedule e.g., the date within 3 months from the anchor date taken into consideration a particular holiday calendar and adjusting it using the *ModFollowing* convention in case it does not fall on a working day.

Algebra Module

The Algebra module implements all of the common operations involving the different abstractions and data structures within *Findates.* It is composed mostly of generic or free functions i.e. functions that are not associated with any particular data type[2]. These aim to be *pure,* that is, they should not produce any side effects [36]. This kind of implementation have

[2] In Rust there are two types of associated functions i.e. inside an *impl* block: the ones that do not take *self* as an argument, and the ones that do. In the case of the latter, they would be called a *method*.

the advantage of providing a clear API: generate an output for a given set of inputs. As a rule, any function that is not tightly associated with a single data structure was defined in this module in this form. Since Rust allows for circular dependencies in modules and due to their generic nature, some of the functions defined in *Algebra* are re-used in other modules.

The first public function implemented is *is_business_day.* It has the simple but essential purpose of checking if a date falls on a business day by taking a *NaiveDate* and a *Calendar* as inputs. Its implementation is simple by using the *contains* method on the *holidays* and *weekend* values of the *Calendar* input – both are *HashSets.* The code snippet *figure 5* presents the full implementation of the function.

The *adjust* function then uses *is_business_day* internally to adjust a given date according to a *Calendar* and a *AdjustRule.* The function implements the logic for each of the *AdjustRule variants* and returns a *NaiveDate* that represents a valid business date. This function is then composed into *bus_day_schedule* function to generate *Vector* of valid business dates between two input dates for a given *Calendar* and *AdjustmentRule.* A convenience *business_days_between* function is then provided that simply returns the length of this resultant *Vector*.

The *day_count_fraction* function utilizes all the functions above to calculate how much a period between two dates represents in terms of a fraction over a year according to a given *Daycount* convention. This can be done with or without an initial adjustment of the start and end dates, represented by the optionality of the *Calendar* and *AdjustRule* arguments. For the *Bd252* day count convention, a *Calendar* needs to be passed or a *panic* will be raised. All the conventions in scope are handled using Rust's *match* control flow construct, enforcing that all possible cases are handled [37].

Testing

Testing was done using *Cargo's* automated testing facilities, with unit tests inside *tests* modules in the same file as the source code that is being tested as per Rust convention [38]. Such modules are annotated with `#[cfg(test)]` so it is only compiled and ran when the `cargo test` command is run. Again, as per convention, integration tests were written in separate files inside the *tests* directory. These are compiled completely separately from the rest of the library – meaning they import *Findates* the same way any external code would, and each file there is compiled separately as its own crate.

A total of forty tests were implemented to check all of the functionalities and the semantics of all possible corner cases. It is worth mentioning that the functional approach adopted in some of the *Findates* implementations, greatly reduced the amount of tests needed given that, once properly tested, a function can be re-used inside another without additional testing required. However, the multiple conventions implemented and the combination between them necessitates a larger number of unit tests.

For the integration testing, the United States Federal holiday calendar was used [39]. Firstly the actual holiday dates were generated for the next ten years, taking into account the observation rules in case of a non-working day; and these dates were all then put inside a calendar that was then used to generate payment dates for a United States Treasury Note [40] and calculate day count fractions. *Findates* was able to successfully generate the dates and fractions with no discrepancies.

Results and Analysis

The results for a Library of functions such as *Findates* can be expressed by the compiling of the source code without any warnings, the use of its functionalities without raising any unexpected errors, and the semantically correct use of it. A real-life example was used for

integration testing – also included in the *examples* directory, with the NY Federal Reserve holiday calendar being modelled, payment dates for an issued U.S. Treasury Note, and day count fractions for interest payments. This example can be run using the command `cargo run --example calendar`.

While the example and tests mentioned above assert the absence of any incorrect or incorrect behavior for these cases, due to time limitations, they provide no benchmark in terms of speed and memory usage. Given the context of where *Findates* would be used – large number of financial operations and products at the same time, a comparison between *Findates* and say *Quantlib* performances for those two elements would be of great value.

Findates proves itself as a valuable addition to the Rust ecosystem of open source tools, as no other crate presently available contain such functionalities for dealing with dates. Considering all of the most widely known open source libraries for any language, its major advances are the use of zero cost abstractions such as infinite or lazy iterators for schedule evaluation, and algebraic data types that guarantee not only memory safety but completeness in implementation. It is also important to highlight how *Findates* aims to deal with a single aspect of calculations in finance: dealing with dates and its conventions. This means that any potential user can find in it a clear and concise API without having to worry about any of the other complexities in quantitative finance.

Evaluation

Findates fulfills its aim of being a concise but comprehensive library for dealing with date operations in a financial industry context. Enumerations for the most common day count conventions were implemented via the *Daycount enum;* all of these conventions are within scope of the *day_count_fraction* function; a *Calendar struct* was implemented, supporting multiple methods including set like operations such as union and intersection; a *Schedule struct* was implemented for modelling a set of rules for business dates and a *generate* method was

provided so an actual list of dates following those rules could be created; business days counting and operations using business days were made possible with the implementation of *ScheduleIterator*; and parsers from *Strings* were also implemented for each of the conventions.

Albeit the main objectives were met, in order for *Findates* to gather wider usage after as an open source library, there are shortcomings that need to be addressed: Firstly, the documentation needs to be considerately more extensive, with more internal links and examples as not to present a big upfront cost in learning the APIs; more day count conventions could be brought into scope as to cover a wider variety of markets; more "convenience" functions should be provided facilitating the creation of varied schedules with different inputs; and parsing the way it was proposed in the original aims was not implemented – the complexity of parsing even short excerpts of texts would be prohibitive in the given timeframe and dependencies on parser libraries, such as *nom* [39], would have to be added.

Another usable feature that could be added in future work would be to maintain different calendars on a separate "companion" crate, the same way that *chrono* keeps the actual time zones in *chrono-tz,* while the logic and abstractions for dealing with date and time are kept in the main crate. This would require an ongoing time commitment on maintaining the library, but it is a worthy feature that could increase usage of *Findates.*

Critical Evaluation

The main challenge in developing *Findates* in Rust was related to how the language forces the developer to think about memory usage, ownership, composability, etc. upfront, not allowing for compilation before these concerns were addressed. On a language like Python, such a library would perhaps take considerably less time to implement and less knowledge of some Computer Science concepts. But one of the main motives in Rust is "empowering everyone to build reliable and efficient software" [39], so this challenge brought with it a deeper understanding as a reward. Perhaps the biggest merit of *Findates* is in the use of the language

and its features, with the use of infinite iterators for schedule generation as the most notable innovation. The more functional approach in the *Algebra* module can also be considered of a more "modern" design, bringing the most benefits if this crate gets integrated into a wider Software ecosystem as it aims for purity and allows concurrency in computation.

This wider array of functionalities and improved documentation perhaps could have been achieved within the timeframe given. As the core implementations were done, these would be simpler tasks but time-consuming nonetheless, so a choice was made to focus on the core functionalities of the library.

Conclusion

Findates achieved its objectives of implementing enumerations for commonly used market conventions; provide functions for calculating day count fractions; creates a calendar abstraction and methods for it; allows for the counting of business days; provide schedule generators; and simple text parser for the conventions were. As the most notable accomplishments delivered by the library, and that could be seen as real advances in terms of the open source libraries presently available, are the use of lazy and infinite structures for the generation of schedules; the use of algebraic data types for the conventions; added safety by the use of Option and Result wrappers; and a functional approach to some of the modules, allowing for a clear API. But, in the future, more documentation needs to be written, additional functions need to be implemented – including a more extensive parsing, and a separate companion crate with actual holiday calendars.

A modern language like Rust provides many advantages but, as with any new technology, adoption will always face the hurdle of lack of available libraries and leave developers with the feeling of "building from scratch" or "reinventing the wheel". There is no shortcut for a vibrant library ecosystem for any new language, it needs to be built brick-by-

brick. *Findates* is a small but important “brick” of that construction: dates for financial calculations.

References

[1] Computerphile, "Computerphile Youtube Channel," 30 December 2013. [Online]. Available: https://youtu.be/-5wpm-gesOY.. [Accessed 10 September 2023].

[2] N. G. Mankiw, Principles of Economics, Boston, MA, USA: Cengage Learning, 2019.

[3] BCS, "When Computers Changed Banking," 03 September 2021. [Online]. Available: https://www.bcs.org/articles-opinion-and-research/when-computers-changedbanking/.. [Accessed 10 September 2023].

[4] Mozilla.org, "Mozilla Welcomes the Rust Foundation," Mozilla.org, 21 February 2021. [Online]. Available: https://blog.mozilla.org/en/mozilla/mozilla-welcomes-the-rust-foundation/. [Accessed 14 September 2023].

[5] Microsoft Security Response Center, "We need a safer systems programming language," Microsoft, 18 July 2019. [Online]. Available: https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/. [Accessed 14 September 2023].

[6] The Chromium Projects, "Memory Safety," Google, [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/. [Accessed 14 September 2023].

[7] S. K. a. C. Nichols, "The Rust Programming Language - Interactive Version," [Online]. Available: https://rust-book.cs.brown.edu/ch04-02-references-and-borrowing.html. [Accessed 14 September 2023].

[8] Stack Overflow, "2023 Developer Survey," Stack Overflow, 2023. [Online]. Available: https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages. [Accessed 14 September 2023].

[9] M. Contributors, "Cargo Github Repository," Th Rust Programming Language, [Online]. Available: https://github.com/rust-lang/cargo. [Accessed 17 September 2023].

[10 M. Thosmas, "Computers, People and the Real World," Gresham College, London, 2016.

[11 J. Pratt, "time Github Repository," [Online]. Available: https://github.com/time-rs/time. [Accessed 14 September 2023].

[12 B. W. O. D. S. K. S. E. &. D. P. Maister, "chrono (Version 0.4.30)," 2023. [Online]. Available: https://github.com/chronotope/chrono. [Accessed 14 September 2023].

[13 M. Authors, "Blessed - An unofficial guide to the Rust ecosystem," [Online]. Available: https://blessed.rs/crates. [Accessed 14 September 2023].

[14 Rust Infrastructure Team, "crates.io," Rust Programming Language, [Online]. Available: https://crates.io/.. [Accessed 14 September 2023].

[15 International Organization for Standarzation, "ISO 8601 Date and time format," [Online]. Available: https://www.iso.org/iso-8601-date-and-time-format.html.. [Accessed 14 September 2023].

[16 The Rust Programming Language Org, "Crate std," [Online]. Available: https://doc.rust-lang.org/std/index.html. [Accessed 14 September 2023].

[17 A. King, "Parse, don't validate," 5 November 2019. [Online]. Available: https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/. [Accessed 14 September 2023].

[18 M. Authors, "RustQuant Github Repository," [Online]. Available: https://github.com/avhz/RustQuant/tree/main. [Accessed 14 September 2023].

[19 M. Rainbow and J. Strong, "QuantMath Github Repository," [Online]. Available: https://github.com/MarcusRainbow/QuantMath/tree/master. [Accessed a4 September 2023].

[20 RustQuant, "RustQuant: a quantitative finance library written in Rust.," 17 May 2023. [Online]. Available: https://www.reddit.com/r/rust/comments/13jzcfj/rustquant_a_quantitative_finance_library_written/.

[21 Quantlib Org, "QuantLib: a free/open-source library for quantitative finance," Quantlib Org, [Online]. Available: https://www.quantlib.org/. [Accessed 15 September 2023].

[22 Quantlib Org, "Official QuantLib Documentation," Quantlib Org, [Online]. Available: https://www.quantlib.org/docs.shtml. [Accessed 15 September 2023].

[23 B. Nikolic, "Introduction to QuantLib and Using QuantLib Programmatically," Skills Matter, 2016. [Online]. Available: https://skillsmatter.com/skillscasts/9325-introduction-to-quantlib-and-using-quantlib-programmatically?tc=3f7216. [Accessed September 2023].

[24 A. Kirsh, "About C++ Dependency Management," 29 November 2021. [Online]. Available: https://www.incredibuild.com/blog/about-cpp-dependency-management. [Accessed 15 September 2023].

[25 Wikipedia, "IMM Dates," [Online]. Available: https://en.wikipedia.org/wiki/IMM_dates. [Accessed 15 September 2023].

[26 Chicago Mercantile Exchange, "Contract Month Codes," Chicago Mercantile Exchange, [Online]. Available: https://www.cmegroup.com/month-codes.html. [Accessed 15 September 2023].

[27 Swig.org, "What is SWIG?," Swig.org, 18 April 2019. [Online]. Available: https://swig.org/exec.html. [Accessed 16 September 2023].

[28 Open Gamma, "Strata Github Repository," Open Gamma, [Online]. Available: https://github.com/OpenGamma/Strata. [Accessed 16 September 2023].

[29 Open Gamma, "Open Gamma," Open Gamma, [Online]. Available: https://opengamma.com/. [Accessed 16 September 2023].

[30 Open Gamma, "Strata Documentation," [Online]. Available: https://strata.opengamma.io/docs/. [Accessed 16 September 2023].

[31 Department for Business and Trade of the UK Government, "The Consumer Credit Regulations 2010," 2010. [Online]. Available: https://www.legislation.gov.uk/uksi/2010/1970/regulation/4/made. [Accessed 19 September 2023].

[32 The Rust Programming Language, "Rust official documentation - Module collections," [Online]. Available: https://doc.rust-lang.org/std/collections/. [Accessed 23 September 2023].

[33 The Rust Programming Language, "Comparing Performance: Loops vs. Iterators," [Online]. Available: https://doc.rust-lang.org/book/ch13-04-performance.html. [Accessed 23 September 2023].

[34 H. contributors, "All About Monads," 19 September 2021. [Online]. Available: https://wiki.haskell.org/All_About_Monads#Maybe_a_monad. [Accessed 23 September 2023].

[35 The Rust Programming Language, "Rust Documentation - Module iter," [Online]. Available: https://doc.rust-lang.org/std/iter/index.html. [Accessed 23 September 2023].

[36 G. Hutton, Programming in Haskell, 2nd Edition, Cambridge: Cambridge University Press, 20116.

[37 The Rust Programming Language, "The match Control Flow Construct," [Online]. Available: https://doc.rust-lang.org/book/ch06-02-match.html. [Accessed 23 September 2023].

[38 The Rust Programming Language, "Writing Automated Tests - Test Organization," [Online]. Available: https://doc.rust-lang.org/book/ch11-03-test-organization.html. [Accessed 23 September 2023].

[39 M. Contributors, "nom github Repository page," [Online]. Available: https://github.com/rust-bakery/nom. [Accessed 19 September 2023].

[40 The Rust Programming Language, "Rust," [Online]. Available: https://www.rust-lang.org/. [Accessed 19 September 2023].

Figures

Figure 1. A business day count calendar. Still a common sight in many trading floors.

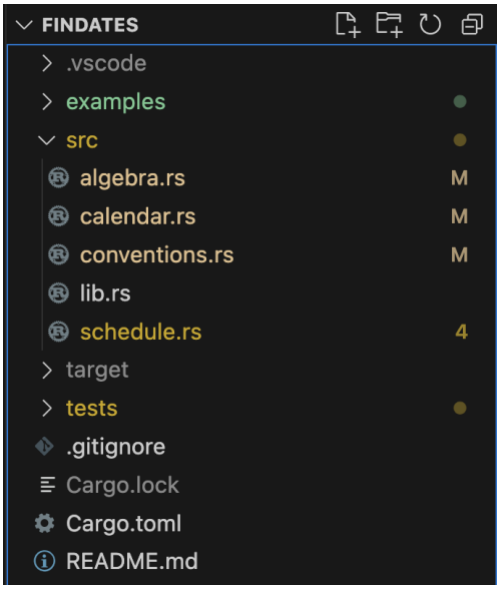

Figure 2, The folder structure for *Findates* generated by *Cargo.*

```rust
#[derive(PartialEq, Eq, Clone, Debug)]
5 implementations
pub struct Calendar {
    pub weekend:   HashSet<Weekday>,
    pub holidays:  HashSet<NaiveDate>,

}
```

Figure 3. The *Calendar struct* implementation

```rust
#[derive(Clone, Debug, PartialEq, Eq)]
5 implementations
pub struct Schedule<'a> {
    pub frequency: Frequency,
    pub calendar: Option<&'a Calendar>,
    pub adjust_rule: Option<AdjustRule>,
}
```

Figure 4. The *Schedule struct* implementation.

```rust
/// Check if a date is a good business day in a given calendar.
pub fn is_business_day (date: &NaiveDate, calendar: &Calendar) -> bool {
    if calendar.weekend.contains(&date.weekday()) {
        return false;
    } else if calendar.holidays.contains(date) {
        return false;
    } else {
        return true;
    }
}
```

Figure 5. is_business_day function implementation.