Edwin Stang

A Simple Expression Language for Automating Strategy Development Processes

Technical Report

Edinburgh Business School

Heriot-Watt University

Supervisors:
Hans-Wolfgang Loidl, Boulis Ibrahim

December 6, 2021

Contents

Abbreviations and Glossary

Term	Meaning
Broker	The financial institution (e.g. a bank) that manages the account equity and executes trades. This can be simulated for backtests in order to evaluate strategies without risking real money.
Strategy	An automated way to buy and sell on one or more financial instruments.
Backtest	A simulation of one or more strategies on historical data.
Instrument	A financial asset or derivative that has a price curve associated with it.
Bar	An aggregation of price ticks in the form of Open, High, Low, Close data points. Most commonly aggregated by timeframes but can also be aggregated by various price ranging algorithms.
Order	An instruction sent to the broker for executing a trade. It might be an entry or an exit order. Configs might be Limit, Stop, Market, ...
Indicator	A mathematical formula to modify a price series to make the information contained within measurable to derive actions from it.
sma	Simple Moving Average as a smoothing of prices.
atr	Average True Range as a smoothed volatility measure of price movements.
pow	Mathematical function to raise one number by the power of another number.
sin	Mathematical function to calculate the trigonometric sine of an angle.
Zig Zag	A lagging indicator to detect extreme points as peaks and valleys of a price curve.
Stop Loss	An instruction to the broker as a price at which a trade should be closed automatically with a loss.
Take Profit	Inverse of a Stop Loss which tells the broker to close a trade in a favourable outcome at a specific price.
Long	An order position profiting from a rising price. First buying to enter a position then selling to exit.
Short	An order position profiting from a falling price. First selling (e.g. via lending) to enter a position, then buying (and covering the debt) to exit.

1 Overview

This technical report defines a simple rule-based expression language for automating strategy development processes for trading financial instruments. The paradigm is declarative and functional.

The language is supposed to be simple enough to formalise building blocks (rules) for trading strategies (grouped rules) such as entry and exit conditions, as well as portfolio-level expressions for risk management, position-sizing, equity curve trading, strategy selection, and robustness testing. These expressions should be simple enough so that traders without programming experience have a low learning curve when trying to automate their decisions.

How is this simplicity achieved? The language is deliberately not Turing complete, and components need to be of limited resource consumption. It is designed as a domain-specific language that supports variables and functions. It supports no recursion, loops, arrays, lists, collections, maps, strings, exceptions, declarations, or assignments. Instead, non-recursive functions are used to emulate loops, mappings, decisions, and selections in a simplified fashion with a limited potential for coding errors. Functions can be nested arbitrarily as dynamic parameters. Thus a mathematical view of higher-order functions is supported as nested transformations on *double* series.[1] Though there is no notion of a functor, function object, or variable for a function in the language. Only the underlying implementation has that representation which is hidden from the user. Instead, the user can think in a series of *doubles* (data) which are provided and transformed by functions to be fed into other functions as parameters. This comes from the fact that the language operates on streamed financial data and is able to access previous data and calculation results. Since it operates solely on *doubles* and indexes, the function algebra is simplified significantly. The language makes the use of functions and variables transparent to the user by making parentheses optional.[2] Typical mathematical and logical operations are supported. Constants are represented as variables in the language. The language is case insensitive.

The language has only a single type *double* for inputs and outputs. Other types like *boolean* get encoded into *double* by seeing numbers above *0* as *TRUE* and all other values as *FALSE*. *Null* values are represented by the *double* value *NaN* which means "Not a Number".

[1] For example, a *double* series consisting of {1.1, 1.2, 1.3} is transformed via nested transformations: "*multiply(round(0), 2)*" into a new *double* series consisting of {2, 4, 6}.

[2] This is realized in the syntax. Variables can be suffixed with parentheses: "*variable()*" and "*variable*" works. Also, parentheses can be omitted for functions with only optional or no parameters: "*function*" and "*function()*" works.

The following chapters provide more detail:

- First the context in which the expression language is embedded is shown.
- Then the actions we want to perform with the expression language are defined.
- And then we explain how to realize that with the language definition in a *Backus Naur Form* with some extensions.[3]
- Afterwards we discuss implementation thoughts and alternatives.
- Lastly we analyse the performance results for the chosen implementation.
- A short conclusion rounds up the report.

1.1 Context

The expression language can be embedded in a trading platform that provides a context. This trading context provides a connection to a broker or simulates one. Both a real and a simulated broker can execute orders and provide data for time-series contexts. Also, the trading-related functions and variable implementations are provided by that platform.

The time-series contexts define a selection of an instrument and a bar aggregation formula. The instruments can come from different namespaces that differentiate between data sources. A default time-series context is given by a defined building block for a rule expression. Though the expression language also supports overriding this default to lookup values in different time-series contexts. This allows comparing multiple instruments and time periods for making decisions.

BarConfig, *OrderConfig*, and *Instrument* can have parameters depending on the given type. For example, a Renko bar config requires a price range expression. An order config for a conditional limit entry requires an open price expression. An instrument data source might be used that generates a randomized price curve from a given seed during robustness testing based on a real price curve provided by a broker connection. The trading platform defines these semantics. Since these details are not too relevant for the language definition, these details were simplified in the below model.

An instrument can provide alternative data *Schemas* besides prices. *Streams* can be extracted from these and aggregated as bars. Because of this, these alternative bars can then be used for calculations and plotting in the same way that price bars can be used by the expression language.

[3] This can be validated online via *BNF Playground* by Bobkov and Georges (2020).

Each trading strategy is among others defined by an entry (e.g. moving average crossover) and exit (e.g. stop loss, take profit, inverted entry) condition and runs in a simulated account. In that simulated account the trading strategy can have only one simultaneous trade. This simplifies the entry and exit expressions. Though the portfolio of those trading strategies can run either with a simulated or with a real broker account. Trades are copied from the simulated brokers of the individual trading strategies into the portfolio account for combined execution. Thus on a portfolio-level, multiple simultaneous trades are allowed.

Here is a context diagram of this:

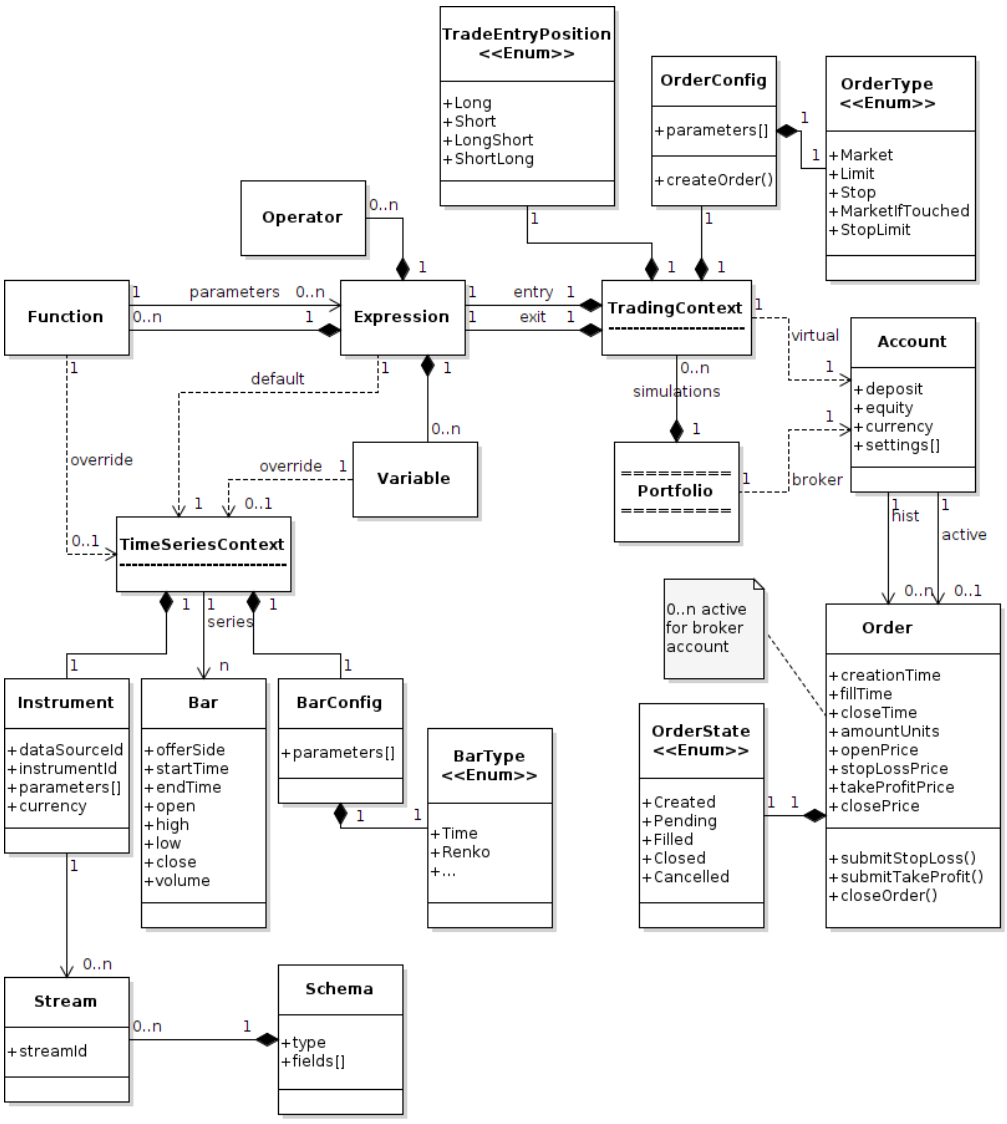

Figure 1: Expression Context Diagram

1.2 Stateless

The language is stateless, thus strategies defined in it are also stateless. Or rather the strategies could be seen as state machines that define all possible states of a trading account. Only the platform and an associated broker connection hold state and only those know how to execute state transitions. They provide the context for the language. The strategies defined in the language only pick a desired state based on the current and past data without caring how it will be realized. By not storing any state in the language (as arbitrary data), the language does not care about connection issues, trading pauses or order management details. The platform handles warming up of unstable periods (relevant for recursive functions like exponential moving average) for time-series calculations and to keep the input stream continuous without gaps (connection stability). These things make it easy to deploy such strategies into a cloud computing environment where a load balancer can move strategies between nodes without having to worry if the strategy state can be transitioned as well.

2 Building Blocks

The notation we are using here to describe the semantics of the expression language rules is:

- *Rule* := *Guard* -> *Action*

This can be translated into:

- *Building Block* is defined by *Boolean Expression* which triggers *Effect*

Here are some examples of entry and exit building blocks with example expressions:

- Entry := sma(25) > sma(5)
  -> OrderConfig.createOrder()
- Exit := !(sma(25) > sma(5)) || exitAtStopLoss(stopLossFromRange(atr(20)*2), true)
  -> Order.closeOrder()

This defines a simple moving average crossover strategy. When using a *TradeEntryPosition* as *Long* this strategy opens a trade that profits from rising prices when the condition is *true*. The exit condition stays out of the market when the inverse of the condition is *false* as defined by "*!(sma(25) > sma(5))*". This could also be written shorter in the exit condition by using the alias "*!entry*". Only one trade can occur, thus the entry condition is skipped by the strategy if a trade already exists. In that case, only the exit condition is evaluated for each arriving bar.

The exit condition says that the trade should be closed when the entry condition is *not true* (a reversal happened).

The exit condition also puts a safety stop-loss on the current order that is trailing (because the second optional parameter for that is *true*). This stop-loss function is creating a side effect in the expression. This is because *exitAtStopLoss* is normally not executed by the client-side trading platform but instead sent to the broker as an asynchronous update to the order parameters or depending on the broker as a new conditional order attached to an existing order. When the broker receives this information, he can act on price changes directly which reduces latency and thus slippage in comparison to evaluating this command on the client-side. Also, the broker can act on this information even if the trading platform is shut down or lost the connection to the broker. Thus, it is explicitly desired to have this kind of side effect in the language for this special command. Though if the broker does not support a stop loss or he can not be trusted with that information, the trading platform can fall back to a phantom stop-loss that is calculated on the client-side (with all its shortcomings and risks). The expression language does not care about such details, since it is the concern of the trading platform to guarantee the semantics of a stop loss in any way possible.

Here are more examples for such functions with side effects:

- exitAtStopLoss := exitAtStopLoss(stopLossFromRange(atr(20)*2), true)
  -> Order.submitStopLoss()
- exitAtTakeProfit := exitAtTakeProfit(stopLossFromRange(atr(20)*2), true)
  -> Order.submitTakeProfit()
- exitAtPriceTarget := exitAtPriceTarget(zigZagLastPeakOrValley(5))
  -> Order.submitStopLoss() || Order.submitTakeProfit()

One could argue they are sub-rules for rules with sub-actions for actions. If a platform provider wanted to keep the expression language free from side effects, these functions would need to be realized instead as independent (top-level) building blocks. Though for our specific trading platform it was found to be simpler and less restrictive to have these nested rules available. These functions with side effects require well-documented semantics and should only be created rarely for good reasons.

The price target is a shorthand for submitting a stop-loss or take profit to the broker depending on if the open price of the trade is higher or lower than the price target. In this case, it uses the most recent peak or valley based on the 5 percent zig zag indicator to find support or resistance price levels. This is also useful for strategies that are always in the market such as *TradeEntryPosition LongShort* or *ShortLong*. Here is an overview of these semantics:

- *Long*: Enter long when the entry condition is *true*. Exit when the exit condition is *true*.
- *Short*: Enter short when the entry condition is *true*. Exit when the exit condition is *true*. Essentially the inverse of *Long*.
- *LongShort*: Enter long when the entry condition is *true*, enter short when the entry condition is *false*, stay out when the entry condition is *NaN*/*Null*. Exit long trade when the exit condition is *true*, exit short trade when the exit condition is *false*, hold when the exit condition is *NaN*/*Null*.
- *ShortLong*: Enter short when the entry condition is *true*, Enter long when the entry condition is *false*, stay out when the entry condition is *NaN*/*Null*. Exit short trade when the exit condition is *true*, exit long trade when the exit condition is *false*, hold when the exit condition is *NaN*/*Null*. Essentially the inverse of *LongShort*.

The inverse options allow testing strategies easier without having to change the expressions. In some markets, patterns might be profitable if they are simply inverted.

With this, one can not only create dual-state strategies that are always in the market but also create tri-state strategies when explicitly returning *NaN*/*Null* to stay out of the market. Here is an example of such a strategy:

- Entry: if(sma(250) > sma(100), sma(25) > sma(5), NaN)
- Exit: !(sma(25) > sma(5)) || exitAtMarket(!(sma(250) > sma(100)))

With *TradeEntryPosition LongShort*, this would be a strategy that goes long and short only while the market is in an uptrend based on a longer simple moving average trend filter. The shorter reversal rule is based on a simple moving average crossover and reversed for the exit condition. We use the function *exitAtMarket* to exit both long and short trades independent of the above semantics when the given condition for the trend filter is *false*. This is another example of a function with side effects that improve the expression language. Normally it would be better to define this simple strategy as two separate long and short strategies. But there are more complex ones that are not easy or impossible to separate into two atomic strategies due to order states that can not be expressed by a formula. It might be interesting to also research this type of strategy and the trader should not be limited in his ideas by the trading platform.

In essence, the entry and exit conditions can be seen as similar to a logic programming language for pattern detection on financial time-series. Functions that transform the series are domain knowledge that is interpreted and combined via conditional operators to form the knowledge of something specific happening. This information is acted upon by the trading platform and other building blocks are used to manage further domain-specific knowledge that focuses on the lifecycle of strategies and trades. The prices are just replaced by other quantifiable event data.

2.1 Modularity

The expression language can also be used to calculate math. In that case, we are not talking about actions and guards, but instead, just return the *double* value to the building block that requests it. This flexibility can be used to realize other kinds of building blocks like portfolio-level money management, equity curve trading, risk management, and position-sizing. In those cases, it might even be interesting to let expressions interoperate not only with orders but also with instruments, strategies, fitness criteria, robustness measures, rankings, or filters. Some of these building blocks can be realized with top-level building blocks, others might be realized as functions with side effects. For this report, we will not go into full details on this since we don't expect that more language semantics will be required to formalise these building blocks. Though some examples for interesting concepts will be explained.

This form of modularization should be sufficient and allow us to keep the expression language as simple as possible. Otherwise, the expression would require semantics for namespaces, objects, modules, or functors. Instead, we let the trading platform add that context for the building blocks without the user having to learn any additional design patterns, syntax, or grammar. This also simplifies the usage of genetic programming. We only have to replace the metadata about functions and variables to generate expressions for any of these building blocks. Thus we can build a machine learning framework that can operate on higher levels of abstraction than just entry and exit conditions. The machine learning framework does not have to invent the logical structure of a trading strategy and portfolio-level decisions. Instead, it can just be plugged into these appropriately based on existing domain knowledge provided by the trading platform.

2.2 Extension Functions

The trading platform can provide another form of modularity by allowing a more complex language for defining functions as extensions. These complex languages (e.g. *Java*, *Kotlin*, *Scala*, *Groovy*, *Clojure*, *Eta*) then provide the full package of modularity and might even allow loading external and native libraries for complex mathematical operations. Though the expression language tries to be viable for the majority of use cases and could also be used for extension functions in the form of parameterized aliases for expressions. Though the expression language could also be embedded in more complex languages so that one does not have to resort back to coding loops where they are not needed. For this, an expression factory could be made accessible to the extension function code in different languages.

To make the expression language more versatile, there are some basic functions that are similar to aggregate or scalar functions in the *Structured Query Language* (*SQL*). Basic functions known from functional languages (e.g. *Haskell*) like *map*, *fold*, or *filter* are not supported because they operate on lists, collections, and sets. The only collection that is available in this expression language is the time-series and derivative series (created by transformations via functions).

Here are some examples of these basic functions:

- **isNaN(value):** Missing values can be determined with this to do limited exception case handling.
- **if(condition, whenTrue, whenFalse):** This can be used for simple decisions to switch between two alternatives.
- **map(index, option1, option2, . . . , optionN):** This can be used for decisions to switch between multiple alternatives.
- **vote(threshold, condition1, condition2, . . . , conditionN):** This becomes true when the threshold % of conditions returned true is exceeded. This can be used to create ensemble strategies.
- **once(condition):** This checks that we switched from false to true, thus making successive true values false. The condition series "*{false, true, true}*" will be transformed into "*{false, true, false}*" by this function.
- **stable(condition, lookback):** This checks that the condition is true over a given lookback. This represents a historical *AND*. The condition series "*{false, true, true}*" would be transformed into "*{false, false, true}*" when a lookback of 2 is used.
- **stableCount(condition, lookback):** This is the same as *stable*, just that it returns the number of consecutive true occurrences in the past starting from the current index.
- **occurs(condition, lookback):** This returns true if the condition has been true at least once over the lookback indexes. This represents a historical *OR*. The condition series "*{true, false, false}*" would be transformed into "*{true, true, false}*" when a lookback of 2 is used.
- **occursCount(condition, lookback):** This is the same as *occurs*, just that it returns the number of occurrences in total without checking for successiveness.
- **lastIndexOf(condition, lookback):** This can be used to create dynamic previous value lookups (for an *[index]* suffix operator) based on the latest index when a condition was true in the series.

- **firstIndexOf(condition, lookback):** The same as *lastIndexOf*, just that it looks as far into the past as the lookback allows.
- **previousIndexOf(condition, steps, lookbackPerStep):** This repeats the *lastIndexOf* for multiple steps to get the index of an occurrence further back in time. If a step has no occurrence, *NaN*/*Null* is returned.
- **latestIndexOf(condition, steps, lookbackPerStep):** This repeats the *lastIndexOf* for multiple steps to get the index of an occurrence further back in time. If a step has no occurrence, the latest index of the occurrence before that step is returned. *NaN*/*Null* is only returned if there was no occurrence over all steps.

The functions *stable*, *stableCount*, *occurs* and *occursCount* also offer two variations with the suffix "*Left*" and "*Right*" where a comparison condition is evaluated and the lookback is only applied to the left or right side of the comparison while the other side is kept constant. These can be used for expressions like: "*stableLeft(close[1] < close[0], 20)*" which checks that the previous 20 prices on the left side "*close[1]*" were lower than the current price on the right side "*close[0]*". "*stable(close[1] < close[0], 20)*" on the other hand would check that the last 20 successive prices were rising. "*stableCount(close[1] < close[0], 20) = 5*" would check the last 5 prices in a window of 20 previous indexes were rising. "*occurs(close[1] < close[0], 20)*" would check that the price was rising at least once in a window of 20 previous indexes. "*occursCount(close[1] < close[0], 20) = 1*" would check that the price was rising exactly once in a window of 20 previous indexes. "*occurs(stable(close[1] < close[0], 5), 20)*" would check that 5 consecutive prices were rising once in a lookback of 20 previous indexes. As demonstrated here, these functions can be used to capture fuzzy patterns where conditions don't have to occur at the same time, but near to each other in a specific lookback tolerance.

It might be interesting to research further basic functions that would enable us to detect patterns on filtered extreme points of the underlying price curve. The indexes would then not represent times of prices, but successive extreme points. With that, it could be possible to capture more complex patterns that are easy for the human eye, but hard for computer detection. For example head and shoulders patterns (Chakerian, 2019). Or one could provide *advise* functions for nested machine learning, statistical, probabilistic, and fuzzy matching (oPgroupGermanyGmbH, 2020). Extension functions like these could be implemented in general-purpose programming languages and made available to the expression language.

2.3 Strategy Filters

Imagine a machine learning technique that generates entry and exit conditions to be evaluated as strategy candidates. Out of these, we want to create a portfolio. For this, we could define a building block that filters these strategy candidates based on fitness criteria. This process could be divided into the following steps:

1. **Generate Candidates:** The machine learning algorithm would generate candidates as fast as possible. Thus it should calculate only one highly optimised fitness function on the in-sample period. For example with genetic programming a population of *500* strategies would compete for selection as a valid candidate. A valid candidate produces the best local fitness value by making at least one trade. If zero trades were valid, then the machine learning algorithm would pick not to trade, because a fitness value of 0 is higher than any negative fitness it can come up with initially. We have to eliminate that so the evolution using crossover and mutation develops a population of trading strategies that rise into positive fitness values. One could suggest filtering strategies that exceed a specific maximum drawdown here (or another threshold), which could be included in the fitness function. But instead, we add a second step for advanced filtering.

2. **Filter Candidates:** The best strategy from the population is considered as a candidate. For this candidate, the process would also calculate the out of sample period. The statistics calculated for this step would be elaborate and slower to calculate but also used significantly less often than the fitness measure of step 1. Thus we would collect a daily equity curve and details about the trades so that we can add variable and function providers to the expression context of this building block. These would then lazily calculate the fitness measures and could be mathematically combined. This expression is then used to discard candidates. Step 1 is then repeated until the desired number of filtered candidates is generated. For example *1000* candidates per instrument or basket of instruments. For baskets, the statistics for the different instruments would be combined into one result per candidate and thus *1000* candidates overall. For individual instruments, the whole process would be separated from the outside. Thus we would have *1000* candidates per instrument. An example expression would be:
   "*profitFactor > 1.2 && maxDrawdownPercent <= 30*"

3. **Create Portfolio:** When a list of the desired candidates is collected, the candidates could be ranked based on a mathematical expression on fitness measures and filtered based on the actual rank. This would reduce the candidates to the *10* best ones.
   An example expression is: "*rank(sharpeRatio*sortinoRatio) <= 10*".
   The rank function is special because on this step the expression is again evaluated per

candidate, but the rank also knows about the list and finds out which rank index the current candidate has based on the list which is sorted based on the fitness parameter. Thus each candidate that has a rank greater than *10* is discarded. All surviving candidates are added to the portfolio.

Steps 1 and 2 can happen in parallel and for multiple instruments. Step 3 could also be parallelized if multiple portfolios should be combined into a larger portfolio with maybe more filtering steps. Though there is no need to write every step as a separate expression, instead we can combine the semantics of Steps 2 and 3 into a single expression: "*profitFactor > 1.2 && maxDrawdownPercent <= 30 && rank(sharpeRatio*sortinoRatio) <= 10*". The expression language does not care about how the trading platform implements the steps or when it parallelizes them. When we are at step 2, the rank function will return *NaN*/*Null* because there is no list of candidates available. Since the expression language treats *NaN*/*Null* as a missing value, the comparison "*NaN <= 10*" results in *NaN* and the boolean combination with that skips that part of the expression and evaluates only the left part. Step 3 would be able to evaluate the full expression for each candidate.

2.3.1 Fitness Variations

We could also add variations of the statistics as variable providers for long/short, different position sizing algorithms (e.g. equity, minimal risk, default risk, volatility adjusted) , and different measures of profit (e.g. pips, percent movement captured, and equity). Though this should be done only for simple variations. Complex position sizing rules should have their own building block and testing process based on portfolios instead. The variations could be added as variable providers with appropriate suffixes to the building block. For example:

> *profitFactorPerUnitPips > 1.2 && maxDrawdownPercentLong <= 15 && maxDrawdownPercentShort <= 15 && rank(sharpeRatioPerDefaultRiskLong * sortinoRatioPerDefaultRiskLong) <= 10 && rank(sharpeRatioPerDefaultRiskShort * sortinoRatioPerDefaultRiskShort) <= 10*

Not all variations might be useful, but they provide the flexibility to filter strategies both with and without money management included in the fitness measures at the same time and to separate measures per long/short if desired. When multiple ranks are used successively, they are treated independently and at the same time for each candidate. Thus the last example could result in less than *10* portfolio components due to using rank twice. This is because its *rank* was implemented to not cause side effects. Each candidate is checked individually in the list, thus the ranks of candidates stay constant between evaluations. The invalid candidates are then removed afterwards. Thus the above expression filters for candidates that are under the top *10*

for both long and short. To keep candidates that are either in the top *10* on long or short, one could write:

rank(sharpeRatioRiskLong) <= 10 || rank(sharpeRatioShort) <= 10

This might result in more than *10* portfolio components. To keep the limit of *10* portfolio components one could use a mathematical expression instead:

(rank(sharpeRatioLong) + rank(sharpeRatioShort)) / 2 <= 10

Or combine both into one *rank* function call:

*rank(sharpeRatioLong * sharpeRatioShort) <= 10*

2.3.2 Sample-Selector Prefixes

Prefixes could be added to selecting the in-sample, out-of-sample, or even a validation-sample. A validation-sample might introduce lookahead bias during walk-forward analysis. This makes it unsuitable for automation, but it might be useful for user interfaces for deeper analysis to come up with better filtering expressions. The default without a sample prefix would be the out-of-sample result as above. Here is an example of explicit prefixes:

inSample_profitFactor > 2 && outSample_profitFactor > 1.2

2.3.3 Portfolio Context

For baskets, it would also be possible to define context prefixes for selecting portfolio components: "*INSTRUMENT@ALGORITHM:profitFactor > 1*". Or to combine the results over all portfolio components based on an individual instrument: "*INSTRUMENT:profitFactor > 1*". Or algorithm: "*ALGORITHM:profitFactor > 1*". For the individual instrument, all algorithm results will be combined into one. For the individual algorithm, all instrument results will be combined into one. This shows that the *TimeSeriesContext* can also be implemented as a *PortfolioContext* instead.

This building block is not based on a series of data points like the entry and exit building blocks, thus historical and lookback based features like the functions *once*, *lastIndexOf* or *stableCount* will be disabled here and would cause parsing errors. The user would see these errors in the expression editor of his user interface.

This context could also be used to reference values from benchmarks like buy and hold or randomized signals and price curves. An example that filters strategies that beat a benchmark would be: "*BUY&HOLD:profitLoss < profitLoss*".

2.3.4 Complex Fitness

Complex fitness functions can be used for ranking which incorporate weights, cokurtosis, coskewness, correlations, covariances, and so on. They can not only look at individual candidates, but the interplay and statistics between candidates as well. Examples functions are *whitesRealityCheck*, *markowitz*, *optimalF*. They can even be process-related to provide statistics about nested optimisations. Examples are *parameterStability*, *walkForwardEfficiency*

2.4 Nested Optimisation

Nested optimisation is the process of finding optimal parameters for functions within strategy candidates. It is nested inside the outer optimisation process of finding the best strategy candidates by finding optimal parameters for the entry and exit building blocks. The outer optimisation process can use machine learning techniques like genetic programming to combine simple expression blocks into more complex strategy entry and exit rules.

An illustrative example for a nested optimisation would be a simple reversal strategy that compares a slow and a fast exponential moving average (ema) for entry:

- Entry: ema(25) > ema(5)
- Exit: !entry

It exits on the inverse entry condition to immediately enter another trade in the opposite direction. Thus the strategy always stays in the market.

How does one know if the values 25 and 5 are optimal values for a given market? The simplest approach would be to try out different values and see which performs the best (based on some fitness score) and use the best values going forward. To automate this, we have a special "*optimize(selected, start, end, increment)*" function to define the values that should be evaluated:

- Entry: ema(optimize(25,5,50,5)) > ema(optimize(5,2,20,3))
- Exit: !entry

This will define a problem space for the entry signal that tries the values {5, 10, 15, 20, 25, 30, 35, 40, 45, 50} for the first optimisation parameter and {2, 5, 8, 11, 14, 17, 20} for the second optimization parameter. This could also be written as "*optimizeValues(selected, value1, value2, . . . , valueN)*" using a variable arguments function:

- Entry: ema(optimizeValues(25, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50)) > ema(optimizeValues(5, 2, 5, 8, 11, 14, 17, 20))
- Exit: !entry

The surrounding platform is then able to scan the expressions for optimisation values and optimise each parameter one after the other in the given order of definition. Changing the order of definition might change the result. Trying out all combinations instead would result in 10*7=70 permutations to try, instead of 10+7=17 permutations when optimising one parameter after the next. In machine learning scenarios it is preferable to use the variation with fewer permutations. Thus more strategy candidates can be tried instead of wasting computation cycles on finding the best parameters for each candidate. Even though optimisation parameters are not independent, iterating over less permutations can reduce curve fitting bias. The same goes for manually defining the increments granular enough so that not too many permutations are created for a given optimisation parameter.

The actual optimisation can happen before (pre) or after (post) the machine learning algorithm combines the simple expressions into more complex strategy entry and exit rules.

2.4.1 Pre Optimisation

The selection is made on individual generator blocks before the machine learning algorithm combines them into expressions for testing strategy candidates. No nested optimisation loops happen here, instead one action follows the other. Thus we only add a bit of computational overhead to the process. Each generator block is a boolean condition like the above exponential moving average crossover that can be combined with another condition using an *AND* operator: "*condition1 && condition2 && condition3 && condition4*". Only when all conditions are true, the entry is performed. Normally one would use generators that combine 1-5 entry conditions to generate strategy candidates. This is how optimisation parameters could be treated on an individual condition level:

- *None*: Use the preselected value, thus disable optimisations entirely.
- *FirstValue*: Always pick the start value as the selected value for any optimisation parameter.
- *MiddleValue*: Always pick the middle value between the start and end values as the selected value.
- *LastValue*: Always pick the end value as the selected value.

- *InSample*: Find the best performing value in-sample which is also used by the machine learning algorithm.
- *OutSample*: Find the best performing value out-sample which is invisible to the machine learning algorithm and used to confirm results.
- *InOutSample*: Find the best performing value in the whole sample that is used by the machine learning algorithm to confirm the results.

The in-sample and out-sample split can be configured as a percent split (e.g. 30% for out-sample) of the whole sample and a decision can be made if the out-sample should be before or after the in-sample. One could argue that using the out-sample in front would be better for live trading because the optimisation will include more recent data. If the out-sample is at the end, trading afterwards might already be using outdated parameters and work profitably for a shorter period. This hypothesis can be tested by an outer walk forward analysis of the machine learning algorithm by comparing both options with varying walk-forward periods. *None*, *FirstValue*, *MiddleValue*, and *LastValue* can be used to verify if the given strategy generator or selection process is sensitive to optimisation.

2.4.2 Post Optimisation

First, the machine learning algorithm combines the simple expressions blocks into strategy candidates. Then the individual strategy candidates are optimised. This happens as a nested optimisation loop inside the machine learning loop, thus the computational effort is multiplied inside the process. Afterwards the optimised results are used to make decisions on the next evolutionary steps for the strategy candidates. When the evolutions are finished, the best candidates are given to the strategy filter module. Since this mode of optimisation is nested within the machine learning loop, only the in-sample is available with the following types:

- *None*: Skip post optimisation.
- *InSample*: Find the best result inside the in-sample and use that for the fitness value of the strategy candidate.
- *WalkForward*: Split the *InSample* into multiple walk forward steps and repeat the optimisation for each iteration. Concatenate all out-sample walk forward steps in order to create one large out-sample result and use that for the fitness value. When using this type, additional statistics for walk forward efficiency or parameter stability could be made available to the strategy filter module.

- *CrossValidation*: Similar to *WalkForward*, just that the optimisation sample does not have to be before the out-sample step. Thus we optimise on each sample and do an out-sample check on all other samples. This test is more thorough with the available data, but can not be replicated in live trading where the future is unknown.

For more complex strategies it might be useful to define an optimisation parameter only once and reuse that value for multiple inputs. We could represent that as aliases as fake variables using a separate expression before the actual logical combination. Thus introducing multi statement expressions. By changing the order of the optimisation functions one could also influence the order of optimisation. Here is an example for such an entry expression:

var fast = optimizeValues(5, 2, 5, 8, 11, 14, 17, 20);
var slow = optimizeValues(25, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50);
ema(slow) > ema(fast) && ema(slow) > close

Here the fast optimisation would occur before the slow optimisation in comparison to the previous examples where it was the other way around. Also, we reuse the slow variable for a second condition by binding it as a parameter. The first two *var* statements which separate the expression using a semicolon define aliases for *fast* and *slow*. If the actual expression in the third statement does not use a given alias, the optimisation can be skipped for it.

This shows that even modules for complex workflows can be built with this language design.

3 Language Definition

Here we discuss semantics and syntax of the language. As a reminder, an expression can be seen as a *Guard* from which an *Action* follows:

- Rule := Guard -> Action
- *Building Block* is defined by *Boolean Expression* which triggers *Effect*
- Entry := sma(25) > sma(5) -> OrderConfig.createOrder()

The *Rule* and *Action* parts are not defined in the syntax of the language, instead they are defined as semantics by the platform context where the language is implemented in. The *Action* (or *Effect*) is determined by the *Rule* (or *Building Block*) of the surrounding platform after evaluating the return value of the *Guard* (or *Boolean Expression*). In this case the platform configuration might define that the default *Action* is to enter a long market order when "*OrderConfig.createOrder()*" is used. The default *Action* also requires that the *Boolean Expression* returned *true* and that no *Side Effect* was triggered that entered a different order. These *Side Effects* can override the default order configuration and trigger *Actions* for creating different orders within their implementation. The *Guard* for a function with *Side Effects* is based on the *Boolean Expression* position it is embedded into:

- Entry := sma(25) > sma(5) && enterLongAtMarket()

"*enterLongAtMarket()*" is a function that executes the platform *Action* "*OrderConfig[Market, Long].createOrder()*" when the slow simple moving average is above the fast simple moving average ("*sma(25) > sma(5)*"). There are specialised functions that allow to enter other order types. For example "*enterShortAtLimit(close[1])*" executes the platform *Action* "*OrderConfig[Limit, Short].createOrder(limitPrice)*" with *limitPrice* being the previous close price. Some functions also define optional parameters for boolean conditions that behave as an additional *Guard* for a *Side Effect*:

- Entry := enterLongAtMarket(close[1] > close[0])

Both *Guards* can also be combined:

- Entry := sma(25) > sma(5) && enterLongAtMarket(close[1] > close[0])

This will enter a long market order only when the slow simple moving average is above the fast simple moving average ("*sma(25) > sma(5)*") and the current close price is above the previous close price ("*close[1] > close[0]*").

The language definition is thus only concerned about how to define an expression as a *Guard*. The *Guard* thus becomes a program that can consist of zero or more aliases and one expression

that defines the return value of the program. This can be defined in the meta language of the Backus Naur Form as:

```
<program> ::= (<alias>)* <expression>
<alias> ::= "var" <variable> "=" <expression> ";"
```

Aliases and the return value of the program consist of expressions.

```
<expression> ::= <function_or_variable>
  | <number>
  | <expression> <operators> <expression>
  | <grouped_expression>
  | <negated_expression>
```

Grouped expressions allow parentheses to be put around expressions to override default mathematical and boolean operation evaluation priorities (e.g. multiplication before addition).

```
<grouped_expression> ::= "(" <expression> ")"
```

Functions and variables can be surrounded by whitespace which is ignored by the parser and not expressed in the grammar. We expect the expression string to be tokenized already here to simplify the language definition. The language also supports comments, though these are omitted here as well for the sake of simplicity.

```
<function_or_variable> ::= <function>
    | <variable>
    | <ts_context_lookup>
    | <previous_lookup>
```

Functions can have expressions as parameters that are separated by a comma. Though optional parameters can be omitted, a default value might get applied then by the function implementation. Parameters can also be defined as variable arguments (*VarArgs* or variadic functions) in the metadata, the function then supports an unlimited number of parameters. Functions and variables are matched case insensitive. Examples for constant variables are "*PI*", "*NaN*", "*TRUE*" or"*FALSE*". Examples for dynamic variables are "*time*", "*close*" or "*random*". Logical functions might be "*if(condition, whenTrue, whenFalse)*" or "*select(random(1, 2), whenOne, whenTwo)*".

```
<function> ::= <terminal> "(" <function_parameters> ")"
<function_parameters> ::= <expression> ( "," <expression> )*
<variable> ::= <terminal>
```

Even though we speak of variables and functions, these can not be defined, declared, or assigned inside the expression language. They are just hooks into the domain-specific language implementation of the surrounding platform. In this case a trading platform. With each bar in the time-series iterated, the expression is evaluated and the variables and functions will calculate new values when requested. Or it accesses transparently cached values which were already requested by other expressions or prior evaluations.

One can access previous elements directly in the price series (which is the basic underlying data type) via an index suffix. The index suffix *[0]* will access the current value while *[1]* will access the previous value up to arbitrary historical lookups *[n]*. "*n*" can also be an expression instead of a constant for dynamic lookups. The expressions are executed per bar during a backtest or live execution. The current reference index or time for *[0]* is set in the surrounding trading context. Thus *close[0]* references the most recent close price, while *close[1]* references the previous close price for an instrumented as provided by the time-series of price bars. If the function is a parameter of another function, the time index on it is shifted by the time index of the outer function. Thus the lookup is sensitive to function nesting. *Integer* conversion for indexes is done transparently by a simple truncating cast in the implementation. A negative or *NaN*/*Null* index will cause the indexed function or variable to return *NaN*/*Null*. In that sense, *close[-1]* would reference a future value that is not allowed to prevent lookahead bias.

```
<previous_lookup> ::= <function_or_variable> "[" <expression>
    "]"
```

The typical mathematical and boolean algebra operators are defined as well with their usual precedence rules. *NaN* in boolean algebra (*AND*, *OR*, *NOT*) is interpreted as a neutral value that is ignored in favor of another existing value. One can override this by wrapping the potentially missing value in an *isNaN(...)* function to convert it into an existing value of *TRUE* or *FALSE*. In comparison operators, *NaN* makes the comparison result *NaN*. Also, a division by *0* results in *0* instead of *Infinity* to also become a neutral operation. This deviates from the standard behavior in *Java*, which implements a subset of the IEEE 753 standard for floating-point numbers. We follow the standard by making any mathematical operation (plus, minus, divide, multiply, ...) with *NaN* result in *NaN*. The deviations from the standard have some benefits for trading strategies by reducing the number of exception cases from missing values in time-series evaluation. This simplifies the expression language again. Function implementations in Java are unaffected by these simplifications and will still use normal mathematical rules so that matrix operations and indicators can be calculated in classical ways.

```
<operators> ::= <math_operator>
    | <boolean_operator>
    | <equal_operator>
    | <compare_operator>
    | <crosses_operator>
<math_operator> ::= "+" | "-" | "*" | "/" | "^"
<boolean_operator> ::= "&&" | " and " | " AND "
    | "||" | " or " | " OR "
    | " xor " | " XOR "
    | " pand " | " PAND "
    | " por " | " POR "
<equal_operator> ::= "==" | "=" | "!=" | "<>" | "><"
<compare_operator> ::= ">" | ">=" | "<" | "<="
```

Parallel boolean operators *PAND*/*POR* produce the same result as *AND*/*OR* except that lazy (short-circuit) evaluation is disabled. This is useful when combining multiple functions with side effects where it is desired that no evaluation is skipped. For example in a dual entry strategy "*filterLong && enterLongAtMarket() PAND filterShort && enterShortAtMarket()*" lazy evaluation (with *AND* instead of *PAND*) would skip entering a short position when "*filterLong=false*" and "*filterShort=true*". The same would happen with "*filterLong && enterLongAtMarket() POR filterShort && enterShortAtMarket()*" on lazy evaluation (with *OR* instead of *POR*) when "*filterLong=true*" and "*filterShort=true*". Eager (parallel) evaluation will evaluate the short entry despite knowing that the overall result will be false. The return value does not matter in this case since the platform will not enter additional positions when a position has been entered by a side effect during an expression evaluation.

Negated expressions provide a "not" operator which can invert boolean results. When boolean operators are used on numbers, they get converted to boolean automatically.

```
<negated_expression> ::= "!" <expression>
```

An additional domain-specific operator family is the "*crosses*" definition. This is a shortcut for a comparison with the previous value. For example: "*ema(25) crosses above ema(5)*" translates to "*ema(25)[0] > ema(5)[1] && ema(25)[1] < ema(5)[0]*".

```
<crosses_operator> ::= " crosses above " | " CROSSES ABOVE "
    | " crosses over " | " CROSSES OVER "
    | " crosses below " | " CROSSES BELOW "
    | " crosses under " | " CROSSES UNDER "
```

Terminal symbols for functions and variables consist of letters. We are calling these terminal symbols since they are defined by the trading platform's metadata. Only the trading platform

might provide mechanisms to let the user define custom functions and variables outside of the expression language. The user could then create aliases for complex expressions as independent metadata. Or the platform could allow coding in a different programming language to be plugged into the expression language via metadata.

```
<terminal> ::= ( <letter> )+
<letter> ::= [a-z] | [A-Z]
```

Constant numbers are supported with decimal points. Other representations for numbers like scientific notations and shortcuts for units are supported but also omitted here to keep the description simple.

```
<number> ::= <positive_number> | <negative_number>
<positive_number> ::= ("0" | [1-9] [0-9]*) ("." [0-9]+ )?
<negative_number> ::= "-" <positive_number>
```

Functions and variables can be prefixed with a time-series context. This time-series context can be used to reference a value from a different instrument and bar type. The default is defined in the surrounding trading context. Other price series are aligned to the default price series so that lookahead bias is prevented and to make the previous element lookup *[n]* work time-series context-sensitive. An example is: "*JFOREX:EURUSD@Time[1 HOUR]:ema(25)*". This references the instrument "*EURUSD*" from the data source "*JFOREX*". It aggregates the prices into one-hour time bars via the bar config "*Time[1 HOUR]*". Then applies an exponential moving average on the close prices of the last 25 bars via the function "*ema(25)*". The language also supports time-series contexts defined as a suffix using an "of" operator: "*ema(25) of JFOREX:EURUSD@Time[1 HOUR]*". These two styles of time-series context definitions are the common ones used by other trading platforms that support similar expression languages.[4]

```
<ts_context_lookup> ::= <ts_context> ":" <function_or_variable>
    | <function_or_variable> " of " <ts_context>
<ts_context> ::= ( <instrument_id> )? ( "@" <stream_id> )? (
   "@" <bar_config_id> )?
```

The time-series context is defined by an optional "*data_source_id*" which can be used to identify a broker connection in combination with an "*instrument_id*". Here is an example without the optional parts which looks up the data sources by a platform-defined priority and uses the default bar config: "*EURUSD:ema(25)*". Some data sources support configuration

[4]This expression language tries to be compatible to allow copy/pasting of strategy entry and exit conditions with minimal modifications. To make this easier, functions and variables also support alias names, suitable default values for optional parameters, and varying parameter sequences in some cases.

parameters for their instruments. For example to generate random samples from an underlying instrument price series suitable for robustness testing.

```
<instrument_id> ::= ( <data_source_id> ":" )? <terminal>
    ( ":" )? ( "[" <context_parameters> "]" )?
<data_source_id> ::= <terminal>
```

An instrument can provide multiple alternative data streams besides the default price stream. Alternative streams are represented as bars extracted and aggregated from a schema. A schema is a custom data type definition that makes it suitable for storage. Examples are:

Schema	**StreamIds**
Sentiment	LongSentiment, ShortSentiment
Open Interest	ShortInterestPercent
Sustainability	SustainabilityScore
Carbon Footprint	CarbonPrice, CarbonEmission
Level 2	AverageAskPrice, MinAskPrice, MaxAskPrice
Order Book	PendingLongOrdersCount, AverageStopLossLong
Fundamentals	EarningsPerShare, NetProfit

A stream can be referenced as: "*JFOREX:EURUSD@LongSentiment@Time[1 HOUR]:close*" to get the latest percent pressure of trades into the long direction (if the data source provides that for the given instrument). A shorter version is: "*@LongSentiment:close*" which references the instrument and bar config from the default time series context.

```
<stream_id> ::= <terminal>
```

The bar config consists of a bar type with parameters. The parameters are separated by a pipe symbol instead of a comma to make parsing easier. This allows us to use nested expressions inside bar config parameters. An example might be: "*PointAndFigure[atr(20)*2 | 5]*". This defines a Point & Figure bar with two parameters: First a dynamic price range of an average true range based on a 20 bars lookback multiplied by 2. Then the reversal box amount is given as 5 bars.

```
<bar_config_id> ::= <terminal> "[" <context_parameters> "]"
```

This is the definition of the time-series context parameters:

```
<context_parameters> ::= <context_parameter> (
    "|" <context_parameter> )*
<context_parameter> ::= <expression>
```

3.1 Bounded Resource Consumption

By not supporting recursion and loops, the expression language can not produce logic that runs infinitely. This provides a guaranteed termination of expression evaluations and makes the resource consumption bounded. The function implementations need to be careful to keep the promise of guaranteed termination and bounded resource consumption.

For example, a function that applies a machine learning concept should handle logical and mathematical exception cases gracefully and end with a neutral value (e.g. *NaN*) if it does not converge to a solution in an acceptable time. An acceptable time is defined by the time-series context which has time gaps between bars. Each bar should be able to lead to a decision. A simple way to enforce this by the trading platform is to abort calculations after a timeout via an interrupt (when a new bar arrives). Though for this the function implementations need to check for interrupts in internal loops and return *NaN* in that case.

This makes it suitable to run expressions on a shared server infrastructure without the risk of one expression compromising the service level of other expressions on a node. This also prevents a genetic programming approach from creating slow-performing expressions. This makes testing trading strategies more efficient. On the other hand, this makes the language not Turing complete, thus many basic or highly complex functions need to be implemented in an underlying general programming language.

An optimisation that can be used inside a node is to cache function results if they are based on indicators. Thus when one expression requires the value of an indicator, it will only be calculated once on that node for a given bar. The result will be shared between all expressions that require this value. This approach could speed up both live trading with a limited lookback period as well as genetic programming tasks with a fully cached indicator series. This also requires considering load balancing of data storage, computational and memory limits with smart batching of tasks between nodes. But these infrastructural optimisations are out of the scope of this report and will be discussed in the following research. We only look at the optimisations for parsing and evaluation of the expression language for now.

4 Implementation

The underlying implementation is based on *Java* and implements optimisations like simplifying expressions and lazy (short-circuit) evaluation of boolean algebra. The expression string parser is optimised for performance. Though expressions can also be operated on as objects to provide an efficient way to generate expressions during machine learning without wasting time on string parsing.

The expression parser is based on *Parsii* by Haufler (2020b). It was forked as a major rewrite to extend it for time-series indexing, to fix some bugs, improve reusability/separation between expression types, and do further performance optimisations. The time-series indexing can be done using date objects for live trading and primitive integers for faster in-memory testing. Though for simple mathematical calculations, the expression can be called without any index. Beware that time-series features will not be available in that mode. The fork/rewrite is available in the *invesdwin-util* project by Stang (2020b). There, one can see the full language definition in the parser code.

Graphical user interface components in the form of an expression editor based on *RSyntaxTextArea* by Futrell (2020) and a charting component based on JFreeChart by ObjectRefinery-Limited (2020) is made available in the *invesdwin-context-client* project by Stang (2020a). The user experience of the charting component is inspired by TradingView (2020) which is a *HTML5* charting library. Features like lazy data loading during scrolling, mouse and keyboard interactions, a dialog for adding indicators, drag and drop support for chart elements, and customization of chart elements have been implemented on top of *JFreeChart* to make it more similar to *TradingView*. Also, the expression editor has been integrated to add custom expressions as dynamic indicators for quick and easy visualization.

The integration with a trading engine - which adds actual financial indicator function providers, contexts, and bar configs - is currently closed source but might be opened up at a later time. The expressions are inspired by and compatible to a high degree with other tools like *BuildAlpha* (Bergstrom, 2020), *GeneticSystemBuilder* (Zwag, 2020), *Adaptrade* (Bryant, 2019) and StrategyQuant (2020). Improving interoperability and compatibility is a continuous development process.

Custom indicators could be provided by the user through storing expressions as named functions or by coding more complex indicators in a different scripting language in the *Java Virtual Machine*.

4.1 Alternatives

Expression languages that might have been suitable as a basis for this implementation and which were used for performance comparisons during evaluation include:

- **Parsii:** *Parsii* claims to be one of the faster (basic mathematical) expression parsers available (Haufler, 2020a). With some rudimentary testing, it was possible to confirm these claims to some degree. Additionally, because of the simplicity of the implementation, it was decided that it is a suitable basis for defining a new expression language designed for trading applications.
- **Spring:** The *Spring Expression Language* (VMwareInc, 2020) supports mathematical expressions and is simple enough to extend and integrate. Though the parser is based on *Java* reflection which makes it slower compared to other solutions.
- **Groovy:** *Groovy* has extensive support for defining domain-specific languages on top of it (ApacheGroovy, 2020) and provides mathematical expressions out of the box. With static compilation and checked types enabled, it can also execute expressions faster than other scripting languages inside the *Java Virtual Machine*.
- **Janino:** *Janino* is a simple API to dynamically compile *Java* code and use it at runtime (Unkrig, 2020). By being compiled into *Java* byte code it executes code at native speed. Also, each expression would represent a class in the class loader which would cause memory leaks in long-running processes or when many expressions need to be compiled (for example during machine learning). The classes can be wrapped in a separate class loader so they can be unloaded by letting the class loader get garbage collected. But this puts stress on the garbage collector and would still induce avoidable memory overhead.

Other general-purpose programming languages inside the *Java Virtual Machine* which could have been potential bases for the expression language were dropped earlier:

- **Kotlin:** *Kotlin* (KotlinFoundation, 2020) might have also been interesting to compare here, but the scripting integration into *Java* for dynamic parsing was not available in a stable state at the time of the evaluation. Though the results would most likely not be too much different from *Groovy* since it also is a feature-rich programming language with lots of optimisations in the *Java Virtual Machine*. It is not useful to compare the performance of multiple alternatives in this class of implementations, thus it was omitted.

- **JavaScript:** *Nashorn* which replaced *Rhino* as a faster implementation of JavaScript in the *Java Virtual Machine* has been deprecated (OracleCorporation, 2015). Also, it was found to have insufficient parsing and even worse execution speed for expressions. Thus it was dropped from the evaluation.
- **Python:** *Jython* (PythonSoftwareFoundation, 2020) is an implementation in the *Java Virtual Machine*. The execution performance was very slow (user7975996, 2019) (about 13 times slower than Groovy in a simple benchmark). Also, it seemed a bit antiquated by not supporting the newer *Python 3* standards in a stable version.
- **GraalVM:** This platform could bring significant performance improvements for dynamic languages in the *Java Virtual Machine* (OracleCorporation, 2020). Though during the evaluation this was still in an experimental stage and not available inside common distributions of the *Java Development Kit*. Instead, it is a *Virtual Machine* that needs to be installed separately. This currently limits the portability of applications built using it and was thus decided to be unsuitable.

Also, most options lacked capabilities to define domain-specific languages on top of them. Though tools designed for this like *JavaCC* (JavaCC-Community, 2020) and *Xtext* (Eclipse-Foundation, 2020) were discarded early as well because they seemed unnecessarily complex when compared to *Parsii* and did not bring mathematical expression support out of the box. It was also assumed that handwritten parser code would be easier to optimise than parser code that was generated by a tool. Thus the heuristic "simpler is better" was applied to discard these tools even though they might provide other benefits.

Other programming languages and environments outside of the *Java Virtual Machine* were not analysed because the existing trading platform, where the new expression language was to be integrated into, has been implemented there. Reimplementing this trading platform in a different environment would not be worth the effort due to many years of effort already put into it. Also integrating an expression engine via *Java Native Interface* or inter-process-communication would induce a large communication overhead. This is unsuitable for short but frequent expression execution calls.

5 Performance

For the performance measurements a computer with an Intel Core i9-9900k (8 physical cores, 16 virtual cores, 16 MB cache, from 3.6 GHz up 5 GHz turbo boost) with 64 GB RAM and SSD storage was used. The experiment consists of running selected expressions in different implementations separately with a single thread and again with 12 threads. Each experiment was repeated 10 times in the same process (and a new process between experiments) and the highest result was picked for the representation. This is to mitigate the risk of thermal variations in the computer or just in time optimisations in the *Java Virtual Machine* negatively influencing the results. Using averages and standard deviation results combined with a warm-up run would be more accurate, but the maximum or best result for operations performed in a given time allows us to see the same tendencies with less effort.

The test run consists of 100,000 trading strategy backtest simulations. To not measure the overhead of the storage device, the price series is loaded completely into memory. It consists of about 18 years worth of daily bars for the instrument “EURUSD” which results in 5302 data points per backtest. This data fits completely into heap memory as it requires less than 1 MB of memory. The multi-threaded experiment divides the backtests into 10 chunks per thread of 833 backtests per chunk. The tests were done in two modes. Once with expression string parsing done for each backtest and once with the parsed expression cached between backtests. This highlights the overhead the expression parser has over the execution speed.

The results are measured in strategies backtested and as bars processed per millisecond “*/ms*” or per microsecond "/µs".

5.1 Mathematical Expression

The first expression is the one that was used for measuring the performance of *Parsii* against other competitors (Haufler, 2020a). This is a mathematical expression that demonstrates the speed improvements that the expression simplification heuristic of *Parsii* brings. Though since we are looking to automate entry and exit conditions in our platform, we wrap that expression into a condition to make the result a *TRUE* or *FALSE* value. The expression was also modified to be compatible with all implementations tested here. Thus the "^" operator was replaced by a function call to "*pow*". Instead of using the variable "*x*" we look up the current price via the "*close()*" function:

```
(2 + (7 - 5) * 3.14159 * pow(close(), (12-10)) + sin(-3.141)) >
    1000
```

Strategies:

Variation	**Invesdwin**	**Parsii**	**Janino**	**Groovy**	**Spring**
Parsing, 1 Thread	12.21/ms	7.14/ms	2.00/ms	0.17/ms	0.13/ms
Parsing, 12 Threads	63.59/ms	36.54/ms	6.92/ms	0.83/ms	0.83/ms
Caching, 1 Thread	20.64/ms	9.09/ms	10.00/ms	3.71/ms	0.13/ms
Caching, 12 Threads	148.75/ms	46.28/ms	49.00/ms	12.47/ms	0.83/ms

Bars:

Variation	**Invesdwin**	**Parsii**	**Janino**	**Groovy**	**Spring**
Parsing, 1 Thread	64.79/µs	35.63/µs	9.29/µs	0.89/µs	0.66/µs
Parsing, 12 Threads	334.71/µs	193.78/µs	36.69/µs	3.23/µs	3.52/µs
Caching, 1 Thread	109.45/µs	45.30/µs	48.43/µs	19.03/µs	0.66/µs
Caching, 12 Threads	788.97/µs	245.45/µs	259.94/µs	66.15/µs	3.48/µs

The new expression parser *Invesdwin* performs best overall.[5] *Spring* loses overall due to the slow reflection access during execution. *Groovy* is still a magnitude slower than compiling *Java* directly with *Janino* despite type checking and static compilation being enabled. It was possible to improve the performance of *Parsii* by more than a factor of two with the rewrite due to more optimisations despite adding more features to the new expression language.

[5] We call it *Invesdwin Expression Langauge* or *InvEL* since it is part of the invesdwin platform in the open-source project invesdwin-util. Similar to the name of the *Spring Expression Language* or *SpEL*.

5.2 Boolean Expression

The second expression demonstrates the lazy (short-circuit) boolean evaluation. For this we add a condition in front of the previous mathematical expression that never becomes true:

```
close() > 3.14 && (2 + (7 - 5) * 3.14159 * pow(close(),
   (12-10)) + sin(-3.141)) > 1000
```

Strategies:

Variation	Invesdwin	Parsii	Janino	Groovy	Spring
Parsing, 1 Thread	22.51/ms	3.70/ms	2.00/ms	0.18/ms	0.59/ms
Parsing, 12 Threads	122.50/ms	21.25/ms	7.61/ms	0.83/ms	2.40/ms
Caching, 1 Thread	60.94/ms	4.35/ms	15.30/ms	12.80/ms	0.63/ms
Caching, 12 Threads	432.73/ms	24.79/ms	71.20/ms	44.31/ms	2.39/ms

Bars:

Variation	Invesdwin	Parsii	Janino	Groovy	Spring
Parsing, 1 Thread	119.41/µs	19.43/µs	9.76/µs	0.92/µs	3.03/µs
Parsing, 12 Threads	649.74/µs	112.71/µs	40.40/µs	3.36/µs	12.73/µs
Caching, 1 Thread	323.23/µs	22.56/µs	81.15/µs	67.93/µs	3.22/µs
Caching, 12 Threads	2295.19/µs	131.50/µs	377.70/µs	235.06/µs	12.65/µs

Only Parsii is missing this optimisation. This is visible in the cached expression test results where the parser overhead is removed. Parsii performs the worst in that scenario. Other implementations benefit significantly from it.

Advanced Optimisations: In normal strategy expressions, the conditions are normally *FALSE* and occasionally become *TRUE* when a pattern is detected based on specific price changes. A micro-optimisation for generating expressions in machine learning from boolean signals could be to reorder the signals with the least frequency of becoming *true* from left to right in *AND* combinations. Another optimisation would be to cache these signals as *Java BitSet* instances to reduce the memory footprint by a factor of 64 compared to primitive *double* arrays. This would also increase the performance impact of CPU prefetching from memory. Here are the results for **Invesdwin** when these advanced optimisations are enabled for this boolean expression:

Variation	Strategies	Bars
Caching, 1 Thread	223.31/ms	1202.55/µs
Caching, 12 Threads	1488.04/ms	8013.07/µs

We will omit these advanced, domain-specific optimisations from the other comparisons to keep the focus on CPU instead of memory-centred optimisations. The performance of other expressions is similar since mathematical evaluations are replaced by a BitSet lookup with this approach.

5.3 Simple Boolean Expression

The third expression demonstrates the overhead of parsing the above mathematical expression by omitting it:

```
close() > 3.14
```

Strategies:

Variation	Invesdwin	Parsii	Janino	Groovy	Spring
Parsing, 1 Thread	97.18/ms	102.77/ms	3.33/ms	0.19/ms	1.67/ms
Parsing, 12 Threads	520.63/ms	555.33/ms	8.22/ms	0.83/ms	6.43/ms
Caching, 1 Thread	181.49/ms	246.91/ms	16.23/ms	13.03/ms	5.00/ms
Caching, 12 Threads	1753.68/ms	833.00/ms	77.85/ms	42.94/ms	18.44/ms

Bars:

Variation	Invesdwin	Parsii	Janino	Groovy	Spring
Parsing, 1 Thread	962.61/µs	545.12/µs	13.27/µs	1.01/µs	8.54/µs
Parsing, 12 Threads	2761.40/µs	2945.49/µs	43.60/µs	3.35/µs	34.13/µs
Caching, 1 Thread	344.86/µs	1309.63/µs	86.11/µs	69.11/µs	23.36/µs
Caching, 12 Threads	9301.54/µs	4418.23/µs	413.00/µs	227.79/µs	97.85/µs

We could gain about 10% performance improvement in this test by using quicker double comparisons. Currently the implementation uses "$| a - b | > 0.000000001$" instead of "$a > b$" to work around issues with incorrect representations of *double* values in the *Java Virtual Machine*. Some indicators are sensitive to this kind of imprecision, thus the quicker variation is not implemented.

5.4 Constant Expression

The fourth expression demonstrates the overhead of looking up the current price from memory when compared with the above results:

```
1.22 > 3.14
```

Strategies:

Variation	Invesdwin	Parsii	Janino	Groovy	Spring
Parsing, 1 Thread	188.32/ms	152.21/ms	3.33/ms	0.19/ms	2.00/ms
Parsing, 12 Threads	925.56/ms	757.27/ms	9.12/ms	0.83/ms	6.38/ms
Caching, 1 Thread	588.24/ms	458.72/ms	16.05/ms	16.38/ms	5.00/ms
Caching, 12 Threads	3446.90/ms	2776.67/ms	80.87/ms	54.80/ms	17.66/ms

Bars:

Variation	Invesdwin	Parsii	Janino	Groovy	Spring
Parsing, 1 Thread	998.87/µs	807.31/µs	15.67/µs	0.99/µs	9.28/µs
Parsing, 12 Threads	4909.15/µs	4016.57/µs	48.39/µs	3.33/µs	33.87/µs
Caching, 1 Thread	3120.00/µs	2433.03/µs	85.15/µs	86.91/µs	22.23/µs
Caching, 12 Threads	18282.34/µs	14727.44/µs	429.04/µs	290.73/µs	93.68/µs

Both *Parsii* and *Invesdwin* simplify the expression into a constant value. The difference is that *Parsii* simplifies it into a constant *double* value *0* while *Invesdwin* simplifies it into a constant *boolean* value *FALSE*. This is slightly more efficient for *boolean* expressions because the result does not have to be cast multiple times.

6 Conclusion

The comparison with general-purpose programming languages might seem unfair but highlights key aspects of the implementation optimisations. It also provides a known benchmark to measure against instead of repeating existing measurements. The results demonstrate that it was possible to create an improvement for expression languages in the *Java Virtual Machine* both from a feature and performance perspective. Though the features focus on limiting possibilities for incorrect expressions and to provide the ability to formalise strategy development processes. How this can be realized will be content for the following research which will cover the aspects that were not discussed in detail here.

7 References

ApacheGroovy (2020).
Groovy Domain-Specific Languages, retrieved at 17.04.2020.
URL: *http://docs.groovy-lang.org/docs/latest/html/documentation/core-domain-specific-languages.html*

Bergstrom, D. (2020).
BuildAlpha, retrieved at 17.04.2020.
URL: *https://www.buildalpha.com*

Bobkov, A. and Georges, S. (2020).
BNF Playground, retrieved at 17.04.2020.
URL: *https://bnfplayground.pauliankline.com/*

Bryant, M. (2019).
Adaptrade Software, retrieved at 17.04.2020.
URL: *http://www.adaptrade.com*

Chakerian, S. (2019).
Algorithmically Detecting (and Trading) Technical Chart Patterns with Python, retrieved at 08.06.2020.
URL: *https://medium.com/automation-generation/algorithmically-detecting-and-trading-technical-chart-patterns-with-python-c577b3a396ed*

EclipseFoundation (2020).
Xtext - language engineering made easy!, retrieved at 17.04.2020.
URL: *https://www.eclipse.org/Xtext*

Futrell, R. (2020).
RSyntaxTextArea, retrieved at 17.04.2020.
URL: *https://github.com/bobbylight/RSyntaxTextArea*

Haufler, A. (2020a).
Andy's Software Engineering Corner: How to write one of the fastest expression evaluators in Java, retrieved at 17.04.2020.
URL: *http://andreas.haufler.info/2013/12/how-to-write-one-of-fastest-expression.html*

Haufler, A. (2020b).
Parsii, retrieved at 17.04.2020.
URL: *https://github.com/scireum/parsii*

JavaCC-Community (2020).
JavaCC, retrieved at 17.04.2020.
URL: *https://javacc.github.io/javacc*

KotlinFoundation (2020).
Kotlin Programming Language, retrieved at 17.04.2020.
URL: *https://kotlinlang.org*

ObjectRefineryLimited (2020).
JFreeChart, retrieved at 17.04.2020.
URL: *http://www.jfree.org/jfreechart/*

oPgroupGermanyGmbH (2020).
Zorro Manual: Machine Learning, retrieved at 08.06.2020.
URL: *https://manual.zorro-project.com/advisor.htm*

OracleCorporation (2015).
JEP 335: Deprecate the Nashorn Javascript Engine, retrieved at 17.04.2020.
URL: *http://openjdk.java.net/jeps/335*

OracleCorporation (2020).
GraalVM, retrieved at 17.04.2020.
URL: *https://www.graalvm.org*

PythonSoftwareFoundation (2020).
Jython, retrieved at 17.04.2020.
URL: *https://www.jython.org/*

Stang, E. (2020a).
invesdwin-context-client, retrieved at 17.04.2020.
URL: *https://github.com/subes/invesdwin-context-client*

Stang, E. (2020b).
invesdwin-util, retrieved at 17.04.2020.
URL: *https://github.com/subes/invesdwin-util*

StrategyQuant (2020).
Strategy Quant, retrieved at 17.04.2020.
URL: *https://strategyquant.com*

TradingView (2020).
Free Charting Library by TradingView, retrieved at 17.04.2020.
URL: *https://www.tradingview.com/HTML5-stock-forex-bitcoin-charting-library/*

Unkrig, A. (2020).
Janino by janino-compiler, retrieved at 17.04.2020.
URL: *https://janino-compiler.github.io/janino*

user7975996 (2019).
Benchmarking Java, Groovy, Jython, and Python, retrieved at 17.04.2020.
URL: *https://stackoverflow.com/questions/54281767/benchmarking-java-groovy-jython-and-python*

VMwareInc (2020).
Spring expression language documentation, retrieved at 17.04.2020.
URL: *https://docs.spring.io/spring/docs/current/spring-framework-reference/core.htmlexpressions*

Zwag, P. (2020).
GeneticSystemBuilder, retrieved at 17.04.2020.
URL: *https://trademaid.info*