A domain-specific language for compliance rules

Rune K. Svendsen
runs@itu.dk

IT UNIVERSITY OF COPENHAGEN

2020-05-15

ITU project title: *Portfolio compliance DSL*
ITU class code: *5354738-Spring 2020*

Abstract

Companies that invest money on behalf of their customers are subject to requirements regarding what they may invest in. The effort spent checking whether requirements have been violated increases with the number of requirements and how often the company buys and sells assets.
SimCorp is a financial software-company whose software solution *SimCorp Dimension* enables automatic checking of investment requirements. However, some users of SimCorp's software solution prefer typing in requirements using their keyboard, which the software does not support. Therefore, SimCorp is interested in the development of a textual language that can describe investment requirements and which permits software to automatically evaluate compliance with the described requirements.
This report describes the development and design of such a language, and also presents a software implementation that can evaluate whether or not a set of investments comply with requirements described using this language.

Preface

This BSc thesis was prepared during the spring semester of 2020 at the IT University of Copenhagen.
I would like to thank my supervisor Peter Sestoft for invaluable feedback on the report, the code and on the overall approach to the challenge presented by the project.
Likewise, I would like to thank Martin Steen Nielsen and Per Langseth Vester from SimCorp for spending countless hours teaching me both about the intricacies of compliance rules as well as SimCorp's approach to solving the challenges of this domain.

Østerbro, May 15, 2020

Rune K. Svendsen

Table of contents

1 Introduction

Companies that invest money on behalf of customers — hereafter referred to as *asset managers* — are subject to restrictions on what they are allowed to invest in. For example — in order to reduce the risk of monetary loss for pension holders — a law might require that a pension fund invest no more than a certain fraction of its funds in a single business sector (e.g. *agriculture*, *manufacturing*, or *construction*). To the asset manager, a law of this nature becomes a *compliance rule*. A compliance rule is a rule that a given *portfolio* (the set of investments owned by the asset manager) must comply with. A large number of portfolio compliance rules apply to asset managers, who must keep track of whether or not their investments adhere to these rules.

SimCorp is a financial software-company, whose customers include asset managers, banks, central banks, pension funds, sovereign wealth funds and insurance companies(Nasdaq 2020). SimCorp's core software product *SimCorp Dimension* includes — among many other features — the ability to automatically check portfolios against a set of compliance rules. This enables asset managers to spend less time dealing with compliance rules, as computer software can automatically perform the task of checking for compliance with rules and regulations.

Per Langseth Vester (Lead Product Owner at SimCorp) describes SimCorp's motivation for looking at new possibilities regarding compliance rule software in Appendix A. In summary, SimCorp's current software solution for compliance rules works well, but there is room for improvement:

- From a usability point-of-view, some users prefer being able to type in rules using the keyboard — including e.g. auto-completion. SimCorp's current solution does not easily allow for adding this.
- When writing compliance rules using SimCorp's software, the rule author can make use of so-called *rule fragments*. A rule fragment is a small, re-usable rule condition. By combining together many of these conditions, complex compliance rules can be created without having to write the entire rule from scratch.
  - SimCorp's intention is to promote the reuse of rule fragments across compliance rules. In reality however, only few fragments are used in more than one rule.
  - Conceptually, fragments are very powerful, but also difficult to master. They allow for many logical constructions that do not make sense in a business context. SimCorp would like a more user-friendly solution with a less steep learning curve.

1.1 Scope

The scope of this project is to design, and implement in Haskell, a domain-specific language (DSL) for expressing compliance rules. The DSL will make use of concepts from functional programming, and must

support at least the six rules listed in section 2.1.2. These rules are representative of the more complex rules in use, but we do not here strive to cover all rules used in practice.

2 Background

2.1 Compliance rule

2.1.1 Definitions

This section defines terms that will be used throughout the report. Note that these definitions do not necessarily correspond to how the terms are used in the industry — they merely specify how the terms are used in this report.

2.1.1.1 Security The term *security* is used to refer to any asset that an investor can be in possession of. This includes, but is not limited to: cash (money in a bank account), bonds (long term debt obligations), company stock (company ownership), money market instruments (short term debt obligations). A single security — e.g. one share of IBM stock — is the smallest unit described in this paper. The term *instrument* may be used as a synonym for security.

2.1.1.2 Position The term *position* refers to one or more of the same type of security. For example, the ownership of five shares of Microsoft stock at a current market value of USD 150 each comprises a *position* in Microsoft stock with a current market value of USD 750. The position is the smallest unit that a portfolio compliance rule can apply to. Thus, no portfolio compliance rule distinguishes between owning e.g. ten shares of Google with a value of USD 1000 each versus eight shares of Google with a value of USD 1250 each — both comprise a position in Google shares with a value of USD 10000.

2.1.1.3 Portfolio The term *portfolio* refers to a *set of positions*. A particular portfolio may contain positions of the same *type* from the same *region*, e.g. Asian stocks; it may contain all the positions of a particular *client* (a given customer of the asset manager); or it may contain all positions governed by a particular portfolio compliance rule. For the purpose of this paper the latter is assumed. That is, a portfolio — containing a number of positions — exists because the positions herein are governed by the same compliance rule(s).

2.1.2 Compliance rule examples

Compliance rules vary greatly in complexity. An example of a very simple compliance rule is "*invest only in bonds*". Evaluating whether a portfolio complies with this rule only requires looking at each position individually, checking whether this position is a bond, and failing if it is not.

A more complex rule may be a limit on the value of the positions that share a specific property. For example, a rule may require that the value of all positions that have the same *issuer* must be at most

two million dollars. Oftentimes a rule will impose a *relative* limit — as opposed to an absolute limit of e.g. two million dollars — for example by requiring that the value of same-issuer positions relative to the total value of the portfolio must be no more than 5%.

The goal is for the DSL to be able to express compliance rules of relatively high complexity. For this purpose, SimCorp has provided a document containing seven complex compliance rules (see Appendix B). Six of these rule have been chosen as the basis of the proposed DSL because of their similarity to each other. In order to restrict the scope of the DSL, the rule concerning *sector index-weights* — labeled as `IV` in Appendix B — has not been included in the example rules that the DSL must support. The six chosen rules are labeled as `I`, `IIa`, `IIb`, `III`, `V`, and `VI` in Appendix B. They are presented below as Rule I, II, III, IV, V, and VI, respectively.

2.1.2.1 Rule I

Maximum 10% of assets in any single issuer. Positions of the same issuer whose value is greater than 5% of total assets may not in aggregate exceed 40% of total assets. (Rule `I` in Appendix B)

This rule is composed of two sub-rules. The overall rule requires compliance with both of the sub-rules.

We begin by separating the positions in the portfolio into groups, such that each group contains all positions that have the same issuer (hereafter referred to as *grouping by issuer*). After this initial step, the two sub-rules proceed as follows:

1. For each group: the sum value of the positions in the group must be at most 10% of the total portfolio value.
2. Remove all of the groups whose value relative to the portfolio is less than or equal to 5%. The aggregate value of the remaining groups must be at most 40% of the total portfolio value.

2.1.2.2 Rule II

No more than 35% in securities of the same issuer, and if more than 30% in one then issuer must be made up of at least 6 different issues. (Rule `IIa` in Appendix B)

This rule is also composed of two sub-rules. As in the previous rule, the first step for both sub-rules is to group by issuer.

1. For each group: the sum value of the positions in the group must be at most 35% of the total portfolio value.
2. For each group whose relative value is greater than 30%: a *count* of the number of different *issues* for this group must be greater than or equal to *six* (where counting the number of different issues in a group amounts to grouping by issue and counting the resulting groups).

2.1.2.3 Rule III

When holding >35% in a single issuer of government and public securities, then there can be no more than 30% in any single issue, and a minimum of 6 different issues is required. (Rule `IIb` in Appendix B)

The first step is to filter off positions that are *not* either a government or public security. Next we group by issuer, followed by:

1. For each *issuer*-group: only if the group value is greater than 35% of the total portfolio value: then group by *issue* and proceed to **2.**
2. a. The *issue*-group count must be at least 6, and
   b. For each *issue*-group: the value of the group must be at most 30% of total portfolio value

2.1.2.4 Rule IV

The risk exposure to a counterparty to an OTC derivative may not exceed 5% of total portfolio exposure; the limit is raised to 10% for approved credit institutions. (Rule `III` in Appendix B)

First, filter off positions that are not an *OTC derivative*. Next, group by counterparty, and for each counterparty-group:

- If the counterparty **is not** an approved credit institution:
  - then the exposure of the counterparty relative to the total portfolio exposure must be at most **5%**
- If the counterparty **is** an approved credit institution:
  - then the exposure of the counterparty relative to the total portfolio exposure must be at most **10%**

2.1.2.5 Rule V

Max 5% in any single security rated better than BBB: otherwise the maximum is 1% of total portfolio value. (Rule `V` in Appendix B)

First, group by security, and for each security-group:

- If the security's rating is **AAA** or **AA** or **A** (i.e. better than **BBB**), then the value of the security must be at most 5% relative to total portfolio value
- Otherwise the value of the security must be at most 1% relative to total portfolio value

2.1.2.6 Rule VI

The portfolio shall invest in at least 5 / 4 / 3 / 2 different foreign countries if aggregate value of foreign countries relative to portfolio >=80% / >=60% / >=40% / <40%, respectively. (Rule `VI` in Appendix B)

First, filter off domestic positions, in order to obtain only foreign-country positions. Next, calculate **(a)** the value of foreign-country positions relative to the *entire* portfolio (i.e. the portfolio including domestic positions), and **(b)** the number of different foreign countries. Then:

- If foreign-country value is at least **80%**: foreign-country count must be at least **5**
- If foreign-country value is at least **60%**: foreign-country count must be at least **4**
- If foreign-country value is at least **40%**: foreign-country count must be at least **3**
- If foreign-country value is less than **40%**: foreign-country count must be at least **2**

2.2 Domain-specific language

A domain-specific language (DSL) is a programming language that is tailored to model a specific business domain. A DSL stands in contrast to a *general-purpose* programming language (GPPL), which is designed to model *any* business domain. A DSL is thus less expressive than a GPPL, in the sense that a DSL intentionally restricts the domain that can be modelled using the language.

DSL examples include: HTML (*Hypertext Markup Language*) for describing the structure of a web page; CSS (*Cascading Style Sheets*) for describing the presentation of a web page (e.g., layout, colors, fonts); and SQL (*Structured Query Language*) for describing queries against a relational database.

2.2.1 Purpose

Due to the restriction in what a DSL must be capable of modeling, it is possible to design a DSL that is significantly simpler than a GPPL. And while this comes with the disadvantage of reducing what is possible to express using the DSL, it also comes with the advantage of a reduction in the time and effort needed to learn it. Consequently, if the goal is to get experts of a particular business domain to easily express their domain knowledge in code, which can be executed by a computer, a simple DSL can be a helpful tool.

2.2.2 Terminology

2.2.2.1 *Abstract* versus *concrete* syntax The *abstract syntax* of a programming language (whether domain-specific or general-purpose) is a data structure that describes an expression in that language. As an example, let us consider a very simple DSL that describes multiplication and addition of integers. This language has two *data constructors*: Mul and Add, which describe multiplication and addition, respectively. Both of these data constructors take two arguments which may be either an integer or another expression — i.e. either a multiplication or addition or a combination hereof. The abstract syntax $Add\ 3\ 5$ thus describes three added to five, $Mul\ 2\ 7$ describes two multiplied by seven, and $Mul\ 4\ (Add\ 1\ 6)$ describes multiplying by four the result of adding one to six. This syntax is called "abstract" because

it refers to abstract objects. The objects Add and Mul are abstract in the sense that Add and Mul are simply *names* used to refer to the abstract operation of addition and multiplication, respectively — we could just as well refer to these objects as A and M.

The *concrete syntax* of a programming language is **a representation** of the abstract syntax. For example, a common representation of $Mul\ 4\ (Add\ 1\ 6)$ — i.e. multiplying by four the result of adding one to six — is `4 × (1 + 6)`. But we may also refer to this same operation in concrete syntax by adding a (redundant) pair of parentheses around each subexpression: `(4) × ((1 + 6))` — both pieces of concrete syntax refer to exactly the same operation. Thus, as can be seen from this example, a single piece of abstract syntax can be represented in multiple ways using concrete syntax.

In programming jargon, concrete syntax is usually referred to simply as "program code" or "source code", whereas the abstract syntax is an internal data structure used by the compiler or interpreter of the language in question.

2.2.2.2 *Printer* versus *parser*

A *parser* converts concrete syntax into abstract syntax, while a *printer* converts abstract syntax into concrete syntax. A parser can fail because it can be given invalid input. Using the above example of multiplication and addition, a parser given the input `4×(1+6` will fail because an ending parenthesis is missing. A printer cannot fail — it should be able to convert any instance of abstract syntax into concrete syntax.

Given a parser and a printer for the same language, feeding to the parser the output of applying the printer to any piece of abstract syntax should yield the same piece of abstract syntax. The opposite, however, is not necessarily the case, as the printer cannot know e.g. how many redundant parentheses there were in the original concrete syntax, and may thus output different concrete syntax.

2.2.3 Comparison with general-purpose language

The two subsections below describe benefits and drawbacks of expressing domain knowledge in a DSL compared to using an existing GPPL for this purpose.

2.2.3.1 Benefits

- A restriction of the constructs available for recursion can guarantee termination. For example, restricting iteration to *"doing something for every item in a list"*, as well as not allowing infinite lists, guarantees that programs will terminate (i.e. not loop infinitely).
- Parallel evaluation made possible by absence of side effects
- Separation of the *language* from the *model described using the language* allows different interpretations of the same model:

  - *Formatting* the model for ease of readability, by printing out using syntax highlighting and standardized formatting
  - *Visualizing* the model, i.e. producing an image that represents the model
  - *Perform static checks* on the model, i.e. checking the model for inconsistencies and errors that may not be possible if the model were expressed in a GPPL
  - Different implementations of programs that evaluate the model — as opposed to being tied to the GPPL in which the model is formulated, e.g.:
    * One evaluator written in C for performance — while sacrificing memory safety (and thus risking security vulnerabilities)
    * Another evaluator written in Haskell for safety — while sacrificing performance

2.2.3.2 Drawbacks

The drawbacks are related primarily to up-front cost and maintenance costs:

- User needs to learn a new language, including both syntax and semantics
- Higher cost of maintaining separate compiler and parser for custom DSL

3 Requirements analysis

In this section, the example rules from section 2.1.2 are analyzed in order to derive requirements for the DSL.

3.1 Analysis of example rules

3.1.1 General

All the example rules refer to a position as having one or more *properties*. For example, a property may be a position's *value*, its *issuer*, or its *country*. Furthermore, all of these properties require a notion of equality — i.e. it must be possible to determine whether e.g. the *country* properties of two different positions are equal or not. In addition, some properties (e.g. *value* and *exposure*) require the ability to compare for *order* (less/greater than), as well as requiring the numeric operations *addition* (for calculating the sum of multiple values) and *division* (for calculating relative value).

3.1.2 Rule constructs

The following constructs have been identified from analyzing the example rules:

- **Grouping:** by a given property *name* (e.g. *"country"*), so that a group is created for each distinct value of this property (e.g. "DK", "US", "GB"), and each group contains all the positions whose property value is equal to the group's value
- **Group filtering:** removing certain groups from a grouping by some condition (e.g. "*relative value < 5%*")
- **Position filtering**: removing certain positions from a grouping by some condition (e.g. "*is not either a government or public security*")
- **Group sum**: calculating the sum of the values of some property name (e.g. *"exposure"*) for all the positions in a group
- **Group count**: counting the number of groups in a grouping
- **Logical *and***: when a single rule is composed of two or more sub-rules then both sub-rules must apply
- **For all**: apply a rule for each group that results from a grouping (e.g. "*For each issuer-group ...*"), with the effect that the given rule must apply for all groups
- **Relative:** calculating the relative value of one value compared to another value (e.g. "*exposure of the counterparty relative to the total portfolio exposure*")
- **Conditional**: apply a requirement only if some condition is true (e.g. *"if foreign-country value is at least 80% then foreign-country count must be at least 5"*)

Note the presence of both a *group filtering* and *position* filtering. In the second sub-rule of Rule I (sec. 2.1.2.1) the groups in the *issuer*-grouping whose relative value is at most 5% are removed. Thus, the condition (*"relative value of group > 5%"*) applies to a group, and determines whether the entire group in question is removed. Compare this to the filtering in Rule III (2.1.2.3) and Rule IV (2.1.2.4), in which individual positions are removed from the portfolio before proceeding. For example, in Rule IV the positions whose type is not *OTC derivative* are removed. Here, the condition (*"is OTC derivative"*) applies to a single position, rather than to a group, and individual positions are removed from the grouping that the filter is applied to.

Table 1 below summarizes which constructs are used by each example rule. Due to a lack of space, the constructs *group filtering* and *position filtering* are represented as a single construct named *filter*.

	Grouping	**Filter**	***and***	***for all***	**Group sum**	**Group count**	**Relative**	**Conditional**
Rule I	x	x	x	x	x		x	
Rule II	x		x	x	x	x	x	x
Rule III	x	x	x	x	x	x	x	x
Rule IV	x	x		x	x		x	x
Rule V	x			x	x		x	x
Rule VI	x	x			x	x	x	x

Table 1: Rule/construct matrix

3.2 Grouping data structure

From the above analysis it is clear that the concept of a *grouping* is required. In addition, Rule III shows (when grouping an *issuer*-grouping by *issue*) that an existing grouping can be grouped once again, thus creating multiple levels of groupings. Due to these requirements, and inspired by SimCorp's use of a tree structure to visualize groupings, a tree has been chosen to describe a grouping.

The following figure shows the simplest grouping of them all — a portfolio. This portfolio contains the positions $P1$ through $P8$.

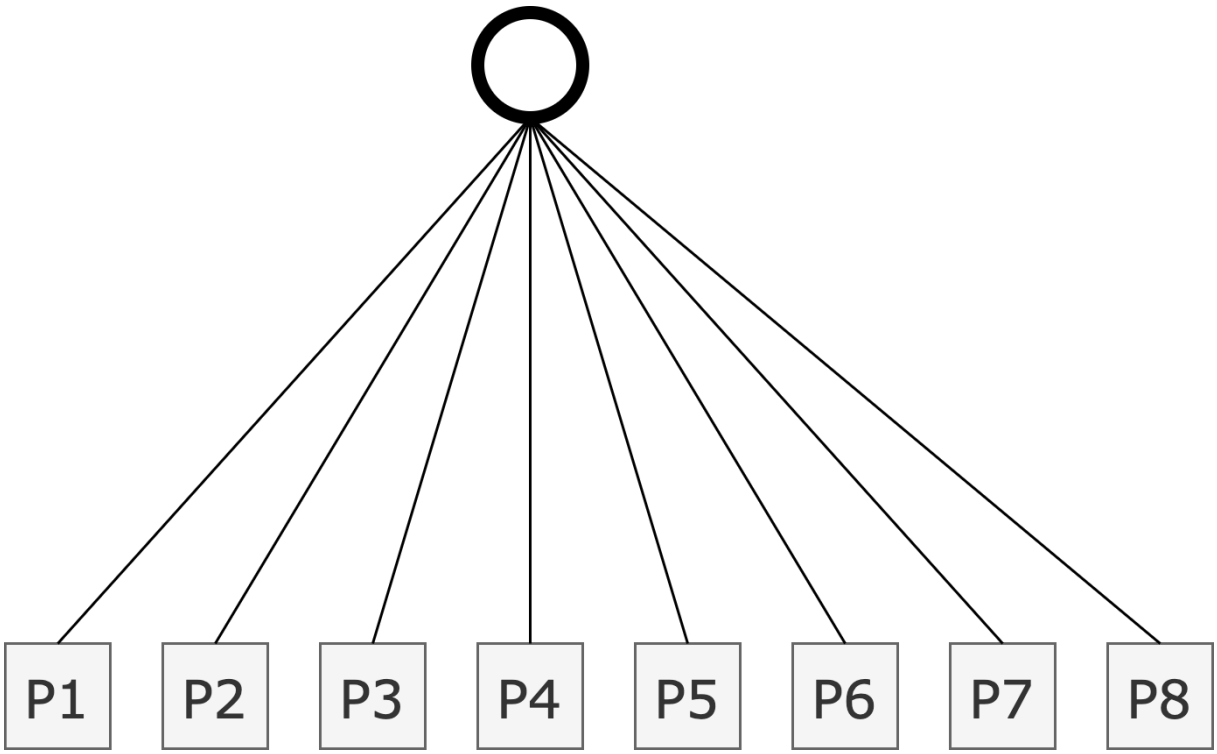

Figure 1: Tree data structure for an ungrouped portfolio

This is represented as a tree with a single non-leaf node (depicted as round), under which a leaf node (depicted as square) is present for each position in the portfolio (named $P1$ through $P8$ in the above figure). A position is thus represented as a leaf node in a tree, where each non-leaf node represents a group.

The figure below depicts the tree that results from grouping the above portfolio (fig. 1) by *country*.

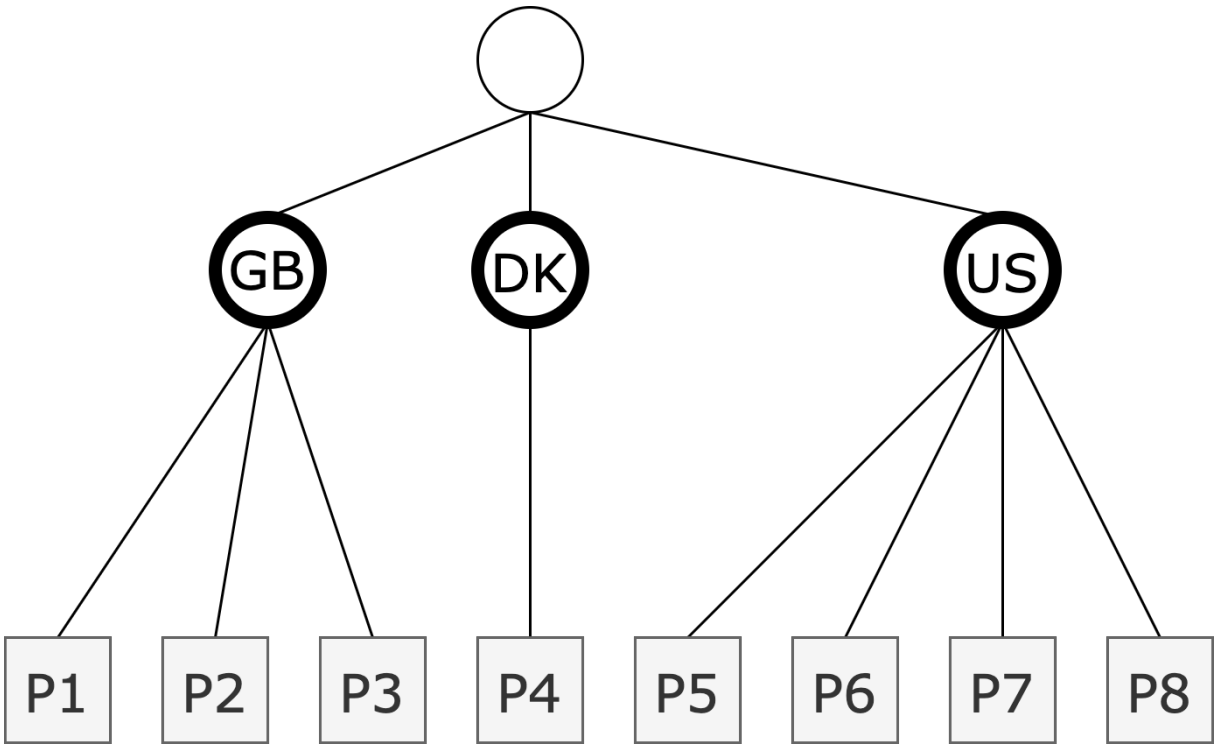

Figure 2: Tree data structure for a portfolio grouped by country

A node for each distinct value of the *country* property is created as children of the parent portfolio-node. This example portfolio contains positions from three different countries: Denmark (*DK*), United States (*US*), and Great Britain (*GB*). Under these new nodes, the positions whose *country* property is equal to the value in the given node are present as leaf nodes.

The figure below depicts the tree that results from grouping by *issuer* the above country-grouping (fig. 2).

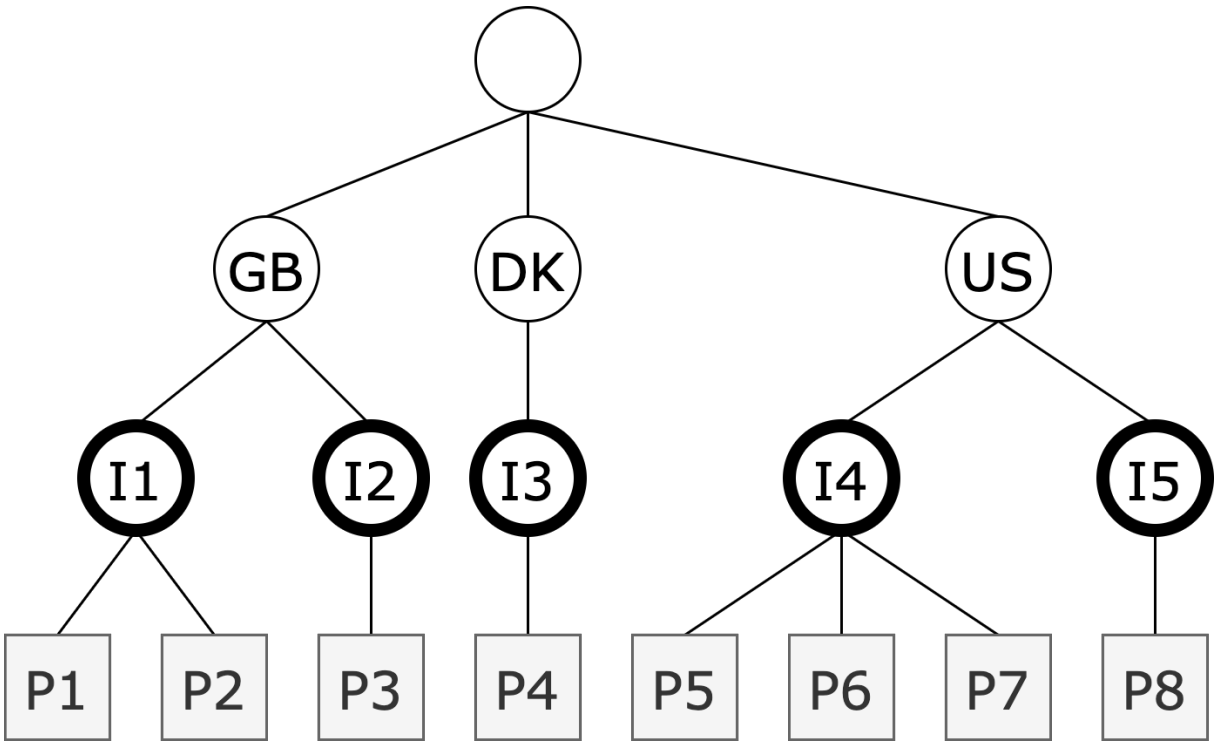

Figure 3: Tree data structure for a portfolio grouped by country and then by issuer

When the existing country-grouping is grouped again by issuer, we see that another level of nodes is added below the country-level. This level contains a node for each distinct issuer under each country node (named `I1` through `I5` in the above example).

Note that any grouping can be represented as a tree, since any node will always have exactly one parent. If, for example, the portfolio used in above example had two positions with different values for the *country*-property (e.g. *JP* and *IT*) but the *same* value for the *issuer*-property (e.g `I6`), then *two* `I6`-nodes would be created — the first with the parent node *JP* and the second with the parent node *IT*. In other words, this would *not* result in a single `I6` node with two parents (*JP* and *IT*).

3.3 Design choices

3.3.1 Rule input

All example rules operate on a portfolio. That is, for the purpose of evaluating whether a portfolio complies with a rule, none of the example rules require any input other than the portfolio in question. Consequently, it has been chosen that all rules operate on an implicit input, present in a variable by the name of `Portfolio`. This simplification may need to be reconsidered in future versions of the language (see Future work/input arguments sec. 6.5).

3.3.2 Input data format

A position is represented as a map *from* a **string** property name *to* a property value which is either a **floating point number**, a **string**, or a **boolean**. *Null*-values are not supported, but a position may omit a particular property. In this way, a position representing e.g. a *commodity future* can have a property called `UnderlyingCommodity`, whereas positions of another type (e.g. *bonds*) may omit this property.

Furthermore, the input data must be preprocessed so that the rule language does not need to perform transformations such as converting between different currencies (in order to have a common measure of value). It is thus required that the input data contains a measure of value that is comparable between positions, even though the underlying positions may be denominated in different currencies.

4 Language specification

This section contains a specification of the syntax and semantics of the compliance rule-DSL, as well as a description of a simple, boolean evaluator for the language.

4.1 Syntax and semantics

The constructs of the language can be divided into two kinds: **(1)** expressions, which evaluate to a value; and **(2)** statements, which *do not* evaluate to a value (but take values/expressions as input).

An expression is one of the following:

1. **Literal**
    1. Property value (number/string/boolean)
    2. Property name
    3. Percentage
2. **Grouping**
    1. Grouping by some property name
    2. Filtering of groups/positions based on a *comparison*
3. **Calculation**
    1. Calculating the sum/average/minimum/maximum for a particular property for all positions in a grouping
    2. Counting the number of groups in a grouping
    3. Calculating a percentage by relative comparison between two of the above (or a literal)
4. **Comparison**
    1. Boolean comparison of equality and/or order (greater/less than) between the result of a calculation and/or a constant
    2. *Boolean logic* comparison (*and/or/not*) between two comparisons

A statement is one of:

1. Variable binding of a *grouping*, *calculation*, *comparison*, or *literal*
2. Rule requirement
3. *If* -statement (for conditional rule requirements)
4. Iteration over a grouping (for applying rule requirements to all groups in a grouping)

4.1.1 Expression

This section describes the syntax and semantics of expressions.

4.1.1.1 Literal A literal is either a property *name*, a property *value*, or a percentage.

As described in section 3.3.2, a position contains multiple named properties. Examples of property names — taken from the example rules in 2.1.2 — include *value*, *issuer*, *issue*, *instrument type* (with a value of e.g. "public security" as in Rule III sec. 2.1.2.3), *counterparty*, *security id* (Rule V sec. 2.1.2.5), and *country*. The syntax for a property name literal is a period (`.`) followed by a capital alphabetic letter followed by zero or more alphanumeric characters. Thus, written in the syntax of the DSL, the property names from the example rules become e.g. `.Value`, `.Issuer`, `.InstrumentType`, and `.SecurityID`.

A property value is either a number, a string, or a boolean. The number type supports both floating point numbers and integers, so that the two number literals `42.0` and `42` are equivalent. Strings are surrounded by double quotes, and support escaping of a double quote or backslash character by prefixing with a backslash character. The two boolean literals are written as **`true`** and **`false`**.

A percentage is simply a number followed immediately by a percentage sign — e.g. `42%` or `42.0%`.

4.1.1.2 Table of operators A *grouping*, *calculation* or *comparison* is performed using one of the operators listed in table 2 below. All of the operators are either *prefix* or *infix*. A prefix-operator takes a single argument and appears *before* its argument, while an infix-operator take *two* arguments and appears *between* its two arguments.

Table 2 lists operator-expressions in descending order of precedence. The operators in a group of rows delimited by a horizontal line have the same precedence, which is higher than that of the operators in the group below it. Thus, **`grouped by`** and **`where`** have the highest precedence, **`of`** has the next-highest precedence, and the **`OR`** operator has the lowest precedence. The precedence of an operator determines which operator-expression is evaluated first — with higher-precedence operators being evaluated before lower-precedence operators. For example, in the expression $2 + 4 * 3$ the multiplication operation is evaluated before the addition operation because the precedence of multiplication is higher than that of addition. Similarly, in the expression **`count`** `a > 7` the **`count`** operation is evaluated before the *greater than*-operation because the precedence of **`count`** is higher than that of the *greater than*-operator — as defined by the table below.

The third column in the table defines the associativity of each infix operator. The associativity of an operator is either *left*, *right* or *none*, and defines how to group a sequence of operators of the same precedence. For example, subtraction and addition have the same precedence, so there are two different ways to interpret the expression $1 - 2 + 3$. The interpretation $(1 - 2) + 3$ is the case when subtraction and addition are *left*-associative, while the interpretation would be $1 - (2 + 3)$ if subtraction and addition

were *right*-associative. Thus, given that e.g. the `relative to` operator is left-associative, the expression `3 relative to 6 relative to 25%` is equivalent to `(3 relative to 6) relative to 25%` (which evaluates to 200%) and not `3 relative to (6 relative to 25%)` (which results in a runtime error because a number cannot be compared to a percentage). Operators with an associativity of *none* are non-associative, which means combining multiple such operators in sequence is not defined — as in e.g. `1 == 1 == 1` (which will cause a parser-error).

The fourth column defines the operator's input argument type(s) while the fifth column defines the operator's result type. The `PropName` type is the type of a property name, while `Bool`, `Number`, and `Percent` are the types of the respective literals described in 4.1.1.1 and also the result type of certain operator expressions. The `Pos` type is the type of a position.
The types contained in these two columns also include *type variables*, which start with a lower case letter (e.g. `tPropVal`), and refer to a *set* of concrete types. The type variable `tPropVal` refers to any property value type (`Number`/`String`/`Bool`); the type variable `tGrouped` refers to any type that can be in a `Grouping` (`Pos`/ `tPropVal`); the type variable `tNum` includes types that support addition and division (`Number`/`Percent`); the type variable `tEq` includes the types that can be compared for equality (`tPropVal`/`Percent`); and finally the type variable `tOrd` includes the types that can be compared for order (`Number`/`Percent`).
When reading the table below, all type variables of the same name must be substituted for a *single* type. Thus, e.g. the operator == can perform an equality-comparison between two `String`-values and two `Percent`-values, but not between one `String`-value and one `Percent`-value.
Lastly, there is the generic type `Grouping` which describes a grouping of values of some type. The type of the grouped value is in angle brackets, e.g. `Grouping<Pos>` refers to a grouping of positions.

Expression	Meaning	Associativity	Input type(s)	Result type
`e1` **`grouped by`** `e2`	Group by property name	left	`Grouping<Pos>`, `PropName`	`Grouping<Pos>`
`e1` **`where`** `e2`	Filter by a condition	left	`Grouping<Pos>`, `Bool`	`Grouping<Pos>`
`e1` **`of`** `e2`	*map* by property name	none	`PropName`, `Grouping<Pos>`	`Grouping<tPropVal>`
`count` `e`	Count the number of groups		`Grouping<tGrouped>`	`Number`
`sum` `e`	Sum of numbers in a grouping		`Grouping<Number>`	`Number`
`average` `e`	Average of numbers in a grouping		`Grouping<Number>`	`Number`
`minimum` `e`	Maximum of numbers in a grouping		`Grouping<Number>`	`Number`
`maximum` `e`	Minimum of numbers in a grouping		`Grouping<Number>`	`Number`
`e1` **`relative to`** `e2`	Relative comparison	left	`tNum`, `tNum`	`Percent`
`e1` **`==`** `e2`	Equals	none	`tEq`, `tEq`	`Bool`
`e1` **`!=`** `e2`	Does not equal	none	`tEq`, `tEq`	`Bool`
`e1` **`>`** `e2`	Greater than	none	`tOrd`, `tOrd`	`Bool`
`e1` **`<`** `e2`	Less than	none	`tOrd`, `tOrd`	`Bool`
`e1` **`>=`** `e2`	Greater than or equal	none	`tOrd`, `tOrd`	`Bool`
`e1` **`<=`** `e2`	Less than or equal	none	`tOrd`, `tOrd`	`Bool`
`NOT` `e`	Logical negation		`Bool`	`Bool`
`e1` **`AND`** `e2`	Logical *and*	left	`Bool`, `Bool`	`Bool`
`e1` **`OR`** `e2`	Logical *or*	left	`Bool`, `Bool`	`Bool`

Table 2: Precedence, associativity, input type(s), and result type of operators

4.1.1.3 Grouping A grouping/filtering operation takes as input an existing grouping and either groups it (by a property name) or removes positions based on a condition. For example, `Portfolio grouped by .Country where (sum .Value of Country relative to Portfolio >= 10%)` first groups the portfolio by the *country* property and then removes the countries whose value relative to the portfolio is less than 10%. Here, `Portfolio` is the variable that is implicitly defined for all rules (see 3.3.1); it has the type `Grouping<Pos>`.

As mentioned in sec. 3.2, a grouping is represented using a tree. The figure below depicts the input and output tree of each operation in the grouping expression `Portfolio grouped by .Country grouped by .Issuer`.

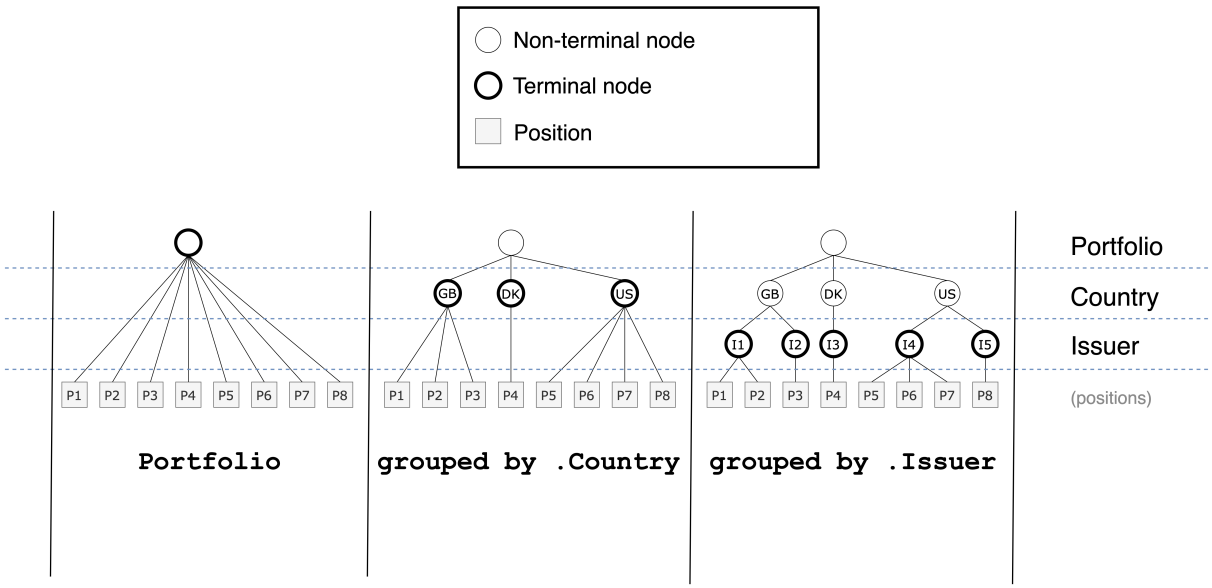

Figure 4: Portfolio grouped first by Country then by Issuer

The leftmost tree is bound to the variable `Portfolio`. Next, the `grouped by .Country` operation takes `Portfolio` as input and produces the tree shown in the middle — which contains a group for each country: Great Britain (GB), Denmark (DK), and the United States (US). After that, `grouped by .Issuer` takes as input the middle tree and produces the rightmost tree — adding a terminal node for each distinct issuer for that country under each country node.

A filter operation (the `where` keyword) evaluates to its left-argument input tree with positions removed based on the right-argument condition. Thus, a filter operation removes zero or more positions from each terminal node in the input tree, and does not modify anything else about the tree. As mentioned in sec. 3.1.2, the condition inside a filter operation applies to either a position or a group:

- ***Position* condition:** `.InstrumentType != "Bond"` applies to a *position*. When used as the condition of a filter operation it removes from all terminal nodes the positions whose `InstrumentType`

property equals `"Bond"`.

- ***Group* condition:** `sum .Value of Country < 10000000` applies to a *group* (in this example the *Country* group). When used as the condition of a filter operation, for each *Country*-level node it removes *all* positions that are children of this node if the sum of the `Value` property for these positions is less than 10 million.

Thus, a position condition evaluates to a boolean for each *position*, while a group comparison evaluates to a boolean for each *group*.

The condition in a filter operation is evaluated inside the same environment as the body of a grouping iteration, which is specified in section 4.1.2.4 below. In effect this means that the condition is evaluated once for each terminal node in the input tree, and that positions are removed from the given terminal node based on what the condition evaluates to inside the environment for this particular terminal node.

4.1.1.4 Calculation

A calculation transforms a grouping into a number. For example: `sum .Value of Portfolio` calculates the sum of the `Value` property for all positions in the `Portfolio` grouping, while `count (Portfolio grouped by .Country)` calculates the number of distinct countries in the `Portfolio` grouping.

A calculation may be a reduction of a particular property of all the positions in a grouping into a value (*sum/average/minimum/maximum*). For example, the *sum* calculation on the `Exposure` property, performed on some grouping, takes the value of the `Exposure` property for all positions in that grouping and returns the sum. The syntax for this is `sum .Exposure of grouping`. In this expression `grouping` is a variable of type `Grouping<Pos>` which `.Exposure of` takes as argument and transforms into the type `Grouping<tPropVal>`. The `sum` operation then takes this value as input — failing at runtime if `tPropVal` is not the `Number` type — and reduces the grouping to a single `Number`.

A calculation may also be a count of the number of groups in a grouping. For example, a *count* on `Portfolio grouped by .Country` returns the total number of `Country` groups, while a count on `Portfolio grouped by .Country grouped by .Sector` returns the total number of `Sector` groups. Thus, the *count*-operation evaluates the input grouping to a tree and returns the number of terminal nodes in this tree (see fig. 4). For example, a *count* performed on the grouping in fig. 2 returns 3, and for the grouping in fig. 3 it returns 5.

Lastly, a relative comparison between two of the above can be performed, resulting in the relative size of the left argument to the right argument in percent. For example, comparing the two constants 7 and 10 — using the concrete syntax `7 relative to 10` — returns 70%.

4.1.1.5 Comparison

The result of a calculation and a constant can be compared to each other. A comparison consists of **(a)** the two values to be compared, and **(b)** the *comparison operation*, which tests

for any combination of equality and greater/less than (==, !=, <, >, <=, >=). For example, the comparison `sum .Value of Country < 100000`) is a comparison between `sum .Value of Country` and the constant `100000` using the comparison operation *less than*. In the Grouping section above (4.1.1.3) the condition (`sum .Value of Country relative to Portfolio >= 10%`) is also a comparison — specifically between the two values `sum .Value of Country relative to Portfolio` and `10%` using the *greater than or equal* comparison operation.

The result of a comparison is a boolean value that can be used as input to any combination of logical *and/or/not*.

4.1.2 Statement

A compliance rule is composed of one or more statements, with a single statement per line. A statement is either a variable definition (`let`), an *if* -statement (`if`), a grouping iteration (`forall`), or a rule requirement (`require`). The following example compliance rule uses all four of these constructs:

```
1  let portfolioBonds = Portfolio where .InstrumentType == "Bond"
2  let countryBonds = portfolioBonds grouped by .Country
3  forall countryBonds {
4     let countryBondValue = sum .Value of Country relative to portfolioBonds
5     require countryBondValue <= 20%
6     if countryBondValue > 15% {
7        require count (Country grouped by .Issuer) >= 8
8     }
9  }
```

The above compliance rule has two requirements for *bonds*, i.e. the positions with an `InstrumentType` property equal to the string `"Bond"`. These two requirements are defined using the `require` keyword on line 5 and 7. The two requirements are: for each country **(1)** the value of bonds of that country relative to all bonds in the portfolio must be at most 20%, **(2)** if this relative value is greater than 15% then the bond positions for that country must be composed of at least 8 different issuers.

4.1.2.1 Block statement

A *block statement* is composed of zero or more statements enclosed in curly braces:

```
1  {
2     statementA
3     statementB
4     statementC
```

```
5 }
```

Block statements never occur in isolation. They are always part of another statement: either an *if*-statement or a grouping iteration.

4.1.2.2 Let-binding The `let` keyword defines a variable. It has the form `let <ident> = <expr>`. Here `<ident>` is the variable name, which is a text string starting with a single lower-case letter followed by zero or more alphanumeric characters. The `<expr>` is an expression, as defined above — i.e. either a grouping, a calculation, a comparison or a literal.

The lines below a variable's definition is the scope of the variable. However — as in languages such as C or Java — variables defined inside a block statement are visible only within this block; i.e. the block is the variable's scope.

A variable-definition shadows a previously defined variable of the same name. For example, in the following code the `require`-statement is evaluated inside an environment with `b` bound to **"hello"** and `a` bound to **false** (thus shadowing the definition of `a` on the first line).

```
1 let a = 7
2 let b = "hello"
3 let a = false
4 require …
```

4.1.2.3 Rule requirement A rule requirement-statement has the form `require <boolExpr>`, where `<boolExpr>` is an expression of type `Bool`. A compliance rule may contain multiple rule requirement-statements (as in the example rule in sec. 4.1.2 above), in which case *all* the requirements must evaluate to **true** for the compliance rule to pass. Thus, there is an implicit logical *and* between two rule requirement-statements, which means the following two example rules are equivalent:

```
1 require a
2 require b
```

and

```
1 require a AND b
```

This also implies that the *order* of rule requirements has no effect on the semantics of the compliance rule. Thus, in the example rule above (sec. 4.1.2), moving the first rule requirement (`require`

`countryBondValue <= 20%`) down below the end of the *if*-statement (below line 8) does not change the semantics of the rule.

However, the actual *effect* of a rule requirement-statement on the process of evaluation depends on the evaluator in question. For example, a "fail-fast" evaluator — whose purpose is to report back as quickly as possible whether a compliance rule has been violated — might only evaluate a single requirement-statement and report back if this requirement evaluates to **false**. In this case, switching two requirement-statements may cause a change in behaviour during evaluation, but the semantics of the compliance rule is unchanged (*all* requirements must evaluate to **true**).

As explained in 4.1.1.3, a position condition is one that applies to a position, rather than to a group. It should be noted that such a comparison is *only* valid as the condition of a **`where`**-operation, and not as the argument to e.g. **`require`**. Thus, the rule requirement **`require`** `(.InstrumentType != "Bond")` is not valid, since the expression evaluates to *multiple* boolean values (one for each position), as opposed to e.g. **`require`** `(sum .Value of Portfolio >= 100000)` which evaluates to a single boolean value.

4.1.2.4 Grouping iteration

A grouping iteration statement has the form **`forall`** `<groupingExpr>` `<blockStatement>`. The `<groupingExpr>` is a grouping expression as defined in sec. 4.1.1.3, and the `<blockStatement>` is a block statement that is the scope of the iteration. This block statement is evaluated once for each root-node-to-terminal-node path in the tree that is the result of evaluating the grouping expression. For each root-node-to-terminal-node path, the block statement is evaluated in a variable-environment that binds the name of the level (e.g. *Country*, *Issuer*) to the node at that particular level in the given path.

For example, consider the tree that results from evaluating the grouping expression `Portfolio` **`grouped by`** `.Country` **`grouped by`** `.Issuer` (depicted as the right-most tree in fig. 4). This tree has three distinct levels (*Portfolio*, *Country*, and *Issuer*) and five distinct root-node-to-terminal-node paths:

1. *Portfolio* → *Country*/**GB** → *Issuer*/**I1**
2. *Portfolio* → *Country*/**GB** → *Issuer*/**I2**
3. *Portfolio* → *Country*/**DK** → *Issuer*/**I3**
4. *Portfolio* → *Country*/**US** → *Issuer*/**I4**
5. *Portfolio* → *Country*/**US** → *Issuer*/**I5**

The block statement in this example is thus evaluated five times. The first time with the variable `Country` bound to the tree starting at the **GB**-node, and the variable `Issuer` bound to the tree starting at the **I1**-node. The second time with the variable `Country` bound to the tree starting at the **GB**-node (same as the first time), and the variable `Issuer` bound to the tree starting at the **I2**-node. And so on and so forth, with the last evaluation binding the `Country`-variable to the tree starting at the **US**-node and the `Issuer`-variable bound to the tree starting at the **I5**-node.

Note that the tree bound to the `Portfolio`-variable is not changed inside the body of a grouping iteration. One option would be to bind the `Portfolio`-variable to the root node of the grouped tree (the right-most tree in fig. 4), while the other option is to leave the `Portfolio`-variable unchanged, so that it remains bound to an ungrouped tree (the left-most tree in fig. 4). The decision has been made to leave the `Portfolio`-binding unchanged, so that in the below example rule, **`count`** `Portfolio` evaluates to 1 both inside the grouping iteration and outside it:

```
1 let a = count Portfolio // evaluates to 1
2 forall Portfolio grouped by .Country grouped by .Issuer {
3     let b = count Portfolio // also evaluates to 1
4 }
```

This has been decided to avoid the confusion that may result from implicitly redefining an existing variable, and because counting the number of issuers can be done simply by binding the grouping `Portfolio` **`grouped by`** `.Country` **`grouped by`** `.Issuer` to a variable and applying **`count`** to this variable.

4.1.2.5 If-statement An if-statement has the form **`if`** `<boolExpr>` `<blockStatement>`, where `<boolExpr>` is an expression of type `Bool` and `<blockStatement>` is a block statement. If the `<boolExpr>` evaluates to **true** then the contents of the block statement is evaluated when evaluating the compliance rule, otherwise it is ignored.

4.2 Example rules

This section expresses the example rules from section 2.1.2 using the proposed DSL.

4.2.1 Rule I

Rule I (sec. 2.1.2.1) expressed in the DSL looks as follows:

```
1 let issuers = Portfolio grouped by .Issuer
2 forall issuers {
3    require sum .Value of Issuer relative to Portfolio <= 10%
4 }
5 let issuersAbove5Pct = issuers where (sum .Value of Issuer relative to Portfolio > 5%)
6 require sum .Value of issuersAbove5Pct <= 40%
```

The first line groups the portfolio positions by issuer, and binds this grouping to the variable `issuers`, so that it can be reused in the two sub-rules (lines 2-4 and line 5-6, respectively) that comprise this rule.

The `forall` keyword on line 2 starts an iteration, with the effect that the single line within the curly braces (line 3) is executed for each *issuer*-group in the `issuers`-grouping — each time with a different group bound to the `Issuer` variable.

Line 5 performs a filtering on the grouping bound to the `issuers` variable, as required by the rule, and binds this to the variable named `issuersAbove5Pct`. Finally, on line 6, the requirement of the second sub-rule is asserted.

4.2.2 Rule II

Rule II (sec. 2.1.2.2) looks as follows:

```
1 let issuers = Portfolio grouped by .Issuer
2 forall issuers {
3    let issuerValue = sum .Value of Issuer relative to Portfolio
4    require issuerValue <= 35%
5    let issueCount = count Issuer grouped by .Issue
6    if issuerValue > 30% {
7       require issueCount >= 6
8    }
9 }
```

As the first two lines are identical to the previous rule, we start at line three, in which the value of an issuer (relative to the portfolio value) is bound to the variable `issuerValue`. Line number four requires that this be less than or equal to 35% for all issuers.

On line five the number of issues for the current issuer is bound to the variable `issueCount`. This variable is then used in the lines below — lines 6-8 — in a conditional statement that requires that if the value of the issuer is greater than 30%, then the issue count must be greater than or equal to six.

4.2.3 Rule III

Rule III (sec. 2.1.2.3) looks as follows:

```
1 let govtSecurities = Portfolio where (.InstrumentType == "GovernmentBond" OR .InstrumentType ==
      "StateBond")
2 forall govtSecurities grouped by .Issuer {
3    let issuerValue = sum .Value of Issuer relative to Portfolio
4    if issuerValue > 35% {
5       let issues = Issuer grouped by .Issue
```

```
6         require count (issues >= 6)
7         forall issues {
8            require sum .Value of Issue relative to Portfolio <= 30%
9         }
10     }
11  }
```

The first line binds to the variable `govtSecurities` the positions from the portfolio that are either government bonds or state bonds. Note that the data format here — namely, that positions which are state/-government bonds have a property by the name of `InstrumentType` that is equal to `"StateBond"` and `"GovernmentBond"`, respectively — is used as an example only (the DSL does not specify the property names/contents for the input data).

On line number two, the above-created variable is grouped by the *Issuer* property and iterated over. Line three binds the relative value of the issuer to the variable `issuerValue`. Line four introduces the conditional that the rule requires: only if the relative value of the issuer is greater than 35% should the issue count and issue value be checked.

4.2.4 Rule IV

Rule IV (sec. 2.1.2.4) looks as follows:

```
1  let otcPositions = Portfolio where (.InstrumentType == "OTC")
2  let nonApprovedCounterparties = otcPositions where (.Counterparty == "SmallCompanyX" OR
         .Counterparty == "SmallCompanyY" OR .Counterparty == "SmallCompanyZ")
3  let approvedCounterparties = otcPositions where (.Counterparty == "HugeCorpA" OR .Counterparty ==
         "HugeCorpB" OR .Counterparty == "HugeCorpC")
4  forall nonApprovedCounterparties grouped by .Counterparty {
5     require sum .Exposure of Counterparty relative to Portfolio <= 5%
6  }
7  forall approvedCounterparties grouped by .Counterparty {
8     require sum .Exposure of Counterparty relative to Portfolio <= 10%
9  }
```

Line 1 filters off the positions whose `InstrumentType` property does not equal `"OTC"`.

Secondly, in order to implement this rule it must be known how to identify approved and non-approved counterparties. In the above interpretation, the property `Counterparty` contains the name of the counterparty, and certain names are approved while others are non-approved. Line 2 and 3 binds to two variables the positions with non-approved and approved counterparties, respectively. The two grouping iterations then require the respective limit for each `Counterparty`-group.

4.2.5 Rule V

Rule V (sec. 2.1.2.5) looks as follows:

```
1  let securities = Portfolio grouped by .SecurityID
2  let betterThanBBB = securities where (.Rating == "AAA" OR .Rating == "AA" OR .Rating == "A")
3  let notBetterThanBBB = securities where (NOT (.Rating == "AAA" OR .Rating == "AA" OR .Rating ==
       "A"))
4  forall betterThanBBB {
5     require sum .Value of SecurityID relative to Portfolio <= 5%
6  }
7  forall notBetterThanBBB {
8     require sum .Value of SecurityID relative to Portfolio <= 1%
9  }
```

Line 1 groups the portfolio positions by the property name `SecurityID`. This property is assumed to be unique for each individual security, thus fulfilling the ".. *in any single security* ..."-part of this rule. Line 2 and 3 are similar to the same lines in the previous rule, except that line 3 simply applies a **`NOT`** to the condition used in line 2. Thus, all positions in the portfolio are either in `betterThanBBB` or `notBetterThanBBB` — whereas the previous rule ignores positions that don't have one of the specified counterparties. Lastly, the last two statements apply the respective limit to each security-group.

4.2.6 Rule VI

Rule VI (sec. 2.1.2.6) looks as follows:

```
1  let homeCountry = "DK"
2  let foreignCountryPositions = Portfolio where (.Country != homeCountry)
3  let foreignCountryValue = sum .Value of foreignCountryPositions relative to Portfolio
4  let foreignCountryCount = count (foreignCountryPositions grouped by .Country)
5  if foreignCountryValue >= 80% {
6      require foreignCountryCount >= 5
7  }
8  if foreignCountryValue >= 60% {
9      require foreignCountryCount >= 4
10 }
11 if foreignCountryValue >= 40% {
12     require foreignCountryCount >= 3
13 }
14 if foreignCountryValue <  40% {
15     require foreignCountryCount >= 2
```

```
16 }
```

This rule first binds all foreign-country positions to the variable `foreignCountryPositions`. Then it calculates the value of these foreign-country positions and binds the result to the variable `foreignCountryValue`. Line 4 counts the number of foreign countries, by grouping `foreignCountryPositions` by the `Country` property and applying *count* to it, and binds this result to `foreignCountryCount`. The foreign-country value and count are then used in a series of *if* -statements to implement the requirements of the rule.

4.3 Transformation pass

In order to allow for simpler concrete syntax, while adding complexity to neither the parser nor the evaluator, the abstract syntax produced by the parser goes through a *transformation pass* before being given as input to the evaluator.

In the current version of the DSL, the only pass is a transformation of a relative comparison in which one of the arguments is a grouping calculation (e.g. `average .Exposure of Country`) and the other argument is simply a grouping (e.g. `Portfolio`). An expression of this form is transformed into the equivalent — but more verbose — form in which both arguments of the relative comparison is a grouping calculation, i.e. `average .Exposure of Country relative to average .Exposure of Portfolio`.

Other transformation passes that could be relevant to consider in future versions of the DSL include optimizations. E.g., moving a let-binding out of the body of `forall`-statement if its right-hand side only references variables in the outer scope. This generic mechanism, of applying one or more transformation passes, separates the complexity of each pass, and is inspired by the Glasgow Haskell Compiler (Peyton Jones and Santos 1998).

4.4 Rule evaluation

This section describes how a portfolio compliance rule is evaluated to a single boolean value, with *false* meaning one or more requirements have been violated and *true* meaning no requirements have been violated. This evaluator is simplified, in order to lower complexity, and will fail in case of any of the following:

- Any form of type error, e.g.:
  - A comparison of two incompatible values: e.g. a *string* and a *number*
  - Applying an operator to a variable of an incorrect type, e.g.:
    - `if` or `require` applied to a non-boolean
    - `count` applied to anything but a grouping

- A reference to a variable that does not exist
- A position that does not contain the specified property name

4.4.1 Runtime values

The evaluator needs to represent three kinds of values at runtime:

1. A constant: *number*, *string*, *boolean*, *property name*, or *percentage*
2. A `Position` type (`Pos` in table 2)
3. A `Tree` type, which describes the result of evaluating a grouping

4.4.1.1 Literals A constant is used to hold all literals present in the given rule. Also, a constant is the result of evaluating both a comparison and a calculation. The evaluation of these two expression types is described below.

4.4.1.2 `Position` A position is represented at runtime as a value of the `Position` type, which is a map from a property name string to a value of type *number*, *string*, or *boolean*.

4.4.1.3 `Tree` The `Tree` data type is an implementation of the grouping data structure described in sec. 3.2. Its definition as a Haskell sum type looks as follows. Refer to section 5.1.1 for an overview on Haskell sum types.

```
1  data Tree termLabel =
2        Node (NodeData [Tree termLabel])
3      | TermNode (NodeData termLabel)
4
5  data NodeData a = NodeData (FieldName, FieldValue) a
```

The `NodeData` type contains information about a particular node. It contains the *property name* of the group that the node represents (e.g. *Country*) as well as the value of that property for this particular group (e.g. *"DK"*).
Note: in the implementation the `FieldName` type is synonymous with the `PropName` type in table 2, and `FieldValue` is synonymous with `tPropVal` in the same table. In general, the implementation uses the word *field* instead of *property*, and these two terms should be considered synonymous.
In the evaluator, the `a` type argument is instantiated to `[Position]`, thus making each terminal node contain zero or more positions.

A `Tree` value is either a:

1. *terminal node* (`TermNode`) containing a `NodeData` (which contains a single value of type `termLabel`); or
2. a *non-terminal node* (`Node`) containing zero or more sub-trees (inside a `NodeData`)

Thus, the example portfolio grouped by country (fig. 2) represented using this tree type looks as follows. Here, the positions named $P1$ through $P8$ in fig. 2 are represented as the strings `"P1"` through `"P8"`. In practice, values of the `Position` type would be present instead of these strings, but the actual positions have been left out for brevity.

```
Node $ NodeData ("Portfolio","")
    [ TermNode $ NodeData ("Country", "GB") ["P1", "P2", "P3"]
    , TermNode $ NodeData ("Country", "DK") ["P4"]
    , TermNode $ NodeData ("Country", "US") ["P5", "P6", "P7", "P8"]
    ,
    ]
```

The tree contains a root node (`Node $ NodeData ("Portfolio","")`) with a property name of "Portfolio" and a property value of the empty string. The children of the root node are present in the list that is the second argument to the root node's `NodeData`. These are the terminal node GB (containing the positions $P1$, $P2$, $P3$), the terminal node DK (containing the position $P4$), and the terminal node US (containing the positions $P5$, $P6$, $P7$, $P8$) — as depicted in fig. 2.

The tree used by the evaluator to represent position groupings at runtime is thus of type `Tree [Position]`. That is, every terminal node in the tree contains zero or more positions. The type-variable `termLabel` is present in order to more easily support the above example (using strings in place of positions), as well as coming in handy when the variable environment for a grouping-iteration is created (see sec 4.4.3.1 below).

4.4.2 Let-binding

The variable environment consists of a list of name/runtime value pairs. New variable definitions are added at the head of the list, and variable names are looked up starting from the head of the list as well. The effect is that more recently declared variables will shadow variables declared earlier in case of duplicate variable names. The initial variable environment contains a single item which pairs the name *Portfolio* with the `Tree` that contains the positions of the portfolio (see 3.3.1).

4.4.3 Statements

4.4.3.1 Grouping iteration As mentioned in sec. 4.1.2.4, the block statement of a grouping iteration is evaluated once for each root-node-to-terminal-node path in the tree that the input grouping expression

evaluates to. In practice, this means that the evaluator needs to create one variable environment for each terminal node.

Fig. 5 below adds a variable environment (depicted as a rounded, red rectangle) to all terminal nodes of fig. 3 (the tree that `Portfolio grouped by .Country grouped by .Issuer` evaluates to). The arrow describes a binding in the variable environment, with the left-hand side being the variable *name* and the right-hand side the *value* that this variable is bound to (which is a tree in the below example). A tree is referred to as the label of its root node. Thus, for example, *Country* $\mapsto$ **US** describes a binding of the variable name *Country* to the tree starting at the node with the label US.

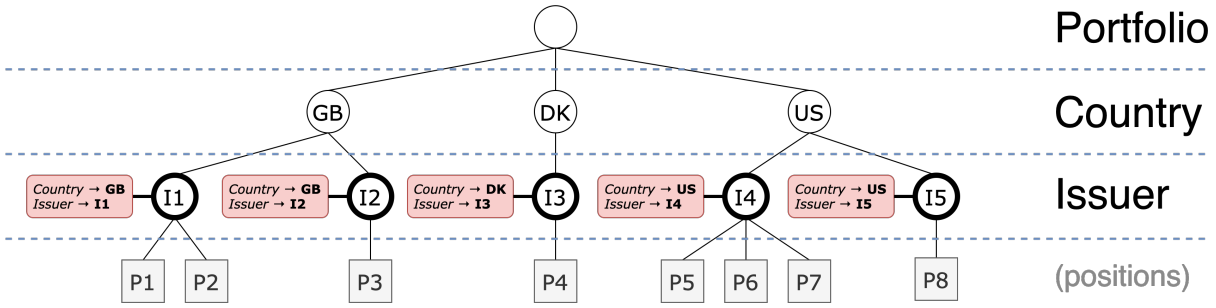

Figure 5: Tree with variable environments as leaf nodes

This variable environment is merged with the existing variable environment (for the code block that contains the grouping iteration), such that variables in the existing variable environment are hidden in case of duplicate variable names.

5 Implementation

The implementation consists of the following parts:

1. Abstract syntax (sec. 5.1 below)
2. Parser (sec. 5.2 below)
3. Pretty-printer (sec. 5.4 below)
4. Simple boolean evaluator (sec. 4.4)
5. Tree data type (sec. 4.4.1.3)

5.1 Abstract syntax

5.1.1 Haskell sum types

The following subsections describe the implementation's abstract syntax as Haskell *sum types*. A Haskell sum type defines a *type*, as well as one or more *values* (separated by `|`) that are all of the specified type. Thus, the definition `data Color = Red | Green | Blue` defines the *type* `Color` and the *values* `Red`, `Green`, `Blue` (which are all of type `Color`). The defined values can also contain other values, which is specified using one or more types after the value name. Thus, `data PersonInfo = AgeYears Int | WeightKilogram Float | FirstLastName String String` defines three values of type `PersonInfo` containing, respectively: age (integer); weight in kilograms (floating point value); first and last name (two strings).

Haskell sum types may be recursive, meaning that the values of a newly defined type may contain values of its own type, such that `data IntList = Empty | ItemAndRest Int IntList` defines a integer list type `IntList` that is either empty or contains a single integer plus another integer list (that, again, may be empty or contain a single integer plus another integer list).

5.1.2 Literals

Literals are constants entered by the user. A `FieldName` is the name of a property in a position. A `FieldValue` is the value of a property. The `Percent` type is used for relative comparisons — a comparison between two value expressions (see 5.1.3).

A `FieldValue` can be one of three types: *number*, *string*, or *boolean*. The implementation uses `Double` to represent numbers, but other implementations are free to use a different type for numbers if, for example, precision is more important than speed.

```
data Literal
    = Percent Number
    | FieldName Text
    | FieldValue FieldValue

data FieldValue
    = Number Double
    | String Text
    | Boolean Bool
```

5.1.3 Value expression

A *value expression* (`ValueExpr`) is either a:

- `GroupCount`: a count of the number of groups in a grouping (**`count`** keyword)
- `FoldMap`: a combined fold and map on a group (e.g. **`sum`** `.Value` **`of`** `Portfolio`)
- `Relative`: a relative comparison between two value expressions (separated by the **`relative to`** keyword)

```
data ValueExpr
    = GroupCount Expr
    | FoldMap Fold Expr
    | Relative Expr Expr

data Fold
    = Sum
    | Avg
    | Max
    | Min
```

For example, the concrete syntax **`sum`** `.Value` **`of`** `Country` **`relative to`** `Portfolio` — the value of the group *Country* relative to the value of the group *Portfolio* — is represented as follows in abstract syntax. *NB: see the section* **Top-level expression** below *(5.1.6) for a specification of* `Map`:

```
Relative
    (ValueExpr (FoldMap Sum (Map (Literal (FieldName "Value")) (Var "Country"))))
    (ValueExpr (FoldMap Sum (Map (Literal (FieldName "Value")) (Var "Portfolio"))))
```

The abstract syntax is thus a bit more verbose in this case, with the benefit of allowing more complex relative comparisons, e.g. **`average`** `.Exposure` **`of`** `Issuer` **`relative to`** `.Value` **`of`** `Country`.

Note that — as mentioned in section 4.3 — there is a difference between, on the one hand, the output of parsing e.g. `sum .Value of Country relative to Portfolio` and, on the other hand, the input expected by the evaluator. The parser will output simply `Var "Portfolio"` as the second argument to `Relative`, whereas the evaluator expects both arguments of `Relative` to be a `ValueExpr`.

5.1.4 Boolean expression

A *boolean expression* (`BoolExpr`) is either a comparison of two value expressions (`Comparison`), a negation of another boolean expression (`Not`), or logical *and/or* between two boolean expressions (`And`, `Or`).

```
1  data BoolExpr
2      = Comparison Expr BoolCompare Expr
3      | And Expr Expr
4      | Or Expr Expr
5      | Not Expr
6
7  data BoolCompare
8      = Eq   -- equals
9      | NEq  -- does not equal
10     | Lt   -- less than
11     | Gt   -- greater than
12     | LtEq -- less than or equal
13     | GtEq -- greater than or equal
```

In order to simplify the pretty-printer, the `BoolCompare` type contains all combinations of equality and order comparisons. In principle, it would be sufficient to have only `Eq` and either `Lt` or `Gt`, which could then be combined using `Or` and `Not` to produce the remaining comparisons. E.g. `a <= b` would be expressed as `Or (BoolExpr (Comparison (Var "a") Lt (Var "b"))) (BoolExpr (Comparison (Var "a") Eq (Var "b")))`.

5.1.5 Grouping expression

A *grouping expression* is modelled as a `DataExpr`:

```
1  data DataExpr
2      = GroupBy Expr Expr
3      | Filter Expr Expr
```

The first argument of both `GroupBy` and `Filter` is the input `DataExpr`. The second argument of `GroupBy` is a property name, by which the input `DataExpr` is to be grouped. The second argument of `Filter` is

a `BoolExpr`. Below, two examples of abstract syntax for a `DataExpr` are given, the first containing a *position* condition and the next containing a *group* condition.

The grouping expression `Portfolio where (.InstrumentType == "OTC") grouped by .Counterparty` — which contains the position comparison `.InstrumentType == "OTC"` — looks as follows in abstract syntax:

```
DataExpr
  (GroupBy
     (DataExpr
        (Filter
           (Var "Portfolio")
           (BoolExpr
              (Comparison
                 (Literal (FieldName (FieldName "InstrumentType")))
                 Eq
                 (Literal (FieldValue (String "OTC")))))))
     (Literal (FieldName (FieldName "Counterparty"))))
```

The grouping expression `Portfolio grouped by .Country where (sum .Value of Country relative to Portfolio >= 10%)` — which contains the group comparison `sum .Value of Country relative to Portfolio >= 10%` — looks as follows in abstract syntax:

```
DataExpr
  (Filter
     (DataExpr (GroupBy (Var "Portfolio") (Var "Country")))
     (BoolExpr
        (Comparison
           (ValueExpr
              (Relative
                 (ValueExpr
                    (FoldMap
                       Sum
                       (Map (Literal (FieldName (FieldName "Value"))) (Var "Country"))))
                 (ValueExpr
                    (FoldMap
                       Sum
                       (Map (Literal (FieldName (FieldName "Value"))) (Var "Portfolio"))))
           GtEq
           (Literal (Percent 10)))))
```

5.1.6 Top-level expression

The `Expr` type has two purposes:

1. It wraps the four types defined above (`Literal`, `ValueExpr`, `BoolExpr`, `DataExpr`)
2. It adds two new expressions:
    1. `Map`: the second argument to `FoldMap`
    2. `Var`: a variable identifier

```
data Expr
    = Literal Literal
    | ValueExpr ValueExpr
    | BoolExpr BoolExpr
    | DataExpr DataExpr
    | Map Expr Expr
    | Var Text
```

The wrapping is done in order to have a single type that can contain all expressions. The four wrapped types are defined as separate types in order to improve code readability.

The `Map` value represents a *map* over a tree of *positions* into a tree of *field values*. The concrete syntax `.Exposure of Country` parses into `Map (Literal (FieldName "Exposure")) (Var "Country")`. It represents the operation of taking the `Country` grouping, which evaluates to a tree with *positions* as leaf nodes, and converting into a tree with *property values* as leaf nodes (specifically, the value of the position's `Exposure` property).

The `Var` value represents a reference to variable name defined via `Let` (see 5.1.7 below).

5.1.7 Rule expression

The `RuleExpr` type represents the four types of statements described in section 4.1.2: let-binding (`Let`), grouping iteration (`Forall`), *if*-statement (`If`), and rule requirement (`Rule`).

```
data RuleExpr
    = Let Text Expr
    | Forall Expr [RuleExpr]
    | If Expr [RuleExpr]
    | Rule Expr
```

A single `RuleExpr`-value describes a single statement. Therefore, a compliance rule is described as a *sequence* of one or more `RuleExpr`-values. Similarly, a block statement (4.1.2.1) is a sequence of *zero or more* `RuleExpr`-values (the second argument to both `Forall` and `If`).

The `Let`-value describes a variable-definition. The first argument is the variable name and the second argument is the expression that the variable is bound to. Given a list of `RuleExpr` that defines a compliance rule, the let-binding has scope in the `RuleExpr` that follow. Thus, `[Let "a" (Literal (FieldValue (Bool True))), Rule (Var "a")]` defines a compliance rule which, first, binds the variable *a* to **true** (`let a = true`), and then uses this definition in a rule requirement (`require a`).

The `Forall`-value describes an iteration over a grouping. The first argument is the `DataExpr` to iterate over, and the second argument is the body of the iteration (see 4.1.2.4).

The `If`-value represents an *if*-statement. The first argument is the boolean condition, and the second argument is the block statement that is executed if the boolean condition evaluates to **true**.

Finally, the `Rule`-value represents a rule requirement (`require` keyword). Its only argument is the boolean condition of the rule requirement.

5.1.8 Examples rule

The following code block shows Rule III (sec. 4.2.3) in abstract syntax. This particular rule has been chosen because it uses all the constructs enumerated in table 1. The abstract syntax is formatted so that arguments to the same constructor appear at equal indentation levels. For example, the two `BoolExpr`-values at line 8 and line 13, respectively, are both arguments to the `Or`-value on line 7.

```
1  [ Let
2      "govtSecurities"
3      (DataExpr
4         (Filter
5            (Var "Portfolio")
6            (BoolExpr
7               (Or
8                  (BoolExpr
9                     (Comparison
10                       (Literal (FieldName (FieldName "InstrumentType")))
11                       Eq
12                       (Literal (FieldValue (String "GovernmentBond")))))
13                 (BoolExpr
14                    (Comparison
15                       (Literal (FieldName (FieldName "InstrumentType")))
16                       Eq
17                       (Literal (FieldValue (String "StateBond"))))))))))
```

```
18  , Let
19      "portfolioValue"
20      (ValueExpr
21         (FoldMap
22            Sum
23            (Map (Literal (FieldName (FieldName "Value"))) (Var "Portfolio"))))
24  , Forall
25      (DataExpr
26         (GroupBy
27            (Var "govtSecurities") (Literal (FieldName (FieldName "Issuer")))))
28      [ Let
29          "issuerValue"
30          (ValueExpr
31             (Relative
32                (ValueExpr
33                   (FoldMap
34                      Sum
35                      (Map (Literal (FieldName (FieldName "Value"))) (Var "Issuer"))))
36                (Var "portfolioValue")))
37      , If
38          (BoolExpr
39             (Comparison (Var "issuerValue") Gt (Literal (Percent 35))))
40          [ Let
41              "issues"
42              (DataExpr
43                 (GroupBy (Var "Issuer") (Literal (FieldName (FieldName "Issue")))))
44          , Rule
45              (BoolExpr
46                 (Comparison
47                    (ValueExpr (GroupCount (Var "issues")))
48                    GtEq
49                    (Literal (FieldValue (Number 6)))))
50          , Forall
51              (Var "issues")
52              [ Rule
53                  (BoolExpr
54                     (Comparison
55                        (ValueExpr
56                           (Relative
57                              (ValueExpr
58                                 (FoldMap
59                                    Sum
60                                    (Map (Literal (FieldName (FieldName "Value"))) (Var "Issue"))))
61                              (Var "portfolioValue")))
62                        LtEq
63                        (Literal (Percent 30))))
```

```
64                ]
65            ]
66        ]
67    ]
```

5.2 Parser

The parser is implemented using *parser combinators*. The two main ideas behind parser combinators are: **(a)** representing a parser as a *value* and, **(b)** combining two or more of these parsers into a new parser using *combinators*.

5.2.1 Parser combinators

As an educational example, consider a hypothetical DSL which consists of two commands: **(1)** `hello` which prints the text *hello*; and **(2)** `exit` which exits. We use the Haskell sum type `data Command = Hello | Exit` to represent one of these two commands as a value. Next, we define two parsers named: **(1)** `pHello`, which accepts the text string `"hello"` as input and returns the value `Hello`; and **(2)** `pExit`, which accepts the text string `"exit"` as input and returns the value `Exit`. Now, we may use the infix parser combinator `<|>` to combine `pHello` and `pExit` into a new parser `pCommand = pHello <|> pExit` which parses *either* the string `"hello"` and returns `Hello`, *or* the string `"exit"` and returns `Exit`.

Now, assume that we want to parse a source file for this language — the format of which is: zero or more commands separated by a newline character. For this we may use two other combinators:

1. The infix parser combinator `<*`, which combines the two parsers given as the left and right argument into a new parser that first runs the left parser, then runs the right parser, and returns the value of the left parser. Given a parser `pNewline` (which parses a single newline character) we can construct a parser for a single line in the source file like so: `pLine = pCommand <* pNewline`.
2. The function `many`, which takes a parser as input and returns a new parser that parses zero or more occurrences of the given parser, returning a list of the parsed values. Using this combinator we can define a parser for a source file: `pSource = many pLine`

We can then run the parser `pSource` on the following input source file:

```
1 hello
2 hello
3 exit
4 hello
5 exit
```

which returns the sequence of commands `[Hello, Hello, Exit, Hello, Exit]`. We permit ourselves to leave as unspecified the semantics of this program.

5.2.2 Overview

The parser implementation uses two Haskell libraries. Firstly, it uses the `megaparsec` library(Martini, Leijen, and Karpov 2020), which enables defining parsers that parse a text string into a Haskell value. Secondly, the implementation uses the library `parser-combinators`(Karpov and Washburn 2019), which offers a way to construct a parser for a top-level expression by combining a set of *operator*-parsers along with information about each operator's *fixity* (prefix/postfix/infix), precedence, and associativity.

The implementation can be divided into three main parts:

1. Parsing of *literals* and *variable names* — using primitives defined in the `megaparsec` library
2. Parsing of *expressions* (see 4.1.1) — using the `parser-combinators` library and a table of operators
3. Parsing of *statements* — using a combination of the two parsers above

In the code below, names prefixed with `M.` are defined by the `megaparsec` library. The combinators `many` and `<|>` are defined in the module `Control.Applicative`, which is part of the Haskell standard library (`base`).

5.2.3 Variable names

A variable name is either a reference to an existing variable, or the name of a new variable (after the **`let`** keyword). The parser for a reference to an existing variable is defined as:

```
1  pVarReference :: Parser Text
2  pVarReference = do
3      firstChar <- M.letterChar
4      remainingChars <- many M.alphaNumChar
5      let identifier = toS $ firstChar : remainingChars
6      -- prevent keywords from being parsed as
7      --  identifiers/variable references
8      if not $ identifier `elem` keywords
9          then return identifier
10         else M.failure Nothing (Set.fromList [])
```

It first parses a single alphabetic character (upper/lower case), followed by zero or more alphanumeric characters (upper/lower case). `toS` is a generic function that converts to/from a list of characters and the `Text` string type. Notably, this parser refuses to parse keywords as variable names, by failing if the

variable name parsed on the first two lines is present in the pre-defined list of keywords (`keywords`). A better solution to the problem of avoiding the parsing of keywords as variable references is desirable, primarily to avoid maintaining a separate list of keywords — which is not guaranteed to agree with the actual keywords used inside various parsers.

The parser for the variable name on the left-hand side of a let-binding is defined as:

```
pDefineVar = do
    word@(firstChar : remainingChars) <- toS <$> pVarReference
    if C.isLower firstChar
        then return (toS word)
        else failParse "Variable name must begin with lower case letter"
                [toS $ C.toLower firstChar : remainingChars]
```

This parser reuses the above parser for variable references (`pVarReference`), but fails unless the first character is lower case. In case of the first character being upper case, a suggestion — substituting the first character for its lower case counterpart — is provided to the user.

5.2.4 Literals

Parsing literals starts with defining parsers for a boolean (`pBool`), a string literal (`pStringLiteral`) and a number (`pNumber`). And then — using the `<|>` parser combinator — combining these three parsers into a parser for field values (`pFieldValue`):

```
pBool :: Parser Bool
pBool =
        pConstant "true" *> return True
    <|> pConstant "false" *> return False
  where
    pConstant str = M.try $ M.chunk str *> M.notFollowedBy M.alphaNumChar

pStringLiteral :: Parser Text
pStringLiteral = fmap toS $
    M.char '"' *> M.manyTill M.charLiteral (M.char '"')

pNumber :: Parser Number
pNumber = signed $
    M.try (fromReal <$> M.float) <|> fromIntegral <$> M.decimal
  where
    signed = M.signed (return ())

```

```
18 pFieldValue :: Parser FieldValue
19 pFieldValue =
20         Bool <$> pBool
21     <|> String <$> pStringLiteral
22     <|> Number <$> pNumber
```

In `pBool`, `M.chunk` matches a string (a sequence of characters). The combination of `M.notFollowedBy` and `M.try` makes sure that the `pConstant` parser does not match a longer string that simply *starts with* `str` — e.g. the string `"trueSomething"`. This is necessary for the `pBool` parser to not match e.g. variable names that are prefixed with **`true`** or **`false`**. See the paragraph regarding the `pNumber` parser below for an explanation of how `M.try` works. The `*>` combinator first runs the left parser, then the right parser, and returns the value of the right parser. Using `return`, a parser is defined which does not consume any input but only returns a value.

In `pStringLiteral`, a string literal is parsed by expecting a starting double quote character (`"`), followed by zero or more of the character defined by `M.charLiteral` until it reaches an ending `"`. `M.charLiteral` is defined by `megaparsec`, and allows parsing string literals containing escaped double quotes (prefixed with a backslash). For example, the input text `"he\"y"` will be parsed as the string `he"y`.

The `pNumber` parser parses either a floating point number or an integer, and in both cases converts this number into the internal representation used for a number (`Number`) using `fromReal` or `fromIntegral`. The `M.try` combinator is important. Given an input of e.g. `47`, the `pNumber` parser will first try to execute the floating point parser. The floating point parser will consume `47` and then fail, since it expects a `.` (followed by the fractional part of the floating point number). The `M.try` combinator makes the floating point parser *backtrack* on failure, meaning that — upon failure — it will set the current position in the input as if it hadn't consumed any input (even though it consumed `47` in the example). Continuing with the example, after the floating point parser fails, the `<|>` combinator will continue with the integer parser, which will succeed on the input `47`. If `M.try` *hadn't* been applied to the floating point parser, the right argument to `<|>` would not be executed, since the right argument to `<|>` is only tried if the left argument parser does not consume any input. Finally, `signed` (defined using `M.signed`) transforms any *number*-parser into the same number-parser that also accepts an optional minus (`-`) or plus (`+`) prefix, and modifies the output number accordingly (negating the number in case of a `-` prefix).

The final parser for literals (`pLiteral`) uses `pFieldValue` defined above, as well as a parser for a field name (`pFieldName`) and a percentage (`pPercentage`):

```
1 pFieldName :: Parser FieldName
2 pFieldName = do
3     varRef <- M.char '.' *> pVarReference
4     let word@(firstChar : remainingChars) = toS varRef
5     if C.isUpper firstChar
```

```
6           then return (fromString word)
7           else failParse "Field name must begin with upper case letter"
8                   [toS $ C.toUpper firstChar : remainingChars]
9
10  pPercentage :: Parser Number
11  pPercentage = pNumber <* M.char '%'
12
13  pLiteral :: Parser Literal
14  pLiteral =
15          FieldName <$> pFieldName
16      <|> percentOrFieldValue
17    where
18      percentOrFieldValue =
19          (Percent <$> M.try pPercentage
20          <|> FieldValue <$> pFieldValue) <* M.notFollowedBy M.alphaNumChar
```

The `pFieldName` parser is very similar to the `pDefineVar` parser. The only difference is that it expects a leading dot (`.`), and it rejects field names starting with a *lower case* character. The `fromString` function converts a list of characters into the internal representation for a field name (`FieldName`).

The `pPercentage` parser simply parses anything `pNumber` parses if it's followed by a percent (`%`) character. How it's used in the final `pLiteral` parser is important, however. Firstly, `M.try` must be applied to `pPercentage`, as `pPercentage` will consume any leading number, and fail unless a `%` follows (making `pLiteral` not try the alternatives). Secondly, `pPercentage` must be tried *before* the `pNumber` parser (part of `pFieldValue`), because otherwise the `pFieldValue` parser will succeed in parsing a percentage as simply a number (stopping before it reaches `%`).

The parser for `FieldValue` and `Percent` are required to not be immediately followed by an alphanumeric character. This prevents e.g. `5relative to 5` (notice the absence of whitespace between the first 5 and **`relative`**) from parsing, which would otherwise be accepted. Applying `M.notFollowedBy M.alphaNumChar` to the `pFieldName` parser would not make sense, as this parser does not stop until it has consumed all consecutive alphanumeric characters.

5.2.5 Expressions

In order to parse expressions, four helper functions are defined: `lexeme`, `kw`, `ks`, and `parens`.

```
1  lexeme :: Parser a -> Parser a
2  lexeme = M.lexeme spaceTab
3
4  kw :: Text -> Parser ()
5  kw input = lexeme . M.try $
```

```
6       M.chunk input *> M.notFollowedBy M.alphaNumChar
7
8   ks :: Text -> Parser ()
9   ks input = lexeme . M.try $
10      M.chunk input *> M.notFollowedBy symbolChar
11    where
12      symbolChar = M.oneOf ['=', '>', '<', '!']
13
14  parens :: Parser a -> Parser a
15  parens = M.between
16      (lexeme $ M.chunk "(")
17      (lexeme $ M.chunk ")")
```

The `lexeme` function transforms its input parser into a parser that discards trailing space and/or tab characters (`spaceTab` consumes zero or more tabs/and or spaces). Note that *leading* tabs/spaces are not discarded by `lexeme` — all parsers assume that leading whitespace has been consumed. Thus, `lexeme` is used to define parsers for input with optional trailing spaces/tabs.

The `kw` function defines a parser for a keyword — e.g. **`let`**, **`count`**, **`where`**, **`NOT`**. As with the parser for boolean constants (`pBool`), `kw` makes use of `M.notFollowedBy M.alphaNumChar` and `M.try` so that it doesn't match keyword-prefixed strings — thus making sure e.g. `counterParty` is parsed as a variable reference, rather than the **`count`** keyword followed by the variable name `erParty`.

The `ks` function defines a parser for a key-*symbol*, e.g. ==, >, <=. It is very similar to `kw`. The difference is that while `kw`'s use of `M.notFollowedBy` ensures that keyword-prefixed variables are not parsed as a keyword followed by a variable, `ks`'s use of `M.notFollowedBy` ensures that `ks` does not parse e.g >= as only *greater than* (>), thus leaving the = in the input behind and causing parser failure. It is a way of specifying that any number of consecutive `symbolChar` characters must be parsed in their entirety, rather than only matching some valid prefix.

The helper function `parens` simply transforms a parser of something into a parser of that same something surrounded in parentheses — with optional whitespace after both the opening and closing paren.

With these helper functions the expression parser `pExpr` can be defined as follows:

```
1   pExpr :: Parser Expr
2   pExpr =
3       makeExprParser term exprOperatorTable
4     where
5       term = lexeme $ parens pExpr <|> pTerm
6       pTerm = Literal <$> pLiteral <|> Var <$> pVarReference
7
8   exprOperatorTable :: [[Operator Parser Expr]]
9   exprOperatorTable =
```

```haskell
    [ [ InfixL $ kw "where" *> return (\a -> DataExpr . Filter a)
      , InfixL $ kw "grouped" *> kw "by" *> return (\a -> DataExpr . GroupBy a)
      ]
    , [ InfixN $ kw "of" *> return Map ]
    , [ Prefix $ kw "count"   *> return (ValueExpr . GroupCount)
      , Prefix $ kw "sum"     *> return (ValueExpr . FoldMap Sum)
      , Prefix $ kw "average" *> return (ValueExpr . FoldMap Avg)
      , Prefix $ kw "minimum" *> return (ValueExpr . FoldMap Min)
      , Prefix $ kw "maximum" *> return (ValueExpr . FoldMap Max)
      ]
    , [ InfixL $ kw "relative" *> kw "to" *> return (\a -> ValueExpr . Relative a) ]
    , [ InfixN $ ks "==" *> return (mkComparison Eq)
      , InfixN $ ks "!=" *> return (mkComparison NEq)
      , InfixN $ ks ">"  *> return (mkComparison Gt)
      , InfixN $ ks "<"  *> return (mkComparison Lt)
      , InfixN $ ks ">=" *> return (mkComparison GtEq)
      , InfixN $ ks "<=" *> return (mkComparison LtEq)
      ]
    , [ Prefix $ kw "NOT" *> return (BoolExpr . Not) ]
    , [ InfixL $ kw "AND" *> return (\a -> BoolExpr . And a) ]
    , [ InfixL $ kw "OR"  *> return (\a -> BoolExpr . Or a) ] ]
  where
    mkComparison numComp a b = BoolExpr $ Comparison a numComp b
```

The expression parser is defined using the `makeExprParser` function from the module `Control.Monad.Combinators.Exp` in the `parser-combinators` package. The `makeExprParser` function takes two arguments: **(1)** a parser for a *term*, which in this case is either literal, a variable reference, or an expression enclosed in parens; **(2)** a table of *operators*.

The table of operators is a list of `[Operator Parser Expr]` — thus making it a list of lists. The outer list is ordered by descending precedence, so that the *first* list of operators have the *highest* precedence. For example, the operator table above defines the **`where`** and `group` **`by`** operators as both having the highest precedence because they are in a list that is the first item in the outer list. This is followed by the **`of`** keyword, which has the next-highest precedence, and so on.

The `Operator` type contains values (`InfixL`, `InfixN`, and `Prefix`) used to define a single operator, by wrapping a *parser* in an `Operator`-value that defines the operator's *fixity* (infix/prefix/postfix) and — in case of infix operators — associativity (left/right). Thus, the parser for e.g. a left-associative infix operator is wrapped in the `InfixL` value, and the parser for a prefix operator is wrapped in the `Prefix` value. The parser, that is wrapped in an `Operator`-value, returns a *function* that either **(a)** takes a single argument (in the case of prefix and postfix operators); or **(b)** takes two arguments (in the case of infix operators). The type of these arguments must be the same as the type of the value returned by the term-parser (the first argument to `makeExprParser`) — which in our case is the `Expr` type (see 5.1.6).

Thus — as an example — given a parser of integers `pInt`, we can define a parser `pIntExpr` for a language that supports addition, subtraction and incrementing (using the `increment` keyword) like so:

```
pIntExpr =
    makeExprParser (lexeme $ parens pIntExpr <|> pInt) table
  where
    table = [ [ Prefix $ kw "increment" *> return (+1) ]
            , [ InfixL $ kw "+" *> return (+) ]
            , [ InfixL $ kw "-" *> return (-) ]
            ]
```

Such that `pIntExpr` parses e.g. the input `5 - 2 + increment 2` to the integer value zero.

5.2.6 Statements

In order to parse a statement, a parser for a block statement is needed (required by **`forall`** and **`if`**). The parser for a block statement `pRuleBlock` is defined as follows:

```
pRuleBlock :: Parser [RuleExpr]
pRuleBlock = M.between
    (lexeme (M.chunk "{") *> M.eol *> spaceTabNewline)
    (lexeme (M.chunk "}"))
    pRules
```

This is very similar to the `parens` function defined above. The difference is that a newline character (`M.eol`) is mandatory after the opening brace, and zero or more newline/space/tab characters (`spaceTabNewline`) may follow it. The `pRules` parser — which parses zero or more newline-separated `RuleExpr` — is defined below.

The way in which newline characters are consumed by `pRuleBlock` results in the following requirements regarding newline-consumption for `pRules` and the parser that precedes `pRuleBlock`. Since `lexeme (M.chunk "{")` does not remove any preceding whitespace, the parser that precedes `pRuleBlock` must consume all whitespace, including newlines, before it runs `pRuleBlock` (such that the current position in the parser is at the opening brace). For `pRules` it is required that all trailing whitespace, including newlines, is consumed, such that the current position in the parser is at the closing brace after `pRules` has been run.

Using `pRuleBlock` — and the previously defined helper functions — parsers are defined for a let-binding (`pLet`), a grouping iteration (`pForall`), an *if*-statement (`pIf`), and a rule requirement (`pRequire`):

```haskell
pLet :: Parser RuleExpr
pLet = do
    varName <- kw "let" *> lexeme pDefineVar
    expr <- lexeme (M.chunk "=") *> lexeme pExpr
    return $ Let varName expr

pForall :: Parser RuleExpr
pForall = do
    dataExpr <- kw "forall" *> lexeme pExpr <* spaceTabNewline
    block <- pRuleBlock
    return $ Forall dataExpr block

pIf :: Parser RuleExpr
pIf = do
    varOrBoolExpr <- kw "if" *> lexeme pExpr <* spaceTabNewline
    block <- pRuleBlock
    return $ If varOrBoolExpr block

pRequire :: Parser RuleExpr
pRequire = kw "require" *> (Rule <$> pExpr)
```

In order to fulfil the above-stated requirements for `pRuleBlock` regarding whitespace, the `pForall` and `pIf` parser consume all whitespace including newlines (via `spaceTabNewline`) before running the `pRuleBlock` parser .

Using the above four parsers, the parser for a single `RuleExpr` can be defined simply as:

```haskell
pRuleExpr :: Parser RuleExpr
pRuleExpr =
    pLet <|> pForall <|> pIf <|> pRequire
```

Since a compliance rule is composed of one or more `RuleExpr` — essentially a block statement without the surrounding braces — the `pRules` parser parses zero or more `pRuleExpr` separated by at least one newline. Here the choice has been made to accept an empty input file (i.e., containing zero rules), but a non-empty input file could easily be required by using `some` instead of `many`.

```haskell
pRules :: Parser [RuleExpr]
pRules = many (lexeme pRuleExpr <* M.eol <* spaceTabNewline)
```

The definition of `pRuleBlock` and `pRules`, as well as how `pRuleBlock` is used inside `pIf` and `pForall`, results in a newline character being optional *before* the opening brace, mandatory *after* the opening brace, and mandatory both before and after the *closing* brace.

The final parser for a source file, `ruleParserDoc`, removes any leading whitespace before running `pRules`. It also requires that the end of the input file is reached after the last `RuleExpr` has been parsed by `pRules`, which makes the parser fail in case the file ends with something unparsable (rather than just succeeding with what *can* be parsed from the file and ignoring the rest).

```
1  ruleParserDoc :: Parser [RuleExpr]
2  ruleParserDoc = spaceTabNewline *> pRules <* M.eof
```

5.2.7 Notes

5.2.7.1 On the choice of `megaparsec` A Haskell parser combinator library was chosen because it allows writing the parser in Haskell. This is in contrast to, for example, a parser generator like *happy*(Gill and Marlow 2019), which requires learning new syntax. A Haskell library was thus seen as the quickest way to arrive at a working parser.
The `megaparsec` library was chosen because, according to its authors, "*[it] tries to find a nice balance between speed, flexibility, and quality of parse errors.*"(Martini, Leijen, and Karpov 2020)

The first version of the parser was written entirely using `megaparsec` — without the use of the `parser-combinators` library. This version had several issues. Firstly, due to the recursive nature of the DSL's expressions — e.g. one of the arguments of a comparison operation can contain a **`where`** which, in turn, may contain another comparison operation — it was difficult to make sure that the parser terminated, and it often ended up in an infinite loop due to non-obvious reasons. In general, the interactions between the various different parsers was difficult to understand, and a change in one parser often broke a different parser that made use of the former parser.
This was solved using the `parser-combinators` library, which constructs the recursive expression-parser through the table of operators. This only requires writing the two non-recursive parsers for a literal and a variable reference.

If the parser were to be rewritten, a parser generator like *happy* would be chosen. Both to avoid the problem of non-termination that the `parser-combinators` library helps avoid, and also to enable warnings in case of an ambiguous grammar. Furthermore, a lexer would be employed, primarily to improve parser errors such as `unexpected "(In", expecting "NOT"`, which happen because the parser does not understand the boundary between tokens (a token is a single character in the current implementation).

5.2.7.2 Performance characteristics Due to the use of backtracking parser combinators, the worst-case theoretical running time of the resulting parser is exponential with respect to the input length. We argue, however, that this is not the case for our parser, because backtracking (use of the `M.try` combinator) is restricted to exactly three places:

1. In `pLiteral`: in which the `pPercentage` parser backtracks if the input is not immediately followed by a percentage sign
2. In `pNumber`: in which the floating-point parser backtracks if the consumed integer is not followed by a period
3. In the parsing of keywords and key-symbols: which backtracks if the keyword is followed immediately by an alphanumeric character (in which case it is parsed as an identifier instead)

and because these parsers call neither themselves nor other backtracking parsers.

5.3 Transformation pass

An implementation of the transformation pass described in sec. 4.3 was attempted, but it was discovered that type inference is needed in order to transform the short-hand **`relative to`**-expression into the longer form. For example, consider the following three statements:

```
1  let portfolioValue = sum .Value of Portfolio
2  require sum .Value of Country relative to portfolioValue
3  require sum .Value of Country relative to Portfolio
```

The first line requires no transformation, but simply defines a variable that is used in line 2. Importantly, line 2 does *not* require the described transformation, because the `portfolioValue` argument to **`relative to`** is a number. Line 3 *does* require the transformation, because the `Portfolio` argument to **`relative to`** is a grouping. In other words, line 2 is actually what we would like to transform line 3 into, but the only way to know that line 2 should *not* be transformed in the same manner as line 3 is by looking at the type of the right argument to **`relative to`**. If this type is a number, then the transformation must not be performed. Only if this type is a grouping (as in the example on line 3) should the expression be transformed. Consequently, due to type inference not being implemented, this transformation pass could not be implemented either.

Note, however, that this transformation is not central to the use of the DSL, as a let-binding can be used as the right-hand side of **`relative to`** (as in the above example) in order to avoid the long form — e.g. **`sum`** `.Value` **`of`** `Country` **`relative to sum`** `.Value` **`of`** `Portfolio`.

5.4 Pretty-printer

For the pretty-printer, a simple solution was chosen that does not use external libraries.

The pretty-printer for statements outputs a list of integer/string pairs, with each item in the list representing a single line in the output. The integer specifies the indentation level of the line while the string is the actual line contents. This indentation level is incremented for the lines inside a block statement.

Pretty-printing of expressions is considered unfinished. The pretty-printer was implemented with the assistance of property-based testing — which generates a piece of abstract syntax, prints it using the pretty-printer, and checks that the parser outputs the original abstract syntax. Using this test, the pretty-printer was adjusted until the test did not output any errors. This method, however, proved insufficient, as the sheer number of different combinations of abstract syntax meant that problematic expressions were not tested within a reasonable time limit. For example, printing the abstract syntax that results from parsing `a AND (b AND c)` incorrectly as `a AND b AND c` — which leaves out the parentheses, thus changing the meaning to `(a AND b) AND c`) — was not detected.

5.5 Bugs

The evaluator contains a bug that arises if a grouping is added to a tree that contains empty groups. If a filter-operation removes all positions from some group, then a subsequent grouping operation will not add a terminal node as child of this group/node, since there are no positions from which this node can be created. This will materialize as a "*Variable not found*"-error when this grouping is used as the argument to a grouping iteration, as the body of the grouping iteration references the group via a variable that has not been added to the environment for the iteration over the empty group.

For example the following example code might fail during evaluation with the error "*Variable 'Issuer' not found*":

```
1  forall Portfolio grouped by .Country where (.InstrumentType != "Bond") grouped by .Issuer {
2     require sum .Value of Issuer <= 10%
3  }
```

If the filter-condition causes all positions to be removed from a country-terminal node, then the subsequent grouping by issuer will not create a new *issuer*-terminal node as a child of the given country node. As a consequence, when this particular combination of groups is iterated over, the evaluator will not add a `Issuer` variable to the environment (because the node doesn't exist), which will cause the body of the iteration to reference a variable that doesn't exist.

6 Future work

6.1 Minor DSL additions

6.1.1 "If-elseif"-statement

In the implementation of Rule VI (sec. 4.2.6), note that the lack of an *if-then-else* construct means than the rule is less clear than could be. If e.g. `foreignCountryValue` is equal to 90%, then *all* of the three first rule requirements apply, since the condition of the *if*-statement is fulfilled in all three cases. This could be solved by adding an "*if-elseif*"-statement to the language, and using a series of "*if-elseif-elseif*..."-statements instead, as in:

```
...
if foreignCountryValue >= 80% {
    require foreignCountryCount >= 5
} else if foreignCountryValue >= 60% {
    require foreignCountryCount >= 4
} else if foreignCountryValue >= 40% {
    require foreignCountryCount >= 3
} else if foreignCountryValue <  40% {
    require foreignCountryCount >= 2
}
```

6.1.2 Property value of group in grouping iteration

In the case of the implementation of Rule IV (sec. 4.2.4), note that creating two distinct groupings and using these two groupings in two separate grouping iterations — instead of a single grouping iteration that contains an *if*-statement — is the only way to implement this rule. This is necessary because the DSL does not have a way to look up a group's property value inside a grouping iteration (i.e. the value inside the node when depicted as a tree in e.g. fig. 3). If this construct were added to the DSL, this rule could be implemented instead using a single grouping iteration (over `otcPositions grouped by .Counterparty`) which contains an *if*-statement that looks at the `Counterparty` value (e.g. `"SmallCompanyX"`) of the current `Counterparty` group, and applies one limit if this name is approved and another if it is not.

6.2 Grouping iteration as a boolean expression

A grouping iteration (`forall`) is currently a statement that can be used to apply a rule requirement for all groups in a grouping. This means that the DSL does not allow using a `forall` as e.g. the condition of an *if*-statement because it requires an expression that evaluates to a boolean. If a `forall`-statement were an

expression instead, a rule could be constructed in which a rule requirement applies only if e.g. the relative value of all groups in a grouping is greater than some value.
In addition, if this ability were added, an `exists` expression (representing existential quantification) could also easily be added, simply by rewriting a hypothetical `exists`-expression of the form `exists grouping { a }` into the equivalent hypothetical **`forall`**-expression: `NOT (forall grouping { NOT a })`.

The challenge regarding adding this capability to the DSL is mostly syntactic, as a **`forall`**-statement spans multiple lines while expressions are restricted to a single line in the source code of the compliance rule.

6.3 Type system

A type system can help reject invalid rules before evaluation is attempted. An informal type system is presented in table 2, but a more thorough analysis of the various constructs of the language is needed to establish a sound type system. For example, in the expression `.Value of Portfolio`, the `.Value` property name is used as a function that maps the positions inside `Portfolio` to each position's value. However, when used in e.g. the position condition `.InstrumentType != "Bond"` (see sec. 4.1.1.3) the property name `.InstrumentType` is compared to a *property value*, thus giving special meaning to a property name when used as part of a position condition. These two different meanings would need to be reconciled in a potential type system.

6.4 Let-binding of functions

In Rule III (sec. 4.2.3), the definition of what constitutes e.g. a "government security" is composed of several logical *or*s between comparisons of the property `InstrumentType` and some string, e.g.: `.InstrumentType == "GovernmentBond" OR .InstrumentType == "StateBond" OR .InstrumentType == "TreasuryBill" OR .InstrumentType == "StateBill"`. Adding the ability to factor out this logic into a function `isGovernmentSecurity p = p.InstrumentType == "GovernmentBond" OR p.InstrumentType == "StateBond" OR p.InstrumentType == "TreasuryBill" OR p.InstrumentType == "StateBill"` reduces duplication in the rule definition, as this logic can be defined once and used multiple times in the rule.

In addition, combined with rule input arguments (sec. 6.5), it separates the data format of positions (e.g. specific string names present in the property `IntrumentType`) from a compliance rule, thus making the rule more generic and increasing reusability.

6.5 Rule input arguments

All of the example rules contain one or more hardcoded constants — e.g. Rule II (sec. 2.1.2.2) specifies a relative limit of exactly 35%. With the goal of generalizing rules, the ability to add input arguments to rules could be added, so that e.g. Rule II keeps its basic structure — of requiring a maximum per-issuer value, as well as a certain issue-count — but accepts as arguments the exact limits, hereby easily changing the rule to have a limit of e.g. 25% and a minimum issue-count of e.g. 5.

Another example is the approved and non-approved creditors in Rule IV (sec. 2.1.2.4). Not only does this rule specify the exact names of which creditors are approved and which are non-approved, it also assumes that each position contains a property by the name of `Counterparty` which contains a string-encoded name of the counterparty. A better solution to this would be to take a *function* as input (see 6.4) — called e.g. `isApprovedCreditor` — which takes a position as input and returns **true** if the creditor is approved and **false** otherwise. How to determine whether a position is from an approved creditor or not would then be completely factored out of the rule, including both *which* creditors are approved but also *how* to determine this when given a position (thus not requiring a string name inside the `Counterparty` property).

6.6 User-defined calculations

The operators `sum`, `average`, `maximum`, and `count` need not be built into the DSL. These could be defined by the customer using a separate DSL and imported by a rule. This allows experts to define calculation primitives, so that the language doesn't have to be changed if a rule needs to calculate e.g. the *median* of a set of values.

In practice these functions are all a form of structural recursion over sets(Buneman et al. 1994) with optional preprocessing (e.g. calculating the average from a *(count, sum)*-pair). The advantage of defining calculation-functions in these terms is that it guarantees termination of the calculation.

6.7 Deriving data schema from rule

The requirements for property names and property types defined by a compliance rule can be statically derived from looking at the source code of the compliance rule. By looking at **(1)** references to a particular property name, and **(2)** the operation performed on the given property value, the required data schema for a rule can be deduced statically. For example, Rule VI (sec. 4.2.6) performs a `sum` operation on the contents of the `Value` property. From this it can be deduced that **(1)** the data schema for positions must include a `Value` property, and **(2)** the `Value`-property must have a number-type, as sum only works on numbers.

6.8 Property abstraction layer

Different customers may have the same position data under different property names. For example, what if a customer wants to use a rule that references e.g. the property `DirtyValue` but instead apply the rule to a property named e.g. `CleanValue`?

Since a property can be considered a function that takes a position as input and returns a property value — e.g. `Value` takes a position as input and returns a number — this ties into taking functions as input (sec. 6.5) and let-bound functions (6.4). If properties are simply functions, and the use of a given property were to imply that a rule takes this property/function as input, then a customer could easily redefine which property names a rule should use by providing different input arguments (that is, properties) to the rule in question.

6.9 Grouping visualizations

A Sunburst chart(Stasko et al. 2000) can be used to visualize a tree data structure. Since a grouping expression evaluates to a tree, these expressions can be visualized in this manner. For example the grouping expression `Portfolio grouped by .Country grouped by .Issuer where sum .Value of Issuer >= 1%` can be visualized as follows (using fictitious position-data):

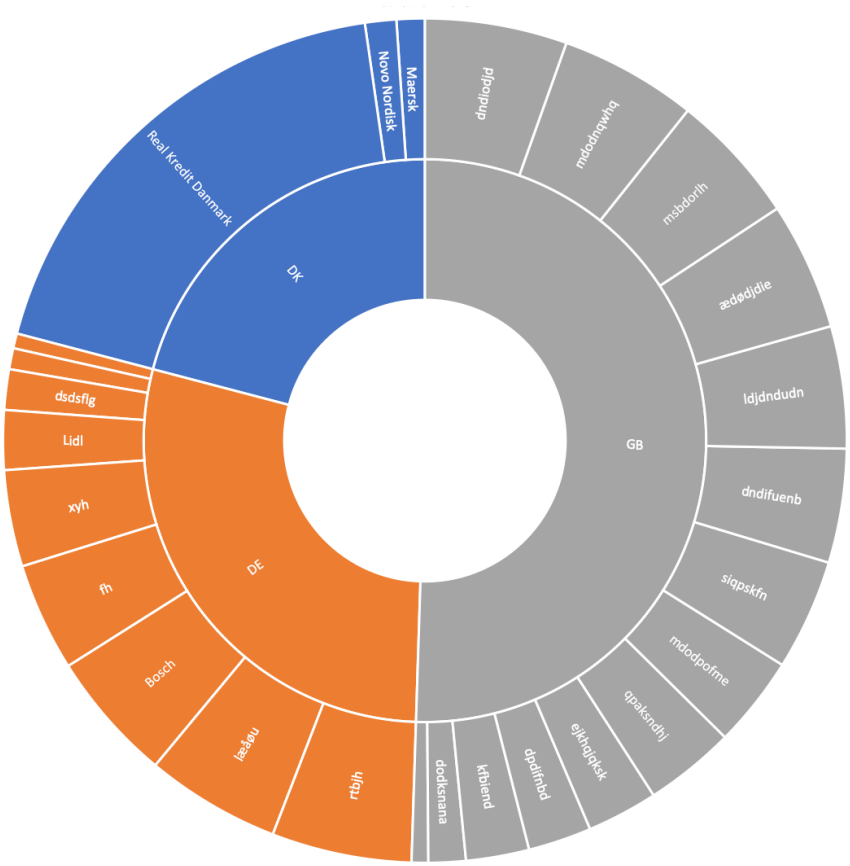

Figure 6: Sunburst chart of grouping by Country and Issuer

Here, the inner ring visualizes the first grouping (by country), while the outer ring visualizes the second grouping (by issuer). Notably, the size of each country/issuer-slice in the visualization is proportional to the `Value`-property of the positions, because this property was used in a filter condition.

A visualization of this form could be helpful when formulating rules, by allowing customers to look at a real-time visualization of their data. For example, the above example visualization reveals that the value of positions of the issuer *Real Kredit Danmark* is relatively high compared to other issuers in the portfolio.

7 Conclusion

A domain-specific language (DSL) for compliance rules enables users to define rules by typing in text using their keyboard. Additionally, a simple DSL makes it easier for non-technical employees to learn how to formulate rules that can be evaluated by a computer — compared to learning a general-purpose programming language. Driven by these advantages, a small DSL was developed, which consists of only four different statements and six different expression types, thus making it significantly simpler than any general-purpose programming language. The developed DSL expresses six out of seven complex compliance rules provided by SimCorp. Expressing the seventh rule would require adding the ability to define additional input to a compliance rule, which currently accepts only a portfolio (see 6.5 in the Future work section). From writing the first version of the parser for the DSL, which used only parser-combinators, it was found that the unrestricted recursion resulted in unintended infinite loops in the top-level parser. The solution was to use a more restricted way to specify the top-level parser of expressions: by using an external library function that takes as input a non-recursive parser for a term, as well as the fixity and precedence for a series of non-recursive operator-parsers, and returns a recursive top-level expression parser. Lastly, the approach of letting property-based testing drive the development of the pretty-printer was deemed unsuccessful, as this approach did not detect e.g. the erroneous pretty-printing of `a AND (b AND c)` as `a AND b AND c`, due to the large number of possible combinations of abstract syntax leading to an explosion in the running time required to generate the relevant test cases.

References

Buneman, Peter, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. 1994. "Comprehension Syntax." *SIGMOD Rec.* 23 (1): 87–96. https://doi.org/10.1145/181550.181564.

Gill, Andy, and Simon Marlow. 2019. "Haskell Parser Generator." 2019. http://hackage.haskell.org/package/happy.

Karpov, Mark, and Alex Washburn. 2019. "Parser-Combinators Haskell Library." 2019. https://hackage.haskell.org/package/parser-combinators.

Martini, Paolo, Daan Leijen, and Mark Karpov. 2020. "Megaparsec Haskell Library." 2020. https://hackage.haskell.org/package/megaparsec.

Nasdaq. 2020. "Nasdaq Company Fact Sheet for SimCorp A/S." 2020. http://lt.morningstar.com/gj8uge2g9k/stockprofile/default.aspx?externalid=DK0060495240&externalidexchange=EX$$$$XCSE&externalidtype=ISIN&LanguageId=en-GB&CurrencyId=DKK&BaseCurrencyId=DKK&tab=-1.

Peyton Jones, Simon L., and André L. M. Santos. 1998. "A Transformation-Based Optimiser for Haskell." *Sci. Comput. Program.* 32 (1–3): 3–47. https://doi.org/10.1016/S0167-6423(97)00029-4.

Stasko, John, Richard Catrambone, Mark Guzdial, and Kevin McDonald. 2000. "An Evaluation of Space-Filling Information Visualizations for Depicting Hierarchical Structures." *International Journal of Human-Computer Studies* 53 (5): 663–94.

Appendix A

Per Langseth Vester's description of SimCorp's motivation to participate in the project:

> **Our Motivation to Participate**
>
> The current way to write complex compliance rules was designed around ten years ago. Overall it works well, but experience has shown us that there is room for improvement.
>
> In terms of specific challenges in the current implementation, these are the main ones:
>
> - Rule fragments are written separately from the compliance rules that use them. The intention is to promote the reuse of fragments across rules. In reality however, only few fragments are used in more than one rule. This leads to a cumbersome process, both upon initial creation and subsequent maintenance.
> - Conceptually, fragments are very powerful, but also difficult to master. They allow for a lot of logical constructions that do not make sense in a business context. A more user-friendly solution with a less steep learning curve will be advantageous.
> - From a usability point-of-view, some users prefer to be able to type in rules using the keyboard including e.g. auto completion. Our current solution does not easily allow for adding this.
>
> The team of testers, developers and product management behind Compliance Manager has been very stable throughout the years. We see this as a great opportunity to get fresh eyes on our solution that can give us new inspiration and let us benefit from some of the latest technological advancements.
>
> Finally, we see the collaboration as a good way to get more ITU students to see SimCorp as an attractive place to work after graduation.

Appendix B

Document from SimCorp entitled *David's Combined Rule Examples with Solutions*, which describes complex compliance rules:

```
I

Maximum 10% of assets in any single issuer. Investments > 5% of assets for any single issuer, may
       not in aggregate exceed 40% of the assets of a UCITS.

Solution : This is essentially two different rules.


Required Fragements:
====================

NoCashPositions
===============
Instrument type <> Cash
=========================


IssuersExcludingCash
====================
Grouping of NoCashPositions
  By Issuer
============================


IssuersAbove5%
==============
Filtering of IssuersExcludingCash
  Where
    Dirty value
  Relative to
    Dirty value
      of NoCashPositions
  > 5%
===================================


Rule:
=====

```

```
39 Limit [<=10%]
40   Dirty Value
41     of IssuersExcludingCash
42 Relative To
43    Dirty Value
44      of NoCashPositions
45
46
47 Limit [<=40%]
48   Dirty Value
49     of IssuersAbove5%
50 Relative To
51    Dirty Value
52      of NoCashPositions
53
54
55 Summary: Limit Dirty value of IssuersExcludingCash relative to Dirty value of NoCashPositions and
       limit Dirty value of IssuersAbove5% relative to Dirty value of NoCashPositions.
56
57 ====================================================
58 IIa
59
60 No more than 35% in securities and money mkt instruments of the same issuer, and if more than 30%
       in one issuer must be made up of at least 6 different issues.
61
62 Solution : This is essentially two different rules.
63
64 The second rule:
65     If more than 30% in one issuer must be made up of at least 6 different issues,
66 can be re-phrased as:
67     Either 30% or less per issuer or at least 6 different issues.
68
69
70 Required Fragement:
71 ====================
72
73 IssuesPerIssuer
74 ===============
75 Grouping of Total Portfolio
76   By Issuer
77 ============================
78
79
80 Rule:
81 ======
82
```

```
Limit [<=35%]
  Dirty Value
    of IssuesPerIssuer
Relative To
   Dirty Value


Only invest / Check
  Where     / If
    Or
      Dirty Value of IssuesPerIssuer  [ <= 30% ]
            Relative To Dirty Value of Portfolio


      Number Of IssuesPerIssuer [ >= 6 ]

Summary: Limit Dirty value of IssuesPerIssuer relative to Dirty value and only invest where (
       check if) Dirty value of IssuesPerIssuer relative to Dirty Value of the portfolio <= 30%
       or Number of IssuesPerIssuer >= 6.

================================================================================================

II b

UCITS Article 23 (1) ñ

When holding > 35% in a single issuer of government and public securities,

Then there can be no more than 30% in any single issue

And a minimum of 6 different issues is required.


Solution: This is again, two different rules, but far more complicated than solution II a.

Required Fragements:
====================


IssuesPerIssuer
===============
Grouping of Total Portfolio
  By Issuer
============================


```

```
IssuersAbove35%
================
Filtering of IssuesPerIssuer
  Where
    Dirty value
  Relative to
    Dirty value
  > 35%
====================================


Rule:
======

Limit [<=30%]
  Dirty Value
    of IssuersAbove35%
        foreach
           Security ( Issue )
Relative To
   Dirty Value



Only invest / Check
  Where     / If
    Or
      Dirty Value of IssuesPerIssuer  [ <= 35% ]
            Relative To Dirty Value of Portfolio


      Number Of IssuesPerIssuer [ >= 6 ]


Summary: Limit Dirty value of IssuersAbove35% foreach issue relative to Dirty value and only
       invest where ( check if) Dirty value of IssuesPerIssuer relative to Dirty Value of the
       portfolio <= 35% or Number of IssuesPerIssuer >= 6.

================================================
III


The risk exposure of a UCITS to a counterparty to an OTC derivative may not exceed 5% NA; the
       limit is raised to 10% for approved credit institutions

Solution : This is essentially two different rules.
```

```
170
171 Rule:
172 ======
173
174 Limit [<=5%]
175  Exposure
176    for each
177       Counterparty
178          Where
179             And
180               InstrumentType = OTC
181               Counterparty NOTIN [approved list]
182 Relative to
183   Exposure
184
185
186 Limit [<=10%]
187  Exposure
188    for each
189       Counterparty
190          Where
191             And
192               InstrumentType = OTC
193               Counterparyt IN [approved list]
194 Relative to
195   Exposure
196
197 Summary: Limit Exposure of each Counterparty where InstrumentType equals OTC and Counterparty not
       in [approved list] relative to Exposure and limit Exposure of each Counterparty where
       InstrumentType equals OTC and Counterparty in [approved list] relative to Exposure.
198
199 ================================================
200 IV
201
202 Max of 15% per Sector or 200% of the index weight in the sector, whichever is greater.
203
204 Solution:
205
206 The rule can be re-phrased as:
207     Either 15% or less per Sector or 200% of the index weight in the sector.
208
209
210 Required Fragment:
211 ==================
212
213 InvestmentPerSector
```

```
======================
Grouping of Total Portfolio
  by Sector
==============================


Rule:
=====

Only invest / Check
  Where     / If
    OR
      Dirty Value of InvestmentPerSector [ <= 15 % ]
           Relative to Dirty value of Portfolio

      Compared % with Benchmark in freely defined benchmark [200%]
          Dirty value of InvestmentPerSector
             Relative To Dirty value


Summary: Only invest where ( check if) Dirty value of InvestmentPerSector relative to Dirty value
       of the portfolio <= 15% or compared relative with benchmark in freely defined benchmark
       Dirty value of InvestmentPerSector relative to Dirty value = 200%.
=========================================
V

Max 5% in any single security rated better than BBB: otherwise the maximum is 1% of fund net
       value.

Solution : This is essentially two different rules.

Rule:
======

Limit [<=5%]
 Dirty Value
   for each
      Security
         Where
            SecurityRating IN [better than BBB]
Relative to
  Dirty Value


Limit [<=1%]
 Dirty Value
```

```
   for each
      Security
         Where
            SecurityRating NOTIN [better than BBB]
Relative to
  Dirty Value

Summary: Limit Dirty value of each Security where SecurityRating in [better than BBB] relative to
      Dirty value and limit Dirty value of each Security where SecurityRating not in [better
      than BBB] relative to Dirty value.
================================================

VI

The portfolio shall invest in no fewer than 5 foreign countries, provided that:
1. If foreign securities comprise less than 80% of its net assets, then it shall invest in
no fewer than 4 foreign countries.
2. If foreign securities comprise less than 60%, then it shall invest in no fewer than 3
foreign countries.
3. If foreign securities comprise less than 40%, then it shall invest in no fewer than 2
foreign countries.


Solution :
This is essentially four different rules with limits as follows :

% in foreign securities     | Number of foreign countries
    80 - 100 ( >= 80% )     |   >= 5
    60 - 80  ( >= 60% )     |   >= 4
    40 - 60  ( >= 40% )     |   >= 3
    0  - 40  ( < 40%  )     |   >= 2


These rules can be rephrased as follows:

Rule 1: Either foreign securities comprise less than 80% of its net assets or the portfolio shall
      invest in
no fewer than 5 foreign countries.


Rule 2: Either foreign securities comprise less than 60% of its net assets or the portfolio shall
      invest in
no fewer than 4 foreign countries.


Rule 3: Either foreign securities comprise less than 40% of its net assets or the portfolio shall
```

```
          invest in
no fewer than 3 foreign countries.

Rule 4: The portfolio shall invest in no fewer than 2 foreign countries.


Required Fragment:
==================

ForeignSecurities
===============
Grouping of
    Country <> Dk
      by Country
=================


Rule:
======

Only invest
  Where
    OR
      Dirty Value of ForeignSecurities  [ < 80% ]
               Relative To Dirty Value of Portfolio

      Number Of Countries of ForeignSecurities [ >= 5 ]


Only invest
  Where
    OR
      Dirty Value of ForeignSecurities  [ < 60% ]
               Relative To Dirty Value of Portfolio

      Number Of Countries of ForeignSecurities [ >= 4 ]


Only invest
  Where
    OR
      Dirty Value of ForeignSecurities  [ < 40% ]
               Relative To Dirty Value of Portfolio

      Number Of Countries of ForeignSecurities [ >= 3 ]

```

```
344
345 Only invest
346   Where
347      Number Of Countries of ForeignSecurities [ >= 2 ]
348
349
350 Summary: Only invest where Dirty value of foreign securities relative to dirty value of the
       portfolio < 80% or number of Countries of ForeignSecurities >= 5 and only invest where
       Dirty value of foreign securities relative to dirty value of the portfolio < 60% or number
       of Countries of ForeignSecurities >= 4 and only invest where Dirty value of foreign
       securities relative to dirty value of the portfolio < 40% or number of Countries of
       ForeignSecurities >= 3 and only invest where number of Countries of ForeignSecurities >=2.
351 ==================================================
```