UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

A unit-aware matrix language and its application in control and auditing

Griffioen, P.

Link to publication

Creative Commons License (see https://creativecommons.org/use-remix/cc-licenses):
Other

Citation for published version (APA):
Griffioen, P. (2019). A unit-aware matrix language and its application in control and auditing.

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (http://dare.uva.nl)

Download date: 21 Feb 2020

A Unit-Aware Matrix Language and its Application in Control and Auditing

Paul Griffioen

A Unit-Aware Matrix Language and its Application in Control and Auditing

A Unit-Aware Matrix Language and its Application in Control and Auditing

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. K. I. J. Maex
ten overstaan van een door het College voor Promoties
ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel
op dinsdag 29 oktober 2019, te 12.00 uur

door

Paul Griffioen

geboren te Zevenhoven

Promotiecommissie:

Promotor:	prof. dr. P. Klint	Universiteit van Amsterdam
Copromotor:	dr. P.I. Elsas	ComputationalAuditing
Overige leden:	prof. dr. T.L.C.M. Groot	Vrije Universiteit Amsterdam
	prof. dr. E.E.O. Roos Lindgreen	Universiteit van Amsterdam
	dr. H. Weigand	Tilburg University
	dr. C.U. Grelck	Universiteit van Amsterdam
	prof. dr. ir. C.T.A.M. de Laat	Universiteit van Amsterdam
Faculteit:	Faculteit der Natuurwetenschappen, Wiskunde en Informatica	

The work in this thesis has been carried out at Centrum Wiskunde & Informatica (CWI) under the auspices of the research school IPA (Institute for Programming research and Algorithmics). This research was supported by the NWO JACQUARD grant #638.001.214 "Next-Generation Auditing: Data-Assurance as a Service".

Thesis cover contains art licensed by Bernard Fallon, BernardFallon.com

ACKNOWLEDGEMENTS

First of all I would like to thank my promotor Paul Klint and co-promotor Philip Elsas. Without their joint effort to start the Next-Generation Auditing project this research would not have been possible. Besides that, the bi-weekly meetings with Paul were critical in finishing this thesis, and for his guidance and positive attitude throughout the project I am very grateful. Philip has been a great mentor for many years. This thesis builds on his work, and our regular discussions helped clear up many questions. For his advice and scientific insights in general I cannot thank him enough. I also would like to thank the committee members prof. dr. Tom Groot, prof. dr. Edo Roos Lindgreen, dr. Hans Weigand, dr. Clemens Grelck, and prof. dr. ir. Cees de Laat. Special thanks to prof. Hans Blokdijk who was a pivotal figure in bringing research in computing and auditing together, and became a regular participant in the project meetings. Unfortunately he is no longer with us, but his enthusiasm remains an inspiration.

One of the nice side-effects of the project is that it brought me in contact with Rob Christiaanse. He independently came to the conclusion that the value cycle is much more attractive in non-financial than in financial units of measurement. This common interest led to a fruitful collaboration and several papers. His insights and vast knowledge of the literature were invaluable, and working together not only brought the research further, but was also a lot of fun.

During the project I was employed at CWI, which is a nice and inspiring environment. I want to thank Jurgen Vinju for his hospitality and pleasant conversations. Part of the time I spent at TU Delft in the SATIN project. I would like to thank Joris Hulstijn, Yao-hua Tan, Marijn Janssen, Albert Veenstra and Han Bosch for the productive cooperation. Other people I would like to mention for their role in making the NGA project possible are Marc van Hilvoorde, Fred van Ipenburg, Jacques de Swart en Jeroen Hoexum.

Among the notable events for me were the visits to the IFL conferences. Many thanks to Rinus Plasmeijer and the people at IFL for all the useful and stimulating feedback. I am also indebted to Jan Brandts for reviewing parts of the material and providing feedback and suggestions. Rob Bisseling was very helpful with feedback on one of the papers and some personal general advise.

Amsterdam — Paul Griffioen
August, 2019.

CONTENTS

1 Introduction **1**
- 1.1 Research Motivation . . . 1
- 1.2 Literature and Related Work . . . 5
- 1.3 Research . . . 14
- 1.4 Thesis Layout . . . 20

2 Short Introduction to Pacioli **23**
- 2.1 The Pacioli Language . . . 23
- 2.2 Unit Inference . . . 27

I Unit-Aware Matrix Programming **31**

3 Introduction **33**
- 3.1 Type Inference for Matrix Programming . . . 33
- 3.2 Dimensioned Matrix Shapes . . . 35
- 3.3 Dimensioned Vectors, Matrices and Tensors . . . 38
- 3.4 The Matrix Type Notation . . . 41
- 3.5 Conclusion . . . 44

4 Dimensioned Linear Algebra **45**
- 4.1 Units of Measurement . . . 45
- 4.2 Dimensioned Vector Spaces . . . 46
- 4.3 Dimensioned Matrices . . . 49
- 4.4 Conclusion . . . 55

5 The Matrix Type **57**
- 5.1 The Matrix Type Syntax . . . 57
- 5.2 Type Inference . . . 58
- 5.3 Exponentiation . . . 63
- 5.4 Expressiveness . . . 67
- 5.5 Conclusion . . . 69

6 Pacioli: Implementation and Examples **71**
- 6.1 Implementation in Pacioli . . . 71
- 6.2 Language Features for Linear Algebra . . . 73
- 6.3 Case Studies . . . 74
- 6.4 Conclusion . . . 90

II Value Nets 93

7 Introduction 95
7.1 The Value Cycle . . . 95
7.2 Value Nets . . . 102
7.3 Normal Behavior: Tour Spaces . . . 107
7.4 Tour Factoring and Causal Chains of Events . . . 111
7.5 Accounting . . . 114
7.6 Fraud Analysis . . . 118
7.7 Conclusion . . . 124

8 Value Net Structure 125
8.1 Value Net Definition . . . 125
8.2 Unit-aware Petri Nets . . . 126
8.3 Value Net Implementation . . . 128
8.4 Value Net Properties . . . 131
8.5 Conclusion . . . 136

9 Value Net Behavior 137
9.1 Tours . . . 137
9.2 The Causal Chain . . . 139
9.3 Fraud Analysis . . . 141
9.4 Conclusion . . . 144

10 Case Study: Public Transport Services 147
10.1 Case Description . . . 147
10.2 Revenue Model . . . 148
10.3 Dispute Resolution . . . 151
10.4 Conclusion . . . 153

Conclusion 157

Appendix 165

References 169

Summary 175

Samenvatting 177

1 INTRODUCTION

1.1 RESEARCH MOTIVATION

Units of measurement are as old as the act of measuring itself since one cannot exist without the other. Many kinds of units have been adopted in history, from the feet, el and pint, to the modern meter and liter, etc. They are so deeply ingrained in our language and daily lives that we mostly use them without noticing.

Programming language support for units of measurement forms the link between the two topics of this thesis, (1) type inference for matrix languages, and (2) process modeling for control and auditing. Since accounting systems record financial information, models used by controllers and auditors to analyze this information are typically also expressed in financial terms, but business processes are most accurately described in physical goods and units of measurement. Financial information always involves some valuation that hides the underlying non-financial facts. Therefore this thesis studies process models for control and auditing in physical units of measurement. This requires however also computation in those units, and that's where the link with matrix programming comes from. The first part of this thesis describes a unit-aware matrix type based on dimensioned linear algebra that infers principle types for linear algebra expressions. This matrix type is used in the second part in the computations involved in the analysis of process models for control and auditing. The dimensioned linear algebra that results from the matrix type is used to analyze process models in non-financial units of measurement.

The process models that are the subject of this thesis are based on the *value cycle* concept from auditing. A value cycle is a specific kind of state-transition system that describes the causality and proportionality in an enterprise's value creation processes from a top-level point of view. Similar to Porter's Value Chain [Por08] it looks at the causal link between buying, transforming and selling products, but by connecting the production processes to the financial processes it gives a complete overview of the circular flow of value in a business. Furthermore, where the value chain is strictly a qualitative conceptual model that explains how to identify competitive advantages in a strategic setting, the value cycle is a quantitative model with roots in system dynamics and costing methods. While still a conceptual model, it is much closer to a process model than the value chain, and when it was formalized into *audit nets* [Els96] it became a complete process modeling formalism. An important concept that the value cycle adds besides causality is the notion of proportionality, the idea that different steps in a process often occur in more or less fixed ratios. The idea is that these ratios are characteristic for a specific enterprise and can be used as a norm when an enterprise's behavior is analyzed. Norms are often associated with standards set

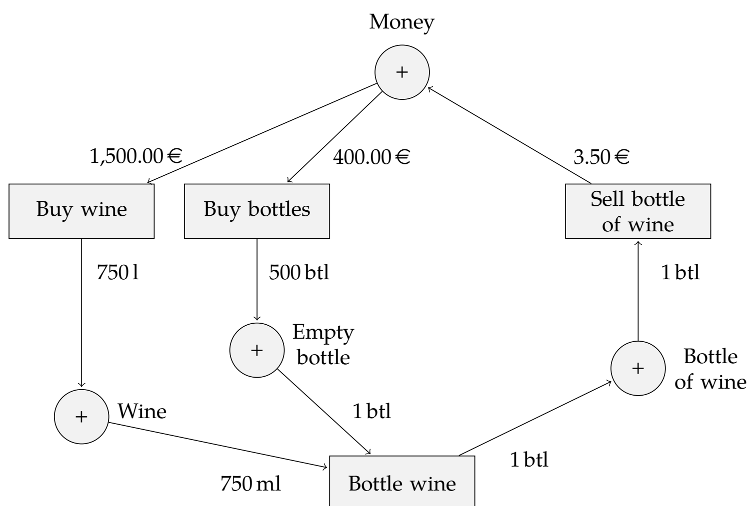


Figure 1.1: A model of a wine bottler's value creation process. The rectangles model economic events or transactions, and the circles denote assets, in this case money and three kinds of goods. The arrows indicate how much of the assets are produced and consumed by each event occurrence.

by regulators, but in this context norms are measurable numerical criteria. With the concept of proportionality the value cycle model gives a top-level quantitative view of the causal relationships in an enterprise's value creation process.

The *value net* formalism developed in this thesis is a generalization of the audit net formalism, that aims to make the causality and proportionality from the value cycle available to other application areas besides auditing. Just like an audit net, a value net is a cyclic Petri net that models the value creation process of a particular business. An example is the wine bottler from Figure 1.1. The wine bottler buys wine and empty bottles, and produces bottles of wine from these products. By connecting the production events to the asset money the figure gives a complete picture of the bottler's circular value creation process from which normal or expected behavior can be deduced. For example, because the capacity of the wine bottles is 750 ml/btl, it is expected that for each purchased empty bottle 750 ml of wine is purchased. In this case the ratio 750 ml/btl is a characteristic property of the product that determines the relationship between different steps in the process. Having a top-level view focused on such proportions allows analysis of process behavior that is useful in areas besides auditing, like planning, scenario analysis, etc..

Analysis of the value net is an example where support for units would be useful. For computational purposes a bill of material matrix can be derived from the wine

bottler's process model. The bill of material matrix B with B_{ij} the amount of product i used in product j is given by

$$B = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 750\,\text{ml/btl} & 0 & 0 \end{bmatrix}$$

where the products are the set {Bottle of wine, Empty bottle, Wine}. The bottles have unit bottle (btl)[1] and the wine has unit millilitre (ml). The first column contains the parts for the bottle of wine. The entry B_{21} in the second row is the consumption of the 1 empty bottle per bottle of wine. The unit is btl/btl which reduces to a dimensionless number. The value 750 ml/btl for entry B_{31} is the amount of wine used. The bill of material matrix is a typical application where units plays an important role. The units in the bill of material are crucial information and support for units of measurement in the computations would improve the quality of the analysis.

If numbers in a programming language are enriched with units of measurement, then the units of a program's output can be computed automatically while checking the correctness of the complete computation, but such support is not commonplace in mainstream programming languages. Although the convenience of automating the tagging of the output and the increased safety are obvious benefits, the issue is that the computation of the units of measurement is inefficient compared to a numerical operation, which is often directly supported by hardware. Languages that compute with units of measurement do exist, but mainstream languages mostly provide libraries that help with the output of units. Computations are done with numbers only and a programmer has to know from specifications or other sources what the units are. However, support for units of measurement has been added to the static type system of the F# programming language.

The type inference with units of measurement developed by Andrew Kennedy in 1994 [Ken94; Ken96] and added to the F# programming language in 2009 is a major step forward because it gives units correctness without sacrificing efficiency [Ken09]. The unification algorithm developed by Kennedy is an extension to type systems with the potential to considerably increase safety of numerical software. Using techniques not found in popular type systems of languages like Haskell or C++, it infers the unit of measurement of numerical expressions and rejects erroneous operations at compile time. Unit unification uses the fact that units of measurement form a free abelian group. These groups have a most general unifier and provide a powerful extension to type systems for mathematical software. This increases type safety of numerical expressions to a much finer level than regular type systems and effectively automates dimensional analysis.

[1]The text between parenthesis after the units's name is the unit's symbol. A unit's symbol is a short string that is appended to a printed or written number to indicate the unit. More details are given in Chapter 4.

Although Kennedy's unit inference is sound and complete, it is insufficient to type a dynamic data type like a vector or a list, because parametric polymorphism limits it to numbers with homogeneous units. A vector with an unknown size at compile time could be given a parametric type like Vector(a), but then each element has the same unit a. This is a general problem for data types of varying size at run time and is similar to a database with numbers where it is insufficient to associate a unit with a column if the units can vary per row. If the elements in numerical data can have different units of measurement then parametric typing with units cannot express the desired type.

The bill of material matrix from the wine bottler case is an example of a matrix with varying units that cannot be typed with the parametric unit type. The underlying issue is that the products are only known at runtime, and not at compile time when the unit inference is done. Since the individual product units are unknown it seems that a parametric type can never type a matrix like the bill of material example. The units in a matrix can however not be just any units, but they are constrained by the matrix' linear transformation. In the bill of material matrix example the units follow from the units of the products in the production process. The unit at every position in the matrix is as follows.

$$\begin{bmatrix} 1 & 1 & \mathrm{btl}/\mathrm{ml} \\ 1 & 1 & \mathrm{btl}/\mathrm{ml} \\ \mathrm{ml}/\mathrm{btl} & \mathrm{ml}/\mathrm{btl} & 1 \end{bmatrix}$$

It's not difficult to see that the rule is that the entry in the matrix for product i and j is the unit of product i divided by the unit of product j. This is a general principle. The form of the units of measurement in any matrix is completely determined by the units of measurement of the source and target vector space of its transformation. This property is the main result of Hart's theory of dimensioned vector spaces [Har95; Har94] and underlies this thesis' matrix type. The general principle translates into a type rule for the matrix product that can be used during type inference. With similar rules for the other linear algebra operations the type can handle any linear algebra expression. The matrix type is an extension of Kennedy's unit type that can handle any matrix, including ones like the bill of material.

The matrix type solves the issue of heterogeneous units of measurement by extending unit-inference to dimensioned vector spaces. A dimensioned vector space types a collection of numbers as a whole instead of typing each element individually. The underlying idea is to split a vector into an element-wise product of a magnitude vector and a unit vector. For example,

$$\text{replace} \begin{bmatrix} 1\,\mathrm{btl} \\ 1\,\mathrm{btl} \\ 750\,\mathrm{ml} \end{bmatrix} \text{ by } \begin{bmatrix} 1 \\ 1 \\ 750 \end{bmatrix} \cdot \begin{bmatrix} \mathrm{btl} \\ \mathrm{btl} \\ \mathrm{ml} \end{bmatrix}$$

A dimensioned vector is an element-wise product of a magnitude vector and a unit vector, just as a dimensioned number is a product of a magnitude and a unit of measurement. And just as a unit of measurement is built from a set of base units, a unit vector is built from base unit vectors. The matrix type extends the base units of measurement with base unit of measurement vectors. At compile time no assumptions about the individual units in the base unit vector are made, just that it has a unique name that can provide the necessary information for shape and unit inference at compile time. With the matrix type all major linear algebra operations can be typed, effectively giving a type for dimensioned linear algebra.

1.2 LITERATURE AND RELATED WORK

This section starts with a discussion of literature related to matrix programming and units of measurement. The efficiency and correctness of matrix and vector operations is a large and active research topic, but despite the numerical nature of the domain it is rather unconnected from language support for units of measurement. Next some background is given on accounting, control and auditing. In a short historic overview we look at the development of the ideas leading to the value cycle.

1.2.1 *Unit-Aware Matrix Programming*

Related research on type systems and units of measurement for mathematical software aims to improve programming in different ways. Language extensions for units of measurement and dimensional analysis mostly aim to increase safety of software, whereas research on types additionally tries to optimize code and remove run-time overhead with improved type inference. Other approaches improve languages or libraries to increase software quality or reuse.

Units of Measurement and Dimensional Analysis

Extending language syntax or typing rules with units of measurement is a common approach to increase the safety of numerical software. Andrew Kennedy's PhD thesis is the most advanced work on units in programming languages [Ken96]. A good overview of his work and a nice hands-on introduction to units in F# is given in [Ken09].

Recent related work besides Kennedy's unit inference is the MetaGen extension of Java that statically checks units of measurement [ACL+06]. Dimensions and units can be formulated in a nominally typed object-oriented language through the use of statically typed meta-classes. Another development is the Osprey tool [JS06]. This tool extends the C programming language with type annotations that can be statically

checked for errors in units of measurement. The prototype was extensively validated on mature code bases of significant size and discovered many errors.

The advantages of polymorphic units is demonstrated in the application of a Haskell unit library to astrophysics research [ME14]. The library is dimension-monomorphic but unit-polymorphic, which means that the library interface specifies the dimensions (e.g. length), but a library user is free to call the library with any units he wants (e.g. feet or metre). None of these approaches provide support for data with heterogeneous units of measurement, or support for matrices or linear algebra.

Support for Linear Algebra

Support for operations like those from linear algebra in general purpose languages is a typical use case for optimizations or verification, and the implementation of efficient data types like arrays is continuously improved. Functional matrix operations are explicitly discussed as application of higher-order vectorization in [LCK+12]. A matrix is represented as an array of rows and operations on it are transformed into vectorized form. This form is implemented efficiently in Data Parallel Haskell (DPH). In [AHK+10] the level of abstraction for sparse matrix programs is raised from imperative code with loops to functional programs with comprehensions and limited reductions. It then uses Isabelle/HOL to verify full functional correctness of programs.

In the Single Assignment C (SaC) array language, shape inference helps prevent runtime errors and also improves the code generation with respect to runtime efficiency [Sch03]. The type system uses a hierarchy of array types, containing varying levels of shape information, and infers shapes as specific as possible. Type checks are postponed to runtime when exact shapes cannot be inferred statically. The shape information allows SaC to generate code that is competitive with hand-optimized code.

Specific support for linear algebra in Haskell is promoted as statically typed linear algebra in [Eat06]. It aims to catch errors in matrix and vector operations by statically verifying properties specific to linear algebra. However, no checks beyond array shape are performed, and it is for instance mentioned that forgetting to invert a matrix would not be detected by statically typed linear algebra. The advantages of statically typed linear algebra are shown in the linear algebra library interface from [AS15]. It verifies the consistency of matrix operations by generative phantom types in ML.

A novel approach to multi-dimensional arrays in Haskell is presented in [KCL+10]. The purely functional arrays support reuse through shape polymorphism, which is embedded in Haskell's type system using type classes and type families. The embedding in Haskell and the strong typing and purity gives it advantages over languages like APL, J and Matlab. Although this approach is more tailored towards matrices, it still suffers from the restriction to homogeneous elements. The parametric

array type expects an element type, so it cannot express different units of measurement for different elements.

Many type systems take advantage of the algebraic properties of matrices when code is written in index-free form. The Rice Vector Library is a collection of C++ classes expressing core concepts (vector, function,...) of calculus in Hilbert space that allows expression of a variety of index-free algorithms from linear algebra and optimization in terms of this system of classes [PSS09]. The type system's support for data abstraction and polymorphism enables creation of reusable numerical code where implementation details are hidden behind abstract interfaces. The Trilinos Project is an effort to facilitate the design, development, integration, and ongoing support of mathematical software libraries within an object-oriented framework for the solution of large-scale, complex multi-physics engineering and scientific problems [HBH+05]. Other approaches that emphasize reusable high-level code are for example a library for grid-free two-dimensional vortex particle methods [RR09] and interfaces for solving linear system [SSH08]. These types focus on other aspects than units of measurement and vector spaces, but it is interesting to note that like the matrix type they favor high-level index-free code under the influence of the type system. Finally, the Formal Linear Algebra Methods Environment (FLAME) doesn't use types to improve quality but derives provably correct algorithms for linear algebra operations [BQG05]. Again a key observation is that intricate indexing should be avoided because it often introduces programming errors.

Units of measurement and support for linear algebra are active areas of research, but the combination of the two as in dimensioned matrices is unexplored.

Type Inference for Matlab

Type inference for matrix languages like Matlab is continuously improved to optimize code and remove run-time overhead, or to increase software quality or reuse. Many type inference techniques have been developed to eliminate runtime checks or to improve code generation for matrix languages. What most of these approaches have in common is some form of shape inference or algebraic rewriting. Just as for the matrix type, the shape or some other property is derived from the operators in an expression. The matrix type's extension of unit unification to linear algebra refines such inference to dimensioned vector spaces. This makes the type system much more powerful than shape inference. For example, when a transpose is omitted from an expression the type system complains even when the matrix is square, because the result would be in the wrong space.

Various Matlab compilers use a lattice-based type inference technique based on fixed-point solutions that was influenced by APL and SETL compilers. The FALCON Matlab compiler uses such type inference to provide the shape and type information necessary for the generation of efficient Fortran 90 code from Matlab programs

[RGG+96]. The Matlab D compiler is a parallel Matlab compiler that uses a telescoping languages framework to generate Fortran with MPI from Matlab scripts [FMJ+07]. Type analysis is performed to obtain high performance parallelism from high-level Matlab scripts. MaJIC (Matlab Just-In-Time Compiler) employs a combination of just-in-time and speculative ahead-of-time compilation [AP02]. It consists of an extremely fast type inference engine that makes a conservative estimate of the types of expressions. Its goal is to remove as much runtime overhead as possible without sacrificing the interactive nature of Matlab. The MAGICA type inference engine overcomes limitations of the lattice-based type inference technique by modeling the language's shape semantics with an algebraic system and applying term rewriting techniques to evaluate expressions under this algebra [JB06]. Like unit inference with unit variables, this enables the deduction of valuable shape information even when array extents are compile-time unknowns. However, this information cannot always be derived and is much less refined than unit inference.

1.2.2 *Value Nets*

The value cycle has its roots in the developments in accounting in Western Europe and the United States in the 19th and 20th century. We present here a condensed overview of the main developments that are relevant for the value cycle. A particular relevant recurring theme in the literature is the distinction between financial and non-financial information. In the analysis of business models there is always a trade-off between financial and non-financial information. Non-financial information is more accurate, but financial information is commensurate. In all periods discussed this plays a role.

Pacioli and Double-Entry Bookkeeping

Just as for units of measurement, the history of bookkeeping goes back a long time. The birth of modern accounting is often credited to Pacioli's publication of the Summa de Arithmetica, Geometria, Proportioni et Proportionalita [Pac94]. Book 9 of this bundle of mathematics texts is devoted to the principles of double-entry bookkeeping as practiced by Venetian merchants at the time. The development of the accounting systems in the renaissance was driven by the enormous growth of the textile trade. According to Pacioli, the double-entry bookkeeping systems provides a businessman with the management information he needs to safeguard his assets and be in control of his business, even when faced with a large volume of business transactions. From a modern programming perspective it is interesting to note that the Summa is in essence a description of a data structure. It explains how data is structured and which operations can be done on it. Key features are the systematic recording and indexing of business transactions for easy aggregation and fast access.

Pacioli describes an accounting system consisting of three books; the Memorandum, the Journal and the Ledger. In the Memorandum all business transactions are recorded during daily operations, typically by non-accountants. This first recording contains as much information as possible, with at least the quantity in the proper unit of measurement and the amount or price. Next the transactions are entered in the journal following the double-entry principle. Each business transaction leads to a double entry in the journal, one for an account to debit and one for an account to credit. Both accounts are updated with the same amount, which makes the transaction financially balanced. After the journal entries have been approved an accountant updates the accounts accordingly. The Ledger is the book that contains the accounts, one for each asset or liability. An account is a growing table with a debit and a credit column. Each debit entry in a journal is recorded as a row with an entry in the debit column and each credit entry as a row with an entry in the credit column. The difference between the column totals is the account's state. All accounts together represent an enterprise's overall financial state.

The journal is the key to the double-entry method and an important feature is that the data is brought into a single unit. The requirement that every journal entry is financially balanced guarantees that the collection of all accounts is always balanced overall. A consequence is that both sides of the journal must be in financial units, even if it concerns physical goods. For example, when the wine bottler purchases wine, this is not recorded as 750 l but as 1500 €. Pacioli explicitly notes the importance of the single financial unit.

> *When you summarize an appropriate number of inventory items, only one unit of currency need be used.* *(Pacioli, Summa, Chapter 12)*

Once the journal entries are made, the rest of the process is fixed and free of other choices. The ledger can be completely reconstructed from the journals when it is lost. It is just a data structure that indexes and groups the journal's mutations for fast access. The journal with the double entries in financial units is the key of the method and used by most accounting systems since its introduction.

Causality and Proportionality in Production Processes

The question of causality and proportionality in production processes, two key concepts in the value cycle, has its origins in the increasing specialization of labor during the industrial revolution. During the industrial revolution the focus in accounting shifted from trade to manufacturing, and the main driver for developments in accounting was no longer the transaction volume, but the increasing complexity of business processes. A standard handbook by Garcke & Fells [GF87] describes elaborate accounting practices for specialized functions like recording and paying labor, purchasing materials, etc. that were developed during that period. According to

the authors the specialization of labor that made accounting more complex occurred, because reorganizing the manufacturing in this way was shown to be economically more advantageous, in spite of the increased expense of administration. It did however raise questions about causality and proportionality in production processes. With the introduction of specialized business functions the relationship between production volume and cost is lost. If for example a specialized purchasing function is introduced, then this raises the question which other departments use the purchased materials and in what ratio. These are questions about causality and proportionality, driven by the specialization of functions in manufacturing.

After the developments in manufacturing in the textile industry, the driver for developments in accounting was the enormous transaction volume in the large transportation, production and distribution enterprises from the 1850-1925 period. In his overview 'The Evolution of Management Accounting', Kaplan explains that after the textile industry, the railways handled a 'vastly greater number and dollar volume of transactions than had any previous business ...', but that despite the enormous capital invested, predictive models to manage cost were still lacking [Kap84]. A scientific approach to manufacturing emerged with Taylor a key figure [Tay11]. His work was more focused on management methods and less on accounting, but new accounting methods were adopted after the integration of traditional financial accounting with the new accounting methods from Garcke & Fells after their 1887 publication. Important topics at the time were the distinction between fixed and variable cost and the allocation of overhead. The scientific approach pioneered in factory accounting and the railways was elaborated further in the large enterprises like DuPont Corporation and General Motors Corporation that arose around the start of the 20th century in the United States. In Europe Schmalenbach was a dominant figure in development of accounting [Sch28]. He also emphasized that accounting systems should not just be used for legal but also for operational purposes and he advocated the use of a predictive model for business planning. In all these developments at the start of the 20th century the proper allocation of cost was a core issue in the development of accounting.

Proportionality and modeling in non-financial units was an important aspect of the research into cost and accounting that became a cornerstone of the value cycle models. The study of proportionality was part of the question of the causal connection between production volume and cost. A key question is what mix of an enterprise's resources is used to achieve a certain production goal. In [vdSch55] van der Schroeff reviews the literature on proportionality and makes a comparison with research on processes in other fields, for example chemical reactions and vegetative processes. He distinguishes a technical and an economical question and explicitly discusses how valuation causes the difference between these two.

> *When the quantities are measured in physical units the analysis of this question is complicated because there is no common norm to compare the resources with each other or with the end result. As soon as the comparison is applied to values a common denominator is found for the resources and the end result.*

In the study of proportionality, economics is viewed as a quantitative science dealing with quantities and their relationships.

Audit and the Value Cycle

Auditing and control concern the safeguarding of assets. The difference is that an audit is a one-off activity, whereas control is a system that continuously operates.

Modern financial auditing is an examination to ascertain how far an enterprise's financial statements are a true and fair representation of its performance [AL84]. Faced with the separation of ownership and management since the industrial revolution and the growing number of limited liability companies with anonymous shareholders, lawmakers in the UK in the 19th century started drafting legislation to protect the various interests of a business' stakeholders. For a brief period of time after the British Joint Stock Companies Act of 1844 stockholders obtained for the first time the right to audit a company's accounts. Later on companies are required to use proper accounting techniques and to publish annual financial statements containing a balance sheet and profit and loss statements. The requirement for a formal external audit as we know it today appeared for the first time in the UK in the Companies act of 1907.

Initially regulations in the US were rather lax, but after the Stock Market Crash had diminished investor confidence the Securities Act of 1933 and the Securities Exchange Act of 1934 were passed. Strict accounting standards and reporting regulations were introduced after companies had overstated profits to look more attractive to potential investors. With the publication of audited financial statements, regulators aim to provide stakeholders with trustworthy information on the state of a business [ZvdWC+92].

The development of the value cycle in The Netherlands has its roots in a scientific approach to auditing and accounting. Historically Dutch regulations regarding financial statements have been less strict as compared to the Anglo-American setting, and laws for financial reports were only finalized in 1929 after decades of deliberation [BDW95; ZvdWC+92]. Under Limperg's influence, in the Netherlands audit developed as a deductive science with much discussion for the auditor's role in society and the limits of auditing and what can reasonably be expected from an auditor [LG64a; LG64b; Lim85]. In this principle-based approach, assessment of profit and loss is approached from a costing viewpoint, based on Limperg's replacement cost, a method of valuation that considers the current amount needed to replace an asset. Consequently, instead

of standardization and strict regulation as in the rule-based auditing approach, more attention is given to the questions of causality and proportionality in costing from the accounting field [vdSch55].

In this analytical climate the value cycle was developed and became an integral part of Dutch auditing practice and education. Inspired by developments in automation and operations research after the second world war, Starreveld developed the value cycle to model information processing in the context of the automation of accounting processes [SdMJ88]. By connecting the production process to the monetary process, the value cycle describes the circular flow of value that is inherent in a business process. Starreveld models this value flow with a state-transition system inspired by Forrester [For78] and gives a theoretical underpinning in the form of accounting equations that relate the states and the transitions numerically. In contrast to Limperg's deductive approach, the value cycle is based on empirical classifications of industries and services into a topology based on the type of business process. A key figure in the application of the value cycle and the equations to external auditing was Frielink [FdH89]. His textbooks have been standard educational material for Dutch auditors for decades.

The value cycle concept as developed in the Dutch auditing tradition has been formalized into *computational auditing* by Elsas [Els96]. Instead of a conceptual model that is used implicitly, audit nets are introduced as process models that describe a specific enterprise. Audit nets are Petri-nets enriched with business concepts important for auditing, like recordings and agents. Furthermore, the models come in two modalities, the *soll* modality and *ist* modality. The soll modality only models licit behavior, while the ist model additionally models illicit behavior. All these intricate refinements give a formalization of the quantitative and qualitative fundamentals of the value cycle.

Recent Developments

Although the value cycle and the audit nets originated in the auditing domain, the generalization into value nets moves it more into the field of management accounting. A major impact on management accounting and business processes in general is the recent development of modern electronic information systems. Computing has become an inexpensive commodity and accounting systems are no longer the main or only source of management information. With the emergence of software like enterprise resource planning (ERP) systems the landscape of accounting has changed drastically, but many of the conceptual issues in accounting still remain a challenge.

In the second half of the 20th century the distinction between financial accounting and management accounting becomes more institutionalized. Financial accounting concerns the aggregation of a business' financial transactions and reporting financial statements for various stakeholders. Management accounting or managerial account-

ing supports management's decision making process with financial and non-financial information. After the second world war management accounting developed as a distinct discipline. According to the International Federation of Accountants (IFAC) management accounting developed in several stages [ADL05]. Initially it concentrated on cost determination and financial control. By 1965 it also started providing information for management planning and control. At that point all reporting was still financial. In the third stage starting in 1985 management accounting aimed to reduce resource waste in business processes. In this stage non-financial performance measures of actual processes rather than their financial consequences start becoming integrated into the formal monitoring system. New techniques are introduced after Kaplan and others give a harsh critique on the state of affairs in management accounting [JK87; Kap84]. In this context non-financials are not just non-financial measures of the production process like produced items and failure rate, but also measures of contingent factors like customer satisfaction or market activity. Kaplan's own business balanced score card (BSC) is an example. It is a visualization of the causal relationships in a business of which the production process is just one aspect. By making the causal relationships explicit they can be measured and controlled [DL03]. With emerging techniques like these in management accounting, business analysis in non-financials has become a common practice.

Research into contingent non-financials has broadened the scope of management accounting, but non-financials for internal business processes is far from a settled issue. After the initial development of management accounting at the start of the 20-th century not much was done on the role of accounting information in the more complex production and assembly operations of manufacturing [Kap84; McC82]. After investigating the performance measurement frameworks of a large number of companies, Ittner and Larcker found that many of them lacked a causal model and failed to connect non-financial measures to financial performance [IL03]. Similarly, a literature review [AE10] on performance measurement systems in supply chains revealed that existing systems cannot handle the requirements of the new supply chain era and that modeling the causal relationships among performance indicators is still an unresolved and challenging issue. The questions of causality and proportionality that are central to the value cycle are seen in a variety of research areas in management accounting.

Besides measurement and analysis in non-financials, the possibility of accounting in non-financials, free from any valuation, has also been investigated by several researchers. A common reference is the work of Ijiri [Iji65; Iji67], in which he proposes accounting in physical units of measurement. His main objection to accounting practices is that it does not record the causal chain of events that is found in business processes. Inspired by Ijiri's proposals McCarthy formulated the Resource, Event and Agent (REA) accounting model [McC82]. REA replaces Pacioli's Memorandum with

a database of events and abandons the journal and the ledger. Ellerman proposes property accounting as a valuation free alternative to financial accounting [Ell82; Ell86]. He suggests a system that can store values in multiple units at the same time by using vectors. All these approaches attempt to eliminate the valuation from accounting, but accounting systems without exception still use Pacioli's system of double entry bookkeeping.

1.3 RESEARCH

This thesis has been written as part of the Next Generation Auditing project. This project is a unique cooperation between experts in Dutch auditing theory and software engineering researchers specialized in software analysis and domain-specific languages.

The grand vision of Next Generation Auditing is

> *... to create auditing services that can perform real-time monitoring of companies in order to fundamentally increase the reliability and transparency of financial systems and supporting IT systems.*

The aim is to create techniques to describe, analyze, and maintain specific company models and to automatically generate auditing services from them. By using and specializing various software engineering techniques like modeling, domain-specific languages, software analysis, software transformation and applying them to model, design, and implement trustworthy auditing services, this impacts both auditing methods and supporting IT systems.

The Netherlands is in the unique position that it has developed and formalized an audit theory [FdH89; SdMJ88]. Although this theory has been applied to large multinationals like Shell, Philips, Unilever and Akzo and generations of Dutch auditors have been involved in the evolution, application and education of this theory, it has never been actively promoted in international literature. Central in the Dutch audit approach is Starreveld's value cycle, based on a classification of industry-specific top-level business process models. Originally more intended as mental models, these models evolved into formal models based upon mathematical linear equation systems.

The initial formalization of the value cycle from Dutch auditing was done in the Computational Auditing research project [Els96]. The mental models from audit were formalized as Petri nets and it was shown that much theory from the audit field, including the audit equations could be derived. The so-called audit nets are similar to the process model from the wine bottler case earlier, but they contain a number of refinements that enable the formulation of a mathematical audit theory.

Further formalization based upon mathematical audit theory in combination with language technology allows digital services to support audit analytics. Examples of services are:

- Specifying a normative model: a top-level business process model showing an enterprise's flow of money and products.
- Confronting this normative model with actual business design, implementation, operations, and loggings.
- Passing judgment on an authorization system.

Language technology is an excellent tool to capture the domain knowledge required for such services in a so-called Domain Specific Language (DSL), see [Deu04, Deu03, Arn01]. A DSL can form the basis for tool development and service provisioning. The definition and analysis of the high-level models from the auditing domain is the core topic of this thesis.

1.3.1 *Research Questions*

The key research question in the Next Generation Auditing project is how formalized models from Dutch audit theory can form the basis for future, worldwide, auditing services. Using an interdisciplinary approach the research aims to connect the fields of computing and auditing and control.

Unit-Aware Matrix Programming

The first of this thesis' topics, type inference for matrix languages, addresses the question how programming language support for process modeling and analysis in control and auditing can be improved.

The example from the opening explained the importance of units of measurement in the financial field, and cases like the bill of material are the motivation to incorporate units of measurement into programming languages. Since Kennedy's unit type is insufficient to handle the models from audit, the question is if this can be remedied by generalizing dimensioned numbers to dimensioned matrices. As was explained, the structure of units in a matrix is described by Hart's theory of dimensioned vector spaces. The first research question is how this structure can be exploited by type systems.

Research question (1)

Can dimensioned vector spaces form the basis for a parametric matrix type for matrix programming?

Type rules based on dimensioned vector spaces would not only solve the issue of heterogeneous units in parametric typing, but would also have the potential to catch other errors that normal checks miss. For example forgetting to transpose or invert a square matrix cannot be detected by shape checks, because these operations do not change a matrix' shape. The outcome would however be in the wrong vector space, so it is detectable by a type that tracks vector spaces. A matrix type for matrix

programming based on dimensioned vector spaces would improve language safety and increase the use case for units in parametric polymorphic type systems.

As a reference for the matrix type, the first part of the research introduces dimensioned matrices as a concrete index-free definition of dimensioned linear algebra. These dimensioned matrices define evaluation rules for linear algebra operations that serve as a reference for the matrix type's inference rules. Since the idea is to base the matrix type on base unit vectors whose individual entries might not be known at compile time, it is a requirement that the dimensioned matrices' evaluation rules are index-free. Index free means that a rule does not refer to individual vector or matrix entries. The question is whether dimensioned matrices in index-free form are a feasible definition of dimensioned linear algebra.

Research question (1a) Dimensioned Linear Algebra

How accurate can dimensioned linear algebra be expressed in index-free form using base unit vectors?

With suitable evaluation rules the matrix type has a definition to build on. Dimensioned matrices serve as a definition of dimensioned linear algebra relative to which the matrix type's rules are defined.

With dimensioned matrices as reference, the next part of the research defines and evaluates the matrix type. A syntax is given for the matrix type, and for each of the dimensioned matrices' evaluation rules a type rule is defined. The question is how the type rules compare to the evaluation rules.

Research question (1b) Matrix Type

How expressive are the type rules for the matrix type compared to the evaluation rules for dimensioned matrices?

The matrix type itself is defined as an extension of Kennedy's type with a specialized unification algorithm. The soundness and completeness of unification for the matrix type is easily established. This means that the expressiveness of the matrix type only depends on the rules. Each type rule must at least be correct, and follow the evaluation as close as possible. Evaluating the matrix type like this shows how well it types dimensioned linear algebra.

Value Nets

The second research topic is the study of process models for control and auditing in non-financial units of measurement. This is not just a question of safety from a programming point of view, but also a more fundamental issue regarding the application of the process models.

The audit field developed mathematical equations to analyze value cycle models, but the utility of these audit equations is limited because they are formulated directly

as integrity rules on an accounting system. An enterprise's accounting system plays a central role in an audit because the financial statements that an auditor has to assess are generated by it. As parts of the tests to assess the accounting system's integrity, auditors developed equations based on the value cycle. These equations use the proportions from the value cycle model to relate the occurrences of the various events and the corresponding changes in accounts. In this way the concepts of causality and proportionality from the value cycle are used to validate process behavior, but this modeled behavior is never made explicit. The normal behavior that is implicitly described by the auditors in their checks is completely interwoven with the application. If the equations can be decoupled from the application to accounting systems then the core concepts from the value cycle can be used for other applications.

Research question (2)

How can the analytical features based on the value cycle's causality and proportionality be generalized from auditing to other applications?

Decoupling the application from checking an accounting system increases the analytical possibilities and opens the value cycle for use in other application areas. The value cycle's use of proportionality gives a powerful analysis tool that is also useful for other purposes besides auditing, but by directly stating the equations in terms of accounting systems it does not use its full potential and other applications are not directly feasible.

The main issue in the generalization of the value cycle is that the audit equations are formulated as conservation laws, but that value is not a conserved quantity. The audit equations are a combination of conservation laws for the value cycle's states, and laws of proportionality for the value cycle's events. The conservation laws express that the difference between a state's input and the output over a period must equal the change in stock. This means that if the change in stock is compensated for, then what goes in must come out, just like Kirchhoff's law for electrical networks. The laws for the events state that in each transformation step the input and the output are related by a constant factor. These laws capture the proportions that are central to the value cycle. Combined with the conservation laws this gives a set of equations for an accounting system. In this overall set of equations there is however no conservation of value, as there is usually a value jump. Auditors address this separately by assessing the valuations involved, but for other applications this defeats the purpose of the equations and is not a general solution. The audit net formalism circumvents the issue, but as a consequence it is only applicable to a limited class of value cycle models. For proper general conservation laws the valuation needs to be eliminated from the equations.

Eliminating the valuation from the audit equations would solve the conservation laws but this is not straightforward because the convenience of a single unit is lost. An

important feature of an accounting system is that all quantities are in a single financial unit. Not only does this enable the required aggregation of quantities for financial reporting, but it is also convenient for auditing purposes. If for example all goods receipts are recorded and stored on an account then it is possible to check that the total purchases equals the total of the received goods. This check can not be expressed in non-financials because the received goods cannot be aggregated in physical units if the products have different units. The downside of a single monetary unit is however that it causes inaccuracies because information is lost. Financial information involves a valuation as is expressed by equation

$$\text{value} = \text{valuation} \times \text{quantity}$$

Since value is the product of a valuation and a quantity, accounting systems lose information when it just stores the value. A conservation law is applicable to quantities, while financial information is usually subject to a fluctuating valuation. Conservation of value would even be in contradiction with an enterprise's goal to create value. Equations in units of measurement make it possible to analyze the quantities without any valuation issues.

Researching this topic was done by the development of value nets. A value net is a formalization of the value cycle that models and analyzes process behavior in non-financial units of measurement. As explained, it is a Petri net that models an enterprise's value-creation process, similar to an audit net, but its goal is to generalize it for uses in other applications. If the equations can be reformulated without any valuation it removes the issue with the value jump in the conservation laws.

Research question (2a) Value Net Model

Can the valuation be eliminated from the value cycle models by reformulating the audit equations in non-financial units of measurement?

Eliminating the valuation complicates any equation set because the lack of aggregation due to difference in units would lead to many duplicate equations to keep all the quantities properly separated. Also the direct relationship with financial and accounting information is lost and has to be recovered. However, the increased accuracy gained by eliminating the valuation increases the precision and therefore the analytical possibilities.

The solution pursued in this thesis formulates the audit equations as a steady-state equation using dimensioned linear algebra, and gives a precise characterization of a value net's normal behavior as a *tour space*. The normal behavior is defined as the solution to the steady state equation where the net's input and output are in equilibrium. This equation uses dimensioned vectors to handle the complexities of the units of measurement. With dimensioned vectors spaces all different products can be given their own unit while still being handled as a single entity. Any vector in

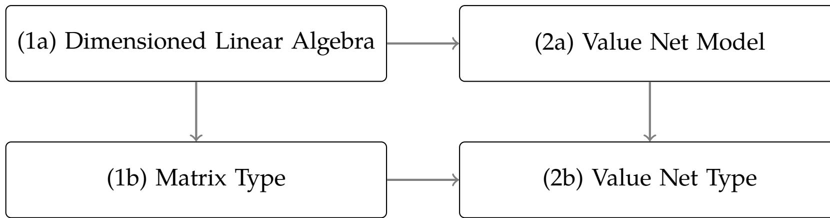


Figure 1.2: An overview of the relationships between the four research topics

the solution space corresponds with a *tour* in the value net and the complete space is called the tour space. This space is the normal behavior that follows from the causal relations and the proportions in the value net. With this approach, instead of directly applying the value net to accounting data, a value net's normal behavior in non-financial units of measurement is derived separately. If desired the comparison with accounting information can be made later by multiplying the quantities with any valuation to get a uniform financial unit, but the derived behavior itself is in non-financial units. The tour space is the normal behavior that follows from the causal relations and the proportions in the value net and is useful for all kinds of analysis. Having the normal behavior as separate result makes it available for other purposes like costing, budgeting etc. With a steady-state behavior in non-financial units the equations become true conservation laws and can be solved to give the normal behavior specified by the causality and proportionality in a value net.

The final research topic is the actual implementation of the value net model with the matrix type in the Pacioli language. The Pacioli language is a functional matrix language with an ML-style parametric type system developed during earlier research and development. It has its own unit-aware runtime system called the Matrix Virtual Machine (MVM), but to experiment with various runtime options the compiler can also generate Matlab and JavaScript code. This provides an environment where the matrix type is easily added and also provides many features that are useful to test the value net and the type. The actual implementation of value nets ties together the other research. It is useful as a proof of concept of the value net models, but it is also a proof of concept for the matrix type.

Research question (2b) Value Net Type

Can a statically checked data type for value nets implement the value net models?

A data structure was developed that provides a library of modeling and analytical primitives for value nets. Experiments to validate the matrix type were done in the Pacioli language. The application of Pacioli to the models from auditing validates the value net models and the matrix type at the same time.

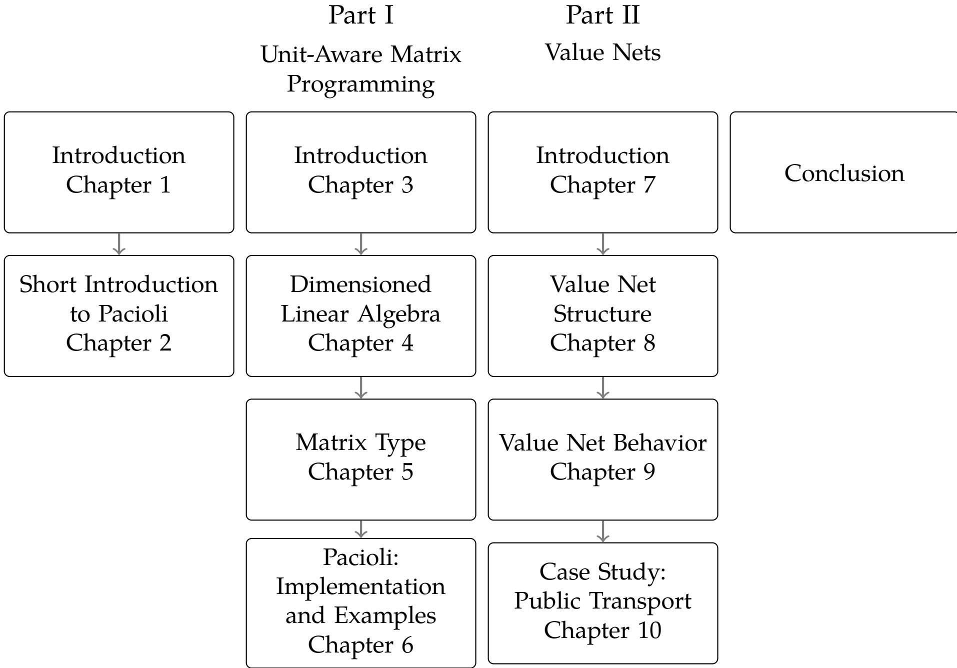


Figure 1.3: A schematic view of the chapters in the thesis' parts.

Figure 1.2 gives an overview of the relationships between the four research topics. In the top row it shows how value net models use dimensioned matrices to support units of measurement. Similarly in the bottom row it shows how the value net type uses the matrix type in its implementation. The topics on the top are conceptual, while the topics on the bottom implement the concepts.

1.4 THESIS LAYOUT

The thesis is divided into a part about unit-aware matrix programming and a part about value nets. The first part is about matrix programming and is not concerned with auditing or process modeling at all. The second part uses the results from the first part to define value nets. Figure 1.3 shows how the thesis' chapters are organized.

Chapter 2 gives a short overview of the Pacioli language and a short explanation of type inference with units of measurement. Section 2.1 discusses Pacioli features that are used in this thesis and that are relevant for units of measurement and the matrix type. Section 2.2 introduces Kennedy's inference for units in the form of a series of small Pacioli examples.

Chapter 3 opens the first part with an overview of dimensioned linear algebra and the matrix type. Section 3.2 explains how a matrix implementation in a programming language can be extended with units of measurement following the principles of dimensioned vector spaces. In Section 3.3 dimensioned matrices are informally explained with various examples. After describing the base case for vectors, the structure of units in matrices and tensors is discussed. Section 3.4 introduces the matrix type notation and its relation with dimensioned matrices.

In Chapter 4 the details of the dimensioned matrices are explained. It starts with an explanation of units of measurement. The mathematical details of Hart's dimensioned vector spaces on which everything is based is discussed next in Section 4.2. Relevant mathematical details are given in the appendix. In Section 4.3 dimensioned matrices are defined.

The matrix type from Chapter 5 is the compile time counterpart of dimensioned matrices. The type's syntax is defined in Section 5.1. In Section 5.2 a unification algorithm is given for the matrix type and accompanying type inference rules are stated. The units in the exponentiation operators are the most complex and discussed in Section 5.3. The expressiveness of the type is discussed in Section 5.4.

Chapter 6 shows how the matrix type is implemented in the Pacioli language and discusses three elaborate examples. First the implementation and experiments with different runtime systems are discussed in Section 6.1. Next in Section 6.2 an overview of language features that are specific for linear algebra is given. The case studies are discussed in the remaining sections. The first case study explains the combination of scalar units and vectors with an implementation of Kirchhoff's law for electrical networks. The second case is an implementation of the Fourier-Motzkin algorithm. The bill of material case shows how the type system handles units as explained in the bottler case earlier.

The results in Section 4.3 and Chapter 5 have been published in [Gri15]. The bill of material case is published in [GCH18]. All cases and examples have been calculated with the Pacioli implementation of the matrix type.

Part II starts with an introduction to the topic of mathematical auditing in Chapter 7. It explains some background of the value cycle and gives an informal introduction to the value nets that will be developed in this part. Throughout the text the wine bottler will be used as a running example.

Chapters 8 and 9 define the value net data structure and give the details of the analysis of a value net's behavior. Instead of separating the definition and the implementation into different chapters as was done for the dimensioned linear algebra and the matrix type, the concepts and their implementation are defined side by side in these two chapters. Chapter 8 deals with the encoding of the information in a value net in a well-typed data structure. Chapter 9 is concerned with the value net's behavior and various tools to analyze it.

Chapter 10 discusses a case study about a public transport provider. The case demonstrates the developed value net analytics and shows how value nets are useful in the strategic design and analysis of enterprise behavior.

The definition of value nets is published in several joint papers with auditing experts. The potential usefulness of the strategic analysis done with value nets for other fields was already published in [GEvdR00]. The importance of normative models for the analysis of recordings as evidence is published in [CGH15b]. The public transport case study was published in [CGH15a].

The following summary gives an overview of the papers that were published as part of this research and the corresponding chapters where parts of the papers have been included.

Paper	Chapter
• Type Inference for Array Programming with Dimensioned Vector Spaces [Gri15]	3, 4, 5
• Controlling production variances in complex business processes [GCH18]	6, 7
• Analyzing enterprises: The value cycle approach [GEvdR00]	7
• Reliability of electronic evidence: An application for model-based auditing [CGH15b]	8, 9
• Adaptive Normative Modelling: A Case Study in the Public-Transport Domain [CGH15a]	8, 9,10

The second and third paper are entirely based on the research into value nets. The last two papers describe joint work with other research projects where the application of value nets was tested with researchers from the domain of control and auditing. The author's contribution to these papers is process modelling, mathematical analysis using value nets and their implementation in Pacioli, whereas the co-authors provided essential knowledge about the respective application domains.

A public Pacioli implementation can be found at

`http://pgriffel.github.io/pacioli/`.

The short overview in the next chapter explains all the basic Pacioli features that are used in this thesis. Extensions for linear algebra are given later in Section 3.4 and Section 6.2 after the matrix type has been developed.

2 SHORT INTRODUCTION TO PACIOLI

After a short introduction to the Pacioli language, type inference with units of measurement is explained in the form of some small Pacioli examples.

2.1 THE PACIOLI LANGUAGE

Pacioli is a functional matrix language. Besides the matrix type it supports types like booleans, lists, tuples and first class functions.

Pacioli fragments in the text typically define a function or print some value. The fragments are shown in typewriter font and are included in the text as a separate block. A definition starts with keyword `define` followed by a name and an expression. Each definition is followed by the type that is derived by the type system. This is indicated by the symbol ↪. When a value is printed the symbol ↪ indicates that the result of an evaluation follows. For example the following example defines and prints the value of `foo`.

```
define foo = 1 + 1;
↪
foo :: 1;

print foo;
↪
2
```

The compiler infers that the type of variable `foo` is 1, the type for dimensionless numbers. When `foo` is printed expression 1+1 is evaluated and the result 2 is shown.

A characteristic property of Pacioli is that vectors and matrices are not indexed with natural numbers but with index sets that can be defined with a special language construct. Instead of a map $\mathbb{N} \to \mathbb{R}$ from the natural numbers to the reals, a vector is of the form $I \to \mathbb{R}$ with I the vector's index set. For example Pacioli's geometry library defines the index set `Geom3` for three-dimensional vectors as follows.

```
defindex Geom3 = {x, y, z};
```

Keyword `defindex` starts an index set definition. It is followed by the index set name and the elements. Vectors with the `Geom3` index set are indexed by the names `x`, `y` and `z`.

A vector's type is the name of its index set followed by an exclamation mark. The call to library function `vector3d` in the next fragment creates a `Geom3` vector.

```
define some_vector = vector3d(5, 12, 13),
↪
some_vector :: Geom3!;

print some_vector;
↪
Geom3        Value
-----------------
x            5.000
y           12.000
z           13.000
```

Pacioli favors an index-free programming style, but if the elements of an index set are known at compile time they can be referred to by the index set name and the index name separated by the `@` symbol. For example in the geometry library the name `Geom3@y` refers to index `y` of `Geom3` vectors.

```
print vector_get(some_vector, Geom3@y);
↪
12.000
```

Function `vector_get` retrieves the value at position `y`.

Pacioli uses square brackets for list literals and list comprehensions. For example the following fragment defines value `some_nums` as the list `[1, 2, 3]` and uses it to create and print list `some_even_nums`.

```
define some_nums = [1, succ(1), sqrt(9)];
↪
some_nums :: List(1);

define some_even_nums = [2*x | x <- some_nums];
↪
some_even_nums :: List(1);

print some_even_nums;
↪
[2, 4, 6]
```

Type `List(1)` in this case means a list of dimensionless numbers. Type `1` is a dimensionless number and type `List(t)` is a list with elements of type `t`.

Another data type used in the fragments in this thesis is the tuple type. The following comprehension creates a list of tuples.

```
define foo = [tuple(x, y) | x <- some_nums, y <- some_nums, x + y = 4];
↪
foo :: List(Tuple(1, 1));

print foo;
↪
[(1, 3), (2, 2), (3, 1)]
```

Function `tuple` creates a tuple from its arguments. The type of a tuple with values of types `t1`, ..., `tn` is written as `Tuple(t1, ..., tn)`. Type `Tuple(1, 1)` in the example is a pair of dimensionless numbers.

Functions are also defined with the `define` keyword. The type of a function with argument types `t1,...,tn` and result type `t` is written as `(t1,....,tn) -> t`. For example, the following fragment is a possible definition of the successor function used earlier.

```
define succ(x) = x + 1;
↪
succ :: (1) -> 1;
```

The inferred type correctly shows that the function expects a dimensionless number and also returns a dimensionless number. After the matrix type is introduced a unit-aware version of the successor function is defined in Section 5.4 to demonstrate how a general unit-aware alternative can be written with the matrix type.

Pacioli has a polymorphic type system in which type variables are introduced with keyword `for_type`. Say we create a function to make pairs of numbers like the ones from the example earlier.

```
define numpair(x, y) = tuple(x, y);
↪
numpair :: for_type a, b: (a, b) -> Tuple(a, b);
```

Function `numpair` expects two values and creates a tuple from them. This is derived by type system and expressed with type variables `a` and `b`. In the remainder of this section function `numpair` is used as a small running example to illustrate the use of units of measurement in Pacioli.

A language feature that proved useful in practice is a facility for optional type declarations. For many functions the inferred type is more general than what the programmer intended. An example is the previous definition of `numpair`. Although

the type is correct, we are interested in the specific case `a = b = 1`. Type declarations allow the type to be strengthened in cases like this. This prevents incorrect usage of the function and improves the inferred types of functions that use this one. For example `numpair`'s type could be declared as follows.

```
declare numpair :: (1, 1) -> Tuple(1, 1);
```

The type is obtained by substituting type 1 for type variables `a` and `b`. The compiler checks that the declared type specializes the inferred type and does not contradict or generalize it. In the next section unit variables will be introduced, and than this function can be generalized from tuples of dimensionless numbers to tuples of dimensioned numbers. But before we can do that we need to see how units are incorporated into the type system and how they can be used to create dimensioned numbers.

Units names are valid types and are part of Pacioli's type system. For example `foo :: metre` indicates that variable `foo` is a dimensioned number with unit `metre`. Units like `metre` can be defined with keyword `defunit`. The definition expects a unit name and a symbol string. The following fragment defines the unit `metre` with symbol string `"m"` and uses it to type function `numpair`. The symbol string is used to represent the unit of a value.

```
defunit metre "m";

declare numpair :: (metre, metre) -> Tuple(metre, metre);
```

This restricts the function to pairs of dimensioned numbers of unit `metre`. In this way a unit name can be directly used as a type in the type system.

Many programming language features are possible to create dimensioned numbers from some given units. One feature from Pacioli that is used in many fragments in this thesis is Pacioli's special notation for unit literals. This notation uses pipes (`|`) as delimiters for unit expressions. For example `|metre|` means one metre and `|metre/second|` means one meter per second. As we will see later the expression between the pipes is actually a matrix type, but since scalars in Pacioli are just 1 by 1 matrices the notation works for scalar numbers as well. The matrix type is constructed in such a way that unit names and the dimensionless 1 are special cases. The following fragment illustrates the use of units expressions. It defines the value of `foo` as the product of the number 10 and the units gram and metre. When the resulting dimensioned number is displayed the unit's symbols are used and it is shown as `10 g*m`.

```
define foo = 10 * |gram*metre|;
↪
foo :: gram*metre

print foo;
↪
10 g*m
```

Note that the unit could also be written as `|gram|*|metre|`. The result would be exactly the same. Both variants build a dimensioned number from the unit literals. More on that in the next section.

With unit literals the dimensioned version of function `numpair` can be used. The following fragments uses unit `|metre|` to pass dimensioned numbers to the function.

```
define foo = numpair(3*|metre|, 4*|metre|);
↪
foo :: Tuple(metre, metre)

print foo
↪
(3 m, 4 m)
```

The function call is type correct and result in a tuple of metres. Although this works, it is often not desirable to commit to a specific unit. So we see that without a type declaration function `numpair` is too generic, but with a type declaration it is too specific. The next section explains how Kennedy's unit variables can be used to give function `numpair` a unit correct parametric type.

2.2 UNIT INFERENCE

Kennedy's unit unification algorithm infers principal types for dimensioned scalar numerical expressions. A key feature is that the type system checks the unit-correctness of the numerical operations at compile time. For example, in the next fragment the sum is incorrect.

```
define foo = |gram| + |metre|;
↪
Error: cannot unify units metre and gram
```

The type system infers that the sum's arguments are not compatible and signals

an error. The error is caught at compile time, before any evaluation takes place at runtime.

Another key feature of the unit type is that it is semantic. The type implements the rules of abelian groups, allowing it to derive the equality of differently expressed units. For example, the type is commutative, which means the order of multiplication is irrelevant. The following definition is unit correct.

```
define foo =
  10*|gram|*|metre| + 20*|metre*gram| + 30*|metre^2|*|gram/metre|;
↪
foo :: gram*metre
```

Each of the three terms is a different way to write the same unit. For the type system the units are equal.

The type also allows unit variables and is able to infer the type of any arithmetic expression. In the following function definition the type of variable `x` is unspecified.

```
define f(x) = x*|metre| + |gram|*|metre|;
↪
f :: (gram) -> gram*metre
```

The type system infers that the unit must be gram and correctly displays the function type. Kennedy's main result is that his type system derives the most general type for any numerical expression. If a second argument is introduced in the previous function, then an infinite number of possibilities arise. Using a unit variable the type system can still infer the type correctly.

```
define g(x,y) = x*y + |gram|*|metre|;
↪
g :: for_unit a: (a, gram*metre/a) -> gram*metre
```

Keyword `for_unit` introduces unit variable `a`. By dividing the required unit `gram*metre` by `a` the type describes all possible cases. For example, if `x` has unit gram as in the previous code fragment, then `y`'s unit is a metre, but this could also be reversed with `x` of unit metre and `y` of unit gram. The first case is obtained when the unit gram is substituted for unit variable `a`. In the second case `a` is the unit metre. The division in the type ensures that for any `a` the units are correct. The following example shows some uses of function `g`.

```
define foo = g(3*|gram|, 5*|metre|) +
             g(2*|metre|, 6*|gram|) +
             g(9*|metre^2|, 2*|gram/metre|) +
             g(4*|metre*second|, 7*|gram/second|);
↪
foo :: gram*metre
```

All four calls are unit correct. In the last one the unit `second` in the first and second argument cancel. This ability to infer the unit correctness of any expression is a novel and powerful feature.

With unit variables a type declaration can completely abstract from concrete units. This is for example useful when writing generic code or a library. In such cases you typically don't want to commit to a specific unit. With unit variables this is possible. For example the following code is unit correct without any reference to a concrete unit.

```
define f(x, y, z) = 2*x*y + 3*y*z;
↪
f :: for_unit a, b: (a, b, a) -> a*b;
```

This combines nicely with type declarations. For example if in the example the unit of arguments `x`, `y` and `z` are known to be the same the type can be strengthened as follows.

```
declare f :: for_unit a: (a, a, a) -> a^2;
```

All arguments have the same unit and the result is the square of that unit. Similarly the `numpair` function can be given a completely general unit correct type. For example:

```
declare numpair :: for_unit a: (a, a) -> Tuple(a, a);
```

In this case the function creates pairs of numbers with the same unit. In general unit variables are a powerful addition to polymorphic type systems that significantly improves type inference for numerical expressions.

Type inference for the matrix type generalizes Kennedy's unification algorithm from dimensioned numbers to dimensioned matrices. Kennedy's type gives powerful support for units of measurement, but as explained it is insufficient to type linear algebra operations. With the discussed language features it is possible to define a parametric vector or matrix type like `Vector(a)` or `Matrix(a)`, but this excludes matrices like the bottler bill of material in which the elements have different units. The matrix type solves this issues and extends the unit type to linear algebra expressions.

Part I

Unit-Aware Matrix Programming

3 INTRODUCTION

This introduction gives an informal overview of dimensioned matrices with various examples. After describing the base case for vectors, the structure of units in matrices and tensors is discussed. The final section introduces the matrix type notation and its relation with dimensioned matrices.

3.1 TYPE INFERENCE FOR MATRIX PROGRAMMING

The combination of parametric polymorphism and type inference in functional programming has enhanced the safety of programs enormously, but support for numbers is still limited. Vectors and matrices are common tools in numerical software, but while specific support for linear algebra operators has the potential to increase safety, it is not typically part of type systems. A parametric type system usually provides some built in primitive type for single numbers. Compound numerical data can then be built with the usual data types like tuples, lists, records, or any form of data abstraction. The usual checks on these types provide many guarantees for the correct usage, but are not readily capable of detecting properties like the correct shape of vectors and matrices. Such dimension checks are too specific for the general type system, and have to be performed at runtime, at the expense of safety and performance.

The matrix type developed in this part of the thesis improves support for numerical data with a unit-aware type that infers types using dimensioned vector spaces. The term *dimensioned* is in this case not about size, but about units of measurement. A dimensioned vector space couples a unit of measurement with each vector entry. The underlying idea is to split a vector into an element-wise product of a vector of numbers and a vector of units of measurement. For example

$$\text{replace} \begin{bmatrix} 48\,\text{hr} \\ 60\,\text{m}^2 \end{bmatrix} \text{ by } \begin{bmatrix} 48 \\ 60 \end{bmatrix} \cdot \begin{bmatrix} \text{hr} \\ \text{m}^2 \end{bmatrix}$$

The separated unit vectors cannot only describe the units in vectors as in this example, but can also be combined to describe the units in matrices and tensors. They are used in dimensioned matrices to define unit-aware operations for linear algebra. Dimensioned matrices could be implemented directly by a dynamically typed language, but in the context of static typing they just serve as a conceptual definition for validation purposes. The rules for the matrix type guarantee at compile time that the operations satisfy the laws of dimensioned vectors spaces, thus increasing the safety of numerical operations without any runtime overhead.

The matrix type is based on Kennedy's result that the abelian group[1] of units extended with type variables can be used to infer the type of expressions build from dimensioned numbers. A dimensioned number combines a number with a unit of measurement. Mathematically, a unit of measurement is from an abelian group finitely generated from a set of base units $\mathcal{B} = \{b_1, \ldots, b_n\}$. The group's identity 1 denotes the dimensionless numbers, and a unit is build from the base units by multiplication and division. Examples of base units are the well known SI units kilogram, metre, second, etc., with symbols kg, m and s. An example unit made from these bases is $\mathrm{kg} \cdot \mathrm{m/s}^2$. Kennedy extended the base units with unit variables $\mathcal{V} = \{v_1, v_2, v_3, \ldots\}$ and showed a unification algorithm that derives a principle type for any numerical expression. Not only does this give the safety of units of measurement without the burden of annotating every numerical variable, but the unit variables also enable programmers to abstract from specific units of measurement. A program can be unit correct without committing to specific units like meter or feet.

The matrix type extends the scalar base units with unit of measurement base vectors $\mathcal{U} = \{u_1, \ldots, u_m\}$. The bases u_i are unit vectors like the separated unit vector from the example earlier. As long as no other than the abelian group operations are performed on the unit vectors, the extension does not alter the group's properties and therefore Kennedy's algorithm still applies. With the unit vectors, evaluation and types rules can be expressed for dimensioned linear algebra.

To support multi-linear algebra the matrix type actually stores indexed unit of measurement base vectors from $\mathcal{U} \times \mathbb{N}$, but matricization mostly hides this technicality. A tensor is equivalent to a matrix with compound indices. To support multi-dimensional data the type system matricizes tensors into matrices. Technically this is done by storing a base pair (u, i) with u a unit vector and i a natural number. For vectors and matrices i always equals 1, but for the higher dimensions of tensors i indicates to which dimension u applies. Although this is irrelevant for a large part of the technical development, the support for multi-linear algebra increases the usability of the matrix type significantly.

Each discussed extension to the abelian group's base increases a type system's capabilities for the support for numbers. The set of base units B of the group of units can be varied in different ways without altering the abelian group's structure. The

[1]See the Appendix for definitions of mathematical terms used in this introduction. In later chapters precise references to the appendix are given at relevant places.

following summary shows how each variation increases the capabilities of a type system.

$B = \{1\}$	dimensionless numbers
$B = \{1\} \cup \mathcal{B}$	dimensioned numbers
$B = \{1\} \cup \mathcal{B} \cup \mathcal{V}$	parametric number type
$B = \{1\} \cup \mathcal{B} \cup \mathcal{V} \cup \mathcal{U}$	parametric matrix type
$B = \{1\} \cup \mathcal{B} \cup \mathcal{V} \cup (\mathcal{U} \times \mathbb{N})$	parametric tensor type

The trivial case where the base units is simply the singleton set {1} corresponds to no unit support at all. Supporting units requires a set of base units $\mathcal{B}$ from which other units can be derived by multiplication and division. This extension of the base units is the least requirement for unit support. The extension with $\mathcal{V}$ in the third case adds Kennedy's inference for units. The unit vectors from the fourth case adds support for dimensioned linear algebra and conceptually corresponds with the matrix type, but to support the multi-linear case the type matricizes tensors and actually uses the indexed unit vectors from the fifth case.

The rest of this chapter explains the concepts from dimensioned linear algebra in terms of the implementation of dimensioned matrices. As explained, dimensioned matrices are a dynamically typed implementation of dimensioned linear algebra that serve as a reference for the statically typed implementation with the matrix type. The next section explains how their implementation extends the multi-linear arrays from matrix programming with units of measurement. We will see that the extension requires the distinction between covariant and contravariant dimensions and factoring out scalar units. After that various examples will explain how vectors, matrices and tensors are represented in dimensioned matrices. Finally the matrix type notation is explained.

3.2 DIMENSIONED MATRIX SHAPES

In matrix programming numerical values like vectors and matrices are represented by multidimensional arrays. A multidimensional array stores numbers in multiple dimensions. To layout the numbers, an array implementation needs to know the sizes of the array's dimensions. This information is typically stored in an array of integers $[s_1, \ldots, s_d]$ with s_i the size of an array's i-th dimension and d the number of dimensions. The array with the dimension sizes is called the array's shape. Shorthand x^n is used for the sequence $x_1, \ldots, x_n$. With this notation an array value can be described as a tuple

$$\langle [s^d], [r^p] \rangle$$

where the size of the array p is the product of the dimension sizes s_i. The array $[s^d]$ is the shape and array $[r^p]$ contains the actual numbers. An implementation uses the

matrix shape to layout the numbers in the number array. Consider for example the three by three matrix from the introduction chapter, but without the units.

$$B = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 750 & 0 & 0 \end{bmatrix}$$

This matrix is represented by the 2-dimensional array

$$\langle [3,3], [0,0,0,1,0,0,750,0,0] \rangle$$

The layout of the numbers is in row major order, meaning that the values are laid out per row. The value 1 from B_{21} is for example mapped to r_4 and and value 750 from B_{31} to r_7. With the shape information a multi-dimensional array can store numbers in multiple dimensions.

Conceptually a dimensioned matrix is a matrix where the numbers have been replaced by dimensioned numbers, but simply adding units of measurement to a matrix is too expensive and not suitable for typing. A unit of measurement object is much more expensive than a primitive number, so coupling a unit with each number would give an enormous overhead. An alternative would be to associate one unit with a vector. This is similar to a static parametric type like Vector(a) and as explained in the introduction this would work well with Kennedy's inference for units, but then each element has the same unit a. Parametric polymorphism cannot type numerical data if the elements can have different units of measurement.

The limitations of parametric polymorphism and the high overhead of units were motivations to investigate a more efficient alternative that adds units of measurement to a matrix' shape. The concept is based on Hart's dimensioned vector spaces. These spaces are typed by vectors of units u built from base unit vectors. A scalar unit can be written as a product $b_1^{\sigma_1} \cdots b_n^{\sigma_n}$ with $b_i \in \mathcal{B}$ a base unit and $\sigma_i \in \mathbb{Z}$ its exponent. For example the unit from the opening paragraph can be written as $\text{kg} \cdot \text{m} \cdot \text{s}^{-2}$. Similarly, unit vectors have the form

$$u = u_1^{\rho_1} \cdots u_m^{\rho_m} \tag{3.1}$$

with $\rho_i \in \mathbb{Z}$ and each $u_i \in \mathcal{U}$ a base unit vector. The multiplication in these expressions is element-wise.

$$(u_1^{\rho_1} \cdots u_m^{\rho_m})[i] = u_1[i]^{\rho_1} \cdots u_m[i]^{\rho_m} \tag{3.2}$$

For instance, using units hour (hr) and metre (m):

$$\begin{bmatrix} \text{hr} \\ \text{m}^2 \end{bmatrix}^{-1} = \begin{bmatrix} \text{hr}^{-1} \\ \text{m}^{-2} \end{bmatrix} \text{ and } \begin{bmatrix} \text{hr} \\ \text{m}^2 \end{bmatrix}^{2} = \begin{bmatrix} \text{hr}^2 \\ \text{m}^4 \end{bmatrix}$$

Dimensioned matrices use unit expressions with vector bases like this instead of individual units per number. Lifting the unit operations to vectors like this allows an efficient implementation of units of measurement in matrices.

An issue to be solved when adding unit vectors to matrices is that the shape must distinguish covariant and contravariant dimensions. Covariance and contravariance in tensor calculus describes how values change with a change of units and should not be confused with covariance and contravariance in types. In a contravariant dimension the values changes in the opposite direction as the units. For example if some units are scaled down a factor thousand by changing from meters to millimeters then the values must be scaled up by a factor thousand. In a covariant dimension the situation is reversed and the values changes in the same direction as the units. In Hart's theory about the structure of units in matrices the distinction between covariant and contravariant dimensions is crucial. To accommodate this distinction the dimensioned matrices implementation splits the shape array $[s^d]$ into a contravariant shape array $[m^k]$ and a covariant shape array $[n^l]$ with $d = k + l$. The pair (k, l) is called the matrix' order. A scalar has order $(0, 0)$, a column vector has order $(1, 0)$, a row vector has order $(0, 1)$, and a matrix $(1, 1)$. The distinction between covariant and contravariant dimensions is not commonly found in matrix implementations, but necessary for the unit support introduced in this thesis.

Another technicality in the dimensioned matrix implementation the handling of scalar unit factors. The matrix shape factors out scalar units from the unit vectors to accommodate the scaling operator from linear algebra. Scaling a vector or matrix is one of the core operators of linear algebra. The product between a scalar a and some vector v can be defined by

$$(a \cdot v)_i \;=\; a \cdot (v_i)$$

but this accesses the unit vector v and we want to avoid that. An alternative definition that doesn't access the vector's elements can be given if scalars are factored out and a vector is seen algebraically as a product $a \cdot v$ between a scalar a and a vector v. This amounts to writing

$$\text{€}\,/\begin{bmatrix}\text{btl}\\ \text{btl}\\ \text{ml}\end{bmatrix} \quad \text{instead of} \quad \begin{bmatrix}\text{€/btl}\\ \text{€/btl}\\ \text{€/ml}\end{bmatrix}$$

With factored out scalars a rule for scaling without looking at v's elements is:

$$\frac{x \Rightarrow a, \quad y \Rightarrow b \cdot v}{x \cdot y \Rightarrow (a \cdot b) \cdot v}$$

By factoring out the scalar units the implementation of dimensioned matrices can be kept index-free.

With the covariant and contravariance and the scalar unit factors resolved the shape of a dimensioned matrix can be given. The extended matrix shape is a tuple

$$\langle a, [m^k], v, [n^l], w, [r^p] \rangle$$

Unit a is the scalar unit factor. Array $[m^k]$ is the contravariant shape array and v a unit expression built from base unit vectors for the k contravariant dimensions. Array $[n^l]$ is the covariant shape array and w a unit expression built from base unit vectors for the l covariant dimensions. Vectors v and w are unit vectors expressions of the form given in equation (3.1). The unit at position $[i^k]$ is written as $v[i^k]$. Together with units scalar a these two unit vectors can describe the units of measurement in any matrix. The overall unit is the unit for the contravariant dimensions divided by the units for the covariant dimensions. So, for indices $[i^k], [j^l]$ the unit at these coordinates is

$$a \cdot \frac{v[i^k]}{w[j^l]} \tag{3.3}$$

The equation shows that units are multiplied for contravariant dimensions and divided for covariant dimensions. This is the general principle that was mentioned in the introduction where the unit in the matrix was the unit in the row dimension divided by the unit in the column dimension. According to Hart's theorem about units of measurement in a matrix the units in any matrix can be described like this. The units in a linear transformation are determined by the units of the source and target vector spaces. Storing the units for the co- and contravariant dimensions is sufficient to support units of measurement and the basis for the units in dimensioned matrices and the matrix type.

3.3 DIMENSIONED VECTORS, MATRICES AND TENSORS

To see how the extended shape is used, consider the following data that may come from a database, or some other source.

Product	prod_unit	purchase_price	sales_price	sales
Butter	gram	0.40 ¢/g	-	-
Flour	gram	0.25 ¢/g	-	-
Apples	gram	0.09 ¢/g	-	-
Sugar	gram	0.08 ¢/g	-	-
Ice water	millilitre	0.00 ¢/ml	-	-
Dough	kilogram	-	-	-
Pie	pie	-	499 ¢/pie	25 pie
Piece of pie	piece	-	49 ¢/pc	100 pc

The data contains numerical data for the purchase price, the sales price, and the sales amount for various products. The units of measurement are an important part of the data. The column with units defines a unit for each product. In this case there is a single unit column, but in other cases it might for example be convenient to have different units like trade_unit and ingredient_unit. Let prod_unit denote the unit vector containing the units from the unit column, then the shapes of the numbers are as follows:

$$\begin{aligned}\text{purchase_price: } &\langle \text{cent}, [8], (\text{prod_unit}, 1)^{-1}, [], 1, [0.40, 0.25, \ldots] \rangle \\ \text{sales_price: } &\langle \text{cent}, [8], (\text{prod_unit}, 1)^{-1}, [], 1, [0, 0, \ldots] \rangle \\ \text{sales: } &\langle 1, [8], (\text{prod_unit}, 1), [], 1, [0, 0, \ldots] \rangle\end{aligned}$$

The shapes contain unit expressions build from the base unit cent and the base unit vector prod_unit. From Equation (3.3) it follows that for any product p the unit for the purchase_price and for the sales_price quantity is cent/prod_unit[p]. For example the purchase price for butter is 0.40¢/g. For the sales data the unit is simply prod_unit[p].

Units of measurement in matrices can be described by combining unit vector bases, without the need for any additional primitives. According to Hart's theorem that will be explained in detail in Section 4.2, units of measurement in a matrix can be completely described by just two unit vectors. In general a matrix has shape $\langle a, [m], v, [n], w \rangle$ where the base vectors in unit expression v have size m and the base vectors in unit expression w have size n. For example, shape $\langle 1, [2], (u_0, 1), [3], (u_1, 1) \rangle$ with $u_0 = \begin{bmatrix} \text{hr} \\ \text{m}^2 \end{bmatrix}$ and $u_1 = \begin{bmatrix} \text{kg} \\ \text{box} \\ \text{bag} \end{bmatrix}$ describes a 2×3 matrix. From Equation (3.3) follows that the units are:

$$\begin{bmatrix} \text{hr}/\text{kg} & \text{hr}/\text{box} & \text{hr}/\text{bag} \\ \text{m}^2/\text{kg} & \text{m}^2/\text{box} & \text{m}^2/\text{bag} \end{bmatrix}$$

The entry for i, j is $u_0[i]/u_1[j]$. The units in a matrix follow from the units in the row and in the column dimension.

The form of the units in a matrix follow from the linear transformation that the matrix performs and is the basis for evaluation and unit checking. Notice that multiplying the example matrix with a vector with units u_1 gives a vector with units u_0. In general a matrix with shape $\langle a, [m], v, [n], w \rangle$ transforms vectors with units w into vectors with units v. Now if a matrix transforms from w to x, and another matrix from x to v, then their product transforms from w to v. This can be stated as an evaluation rule for the matrix product between dimensioned matrices:

$$\frac{x \Rightarrow \langle a, [m], v, [k], x \rangle, \quad y \Rightarrow \langle b, [k], x, [n], w \rangle}{x \cdot y \Rightarrow \langle a \cdot b, [m], v, [n], w \rangle} \tag{3.4}$$

This rule is just an illustration for the units in matrices only, but in Section 4.3 this will be generalized to any shape and numbers will be included. Note the similarity to the well-known matrix product rule $\mathbb{R}^{m\times k} \times \mathbb{R}^{k\times n} \longrightarrow \mathbb{R}^{\mathrm{m}\times\mathrm{n}}$.

An example that illustrates units of measurement in matrices well is the 'explosion' and 'netting' problem in requirements planning from operations research. It involves computations with a bill of material matrix where units of measurement are vital. A miniature example of a bill of material is the following ingredient list from a recipe for apple pie.

Product	Product	BoM
Butter	Dough	360.00 g/kg
Flour	Dough	550.00 g/kg
Ice water	Dough	100.00 ml/kg
Dough	Pie	0.40 kg/pie
Apples	Pie	700.00 g/pie
Sugar	Pie	225.00 g/pie
Butter	Pie	115.00 g/pie
Pie	Piece of pie	0.13 pie/pc

Dough is an intermediate product, made from butter, flour and ice water. The dough is combined with apples, sugar and butter into a pie. Finally a pie is cut into 8 pieces, which means that a piece of pie 'consists' of 0.13 pie. The computations for the explosion and netting problem in requirements planning are done with a bill of material matrix. This square matrix is indexed by products in the row and the column dimension and tells how much of a product is contained in an other product. In this case it would be an 8×8 matrix with the shape as follows:

$$\text{BoM: } \langle 1, [8], (\text{prod_unit}, 1), [8], (\text{prod_unit}, 1)\rangle$$

The units are $\text{prod_unit}[i]/\text{prod_unit}[j]$. Inspection of the units in the table shows that is indeed the case.

The improved error detection capabilities can be illustrated with matrix multiplication. The bill of material matrix can be multiplied with the sales amounts to compute the amounts for parts. It can also be multiplied with the the purchase price to compute part prices, but this requires the transposed matrix.

$$\text{BoM} \cdot \text{sales: } \langle 1, [8], (\text{prod_unit}, 1), [\,], 1\rangle$$
$$\text{BoM}^T \cdot \text{ purchase_price: } \langle \text{cent}, [8], \text{prod_unit}^{-1}, [\,], 1\rangle$$

The rule for the transpose follows from the mathematical properties of units and is as follows

$$\frac{x \Rightarrow \langle a, [m], v, [n], w\rangle}{x^T \Rightarrow \langle a, [n], w^{-1}, [m], v^{-1}\rangle} \tag{3.5}$$

The rule swaps the row and column dimensions and also takes the reciprocal of the units of measurement. It is an easy mistake to mix up the direction or forget a transpose. The matrix type catches these errors because prod_unit does not equal prod_unit^{-1}. Shape information without units cannot detect these errors, because the bill of material matrix is square and therefore its shape and the shape of its transpose would be the same.

Multi-linear algebra generalizes a matrix to a tensor. A matrix is a map $I \times J \to \mathbb{R}$ with exactly one contravariant and one covariant index, whereas a tensor is a map $I_1 \times \ldots \times I_k \times J_1 \times \ldots \times J_l \to \mathbb{R}$ with k contravariant and l covariant indices. Pair (k, l) is the tensor's order.

The units of measurement in a tensor can be described by associating unit vectors with the various dimensions, and this is where index i in the unit bases (x, i) comes into play. For vectors and matrices this index is always 1, but for tensors it tells to which dimension unit vector x belongs.

The columns sales and amount from the following table are examples of multi-dimensional data. The sales column has homogeneous unit dollar. Oil amounts are in barrels and gold amounts in ounces.

Region	Year	Commodity	sales	amount
North	1990	Oil	5,000.00 €	155.00 bbl
South	1990	Oil	87,000.00 €	400.00 bbl
West	1990	Gold	64,000.00 €	150.00 oz
South	1990	Gold	99,000.00 €	235.00 oz
North	1991	Gold	8,000.00 €	18.00 oz
East	1991	Gold	7,000.00 €	16.00 oz

Let's say there are four regions, two years of data, and just the two commodities oil and gold. If it is further given that for the commodities' units

$$\text{commodity_unit} = \begin{bmatrix} \text{bbl} \\ \text{oz} \end{bmatrix}$$

then the shapes for the data is:

$$\text{sales: } \langle \text{dollar}, [4, 2, 2], 1, [], 1 \rangle$$
$$\text{amount: } \langle 1, [4, 2, 2], (\text{commodity_unit}, 3), [], 1 \rangle$$

The sales and amount columns are examples of $(3, 0)$ tensors. For any entry $[r, y, c]$, the unit for the amount is $\text{commodity_unit}[c]$.

3.4 THE MATRIX TYPE NOTATION

The matrix type that will be developed in Chapter 5 is an addition to parametric type systems that infers the dimensioned vectors spaces of dimensioned matrices at

compile time. For each evaluation rule for dimensioned matrices a corresponding type rule is given. The form of these type rules is very similar to the evaluation rules, but uses unit vector names instead of unit vectors.

The notation for the matrix type is based on a special naming scheme for units vectors that incorporates the vector's index set name. This naming scheme works with generalized index sets instead of natural numbers indices, which means mathematically that matrices are of the form $I \times J \to \mathbb{R}$ with index sets I and J, instead of the from $\mathbb{N} \times \mathbb{N} \to \mathbb{R}$. At compile time a unit vector is identified by the combination of an index set name and some identifier, separated by an exclamation mark. For example, names like $I!v$, $I!w$, etc. denote unit vectors with index set I, and names like $J!v$, $J!w$, etc denote unit vectors with index set J. This creates a simple two-level hierarchical name space; the same unit name can be used with different index sets but it is unique in combination with an index set. A concrete example would be the name Product!unit for the prod_unit vector from the previous section. The name Product denotes the index set, and the unit part identifies a particular unit vector for products. Different units may exist for the Product index set, for example Product!trade_unit and Product!ingredient_unit. The notation acts as a kind of constraint or quantification on a unit vector identifier that states that the vector's indices are from some given index set. This naming scheme provides the necessary information about the unit vectors for the matrix type.

The naming scheme allows the identification of dimensioned vector spaces. Dimensioned vector spaces will be defined in Section 4.2, but informally two dimensioned vector spaces $I!v$ and $J!w$ are guaranteed to have the same size if and only if index set I equals index set J, and they are guaranteed to have the same units as well if and only if additionally v equals w. For each index set I there exists one unique dimensionless vector space which is written as $I!$, for example Good! denotes a dimensionless good vector.

The unit vector names are combined with the 'per' operator into the matrix type. A matrix that transforms vectors from the $J!w$ space to the $I!v$ space is written as a $I!v \text{ per } J!w$ matrix. A vector is a $I!v \text{ per } 1$ matrix, which is simply written as $I!v$. In the following matrix product using the 2×3 matrix from the previous section, the compile time types are shown above the runtime values. The product computes the required resources given some desired output of goods.

$$\overbrace{\begin{bmatrix} 400\,\text{hr} \\ 185\,\text{m}^2 \end{bmatrix}}^{\text{Resource!unit}} = \overbrace{\begin{bmatrix} 5\,\text{hr/kg} & 4\,\text{hr/box} & 5\,\text{hr/bag} \\ 1\,\text{m}^2/\text{kg} & 3\,\text{m}^2/\text{box} & 2\,\text{m}^2/\text{bag} \end{bmatrix}}^{\text{Resource!unit per Good!unit}} \cdot \overbrace{\begin{bmatrix} 10\,\text{kg} \\ 25\,\text{box} \\ 50\,\text{bag} \end{bmatrix}}^{\text{Good!unit}}$$

The names Good!unit and Resource!unit that the types use are unit vector names. The type Resource!unit per Good!unit denotes a matrix that transforms vectors from the Resource!unit space to the Good!unit space. The interpretation of the per operator is that it yields a matrix with the units as follows.

$$(v \text{ per } w)[i,j] = \frac{v[i]}{w[j]} \tag{3.6}$$

This interpretation is in line with Equation (3.3) and is based on Hart's theory about the structure of units of measurement in matrices. With the per operator the matrix type uses the unit vector names to describe the matrices.

The matrix type notation maps directly to dimensioned matrix shapes. The algebraic matrix type factors out scalar units just like the matrix shape. The most general matrix type is

$$a \cdot I!v \text{ per } J!w$$

It matches any matrix type. Or put differently, any other matrix type can be obtained from it by a proper substitution. If $I!v$ denotes unit vector u_0, $J!w$ denotes unit vector u_1, m is u_0's size, and n is u_1's size, then $\langle a, [m], (u_0, 1), [n], (u_1, 1)\rangle$ is the corresponding matrix shape. For the vectors and matrix from the example the mapping is as follows.

$$\begin{aligned} \text{Resource!unit} &\mapsto \langle 1, [2], (u_0, 1), [1], 1\rangle \\ \text{Good!unit} &\mapsto \langle 1, [3], (u_1, 1), [1], 1\rangle \\ \text{Resource!unit per Good!unit} &\mapsto \langle 1, [2], (u_0, 1), [3], (u_1, 1)\rangle \end{aligned}$$

The compile time unit vector names from the matrix type map to the actual unit vectors in the matrix shape.

The matrix type is more restrictive than dimensioned matrix shapes, which excludes some runtime behavior but also can improve safety. Normal matrix shapes see vectors and matrices of the same size as compatible, while they conceptually may have different index sets and not be compatible at all. Consequently, the matrix type can be mapped to a dimensioned matrix shape, but not the other way around.

The type system infers the type of any linear algebra expression. An example is the following Pacioli function to compute the area of a triangle in three dimensional geometric space. Remember from Section 2.1 that vectors in that space are indexed with the index set `Geom3`.

```
define triangle_area(v0, v1, v2) =
  norm(cross(v1 - v0, v2 - v0)) / 2;
↪
triangle_area :: for_unit a:
  (a*Geom3!, a*Geom3!, a*Geom3!) -> a^2
```

The type `a*Geom3!` consists of a scalar unit variable `a` multiplied by the dimensionless space vector which gives a homogeneous vector of unit `a`. The function expects three points in space given as space vectors and computes the area of the triangle enclosed by these points. The result is correctly derived as the square of the unit of measurement of the geometric space. So, if for example `a` is the unit metre, the result would have unit m^2.

3.5 CONCLUSION

The informal introduction of the dimensioned matrix shape shows the structure of units in matrices and tensors in dimensioned linear algebra, and their implementation in dimensioned matrices. The dimensioned matrix implementation factors out scalar units to facilitate index-free operations, and distinguishes covariant and contravariant dimensions. The matrix type has a similar structure as the matrix shape and maps directly on dimensioned matrices.

In Chapter 4 the details of the dimensioned matrices are explained. Next the matrix type is discussed in Chapter 5. Chapter 6 shows how the matrix type is implemented in the Pacioli language and discusses three elaborate examples.

4 DIMENSIONED LINEAR ALGEBRA

Dimensioned linear algebra is a unit-aware variant of linear algebra with unit-aware vectors and matrices. The dimensioned matrices defined in this chapter are an index-free implementation of dimensioned linear algebra based on Hart's dimensioned vector spaces. They serve as a reference for the static matrix type to be described Chapter 5.

4.1 UNITS OF MEASUREMENT

Traditionally units of measurement were just tags appended to numbers, but developments in physics have led to a mathematical interpretation. A mathematical treatment of units of measurement leads back via Maxwell to Fourier and even further. Birkhoff remarks in his historical a overview [Bir60] that Fourier was the first to note that there are certain fundamental units of which every physical quantity has certain "dimensions", to be written as exponents [Fou22]. Nowadays an international standard of units has been defined, called the Système International d'Unités [dPoi06], and is maintained by the Bureau International des Poids et Mesures (BIPM). Internationally this is known as The International System of Units, or SI. The document describes all aspects of the SI. For precise definitions it refers to the International vocabulary of metrology [BIP08].

The international standard of units (SI) defines the core concepts for systems of units. A *quantity* is a property of a phenomenon, body or substance, where the property has a unique magnitude that can be expressed as a number and a reference. The value of a quantity is the product of a number and a *unit*. The unit is a particular example of the quantity concerned.

Quantities are organized in a *system of dimensions*. Any quantity can be measured by many different units. The notion of a dimension abstracts from the concrete units. Units that measure the same quantity are said to have the same dimension.

Some quantities are called base quantities and all other quantities are derived from them. Typical base quantities are length, mass, time, etc. All other quantities are derived quantities, which may be written in terms of the base quantities by the equations of physics. A derived quantity's dimension follows from the equations by which it is derived.

The SI defines a base unit for each base quantity. Figure 4.1 show the dimensions and base units for the seven base quantities defined by the SI. For the use in a type system the choice between units or dimensions is mostly a matter of taste, because the relations that hold between dimensions also holds between the units in which those dimensions are measured [Ken96]. We consider units of measurement only.

Dimension	Symbol	Unit	Symbol
Length	L	metre	m
Mass	M	kilogram	kg
Time, duration	T	second	s
Electric current	I	ampere	A
Thermodynamic temperature	Θ	kelvin	K
Amount of substance	N	mole	mol
Luminous intensity	J	candela	cd

Figure 4.1: SI base units

A unit of measurement is build from base units by multiplication and division. Formally this means that units form an abelian group[1] freely generated from a set of base units. We use symbol $\mathbb{U}$ for the group of units given some base units.

Definition 1 *Let $\mathcal{B}$ be a set of base units. The units of measurement group* $(\mathbb{U}, \cdot)$ *is the abelian group freely generated from the base units $\mathcal{B}$.*

As usual division is multiplication by the reciprocal, and exponents are repeated multiplication. The group's unit 1 denotes the dimensionless numbers.

With units defined the question is how dimensioned linear algebra can be constructed from them. Intuitively a dimensioned scalar is a pair ru with $r \in \mathbb{R}$ and $u \in \mathbb{U}$. The unit in such a pair can be used to check the validity of any operation on the pair. A consequence is that dimensioned numbers cannot be directly used to construct a vector space. A vector space is defined over a field[2], but the pairs no longer form a field because the units prevents certain operations. Hart's dimensioned vector spaces discussed in the next section define dimensioned numbers properly using a typed family of fields.

4.2 Dimensioned vector spaces

To define dimensioned numbers mathematically, Hart defines a typed family of fields. The idea is to combine a field for the number's magnitude and a group for the number's units.

[1] See Appendix A.2.
[2] See Appendix A.3.

Definition 2 (Hart) *A typed family of fields (TFF) over a field F and group G is the cartesian product $F \times G$ with addition and multiplication satisfying for all $f, f_1, f_2 \in F$ and $g, g_1, g_2 \in G$*

$$\begin{aligned}(f_1, g) + (f_2, g) &= (f_1 + f_2, g) \\ (f_1, g_1) \cdot (f_2, g_2) &= (f_1 \cdot f_2, g_1 \cdot g_2)\end{aligned}$$

Type function T is defined by $T((f, g)) = g$. An element x is dimensionless if $T(x) = 1$. From the definition immediately follow the following properties.

Property 1 (Hart) *In a typed family of fields over a field F and group G*

1. *pair $(0, g)$ is a unique additive identity for all $g \in G$*
2. *pair $(1, 1)$ is the unique multiplicative identity*
3. *the operations in the TFF obey the usual commutative, associative and distributive axioms of fields, whenever the sums are defined, except that multiplication is only commutative if G is abelian.*

An element from such a family of fields is called a dimensioned scalar.

A dimensioned matrix is a matrix where the real scalars are replaces by dimensioned scalars.

Definition 3 (Hart) *An $m \times n$ dimensioned matrix is an array of dimensioned scalars. A dimensioned vector is an $n \times 1$ dimensioned matrix.*

This is the mathematical definition of dimensioned matrices on which the dimensioned matrix implementation is based. Dimensioned matrices have the same operations as regular matrices, except that the sum is undefined if any of the sums on the underlying dimensioned scalars is undefined.

Hart defines a natural equivalence relation on dimensional matrices using the unit parts.

Definition 4 (Hart) *For all matrices A and B*

$$A \sim B \Leftrightarrow \forall_{i,j}\; T(A_{ij}) = T(B_{ij}) \qquad \textit{dimensional similarity}$$

The relation equates two matrices when they have the same units in corresponding components. The equivalence relation is used to define dimensioned vector spaces.

Definition 5 (Hart) *Let v be a dimensioned vector. The* complete dimensioned vector space *of type v is the set of all w such that $v \sim w$. A* dimensioned vector space *of type v is a subset of a complete dimensioned vector space which is closed under addition and scalar multiplication by dimensionless scalars. A dimensioned vector space of type v is called a v-space.*

In dimensioned vector spaces the adjoint isn't the transpose, but the *dimensional inverse*. This unary matrix operation is used later to show that the units in a dimensioned matrix are a rank-one matrix.

Definition 6 (Hart) *The* dimensional inverse $A^{\sim}$ *of a matrix is any matrix with*

$$\forall_{i,j}\ T(A^{\sim}_{ij}) = T(A_{ji})^{-1}$$

Note that this definition says nothing about the field, it only constraints the group. Hart only uses it to state the type of a matrix. An alternative that would completely define the dimensional inverse is given by $A^{\sim}_{ij} \cdot A_{ji} = 1$. With this version the reciprocal may however not exist because of the underlying field, so care has to be taken. Either way, the dimensional inverse combines the transpose and the reciprocal. If we denote the reciprocal of a matrix by A^{-1} then it is easy to show that $A^{\sim} = (A^T)^{-1}$.

With the dimensional inverse the the form of units in a matrix can be stated. The units of measurement in a linear transformation between two spaces are completely determined by the spaces' units.

Theorem 1 (Hart) *Any matrix of a linear transformation from a w-space to a v-space has the form $vw^{\sim}$*

If v is of size m and w of size n then the form $vw^{\sim}$ is a $m \times n$ matrix. From the definition of the dimensional inverse follows that entries in this matrix have the following property.

$$(vw^{\sim})[i, j] \sim \frac{v[i]}{w[j]} \tag{4.1}$$

The units in the matrix are the units in v divided by the units w. The structure of units in a matrix is determined by the units in the spaces between which it transforms.

The form of the units in a matrix provides the structure for the dimensioned matrix implementation. The form $vw^{\sim}$ from Theorem 1 is the structure of units that was referred to in the wine bottler example in Section 1.1 from the introduction, and the relationship shown in (4.1) was illustrated for matrix shapes in equation (3.3) and for the matrix type in equation (3.6). The relationship shows how the units in a matrix are build from a contravariant part v and a co-variant part w. The entire matrix is constructed from an outer product of these two vectors. In the next section the implementation of a dimensioned matrix is structured as a typed family of fields $\mathbb{R}$ over group $\mathbb{U}$ obtained from the field of real numbers and the abelian group of units. This gives a dimensioned variant of vector space $\mathbb{R}^n$ using unit vectors from $\mathbb{U}^n$.

The dimensioned matrix implementation supports the multi-linear case by matricization of tensors. Let $\mathrm{Lin}(V, W)$ stand for the linear transformations from V to W. The multi-linear algebra involves the generalization from $\mathrm{Lin}(V, W)$ to $\mathrm{Lin}(V_1 \otimes \cdots \otimes V_k, W_1 \otimes \cdots \otimes W_l)$. Tensor product $V_1 \otimes \cdots \otimes V_k$ replaces space V and tensor product $W_1 \otimes \cdots \otimes W_l$ replaces space W, resulting in a (k, l) tensor. Instead of supporting actual tensors, the dimensioned matrix implementation matricizes them with the Kronecker product. This means Hart's form $vw^{\sim}$ for the units in a matrix now becomes

$$(v_1 \otimes \ldots \otimes v_k) \cdot (w_1 \otimes \ldots \otimes w_l)^{\sim}$$

with $\otimes$ the Kronecker product. The unit at coordinates $i_1, \ldots i_k$ in such a Kronecker product $v_1 \otimes \ldots \otimes v_k$ is the product of each dimension's unit:

$$(v_1 \otimes \ldots \otimes v_k)[i^k] = v_1[i_1] \cdots v_k[i_k]$$

The effect of matricization is that a tensor is turned into a matrix where the row dimension and the column dimension have additional structure. The crucial difference for the dimensioned matrices is that this additional structure is hidden.

The tensor structure in the dimensioned matrix implementation is encoded in the unit bases. Base units have the form (u, i) with $u \in \mathbb{U}$ and $i \in \mathbb{N}$. Number i indicates the dimension to which unit vector u applies. For example product ${u_0}^{-1} \otimes u_1 / {u_2}^2 \otimes {u_3}^3$ can be written as unit product

$$(u_0, 1)^{-1} \cdot (u_1, 2) \cdot (u_2, 2)^{-2} \cdot (u_3, 3)^3$$

This form is the basis for the units v and w in the matrix shape introduced in Section 3.2. The units at position $[i^k]$ for such bases (u, i) is given by

$$(u, n)[i^k] = u[i_n]$$

It is unit from u at position i_n, exactly what the unit in an actual tensor would be. Encoding the structure of tensors like this is makes support for tensors possible, but allows us to ignore it when evaluation rules are defined for dimensioned matrices.

4.3 Dimensioned matrices

Dimensioned linear algebra applies the customary linear algebra operators to dimensioned vector spaces. Instead of the real vectors from $\mathbb{R}^n$, it uses dimensioned vectors based on the typed family of fields $\mathbb{R}$ over group $\mathbb{U}$. To support this, matrix shapes are extended with units of measurement, and the operations are defined on this extended shape.

The operations on dimensioned matrices are primitive functions for linear algebra. The table in Figure 4.2 shows the operations for which evaluation rules will be defined.

Operation	Function	Pacioli notation	Matlab notation
Addition	*sum*(x, y)	`x + y`	`x + y`
Subtraction	*diff*(x, y)	`x - y`	`x - y`
Multiplication	*mult*(x, y)	`x * y`	`x .* y`
Division	*div*(x, y)	`x / y`	`x ./ y`
Exponent	*expt*(x, y)		`x .^ y`
Matrix product	*mmult*(x, y)	`x '*' y`	`x * y`
Matrix exponent	*mexpt*(x, y)		`x ^ y`
Scaling	*scale*(x, y)	`x '.*' y`	
Transpose	*transpose*(x)	`x^T`	`x.'`
Reciprocal	*reciprocal*(x)	`x^R`	
Dimensional inverse	*diminv*(x)	`x^D`	

Figure 4.2: Linear Algebra Operations

Besides the operation's Pacioli syntax, the Matlab syntax is also given for comparison. The infix operators are syntactic sugar that translate to a function application as shown in the table. Pacioli's naming for operators deviates from Matlab's convention. In Matlab it is the convention to prefix element-wise operators with a dot. Since the language focuses on linear algebra this makes sense. In Pacioli element-wise multiplication plays a more prominent role, and therefore the `+`, `-`, `*`, and `/` operators translate to the corresponding element-wise operators. The linear algebra operators have special symbols using an experimental language facility to add infix operators delimited by single quotes. For the exponentiation functions *expt* and *mexpt* there are no infix operators for the general case. Instead the syntax is limited to constant powers to enable better type inference. This will be explained in detail in Section 5.3.

A difference between Pacioli and Matlab is in the operators for scaling. In Matlab the $*$ operator overloads the matrix product and scaling, but the types of these two operators cannot be expressed in a single rule. Therefore Pacioli has the operator `'*'` for the matrix product and the separate operator `'.*'` for scaling. The dot indicates that the operator expects a scalar as first argument and scales the second argument with it. Another difference between operators in Matlab and Pacioli is the transpose function. As the table shows, instead of Matlab's transpose `.'`, there are operators for the transpose `^T`, the dimensional inverse `^D` and the reciprocal `^R`. These three operators form a coherent group in the context of units of measurement and therefore were given a similar syntax[1].

[1]Together with the identity operator they form a Klein four group under function composition. See Appendix A.2.

The structure of dimensioned matrices combines the extended matrix shape introduced in Section 3.2 with an array of numbers.

Definition 7 *Let $\mathcal{B}$ be a set of scalar base units and let $\mathcal{U}$ be a set of base unit vectors. A dimensioned matrix is a tuple*

$$\langle a, [m^k], v, [n^l], w, [r^p] \rangle$$

where

- *a is a unit with bases $b \in \mathcal{B}$.*
- *array $[m^k]$ with $m_1, \dots, m_k \in \mathbb{N}$ the sizes of the contravariant dimensions,*
- *v is a unit $\Pi_i v_i^{\rho_i}$ with bases $v_i \in \mathcal{U} \times \mathbb{N}$ and exponents $\rho_i \in \mathbb{Z}$.*
- *array $[n^l]$ with $n_i \in \mathbb{N}$ the sizes of the covariant dimensions,*
- *w is a unit vector $\Pi_i w_i^{\sigma_i}$ with bases $w_i \in \mathcal{U} \times \mathbb{N}$ and exponents $\sigma_i \in \mathbb{Z}$.*
- *array $[r^p]$ with $p = (\Pi_i\ m_i) \cdot (\Pi_i\ n_i)$ is an array of numbers,*

and the unit at coordinates $[x^k, y^l]$ is

$$a \times \frac{\Pi_i\ v_i[x^k]^{\rho_i}}{\Pi_i\ w_i[y^l]^{\sigma_i}}$$

This details the shape that was announced in Equation (3.3). The pair (k, l) is the matrix' order. Unit a is the scalar unit factor. Number m_i is the length of the i-th contravariant dimension, and n_i is the length of the i-th covariant dimension. For any base (u, i) in v the size of u must equal m_i and for any base (u, i) in w the size of u must equal n_i. Array $[r^p]$ stored the numbers. How the numbers are stored exactly is left unspecified. The matrix may for example be sparse and not store all numbers, but such a choice would be implementation dependent.

With the dimensioned matrix structure defined, evaluation rules for the operations from the opening of the section can be defined. Figure 4.3 gives evaluation rules for the common linear algebra operations. For all operators it is assumed that the operation on the numbers is understood.

The addition operation adds two dimensioned matrices. It requires that the two matrices have the same sizes and also the same units. The subtraction operation is similar, but it subtracts the numbers. The element-wise product requires that the matrices have the same sizes, but the units can be different. The unit of the product is the product of the units. Division is similar, except that the units get divided. Element-wise exponentiation is dimensionless. It expects a single number as second argument and raises the numbers to that power. The matrix product generalizes Equation (3.4) and adds numbers to it. Matrix exponentiation expects and produces a square dimensionless matrix. The matrix product is not overloaded for scaling. The separate scaling operator expects a scalar as first argument and multiplies it with

every element from the second argument. Scaling from the right could be added similarly.

The rule for the transpose is in line with Equation (3.5). While swapping the indices and the units, it also takes the reciprocal of the units. The rule for the reciprocal is given next. Its notation should not be confused with the inverse. The final rule is the dimensional inverse. It combines the transpose and the reciprocal as explained in Section 4.2. All these operators except for the last one, are dimensioned variants of the regular linear algebra operations.

Dimensioned matrices give rise to various operators that are specific for the extension with units of measurement. Figure 4.4 gives evaluation rules for a selection of primitive operations on matrix shapes that access the numbers and units. This set of functions can access any number or unit in a matrix and is therefore a good indication of the requirements for unit support besides the linear algebra operators. Just like for the operators from Figure 4.3 it is assumed that operations on the numbers exist.

Functions *unit_factor*, *row_unit* and *column_unit* retrieve the individual unit parts from the shape. In each case the result is a matrix of the appropriate shape and magnitude one. The result of *unit_factor* is a dimensioned scalar, the result of *row_unit* a dimensioned column vector and the result of *column_unit* a dimensioned row vector. The vectors have magnitude one for every entry.

The next four operators access various matrix parts and are not index-free. Operator *get_num* retrieves an individual number from a dimensioned matrix. The result is a dimensionless number. Operator *get_unit* retrieves an individual unit from a dimensioned matrix. The result is a dimensioned number with the asked unit and magnitude one. Functions *get_row* and *get_column* get a single row or column from a dimensioned matrix. They return a row or column vector with the right numbers and units. Since these functions access the matrix entries they are obviously not index-free. The rules show precisely where the operations refer to individual units. Operator *get_num* ignores all unit information. Operator *get_unit* is basically an implementation of Equation (3.3). The row and column function interestingly access just one of the unit vectors, which makes it kind of half index-free. But by the nature of these function it is to be expected they are not index-free.

The Kronecker product combines two dimensioned matrices into a higher ranked one, whereas the projection operator reduces a dimensioned matrix' rank. In a Kronecker product, the shape arrays are concatenated and the numbers are combined into a tensor. The $\otimes_k$ operator does the required operation on units by shifting the bases in the second argument k positions:

$$v \ \otimes_k \Pi_i(b_i, n_i)^{\rho_i} = v \times \Pi_i(b_i, n_i + k)^{\rho_i}$$

This is in line with $\otimes$'s definitions in Section 4.2. The projection operator *project* reduces a matrix' rank in the contravariant dimension, by summarizing the matrix

$$\frac{x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle, \quad y \Rightarrow \langle a, [m^k], v, [n^l], w, [s^p] \rangle}{sum(x, y) \Rightarrow \langle a, [m^k], v, [n^l], w, sum([r^p], [s^p]) \rangle}$$

$$\frac{x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle, \quad y \Rightarrow \langle a, [m^k], v, [n^l], w, [s^p] \rangle}{diff(x, y) \Rightarrow \langle a, [m^k], v, [n^l], w, diff([r^p], [s^p]) \rangle}$$

$$\frac{x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle, \quad y \Rightarrow \langle b, [m^k], x, [n^l], y, [s^p] \rangle}{mult(x, y) \Rightarrow \langle a \cdot b, [m^k], v \cdot x, [n^l], w \cdot y, mult([r^p], [s^p]) \rangle}$$

$$\frac{x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle, \quad y \Rightarrow \langle b, [m^k], x, [n^l], y, [s^p] \rangle}{div(x, y) \Rightarrow \langle a/b, [m^k], v/x, [n^l], w/y, div([r^p], [s^p]) \rangle}$$

$$\frac{x \Rightarrow \langle 1, [m^k], 1, [n^l], 1, [r^p] \rangle, \quad y \Rightarrow \langle 1, [], 1, [], 1, [i] \rangle}{expt(x, y) \Rightarrow \langle 1, [m^k], 1, [n^l], 1, expt([r^p], i) \rangle}$$

$$\frac{x \Rightarrow \langle a, [m^k], v, [o^d], x, [r^p] \rangle, \quad y \Rightarrow \langle b, [o^d], x, [n^l], w, [s^q] \rangle}{mmult(x, y) \Rightarrow \langle a \cdot b, [m^k], v, [n^l], w, mmult([r^p], [s^q]) \rangle}$$

$$\frac{x \Rightarrow \langle 1, [m^k], 1, [m^k], 1, [r^p] \rangle, \quad y \Rightarrow \langle 1, [], 1, [], 1, [i] \rangle}{mexpt(x, y) \Rightarrow \langle 1, [m^k], 1, [m^k], 1, mexpt([r^p], i) \rangle}$$

$$\frac{x \Rightarrow \langle a, [], 1, [], 1, [r] \rangle, \quad y \Rightarrow \langle b, [m^k], v, [n^l], w, [s^p] \rangle}{scale(x, y) \Rightarrow \langle a \cdot b, [m^k], v, [n^l], w, scale(r, [s^p]) \rangle}$$

$$\frac{x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle}{transpose(x) \Rightarrow \langle a, [n^l], w^{-1}, [m^k], v^{-1}, transpose([r^p]) \rangle}$$

$$\frac{x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle}{reciprocal(x) \Rightarrow \langle a^{-1}, [m^k], v^{-1}, [n^l], w^{-1}, reciprocal([r^p]) \rangle}$$

$$\frac{x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle}{diminv(x) \Rightarrow \langle a^{-1}, [n^l], w, [m^k], v, diminv([r^p]) \rangle}$$

Figure 4.3: Evaluation of dimensioned matrix operations.

$$\frac{x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle}{unit_factor(x) \Rightarrow \langle a, [], 1, [], 1, [1] \rangle}$$

$$\frac{x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle}{row_unit(x) \Rightarrow \langle 1, [m^k], v, [], 1, [1^p] \rangle}$$

$$\frac{x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle}{column_unit(x) \Rightarrow \langle 1, [n^l], w, [], 1, [1^p] \rangle}$$

$$\frac{x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle}{magnitude(x) \Rightarrow \langle 1, [m^k], 1, [n^l], 1, [r^p] \rangle}$$

$$\frac{\begin{array}{c} x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle, \\ i \Rightarrow \langle 1, [k], 1, [], 1, [i^k] \rangle, \\ j \Rightarrow \langle 1, [l], 1, [], 1, [j^l] \rangle \end{array}}{get_unit(x, i, j) \Rightarrow \langle a \cdot v[i^k]/w[j^l], [], 1, [], 1, 1 \rangle}$$

$$\frac{\begin{array}{c} x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle, \\ i \Rightarrow \langle 1, [k], 1, [], 1, [i^k] \rangle, \\ j \Rightarrow \langle 1, [l], 1, [], 1, [j^l] \rangle \end{array}}{get_num(x, i, j) \Rightarrow \langle 1, [], 1, [], 1, get_num([r^p], [i^k], [j^l]) \rangle}$$

$$\frac{\begin{array}{c} x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle, \\ i \Rightarrow \langle 1, [k], 1, [], 1, [i^k] \rangle \end{array}}{get_row(x, i) \Rightarrow \langle a \cdot v[i^k], [], 1, [n^l], w, get_row([r^p], [i^k]) \rangle}$$

$$\frac{\begin{array}{c} x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle, \\ j \Rightarrow \langle 1, [l], 1, [], 1, [j^l] \rangle \end{array}}{get_column(j, a) \Rightarrow \langle a/w[j^l], [m^k], v, [], 1, get_column([r^p], [j^l]) \rangle}$$

$$\frac{\begin{array}{c} x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle, \\ y \Rightarrow \langle b, [o^d], x, [z^e], y, [s^q] \rangle \end{array}}{kronecker(x, y) \Rightarrow \langle a \cdot b, [m^k, o^d], v \otimes_k x, [n^l, z^e], w \otimes_l y, [r^p] \otimes [s^q] \rangle}$$

$$\frac{\begin{array}{c} x \Rightarrow \langle a, [m^k], v, [n^l], w, [r^p] \rangle, \\ y \Rightarrow \langle 1, [], 1, [], 1, [i] \rangle \end{array}}{project(x, y) \Rightarrow \langle a, project([m^k], i), project(v, i), [n^l], w, project([r^p], i) \rangle}$$

Figure 4.4: Evaluation of dimensioned matrix operations concerning shape and units.

values. The summary can be the usual operations like summing, averaging, or counting. It is a requirement that the projected columns have appropriate units for the summarizing operation. For example if some dimensions are summed, the units of all the summed entries must be the same.

4.4 CONCLUSION

The structure of dimensioned matrices and the operations defined on it give a complete index-free description of dimensioned linear algebra. All operations are compliant with Hart's definition of dimensioned vector spaces. The only shortcoming is that scaling and the matrix product cannot be expressed in one rule and have to be split into two operations, but that doesn't impact the correctness of the rules. All dimensioned matrix operations can be given a correct evaluation rule. This shows that factoring out the scalar units is sufficient for an index-free description of dimensioned linear algebra.

Together with the shape and unit operations the dimensioned matrices give useful baseline for the matrix type to be defined in the next chapter. The shape and unit operations give access to the units in a dimensioned matrix. These operations will play a role when we look at the expressiveness of the matrix type.

THE MATRIX TYPE 5

The matrix type is a parametric number type that describes the shape and units of measurement of a matrix. Syntactically it is an extension of Kennedy's type for units of measurement. This algebraic form is however not directly suitable for unification, so it is translated to a form that is similar to a dimensioned matrix' shape. In this form the matrix type can easily be unified and mapped to dimensioned matrices.

5.1 THE MATRIX TYPE SYNTAX

The matrix type is an expression of the form δ per δ, where each dimension expression δ is an algebraic expression with names of units and dimensioned vector spaces as base elements. The syntax of dimension expressions δ is given in the grammar from Figure 5.1. It extends Kennedy's scalar dimension expressions with dimensioned vectors and the Kronecker product. The first dimension expression in the matrix type describes a matrix' row dimension and the second one a matrix' column dimension. Together they describe the shape of a matrix, just as the notation $\mathbb{R}^{m\times n}$ from mathematics, but refines it to dimensioned vector spaces.

The grammar describes the abelian group of units of measurement with the various extensions to the base units that was explained in the opening of Chapter 3. The unit dimension, the reciprocal and the elementwise product form the abelian group. The base dimensions B add the base units and the dimension variables d add Kennedy's unit variables. The remaining three cases are additions for the matrix type. The rule for dimensioned vectors adds the names for base unit vectors. The part on the left of the exclamation mark is the index set and the part on the right is the vector name. Both of there parts can be variable. Dimensionless vectors are a special case and have their own rule. The Kronecker product adds support for the multi-linear case.

With type expressions, a large variety of matrix types can be defined. Figure 5.2 lists some typical examples. The notation uses the convention to omit the per 1 part when the column dimension is 1. This makes the type more readable and a proper extension of the scalar case. For the vector case it is useful to distinguish dimensionless, homogeneous and heterogeneous vectors. Dimensionless and heterogeneous vectors have their own notation. A homogeneous vector type is a product of a scalar unit with a dimensionless vector. It is a special subset of all the possible matrix types that plays a role when typing indexing operations.

The Kronecker product adds support for multi-dimensional numerical data via the matricization of tensors. A (k, l) tensor has type $X_1 \otimes \cdots \otimes X_k$ per $Y_1 \otimes \cdots \otimes Y_l$ where expressions X_i type the row dimensions and expressions Y_i type the column

δ	$::= 1$	unit dimension
	$\mid B$	base dimension
	$\mid d$	dimension variable
	$\mid \delta^{-1}$	reciprocal
	$\mid \delta \times \delta$	element-wise product
	$\mid \xi!\mu$	dimensioned vector
	$\mid \xi!$	dimensionless vector
	$\mid \delta \otimes \delta$	Kronecker product (δ without variables)
ξ	$::= I$	index set
	$\mid p$	index variable
μ	$::= U$	base unit vector
	$\mid d$	dimension variable

Figure 5.1: Syntax of Kennedy's dimension expressions, extended with the $\xi!\mu$ notation for dimensioned vector spaces. The ξ part denotes an index set and the μ part a particular space.

dimensions. In type expressions the Kronecker product separates the different matrix dimensions. For example the $(3, 0)$ tensor from Section 3.3 is typed as Region! $\otimes$ Year! $\otimes$ Commodity!unit. With matricization, tensors are turned into matrices with compound indices. This hides the extension from the matrix type and gives only the multi-dimensional data structure of tensors, not the tensor calculus. This is achieved by disallowing variables in Kronecker products. So expressions like Region! $\otimes$ P $\otimes$ Commodity!unit with P a variable is not allowed. Unification variables can only match the whole row and column dimensions, and not parts of them. The matrix type is unsuitable for the varying number of indices in tensor calculus, but via matricization multi-dimensional data can be handled.

5.2 TYPE INFERENCE

The type inference for the matrix type from the next section requires a unification algorithm for matrices. This section explains Kennedy's unification algorithm for units of measurement and how it is used in the unification of the matrix type.

Unification for units of measurement requires the notion of dimension equivalence and a normal form, because it is a semantic unification problem. Kennedy defines

Type	Description	Full type
1	dimensionless number	1 per 1
a	dimensioned number	a per 1
$I!$	dimensionless vector	$I!$ per 1
$I!v$	dimensioned vector	$I!v$ per 1
$a \cdot I!$	homogeneous dimensioned vector	$a \cdot I!$ per 1
1 per $J!w$	dimensioned row vector	
$I!$ per $J!$	dimensionless rectangular matrix	
$I!v$ per $I!w$	square matrix	
$I!v$ per $I!v$	even more square matrix	
$I!v^2$ per $I!v^2$	matrix with squared untis	
...	...	

Figure 5.2: Some examples of matrices that can be typed with the matrix type. The type can indicate a scalar, a vector or a matrix. The full type is shown to illustrate how the type allows the omission of an empty column dimension.

equivalence relation $=_D$ as the congruence on dimension expressions generated by the following equations:

$$\begin{aligned} \delta_1 \times \delta_2 &=_D \delta_2 \times \delta_1 && \text{commutativity} \\ (\delta_1 \times \delta_2) \times \delta_3 &=_D \delta_1 \times (\delta_2 \times \delta_3) && \text{associativity} \\ 1 \times \delta &=_D \delta && \text{identity} \\ \delta \times \delta^{-1} &=_D 1 && \text{inverses} \end{aligned}$$

This equivalence gives the abelian group required for units of measurement. A dimension expression is normalized if it is of the form $d_1^{x_1} \cdots d_m^{x_m} \cdot B_1^{y_1} \cdots B_n^{y_n}$. This assumes some ordering on the unit variables d and base units B. The normal form of some dimension expression μ is written as $nf(\mu)$.

Kennedy's unification algorithm uses the normal form in the computation of the most general unifier for two units. See Figure 5.3. The algorithm is based on Knuth's adaptation of Euclid's greatest common divisor algorithm. For two elements from the free abelian group of units it computes the most general unifier, or it fails if no such unifier exists. The algorithm is sound and complete.

The development of the matrix type depends on the property that the base units in $\mathcal{B}$ can have additional structure as long as the group remains free. Base elements can be compound structures, for example the combination of a prefix and a base-unit, but crucial is that variables only identify such a compound, not an operator or any substructure in the compound. For example a kilogram is an entity on its own and for the unification algorithm it doesn't matter that it is a pair, as long as equality

$$
\begin{aligned}
&\text{UnifyUnits}(\delta_0, \delta_1) = \text{UnitUnify}(\delta_0 \cdot \delta_1^{-1}) \\
\\
&\text{UnitUnify}(\delta) = \\
&\quad \text{let nf}(\delta) = d_1^{x_1} \cdots d_m^{x_m} \cdot B_1^{y_1} \cdots B_n^{y_n} \\
&\quad \text{where } |x_1| \leq \cdots \leq |x_m| \\
&\quad \text{in} \\
&\qquad \text{if } m = 0 \text{ and } n = 0 \text{ then } \{\} \\
&\qquad \text{if } m = 0 \text{ and } n \neq 0 \text{ then fail} \\
&\qquad \text{else if } m = 1 \text{ and } x_1 | y_i \text{ for all } i \text{ then} \\
&\qquad\quad \{d_1 \mapsto B_1^{-y_1/x_1} \cdots B_n^{-y_n/x_1}\} \\
&\qquad \text{else if } m = 1 \text{ then fail} \\
&\qquad \text{else } S \circ U \text{ where} \\
&\qquad\quad U = \{d_1 \mapsto d_1 \cdot d_2^{-\lfloor x_2/x_1 \rfloor} \cdots d_m^{-\lfloor x_m/x_1 \rfloor} \cdot \\
&\qquad\qquad\qquad\qquad \cdot B_1^{-\lfloor y_1/x_1 \rfloor} \cdots B_n^{-\lfloor y_n/x_1 \rfloor}\} \\
&\qquad\quad S = \text{UnitUnify}(U(\delta))
\end{aligned}
$$

Figure 5.3: Kennedy's unification algorithm for units of measurement.

is defined. There is a still a unique representation in the base elements, the group remains free and the unification algorithm can be applied.

The unification of the matrix also requires a normal form, and additionally requires breaking down the compound identifiers in the $\xi!\mu$ notation. The main operators in the syntax are the normal element-wise product and the Kronecker product. The Kronecker product forms the dimensions while the element-wise product is used within each dimensions. Therefore the row and the column dimension of the matrix type are arranged into Kronecker products of element-wise products. If furthermore multi-linearity is used to move all constants to the front then any matrix type can be written as

$$a_1^{x_1} \cdots a_s^{x_s} \cdot \bigotimes_{1 \leq i \leq k} \prod_{1 \leq j \leq m_i} I_i!v_{ij}^{y_{ij}} \text{ per } \bigotimes_{1 \leq i \leq l} \prod_{1 \leq j \leq n_i} J_i!w_{ij}^{z_{ij}}$$

with m_i the number of bases in the units of the i-th row dimension and with n_i the number of bases in the units of the i-th column dimension. Essentially this form is the $\text{Lin}(V, W)$ space of linear maps $V \to W$. This form is still not suitable for unification

because the combination of two identifiers in the $\xi!\mu$ notation is not allowed in the abelian group's base. Instead the following isomorphic tuple form is chosen:

$$\langle a_1^{x_1} \cdots a_s^{x_s},\ [I_1, \ldots, J_k],\ v,\ [J_1, \ldots, J_l],\ w \rangle$$

with

$$v = \prod_{1 \le i \le k} \prod_{1 \le j \le m_i} (u_{ij}, i)^{y_{ij}} \text{ and } w = \prod_{1 \le i \le l} \prod_{1 \le j \le n_i} (v_{ij}, i)^{z_{ij}}$$

Units v and w contain the unit vectors paired with an integer that indicates to which dimension it belongs. Since the index sets I_i and J_i must be the same within each dimension i because it is an element-wise product, it follows that the pairing of the units vectors is sufficient to reconstruct the matrix type from the tuple. In this form the index sets are separated from the vector space identifiers. The remaining expressions in the first, the third and the fifth position are dimension expressions in the Kennedy's original form and suitable for unit unification.

The tuple form of the matrix type is almost the form of dimensioned matrices. If the index sets are replaced by their sizes then the forms are the same. This is the mapping from matrix types to dimensioned matrices that was discussed in Section 3.4.

Unification for the matrix type is defined component-wise on the tuple form. See Figure 5.4. The three cases with dimension expressions are unified with the unification algorithm for units, the other two with Hindley-Damas-Milner style unification [DM82]. The correctness of this matrix type unification follows directly from the correctness of unit and standard unification.

Theorem 2 (Kennedy) *For any δ the algorithm UnitUnify(δ) terminates with failure or with a substitution S, and:*

- *If UnifUnify(δ) = S then $S(\delta) =_D 1$*
- *If $S'(\delta) = 1$ then UnitUnify(δ) = S such that $S' =_D R \circ S$ for some substitution R.*

Corollary 1 *For every matrix type τ_0 and τ_1 the unification algorithm UnifyMatrices(τ_0, τ_1) terminates with failure or with a substitution S satisfying $S(\tau_0) =_D S(\tau_1)$*

Proof 1 *Since the dimension expressions in a normalized matrix type are reduced to Kennedy's original syntax, it follows that unit unification is applicable to these three cases. The other two cases are the index sets. These are just variables or identifiers and can trivially be unified with standard syntactic unification. Since unification is composable the result follows from each of the five unification occurrences in Algorithm 5.4.*

With the unification algorithm for matrices we can define type inference for linear algebra. The type inference is an extension of Hindley-Damas-Milner style unification. As usual we start with the definition of types and expressions. A type schema

$$
\begin{aligned}
&\text{UnifyMatrices}(\tau, \tau') = S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1\\
&\quad\text{with}\\
&\qquad nf(\tau) = \langle a, I, v, J, w\rangle\\
&\qquad nf(\tau') = \langle a', I', v', J', w'\rangle\\
&\quad\text{and}\\
&\qquad S_1 = \text{UnifyUnits}(a, a')\\
&\qquad S_2 = \text{Unify}(I, I')\\
&\qquad S_3 = \text{Unify}(S_2(J), S_2(J'))\\
&\qquad S_4 = \text{UnifyUnits}(v, v')\\
&\qquad S_5 = \text{UnifyUnits}(S_4(w), S_4(w'))
\end{aligned}
$$

Figure 5.4: Unification for matrices. The unit parts are unified by Kennedy's semantic UnifyUnits algorithm, the index set parts by standard syntactic unification. The result is the composition of the partial results.

introduces quantified variables. Index set variables p are added to the unit variables d and the normal type variables t.

$\sigma ::= \tau$	types
$\mid \forall t.\sigma$	type quantification
$\mid \forall d.\sigma$	dimension quantification
$\mid \forall p.\sigma$	index set quantification

$\tau ::= t$	type variable
$\mid \tau \to \tau$	function
$\mid \delta \text{ per } \delta$	matrix type

A type τ is a variable, a function, or a matrix type. The δ in the matrix type is given in Figure 5.1. In a practical setting other types could be added.

The syntax for expressions e is exactly the same as Damas' syntax. Lambda abstraction and function application are the core primitives.

$e ::= x$	variable
$\mid \lambda x.e$	abstraction
$\mid e\ e$	application
$\mid \text{let } x = e \text{ in}$	let binding

As usual the let binding introduces type variables.

A context Γ contains type statements of the form $x : \sigma$. Type statement $\Gamma \vdash e : \tau$ means expression e has type τ given the types in context Γ. The following derivation rules are defined.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \text{(Var)}$$

$$\frac{\Gamma \cup \{x : \tau\} \vdash e : \tau'}{\Gamma \vdash (\lambda x.e) : \tau \rightarrow \tau'} \quad \text{(Abs)}$$

$$\frac{\Gamma \vdash e : \tau' \rightarrow \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e\ e') : \tau} \quad \text{(App)}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \cup \{x : \sigma\} \vdash e' : \tau}{\Gamma \vdash (\text{let } x = e \text{ in } e') : \tau} \quad \text{(Let)}$$

Next a rule for the index set variables is added to the generalization and instantiation rules for type variables and units variables.

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall t.\sigma} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall u.\sigma} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall p.\sigma} \quad \text{(Gen)}$$

$$\frac{\Gamma \vdash e : \forall t.\sigma}{\Gamma \vdash e : \sigma[t \rightarrow \tau]} \qquad \frac{\Gamma \vdash e : \forall u.\sigma}{\Gamma \vdash e : \sigma[u \rightarrow \mu]} \qquad \frac{\Gamma \vdash e : \forall p.\sigma}{\Gamma \vdash e : \sigma[p \rightarrow \varepsilon]} \quad \text{(Inst)}$$

These rules handle the three kinds of variables.

Finally type rules for the linear algebra operators are defined. Figure 5.5 gives type rules for common linear algebra operations. The type rules correspond one-to-one with the evaluation rules from Figure 4.3. Type rules for shape and unit operations are given in Figure 5.6. These rules corresponds with the evaluation rules from Figure 4.4.

5.3 EXPONENTIATION

The rules for exponentiation are without units because exponentiation is generally dimensionless, but with some syntactic sugar unit support is possible for constant powers. Constant powers like squares and cubes are quite common. If an infix operator for exponentiation would tranlate to the corresponding functions as the other operators in the table from Figure 4.2, then expressions involving constant powers would also be dimensionless. Therefore the syntax for infix exponentiation operators is restricted to constant integers. The general case is still available via a function call, but for constant integers the infix operators allow improved type inference.

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } J!w, \quad \Gamma \vdash y : a \cdot I!v \text{ per } J!w}{\Gamma \vdash \mathit{sum}(x, y) : a \cdot I!v \text{ per } J!w}$$

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } J!w, \quad \Gamma \vdash y : a \cdot I!v \text{ per } J!w}{\Gamma \vdash \mathit{diff}(x, y) : a \cdot I!v \text{ per } J!w}$$

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } J!w, \quad \Gamma \vdash y : b \cdot I!x \text{ per } J!y}{\Gamma \vdash \mathit{mult}(x, y) : a \cdot b \cdot I!v \cdot I!x \text{ per } J!w \cdot J!y}$$

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } J!w, \quad \Gamma \vdash y : b \cdot I!x \text{ per } J!y}{\Gamma \vdash \mathit{div}(x, y) : a \cdot I!v / I!x \text{ per } b \cdot J!w / J!y}$$

$$\frac{\Gamma \vdash x : I! \text{ per } J!, \quad \Gamma \vdash y : 1}{\Gamma \vdash \mathit{expt}(x, y) : I! \text{ per } J!}$$

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } K!x, \quad \Gamma \vdash y : b \cdot K!x \text{ per } J!w}{\Gamma \vdash \mathit{mmult}(x, y) : a \cdot b \cdot I!v \text{ per } J!w}$$

$$\frac{\Gamma \vdash x : I! \text{ per } I!, \quad \Gamma \vdash y : 1}{\Gamma \vdash \mathit{mexpt}(x, y) : I! \text{ per } I!}$$

$$\frac{\Gamma \vdash x : a, \quad \Gamma \vdash y : b \cdot I!v \text{ per } J!w}{\Gamma \vdash \mathit{scale}(x, y) : a \cdot b \cdot I!v \text{ per } J!w}$$

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } J!w}{\Gamma \vdash \mathit{transpose}(x) : a \cdot J!w^{-1} \text{ per } I!v^{-1}}$$

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } J!w}{\Gamma \vdash \mathit{reciprocal}(x) : a^{-1} \cdot I!v^{-1} \text{ per } J!w^{-1}}$$

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } J!w}{\Gamma \vdash \mathit{diminv}(x) : a^{-1} \cdot J!w \text{ per } I!v}$$

Figure 5.5: Types rules for linear algebra operations.

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } J!w}{\Gamma \vdash \mathit{unit_factor}(x) : a}$$

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } J!w}{\Gamma \vdash \mathit{row_unit}(x) : I!v}$$

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } J!w}{\Gamma \vdash \mathit{column_unit}(x) : J!w}$$

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } J!w}{\Gamma \vdash \mathit{magnitude}(x) : I! \text{ per } J!}$$

$$\frac{\Gamma \vdash x : a \cdot I! \text{ per } J!, \quad \Gamma \vdash i : I, \quad \Gamma \vdash j : J}{\Gamma \vdash \mathit{get_unit}(x, i, j) : a}$$

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } J!w, \quad \Gamma \vdash i : I, \quad \Gamma \vdash j : J}{\Gamma \vdash \mathit{get_num}(x, i, j) : 1}$$

$$\frac{\Gamma \vdash x : a \cdot I! \text{ per } J!w \quad \Gamma \vdash i : I}{\Gamma \vdash \mathit{get_row}(x, i) : a \text{ per } J!w}$$

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } J!, \quad \Gamma \vdash j : J}{\Gamma \vdash \mathit{get_column}(x, j) : a \cdot I!v}$$

$$\frac{\Gamma \vdash x : a \cdot I_1!v_1 \otimes \ldots \otimes I_y! \otimes \ldots \otimes I_k!v_k \text{ per } J!w, \quad \Gamma \vdash y : 1}{\Gamma \vdash \mathit{project}(x, y) : a \cdot I_1!v_1 \otimes \ldots \otimes I_k!v_k \text{ per } J!w}$$

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } J!w, \quad \Gamma \vdash y : b \cdot K!x \text{ per } L!y}{\Gamma \vdash \mathit{kronecker}(x, y) : a \cdot b \cdot I!v \otimes K!x \text{ per } J!w \otimes L!y}$$

Figure 5.6: Types rules for shape and unit operations.

In general the fact that the exponentiation is dimensionless is mitigated by the ability to define static powers like the square and cube functions. For example the square function is simply the product of its argument with itself and can be typed correctly because the exponent is known at compile time. The next code fragment defines a square function for the element-wise and the matrix product.

```
define elt_square(x) = x * x;
↪
elt_square :: for_index I, J: for_unit a, I!v, J!w:
  (a*I!v per J!w) -> a^2*I!v^2 per J!w^2

define mat_square(x) = x '*' x;
↪
mat_square :: for_index I: for_unit a, I!v:
  (a*I!v per I!v) -> a^2*I!v per I!v
```

In the inferred types all units are squared as they are supposed to be. Similarly a cube function can be defined and so on. The only limitation is that the type cannot express a rule for arbitrary exponents. A variable exponent cannot be handled by static typing.

To improve unit inference for expressions with constant powers Pacioli has the operators `^` and `'^'` and translates them to a form that is equivalent to a repeated product. In this way the operation gets typed by the regular product operations, just as in the example for squares above. It is as if the following rules are added to the language.

$$\frac{\Gamma \vdash x : a \cdot I!v \text{ per } J!w, \quad n \in \mathbb{Z}}{\Gamma \vdash \texttt{x\,\char`\^\,n} : a^n \cdot I!v^n \text{ per } J!w^n} \qquad \frac{\Gamma \vdash x : a \cdot I!v \text{ per } I!v, \quad n \in \mathbb{Z}}{\Gamma \vdash \texttt{x\,'\char`\^'\,n} : a^n \cdot I!v \text{ per } I!v}$$

These rules are in line with the square functions above. The rule on the left is element-wise exponentiation. It expects a single number as second argument and raises the numbers and the units to that power. The rule on the right types matrix exponentiation. It operates on square matrices with equal row and column units. When a equals 1 the case corresponds with a closed matrix product and that is exactly what the type describes.

A fun example from Kennedy that can be expressed conveniently with the exponentiation operators is the function $f(x, y, z) = x^2 + y^5 + z^6$. The unit unification algorithm figures out all the correct powers.

```
define f(x, y, z) = 1*x^2 + y^5 + z^6;
↪
f :: for_unit a: (a^15, a^6, a^5) -> a^30;
```

The multiplication with 1 forces the type to singleton numbers to make it comparable with Kennedy's outcome.

The matrix variants of the example handle the units correctly. By removing the multiplication with 1 the types become full matrices.

```
define g(x, y, z) = x^2 + y^5 + z^6;
↪
g :: for_index I, J: for_unit a, I!u, J!v:
  (a^15*I!u^15 per J!v^15,
    a^6*I!u^6  per J!v^6,
    a^5*I!u^5  per J!v^5) -> a^30*I!u^30 per J!v^30;

define h(x, y, z) = x'^'2 + y'^'5 + z'^'6;
↪
h :: for_index I: for_unit a, I!u:
  (a^15*I!u per I!u,
    a^6*I!u per I!u,
    a^5*I!u per I!u) -> a^30*I!u per I!u;
```

In the element wise exponent the units of all elements get raised to the required power. In the matrix exponent it is derived correctly that it is a square matrix. All the powers have correct values. Without the special syntax for constant powers these functions would be dimensionless.

5.4 EXPRESSIVENESS

Comparing the type rules from Figure 5.5 with the evaluation rules from Figure 4.3 shows that the types of the linear algebra operators match the evaluation rules exactly. All type rules are correct and completely general.

For the shape and units operations that access individual matrix elements the type rules fall back to homogeneous units of measurement. If we compare the type rules from Figure 5.6 with the evaluation rules from Figure 4.4 we see that all rule parts that are indexed are restricted to homogeneous units. For the *get_num* operation this makes no difference since it disregards the unit. The *get_unit* operation is restricted to the homogeneous matrix type $a \cdot I! \text{ per } J!$ where each matrix element has unit a. The *get_row* and *get_column* functions are just homogeneous in one dimension. The result of *get_row* can be heterogeneous, as long as the units are the same for each row. In type $a \cdot I! \text{ per } J!w$ this is indeed the case. Similarly for *get_column* each column must have the same units. Since the type system relies on index-free, these four functions that access matrix elements cannot support heterogeneous units of measurement.

The remaining four functions show that the factorization of a matrix into an element-wise product of a magnitude matrix and a unit matrix can be typed correctly, giving an escape mechanism for cases where support for heterogeneous units is not possible. The discussion so far shows that when indexing is used the type system has to fall back on homogeneous units and full support for heterogeneous units is not possible. In these cases the type system can be circumvented by factoring a matrix into an element-wise product of its magnitude and its units. Functions *unit_factor*, *row_unit* and *column_unit* give all three unit components of the matrix type. When they are combined into function `unit` as explained in Section 4.3 the type system derives the correct type.

```
define unit(x) = unit_factor(x) '.*' row_unit(x) '*' column_unit(x)^D;
↪
unit :: for_index I, J: for_unit a, I!v, J!w:
  (a*I!v per J!w) -> a*I!v per J!w
```

For any matrix `x` the unit matrix `unit(x)` has the same units as `x` and value 1 everywhere. Together with function *magnitude* it can factor any matrix into its magnitude and unit. The next property is true by definition.

```
define tautology(x) = x = magnitude(x) * unit(x);
↪
tautology :: for_index I, J: for_unit a, I!u, J!v:
  (a*I!u per J!v) -> Boole()
```

The tautology shows that any matrix is the product of a magnitude and a unit matrix.

With the `unit` function a unit-aware version of the successor function from Section 2.1 can be defined. It simply adds a unit to its argument.

```
define succ(x) = x + unit(x);
↪
for_index I, J: for_unit a, I!v, J!w ::
  (a * I!v per J!w) -> a * I!v per J!w
```

The unit-ware version of function `succ` expects a matrix of any type and returns a matrix of the same type.

The algebraic properties of the matrix type can lead to subtle results. Mistakes do for example not always lead to rejection of code because there might be special cases for which the code is safe. If a most general type exists the type system will infer it. Consider the following incorrect inner product between a vector and its transpose.

```
define f(x) = inner(x^T, x)     # Incorrect?!
↪
f :: for_unit a: (a) -> a^2
```

An inner product is between two vectors of the same shape, so it may seem that the function is wrong because arguments `x^T` and `x` are certainly of different shape. However, `x` and its transpose are of the same shape for scalars. In that case the inner product is indeed safe and the function reduces to squaring of scalars. The reasoning can become quite elaborate, but the type system will always infer the most general type.

5.5 CONCLUSION

The extension of Kennedy's syntax for unit expression with unit vectors and the Kronecker product makes the matrix type sufficiently expressive to state effective inference rules for dimensioned linear algebra. With just these two extensions the matrix type can infer the most general type for linear algebra expressions. The inclusion of an index set in the naming of unit vectors gives it enough information to specify the units of measurement in any scalar, vector, matrix or tensor. It can distinguish dimensionless matrices, matrices with homogeneous units and matrices with heterogeneous units. The extension with the Kronecker product adds tensors and generalizes the type from the linear to the multi-linear case. Finally some syntactic sugar allows units of measurement for exponentiation with constant integer exponents.

The shape and unit operations give access to all the unit elements in a dimensioned matrix. Since these operators access individual elements they cannot be index-free and have to fall back to homogeneous units of measurement. However, any dimensioned matrix can be factored into an element-wise product of a dimensionless magnitude matrix and a unit matrix. This gives an escape mechanism for cases where support for heterogeneous units is not possible.

PACIOLI: IMPLEMENTATION AND EXAMPLES 6

The Pacioli implementation of the matrix type is a proof of concept for a unit-aware matrix language. Implementing it for various environments gave valuable insight in details of the matrix type and the required support from the runtime system. This chapter discusses Pacioli's implementation and three cases that illustrate different aspects of unit-aware matrix programming.

6.1 implementation in pacioli

Besides the implementation of the matrix type in Pacioli's compiler, the implementation of unit-aware matrices also requires support at runtime. The matrix type fits well in a parametric polymorphic type system and besides the matrix type the compiler is fairly standard. The matrix type is a sizable but localized extension of the type system. The effect on the runtime system is more substantial. The runtime needs to implement primitive functions and data types, conversion factors for units of measurement, unit symbols, etc. Support for unit-aware matrices requires implementing mechanisms to fetch data like index sets and unit vectors dynamically.

An implementation can be unit-aware in various degrees without impacting the actual numerical computation or type correctness. Dimensioned matrices have all the information for a completely unit-aware implementation. The shape contains all information needed to read and write in proper units of measurement, check the unit correctness of computations, do conversions, etc., but the extra bookkeeping results in some performance overhead. The matrix type catches errors early, so when the units are already shown correct at compile time safety checks can be omitted, and correctness is still guaranteed. But the matrix type can even improve safety without any runtime support for units. In that case input and output with units of measurement is impossible, but the computed numbers are the same as when the units would be attached. This implementation support is the lowest possible level of unit support and experimented with in the Matlab environment.

Pacioli programs are compiled to a high-level intermediate language for a unit-aware run time system called the Matrix Virtual Machine (MVM) that is part of the implementation, or they can be compiled to Matlab or JavaScript. The compiler and the MVM are completely written in Java. The compiler translates a Pacioli program into target code in several phases. See Figure 6.1. Each module of the source program is parsed into an abstract syntax tree. After semantic analysis of this tree, the types of the definitions are inferred. If this is successful the tree is transformed into an intermediate representation. The back end of the compiler can generate code for several target languages from this representation. After code generation the compiled

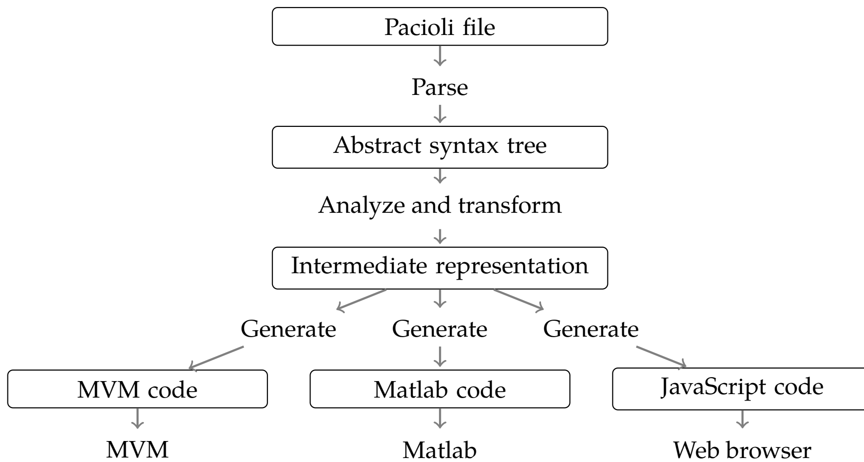


Figure 6.1: Architecture of the Pacioli implementation. A Pacioli program can be compiled to the MVM, to Matlab or to JavaScript.

modules are linked and the overall outcome of all compiler phases together is an executable form of the program. Each of the three targets implement a different runtime environment that can be targeted from the same Pacioli source.

The Matrix Virtual Machine (MVM) is Pacioli's own light-weight unit aware runtime system. It interprets a language that maps by design exactly on the compiler's internal representation. For the numbers it uses an existing matrix implementation from a library and tags any matrix instance with its shape. The shape provides all information to determine a matrix' units and do all the tasks necessary for a unit-aware runtime system. Since it is completely unit-aware it can support dynamically typed languages and statically typed languages, but the combination of units and numbers in dimensioned matrices leads to constant boxing and unboxing. The MVM is a proof of concept without any optimizations that provides an environment to experiment with unit-aware language features and computations.

The Matlab runtime supports all required primitives, but without units of measurement. The generated Matlab code is not unit-aware but since the type system already guarantees unit correct operations it is still unit correct. Any computation will produce the same numbers, the only difference is that the units symbols are missing from the output. Obviously in this case there is no overhead from the units of measurement. The only critical runtime requirement for correct numbers in a runtime with minimal support like in the Matlab case is the computation of conversion factors for units that are not yet known at compile time. Conversion factors are critical because they impact the correctness of the numerical values. Every conversion needs

to determine the multiplication factor between the units involved. If the units are known at compile time then conversion is just multiplication with a constant. If the units in a conversion are not known at compile time, then the runtime system must determine the proper conversion factors dynamically. Other than that the Matlab implementation does not require any unit-aware features for correct computations.

The JavaScript language does not support matrices or units of measurement so both are implemented in order to fully support Pacioli's primitive functions and data types. Pacioli's JavaScript runtime library allows Pacioli scripts to be run in the browser. Vectors and matrices can be displayed in a web page with all numbers annotated with units of measurement. Furthermore, graphical representations like charts and 3D models are supported, all with automatic support for units of measurement where applicable.

In the JavaScript implementation the unit computation is completely decoupled from the numeric computation and reuses the types that were derived by the compiler. This reduces the overhead to an insignificant fraction of the total computation.

6.2 LANGUAGE FEATURES FOR LINEAR ALGEBRA

The Pacioli language implements all linear algebra operators from the previous chapters. The matrix type is part of its type system and the compiler supports the matrix unification algorithm.

A language feature that is specific for the matrix type is the notation for unit literals. It was already explained that unit expressions are distinguished from ordinary language expressions by surrounding them with the `|` symbol like `|metre|` or `|newton/metre^2|`. For example, expression `26.22*|mile|` evaluates to a distance in miles and this value will be printed like `26.22 mi`. The unit expression between the `|` delimiters cannot only be a scalar unit, but any matrix type expression. For example expression `|Resource!unit per Good!unit|` creates a rectangular matrix with value 1 everywhere. The unit notation gives access to the unit name-space and automatically acts as a constructor for unit matrices.

With language primitive `defmatrix` constant matrices can be defined. The keyword is followed by the matrix' name and type and a comma-separated list for the matrix entries. For example the following fragment defines marix `foo`.

```
defmatrix foo :: Geom3! = {
  x -> 1,
  y -> 2,
  z -> 3
};
```

The matrix has type `Geom3!` and values 1, 2 and 3.

The indices of a matrix can be obtained dynamically with functions `row_domain` and `column_domain`. For any matrix value these functions return the matrix' index sets. Remember from Section 2.1 that index set `Geom3` is defined as `{x, y, z}`. This set can be obtained from any vector of that type. For example the call `row_domain(|Geom3!|)` evaluates to the list `[Geom3@x, Geom3@y, Geom3@z]`. Together with the literal notation the two functions give access to any matrix' indices.

Base vectors created with the following `delta` functions are a convenient way to work with indices without leaving the index-free world. The Kronecker delta is common idiom in linear algebra. The `delta` function creates a delta vector.

```
delta :: for_index I: (I) -> I!
```

Given an index the function creates a vector with the proper index set with value 1 at the index position and 0 everywhere else. For example, the geometry library uses `delta(Geom@x)` as a base vector for the x position.

```
print delta(Geom3@x);
↪
Geom3     Value
---------------
x             1
y             0
z             0
```

The vector is a dimensionless space vector of type `Geom3!`. Bases for the y and z coordinates are constructed similarly. With these vectors any vector can be constructed with index-free code. Function `vector3d` uses them to construct a `Geom3` vector.

```
define vector3d(x, y, z) =
  x '.*' delta(Geom3@x) + y '.*' delta(Geom3@y) + z '.*' delta(Geom3@z);
↪
vector3d :: for_unit a: (a, a, a) -> a*Geom3!
```

Arguments `x`, `y` and `z` are scalars of unit `a` and the result is a homogeneous space vector of that unit. The function creates a vector from the three scalar values without any indexing. A more elaborate example of index-free coding combined with scalar units is the electrical network case from the next section.

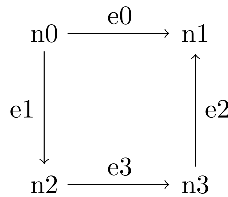
6.3 CASE STUDIES

Three case studies highlight different aspects of the matrix type. The first case is the computation of the equilibrium in an electrical network. It demonstrates the matrix

type's scalar unit factor and its use in homogeneous unit vectors. The second case is the computation of an integer null space of a matrix with the Fourier-Motzkin algorithm. This a typical example where the dual use of matrices as linear transformations and as general data structure is rejected by the matrix type. The last case looks at the explosion problem in a bill of material matrix. This case demonstrates the use of dimensioned vector spaces in the matrix type. Besides demonstrating Pacioli features, the computation is also relevant for the value nets from part II.

6.3.1 *Case: Electrical Network*

An example that demonstrates the matrix type's scalar unit factor and its use in homogeneous unit vectors is the computation of the equilibrium in an electrical network. This typical example of solving a set of linear equations shows how the matrix type helps understand the computations. The problem comes from one of Strang's textbooks [Str88] and is a classical application of linear algebra:

> A network consists of nodes $\{n_0, n_1, n_2, n_3\}$ and edges $\{e_0, e_1, e_2, e_3\}$ connected as follows
>
> 
>
>
> The conductivity on the edges is 1, 2, 2 and 1 A/V. What are the equilibrium potentials and currents given battery b_i on the edges and external currents f_i that flow into the nodes? Node n3 is the ground node. Solve the case when the inflow on nodes n_0 and n_1 is 1, the inflow on node n_2 is 6, and there is no battery.

The problem is solved with Kirchhoff and Ohm's equilibrium equations. Let unknown x contain potentials at the nodes, let unknown y contain currents at the edges, let matrix A be the incidence matrix and let diagonal matrix C the conductance. The equilibrium equations are:

$$y = C(b - Ax) \quad \text{and} \quad A^T y = f$$

Rearranging gives that the equation to solve for x is $A^T CAx = A^T Cb - f$. Key to the solution for x are matrices $A^T CA$ and $A^T C$. The value for y then follows from the substitution of x's value. The solution is unique if one of the nodes is picked as

ground node and set to zero. This is achieved by removing the corresponding column from incidence matrix A.

The first step in the implementation of the solution is to model the network with an incidence matrix and conductance vector. Language primitive `defindex` is used to create index sets for the nodes and edges. The electrical network is modeled with matrix `incidence` and vector `conductance`.

```
defindex Node = {n0, n1, n2, n3};

defindex Edge = {e0, e1, e2, e3};

defmatrix incidence :: Edge! per Node! = {
  e0, n0 -> -1,  e0, n1 -> 1,
  e1, n0 -> -1,  e1, n2 -> 1,
  e2, n3 -> -1,  e2, n1 -> 1,
  e3, n2 -> -1,  e3, n3 -> 1
};

defmatrix conductance :: ampere/volt*Edge! = {
  e0 -> 1,
  e1 -> 2,
  e2 -> 2,
  e3 -> 1
};
```

The types express the dimensioned vector spaces of the two values.

The network is grounded by setting the column of the ground node to zero instead of removing it. Such masking is in general preferred over removing rows or columns because it leaves the index sets unchanged. This solution does however depend on the solver to calculate a least-norm solution because the matrix has one column too many, but it gives the same answer. Setting the column to zero is done by multiplying the incidence matrix with the proper diagonal matrix.

```
define grounded_incidence =
  incidence '*' diagonal(|Node!| - delta(Node@n3));
↪
grounded_incidence :: Edge! per Node!
```

Vector `|Node!| - delta(Node@n3)` contains a 1 for all nodes except for the ground node `n3`. Function `diagonal` from Pacioli's standard library creates a diagonal matrix from a vector. It has type `(a·I!v) -> a·I!v per I!v`. This diagonal matrix filters the column for the ground node.

Next the matrices $A^T C A$ and $A^T C$ from the solution are computed.

```
define matrix1 =
  grounded_incidence^T * diagonal(conductance);
↪
matrix1 :: ampere/volt*Node! per Edge!

define matrix2 =
  matrix1 * incidence;
↪
matrix2 :: ampere/volt*Node! per Node!
```

The types are of the right shape and give helpful insight into the transformations of the matrices. For example, multiplying matrix1 with a `volt*Edge!` vector yields an `ampere*Node!` vector, so it transforms between these spaces. Similarly we see that matrix2 transforms `volt*Node!` vectors into `ampere*Node!` vectors.

Function `potential` solves the potential equilibrium and function `current` computes the current by back substitution. The equation is solved with primitive function `solve`. For equation $A \cdot x = b$ the call `solve(A,b)` gives the solution for x.

```
define potential(b, f) =
  solve(matrix2, matrix1 '*' b - f);
↪
potential :: for_index I: for_unit a, I!v:
  (a*Edge! per I!v, a*ampere*Node! per volt*Iv) -> a*Node! per I!v

define current(b, f) =
  conductance * (b - incidence '*' potential(b, f));
↪
current :: for_unit a:
  (a*Edge!, a*ampere*Node!/volt) -> a*ampere*Edge!/volt
```

The types are correct, but too general. The most general type for the potential function even allows matrices as arguments, but we are interested in the case of an empty column dimension and unit `volt` instead of variable `a`. The following type declarations express this. The type system checks these declarations against the inferred type.

```
declare potential :: (volt*Edge!, ampere*Node!) -> volt*Node!;
declare current :: (volt*Edge!, ampere*Node!) -> ampere*Edge!;
```

According to these types the volt on the edges and the ampere on the nodes is given.

Function `potential` computes the volt per node and function `current` computes the ampere per edge.

To compute the case we define the `inflow` and the `battery` vectors as follows.

```
defmatrix inflow :: ampere*Node! = {
  n0 -> 1,
  n1 -> 1,
  n2 -> 6,
  n3 -> 0
};

defmatrix battery :: volt*Edge! = {
  e0 -> 0,
  e1 -> 0,
  e2 -> 0,
  e3 -> 0
};
```

The inflow vector contains the inflow numbers and the battery vector is zero. With these vectors defined the solution can be computed.

```
print potential(battery, inflow);
↪
Node          Value
--------------------
n0       -4.000000 V
n1       -1.666667 V
n2       -4.666667 V
n4        0.000000 V

print current(battery, inflow);
↪
Edge          Value
--------------------
e0       -2.333333 A
e1        1.333333 A
e2        3.333333 A
e3       -4.666667 A
```

These numbers are the correct solution to the case. The printed vectors are the potential and current for the given inflow.

```
function FOURIER_MOTZKIN(A)
    B ← I_m
    for i := 1 to n do
        (1) Append to matrix [A|B] all rows which results as
        positive least common multiple linear combinations of
        row pairs out of [A|B] that both annul the i-th column
        of A and are minimal rows in B
        (2) Eliminate from matrix [A|B] the rows in which
        the i-th column of A is non null
    end for
    return B's rows
end function
```

Figure 6.2: The Fourier-Motzkin algorithm.

The electricity network case shows that the matrix type gives good insight into linear algebra computations. The scalar factor in the homogeneous node and edge vectors show exactly what linear transformations the matrices perform and what quantity is in each vector. This prevents many types of errors since the system will complain if vectors or matrices are from the wrong space.

6.3.2 *Case: Fourier Motzkin*

The Fourier-Motzkin algorithm computes the integer null space of a matrix [CS91]. The algorithm is a typical example where the dual use of matrices as linear transformations and as general data structure is rejected by the matrix type. Mathematically a matrix represents a linear transformation, but matrices and vectors are often used as general arrays or list. Such dual use is not possible with the strict matrix type and a separate array or list data type is needed in such cases. The matrix type can infer very detailed types because it is strictly limited to linear algebra expressions. Data types like tuples, arrays and lists are required for other uses.

The Fourier-Motzkin implementation in this case is an adaptation of Martinez and Silva's algorithm to compute all invariants of a generalized Petri Net [CS91]. The algorithm is shown in Figure 6.2. For any m×n integer matrix A, the algorithm computes all positive integer vectors y^T such that $y^T \cdot A = 0$.

The algorithm in this form cannot be typed, because the operations on the matrix in the algorithm are array or list operations instead of linear algebra operations. First matrices A and B are glued together. Next this matrix grows and shrinks, until it is broken down into a list at the end. The growing and shrinking matrix is not a linear transformation and cannot be given a meaningful matrix type. The algorithm actually uses the matrix as a list of vectors. During the loop it operates on the matrix' rows.

The effect of gluing matrices A and B together is that row operations on A are also carried out on B. At the end it returns the matrix' rows as list and the matrix itself is no longer relevant.

To enable meaningful types the Pacioli version replaces the matrix by a list of vector pairs. Instead of operating on the matrix rows and turning them into a list at the end, the matrix is turned into a list of vectors first and the algorithm then operates on this list. And instead of combining the rows by gluing them together, they are combined by putting them in a tuple. A minor change is that the Pacioli version operates on columns instead of rows. In practice column vectors are more convenient than row vectors, but for the algorithm this is irrelevant.

To implement these ideas the Pacioli version does not create matrix $[A|I]$ but creates the list of vector quadruples $[(a_k, i_k, s_k, t_k) \mid k < n]$. Vectors a_k are the columns from A, vectors i_k are the columns from I, vectors s_k is the support of a_k and t_k is the support of and i_k. The supports are not strictly necessary, but they are used often so storing them provides some caching. Instead of row operations, operations are lifted to the quadruples. For example scaling is lifted by $\alpha(a, b, c, d) = (\alpha a, \alpha b, c, d)$. With these changes the algorithm can be written using standard idiom for lists.

```
declare fourier_motzkin ::
  for_index I, J: for_unit a, I!v: (a*I!v per J!) -> List(J!);

define fourier_motzkin(matrix) =
  let
    pairs = zip(columns(matrix), columns(right_identity(matrix))),
    quads = [make_quad(v,w) | (v,w) <- pairs],
    eliminated = loop_list(quads, eliminate, row_domain(matrix))
  in
    [quad_right(q) | q <- eliminated]
  end;
```

First the vector pairs are created by zipping the columns of the matrix and the identity matrix. Next a list of quadruples is created from them. Function `make_quad` computes the supports and creates the quadruple. The implementation of this and other quadruple functions is straightforward and not explained further. Finally the algorithm's loop over the rows of the matrix with function `eliminate`. Function eliminate implements the growing and shrinking of the list that is described in steps (1) and (2) in the algorithm and is a direct translation of the mathematical description into code. It uses list comprehensions to perform the same computation as the manipulation of the matrix rows in the original version.

```
define eliminate(quads, row) =
  let
    combined = [canonical(x) |
                  (q,r) <- combis(quads),
                  alpha := quad_magnitude(q, row),
                  beta := quad_magnitude(r, row),
                  alpha * beta < 0,
                  x := combine_quads(abs(beta), q, abs(alpha), r)]
  in
    minimize([q | q <- append(combined, quads),
                  quad_magnitude(q, row) = 0])
  end;
↪
eliminate :: for_index I, J: for_unit a, b, I!v, J!w:
  (List(Tuple(a*I!v, b*J!w, I!, J!)),
   Index(I)) -> List(Tuple(a*I!v, b*J!w, I!, J!))

define minimize(quads) =
  [q | q <- quads, all[r = q | r <- quads, support_sub(r,q)]];
↪
minimize :: for_index I, J: for_unit a, b, I!v, J!w:
  (List(Tuple(a*I!v, b*J!w, I!, J!))) -> List(Tuple(a*I!v, b*J!w, I!, J!))
```

The type of functions `eliminate` and `minimize` and the type of the Fourier-Motzkin function are derived correctly without any annotation.

The case shows that the matrix type enforces a strict distinction between matrices and arrays. The strict mathematical definition of the matrix type eliminates any use besides linear algebra. The use of matrices as general arrays as in the original algorithm is rejected. Combining the matrix type with other parametric types as was done in the example gives alternative solutions that are completely type correct. The strictness of the matrix type together with other data types like lists and arrays improves the safety of the solution.

6.3.3 *Case: Bill of Material*

The bill of material case demonstrates the use of dimensioned vector spaces in the matrix type. It involves vectors and matrices with heterogeneous units of measurement that have to be typed correctly. The types in the solution show how the computation is unit correct without any knowledge of individual products and units. The individual units only play a role at runtime when the concrete case is solved.

Problem Statement

The quantitative relationship between a product and its parts that is found in the bill of material provides a useful norm for controlling the cumulative effects of variances in production processes. The bill of material in production processes tells how much of one product is used in the production of the other. Products can consist of many sub-assemblies and small disturbances in a production process can lead to larger negative effects downstream or extra requirements upstream. Such variances in production are not only a challenge from a quality control and operational risk management perspective, but they also distort the assurance processes from an auditing perspective. With the bill of material the cumulative effects in production process can be analyzed. Given the output of a production process, it can be used to compute the production volume in all intermediate production steps. In operations research and production planning this computation is called the 'explosion' of the bill of material. This operation solves the recursion in the product-part relation that causes the cumulative effects. This helps to analyze and explain issues like the variances in production volumes.

The explosion operation solves the recursion in the bill of material, but for the application as a normative model the explosion operation needs to take rejection of products into account. The volumes computed with the exploded bill of material are all ideal, and do not take any waste into account. Actual volumes will likely deviate from these ideal numbers, and these deviations add up further downstream the production process. To compensate for this cumulative effect, the amount of waste must be part of the explosion.

An example of a production process where the varying reject rate of products have a significant cumulative effect is the production of Integrated Circuits (IC). The following case will be analyzed in the remainder of the section:

> The production of an integrated circuit transforms silicon and various other materials into an integrated circuit. The first step is to produce wafers from silicon. Wafers are discs of silicon on which various layers of other materials are placed. These layers make up the logical circuits. Many identical circuits are printed and in a next production step the wafer is cut into dies. Each die is placed in a case and connected with wires to pins.
>
> The following table gives a selection of some material usage in a finished IC. In this case 200 dies are cut from a single wafer. Each IC uses one die, one case and 0.8 mg of wire during assembly. A wafer uses 10 mg silicon and 0.12 mg other metals.

Product	Material
IC	Die, Case, Wire 0.8 mg
Die	Wafer 1/200
Wafer	Silicon 10 gr, Metal 0.12 gr

After a certain production run the following numbers are reported.

Product	Planned output	Actual production
IC	50,000 ic	53,800 ic
Die		125,900 die
Case		53,900 case
Wafer	400 wfr	1,460 wfr
Wire		43,700 g
Silicon		14,700 g
Metal		176 g

The numbers in column *Planned output* is the planned production of end products. Planned output is 50,000 completed ICs and 400 uncut wafers that will be processed further somewhere else. Column *Actual production* is the actual production volume of each individual product or part in each step of the process. It is the total production that is necessary to create the end products.

Suppose that managers or controllers are faced with such reported production volumes. How can they judge these numbers?

Judgment of the numbers in the case is complicated by the cumulative effect in waste. Waste is caused by the rejection of products that are considered unfit for use. Since a product becomes waste when it is itself rejected or when it is part of another wasted product, the fraction of a product that is wasted has a cumulative effect and is at least as large as the fraction of the product that gets rejected. Consider for example the 1460 produced wafers in the case. From the bill of material follows that each wafer gives 200 dies and thus also 200 ICs, so to produce 50,000 ICs requires 250 wafers. Together with the 400 uncut wafers this totals 650 wafers, which is considerably lower than the 1460 produced. Overall 55.48 % of the produced wafers are wasted. But a large part of the wasted wafers are caused by the rejection of dies and ICs, and not by the rejection of faulty wafers.

The question is how the cumulative effect can be undone and the individual reject rate for each product can be determined. Specifically the question is how the following quantities are related.

$$\begin{aligned}
BoM_{ij} &= \text{The amount of product } i \text{ that is used by product } j. \\
totvol_i &= \text{The total production volume of the i-th product.} \\
output_i &= \text{The end-product volume of the i-th product.} \\
accept_i &= \text{The fraction of product } i\text{'s volume that is accepted.}
\end{aligned}$$

Matrix *BoM* is the bill of material. Vector *totvol* corresponds with the total volume from the case, and vector *output* with the output volume. The accept rate is the complement of the reject rate. This is mathematically more convenient. Computing *totvol* from *output* given some *BoM* is the explosion problem. The question is how this computation can take the accept rate *accept* into account.

Leontief Matrices and the Bill of Material

The explosion of the bill of material from operations research has its origins in Leontief's macro-economic input-output models [LL86]. Leontief's models are used to calculate equilibria of the productivity of regions in an economy. The calculation of the model involves the geometric matrix series $I + A + A^2 + A^3 + \ldots$ with A an input-output matrix for an economy. The case of the explosion problem from production planning involves the same series, but input-output matrix A is now replaced by the bill of material. The resulting matrix is called the exploded bill of material and forms the relation between *totvol* from *output* The computations are similar, but now in a micro economic setting.

The explosion of a matrix into the geometric series is the Kleene star operator for square matrices. Leontief models are solved with matrices A^* and A^+ that satisfy the following equations.

$$\begin{aligned}
A^* &= I + A + A^2 + A^3 + \ldots \\
&= I + A \cdot A^* \\
&= (I - A)^{-1}
\end{aligned} \tag{6.1}$$

$$\begin{aligned}
A^+ &= A + A^2 + A^3 + \ldots \\
&= A^* - I
\end{aligned} \tag{6.2}$$

The equations correspond with a Kleene algebra for square matrices, and in that case the star operator generates the geometric series. If the series converges the solution $A^* = (I - A)^{-1}$ gives a way to compute the Kleen star A^*. Rearranging equation(6.1)

into the following form shows how the exploded matrix is used to solve recursive equations.

$$x = A^* \cdot v \;\Leftrightarrow\; x = v + A \cdot x \tag{6.3}$$

This property is used in the solution in the remainder.

The explosion of the bill of material appears when the total production volume is computed given some desired output volume. For any vector v containing some product volume, multiplication $BoM \cdot v$ is the volume of all parts from which the products are directly composed. We could compute $BoM \cdot (BoM \cdot v) = BoM^2 \cdot v$ to obtain the volume at the next level and if we repeat this enough times and add all results we obtain the volume of all direct and indirect parts. The recursive nature of the bill of material leads to the series $BoM + BoM^2 + BoM^3 + \ldots$. Using the Kleene star the total volume can be computed as

$$totvol = BoM^* \cdot output$$

and equation (6.3) tells that the corresponding recursive is

$$totvol = output + BoM \cdot totvol$$

This is the solution for the bill of material explosion from operations research that we want to extend.

To incorporate the accept fraction into the explosion the recursive equation has to take it into account. The following equation relates the quantities from the problem statement. It is the recursive equation for the bill of material in which *totvol* in the left hand side is multiplied by the accept fraction.

$$accept \times totvol = output \;+\; BoM \cdot totvol$$

Value $BoM \cdot totvol$ is the key to the interpretation of the equation. Earlier we saw the $BoM.x$ is the volume of all parts from which the products are directly composed, but now we are interested in special case $BoM \cdot totvol$. Since *totvol* is the total flow $BoM \cdot totvol$ contains each end product's direct parts but also the parts at deeper levels of composition, but in ideal amounts. The amounts have to be corrected for rejected parts. So the interpretation of $BoM \cdot totvol$ is that it is the volume of all non rejected parts flowing through the process. Continuing from the interpretation of $BoM \cdot totvol$ of the previous paragraph we derive that since *output* is the ideal end-product volume that the sum of *output* and $BoM \cdot totvol$ must equal the total ideal flow. Another expression for the total ideal flow is $accept \times totvol$. Putting both expressions together gives the equation.

The hardest problem is to calculate *totvol* given *output* and *accept*. If *totvol* and *output* are given we can directly calculate *accept* by rewriting the equation to

$$accept = \frac{output \;+\; BoM \cdot totvol}{totvol} \tag{6.4}$$

Calculation *output* from *totvol* and *accept* is done by rewriting the equation to

$$output = accept \times totvol \; - \; BoM \cdot totvol \tag{6.5}$$

For real physical production processes this result cannot contain negative numbers. To get the solution if *output* and *accept* are known the equation is rewritten with (6.3) to use the Kleene star

$$\begin{aligned} totvol &= output/accept + BoM \cdot totvol/accept \\ &= output/accept + diagonal(accept^{-1}) \cdot BoM \cdot totvol \\ &= (diagonal(accept^{-1}) \cdot BoM)^* \cdot (output/accept) \end{aligned} \tag{6.6}$$

Matrix $diagonal(accept^{-1})$ is a diagonal matrix with entry $1/accept_i$ at the i-th diagonal position. The product $diagonal(accept^{-1}) \cdot BoM$ is the bill of material adjusted for rejection. This adjustment fixes the cumulative effect of waste in the recursion.

Leontief Matrices in Pacioli

The Leontief matrices and and the solution to the problem statement can be implemented in Pacioli with the help of standard functions `inverse` and `left_identity`. The following signatures show the types of these functions.

```
left_identity :: for_index I, J: for_unit a, I!v, J!w:
  (a*I!v per J!w) -> I!v per I!v

inverse :: for_index I, J: for_unit a, I!v, J!w:
  (a*I!v per J!w) -> J!w per a*I!v
```

The left identity is a square matrix of the proper type `I!v per I!v`. If the original matrix is multiplied by it from the left it leaves the type unchanged. Similarly the type of the inverse function shows that any matrix can be multiplied from the left or right by the inverse and the result will be the proper identity matrix.

With the identity and the inverse functions the A^* and A^+ operations can be implemented. The following definitions implement equations (6.1) and (6.2).

```
define kleene(x) = inverse(left_identity(x) - x);
↪
for_index I: for_unit I!v: (I!v per I!v) -> I!v per I!v;

define kleene_plus(x) = kleene(x) - left_identity(x);
↪
for_index I: for_unit I!v: (I!v per I!v) -> I!v per I!v;
```

In both cases the type is correctly infered as a function from square matrices to square matrices. Now the the solutions for *accept*, *output* and *totvol* can be implemented. The following declarations shows the types of the solutions. As a hint that the functions work on product vectors the variable names `P` and `P!u` are chosen.

```
declare accept_fraction ::
   for_index P: for_unit P!u: (P!u per P!u, P!u, P!u) -> P!;

declare output, volume ::
  for_index P: for_unit P!u: (P!u per P!u, P!, P!u) -> P!u;
```

The accept fraction is a dimensionless product vector. The output and volume are dimensioned product vectors. The following functions implement the definitions from equations (6.4), (6.5) and (6.6).

```
define accept_fraction(bom, out, tot) = (out + bom ’*’ tot) / tot;

define output(bom, acc, tot) =  acc * tot - bom ’*’ tot;

define totvol(bom, acc, out) = kleene(adjust(bom, acc)) ’*’ (out / acc);

define adjust(bom, acc) = diagonal(acc^R * row_unit(bom)) ’*’ bom;
```

All inferred types satisfy the declared types. The solution directly follows the mathematical definitions and the types correctly express the units involved without knowledge of the actual units or products. The correctness is established for any actual products and units.

Case Solution

The solution for the IC case involves actual products and units, and shows the runtime aspects of unit-aware matrices. The following index and unit definitions define the actual products and units in the IC case.

```
defindex Product = {IC, Die, Case, Wafer, Wire, Silicon, Metal};

defunit ic "ic";
defunit die "die";
defunit case "case";
defunit wafer "wfr";
```

The index definition contains all products. The unit definitions define pseudo units

for the various products. The following unit definition of `Product!unit` defines a base unit vector with the appropriate unit for each product from the index set.

```
defunit Product!unit = {
  IC: ic,
  Die: die,
  Case: case,
  Wafer: wafer,
  Wire: milli:gram,
  Silicon: gram,
  Metal: gram
};
```

With the products and their units defined the values can be defined next, just as in the electrical network case. For the sake of brevity the actual numbers have been omitted.

```
defmatrix BoM :: Product!unit per Product!unit = { ... };
defmatrix planned_output :: Product!unit = { ... };
defmatrix actual_volume :: Product!unit = { ... };
```

If the bill of material is exploded we see that all numbers have the right units.

```
print kleene_plus(BoM);
↪
Product, Product                   Value
-----------------------------------------------
Die, IC                         1.000000 die/ic
Case, IC                        1.000000 case/ic
Wafer, IC                       0.005000 wfr/ic
Wafer, Die                      0.005000 wfr/die
Wire, IC                        0.800000 mg/ic
Silicon, IC                     0.050000 g/ic
Silicon, Die                    0.050000 g/die
Silicon, Wafer                 10.000000 g/wfr
Metal, IC                       0.000600 g/ic
Metal, Die                      0.000600 g/die
Metal, Wafer                    0.120000 g/wfr
```

The exploded matrix gives the direct and indirect part of relation in proper units of measurement.

Using the functions from the previous section the ideal volume can be computed. The unit vector `|Product!|` is passed as the ideal accept fraction.

```
define ideal_volume = totvol(BoM, |Product!|, planned_output);
↪
ideal_volume :: Product!unit

print ideal_volume;
↪
Product            Value
----------------------------
IC          50000.000000 ic
Die         50000.000000 die
Case        50000.000000 case
Wafer         650.000000 wfr
Wire        40000.000000 mg
Silicon      6500.000000 g
Metal          78.000000 g
```

The ideal volume for wafers is the requirement 650 wfr that was mentioned in the problem statement section. Waste is the difference between the actual volume and the ideal volume. In the following code the waste percentage is computed.

```
define actual_waste = actual_volume - ideal_volume;
↪
actual_waste :: Product!unit

print to_percentage(actual_waste / actual_volume);
↪
Product         Value
-----------------------
IC           7.063197 %
Die         60.285941 %
Case         7.235622 %
Wafer       55.479452 %
Wire         8.466819 %
Silicon     55.782313 %
Metal       55.681818 %
```

Here we see the 55.48 % waste from the problem statement section.

Finally the reject fraction is computed. The complement is taken by subtracting the accept fraction from a product unit vector with the standard function `complement`. This standard function gives a matrix with value $1 - A_{ij}$ for each matrix entry A_{ij}. It is implemented as follows.

```
define complement(x) = unit(x) - x;
↪
complement :: for_index I, J: for_unit a, I!v, J!w:
  (a * I!v per J!w) -> a * I!v per J!w
```

In the case of dimensionless product vectors the unit is `Product!`, and for any value `x` `complement(x)` equals `|Product!| - x`. The reject fraction is computed as follows.

```
define reject =
  complement(accept_fraction(BoM, planned_output, actual_volume));
↪
reject :: Product!

print to_percentage(reject);
↪
Product         Value
-----------------------
IC           7.063197 %
Die         57.267673 %
Case         0.185529 %
Wafer       29.486301 %
Wire         1.510297 %
Silicon      0.680272 %
Metal        0.454545 %
```

The results show that reject is mostly negligible, except the IC around 7 %, the wafer just below 30% and the die almost 60 %. Eliminating the cumulative effect in waste helps judgment of the production numbers.

The solution of the case is completely unit-correct. All numbers have the proper unit of measurement. The single unit vector is sufficient to type the heterogeneous units of measurement in the bill of material.

6.4 CONCLUSION

Implementing the matrix type showed practical considerations and feasibility of the matrix type. The implementation with different runtime environments allowed

experimenting with different levels of unit support. The matrix type guarantees unit correctness at compile time, but two features that still depend on runtime support for units are unit conversions and printing of unit symbols.

The three cases highlight different aspects of the matrix type. The electricity case shows use of general index sets and how the type system gives insight into the involved vector spaces. The Fourier-Motzkin case illustrates the difference between matrices and arrays and that an index-free style is preferred for matrix programming with the matrix type. The bill of material case is presented as an example of a matrix with heterogeneous units, but it is also relevant for the analysis done in part II. The proportions between the products in the bill of material is an example of proportionality in business processes that is central to the value cycle. A topic for future research is how the analysis with the bill of material can be integrated with value nets.

Part II

Value Nets

7 INTRODUCTION

In this second part of the thesis we develop value net models as the basis for the analysis and design of organizations. Value nets are cyclic enterprise models that focus on the value creation part of an organization. The aim is to build a computational foundation for the numerical aspects of control, auditing and related fields.

7.1 THE VALUE CYCLE

The value cycle from auditing is a conceptual model of an enterprise's value creation process that relates the various organizational parts for analytical purposes. As explained in the introduction, it is a cyclic model that focuses on causality and proportionality in enterprise processes. An enterprise is seen as a state transition system in which the state is the value of the enterprise's assets and liabilities, and in which the transitions are economic events consuming and producing value [SdMJ88].

The image in Figure 7.1 is a translation of a value cycle figure from a Dutch auditing manual [Vee72]. According to accounting convention, the circles are transitions, the squares are assets and liabilities, and the signs indicate in which direction value flows. The *Produce* transition consumes resources and produces goods. Next the *Sell* transition consumes these goods and produces receivables, and after that the *Receive* transition consumes the receivables and produces money. The *Pay* transition is different because it has two minus signs on its connecting inflow and outflow edges. The reason is that payables is a liability and paying consumes money as well as payables. The *Buy* transition completes the cycle. It produces resources and payables. The *Buy* and the *Sell* transitions connect the financial part in the upper half and the operational part in the lower half. By connecting finance and operations, the value cycle gives a picture of the circular flow of values in an organization.

In the audit literature the value cycle is the basis for quantitative analysis [ACR86; FdH89; SdMJ88; Vee72]. An auditor faces the question whether an enterprise's financial statements faithfully represent what happened in reality, or that they are materially overstated or understated [AL84; FdH89]. From an auditor's point of view the question of soundness and completeness is complicated due to the fact that audit evidence is often recorded by the audited enterprise itself. The causality and proportionality from the value cycle helps an auditor to reason about evidence by providing essential knowledge about the business processes, especially transformation coefficients in the production part. Recall the running example of the wine bottler from Figure 1.1 in the introduction. If there is evidence that the wine bottler bought 750 litres of wine, then you would expect 1000 bottles of wine to be bottled and sold since a bottle contains 0.75 litre of wine. The equations from the audit field were

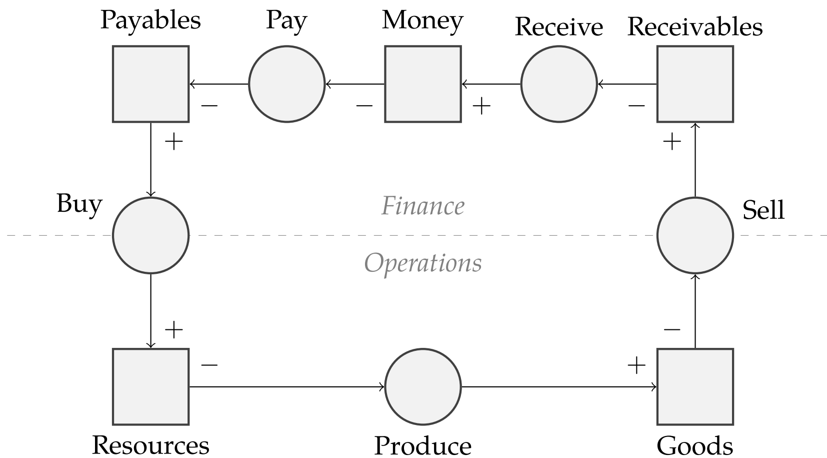


Figure 7.1: The Value Cycle model from the audit literature. It shows the circular flow of values in an enterprise. Circles denote actions and squares denote state. The signs indicate the direction of the flow of value.

developed to support such quantitative analysis. Coefficients like 0.75 l/bottle are production norms in the cybernetic character of business behavior and are essential for the audit equations. In this way the value cycle allows comparison and reconciliation of independently recorded evidence [FdH89; SdMJ88].

An example of a set of audit equations is Frielink's equations for an elementary trading company as shown in Figure 7.2. The equations relate various period totals in a trading company's accounting system [FdH89]. This case is a little more complicated than the model from Figure 7.1 because it includes value added tax (VAT), but at the same time it is simpler because there is no transformation process in a trading company. The form of the equations reflects how they are used in an audit to check for understatement or overstatement. Checking the value on the right hand side for overstatement requires checking the left hand side values with a positive sign for overstatement and the ones with a negative sign for understatement. Similarly, checking the right hand side for understatement requires checking positive values on the left hand for understatement and the negative ones for overstatement. This form of the equations has been developed to guide the reconciliation checks that an auditor performs during an audit. Equation sets like this have been developed for the value cycles of various types of organizations [FdH89; SdMJ88].

Besides the quantitative analysis with the audit equations the value cycle is also the basis for the qualitative analysis of segregation of duties. When hierarchies are introduced in an organization the need for internal control emerges. Typically

$$
\begin{array}{ll}
\text{Receivables (R):} & R_{initial} - R_{final} + T_{sales} + S_{sub} \times SP = M_{add} \\
\text{Sales:} & J + S_{sub} \times PP = S_{sub} \times SP \\
\text{Stock (S):} & S_{initial} \times PP - S_{final} \times SP + S_{add} \times PP = S_{sub} \times PP \\
\text{Payables (P):} & P_{initial} - P_{final} - T_{purchases} + P_{sub} = S_{add} \times PP \\
\text{Money (M):} & M_{initial} - M_{final} - T_{sub} + M_{add} = P_{sub} \\
\text{Tax (T):} & T_{initial} + T_{sales} - T_{purchases} - T_{sub} = T_{final} \\
\text{Sales:} & T_{sales} = S_{sub} \times SP \times p \\
\text{Purchases:} & T_{purchases} = S_{add} \times PP \times p
\end{array}
$$

Figure 7.2: Frielink's audit equations for a trading company. Each equation describes a relationship that must hold between various values in an accounting system. The character in parenthesis after some of the names indicate variables that are used in the equations. Subscript $_{add}$ indicates the added value and subscript $_{sub}$ indicates the subtracted value. Additional variables are SP for the sales price, PP for the purchase price, p for the VAT rate and J for the value jump or margin.

control mechanisms like planning and budgeting procedures, managerial performance evaluation and instruments like transfer pricing and activity-based costing are often used to ensure that assets of the organization are safeguarded and the chances of misconduct by either one of the participants is minimized. Sound bookkeeping practices lie at the heart of good control [RS06; Sim90; Sim95; Sim87]. A control measure of special importance to auditors is segregation of duties. Good segregation of duties makes it harder to manipulate recordings and is a requirement when relying on reconciliation relations. Setting up a proper segregation of duties is a design issue for which the value cycle is the basis. It is a qualitative aspect of a value cycle that is a requirement for the quantitative analysis with the audit equations [AL84; BDW95].

Segregation of duties is an example of a control measure on whose continuous functioning the reliability of the first recording of financial facts depends. Financial reporting requires reliable data. For external audit these requirements even give a formal proviso on an auditor's judgment [BDW95]. As explained, the reliability of first recordings necessarily depends on the proper functioning of the organization's internal control system. The question whether information has been recorded with sufficient segregation of duties cannot be established by inspection of the recorded data or even inspection of the information system that recorded the data. The proper functioning of the information system during the audited period has to be demonstrated, and this requires a set of controls that guarantee that [BDW95; FdH89; SdMJ88]. The design of such a set of controls takes the value cycle as input.

This analysis of the segregation of duties is the qualitative aspect of the value net's application as a modeling tool.

7.1.1 *Process Modeling and Auditing*

The value cycle model has some specific characteristics that follow from its application as a business analytics tool and its emphasis on the quantitative relationship between business processes. The value cycle is a process specification that does not aim at process execution but at process diagnosis and analysis. It is often used for comparison with actual behavior or to specify future or expected behavior. For example in business planning, or cost management. A specific characteristic is that modeled behavior is usually in aggregates. Actual or historical behavior can be captured with a trace, but this is unsuitable for predicting behavior. We can predict that 1000 sell occurrences will happen, but not the individual amounts, dates, etc. A related difference is that modeled behavior is often ideal, or without the details of exceptions and error handling. Properties like failure rates are important aspects of process models, but typically modeled as a parameter. The actual process that leads to the failure rate is of less concern. Another characteristic is that the modeled behavior is often required to be deterministic and reproducible. For example, in a budget or costing computation, a probability distribution is not useful. Predicted behavior can therefore not be computed with simulations, but simulations can play an important role. They can for example be used to analyze the potential efficiency of an operational process design. The outcome can then be used as input parameter for a budget or costing model. All these characteristics follow from the specific applications for which the value cycle is used.

The value cycle's causality and proportionality provide a unique combination of precise analytics on models at a top level view of an organization [GEvdR00; SdMJ88]. Process models have been used extensively for business purposes, but no method covers the applications for which value cycles are used, like auditing, budgeting, costing, etc. Inquiry into the usage of conceptual modeling in practice showed that specialized tasks like identifying activities for activity-based costing and internal control purposes in auditing are among the reported purposes for modeling [DGR+06], but such usage is more about modeling as an elicitation method than using models as a computational and analytical tool. Diagnosis and adaptation of business processes is more advanced in the business process modeling (BPM) field [AHW03], but this is about an individual execution in a workflow system. It is concerned with handling a single job or case, while the application areas of the value cycle are concerned with handling all jobs, not individual ones. The difference can be illustrated with compliance testing. A process specification can be used as a norm to test whether the actual execution complies with it, but this kind of check does not go beyond the correctness of the execution sequence. This also holds for

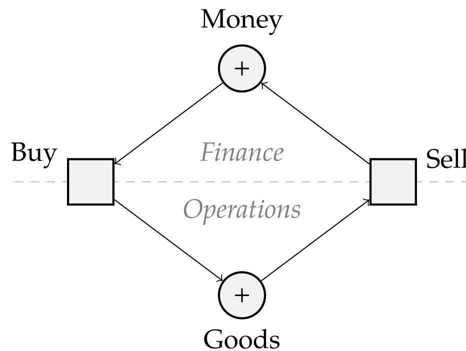


Figure 7.3: An elementary value cycle. Following Petri net convention a rectangle depicts a business event and a circle depicts an asset or liability.

methods like behavioral profiles or footprints [WPM+10]. From a business perspective, this kind of compliance test is of limited interest [Bar09]. A practical reason is that in business there is always a large variation in the actual performance, due to the dynamic environment, human errors or opportunistic behavior. In such a situation, creating value is more important than correctness. A more fundamental reason is that compliance checking usually assumes that the data log is complete, while from an auditing perspective this cannot be assumed but instead has to be proven. Assessing whether data is missing is part of the goal of auditing. This completeness question requires a less operational kind of reasoning that looks at enterprise behavior as a whole. The value cycle is a holistic model that supports such analysis.

7.1.2 *Computational Auditing*

The introduction of audit nets in computational auditing has changed the value cycle from a conceptual model to a process modeling technique [Els96]. Instead of serving as a general conceptual model, an audit net is instantiated for a specific business.

Audit nets are built from parameterized building blocks that represent events and assets or liabilities. With the parameters the building blocks can be instantiated for a specific situation. A parameter may for example be a certain production ratio, or a storage capacity. The overall structure of an audit net is defined inductively with a grammar that specifies how the building blocks can be connected. The grammar's start configurations are elementary audit nets. General rules exist for adding new intermediate steps and for adding assembly steps. These rules have several variants for assets and liabilities. Other rules exist for specific cases like value added tax (VAT) and illicit events.

Figure 7.3 is an example of a start configuration. It is the smallest possible audit net and models the value cycle of an elementary trading company. Following the Petri net convention, events are drawn as squares, and assets and liabilities are drawn as circles. The audit net formalism uses special symbols for the building blocks, but here we draw them as normal Petri nets. A plus sign in a place denotes an asset and a minus sign a liability. The *Buy* transitions is an event that consumes money and produces goods. *Money* and *Goods* are both assets. The *Sell* event consumes the bought goods and produces money. This cyclic economic behavior is the essence of the value cycle.

In the extended model from Figure 7.4 the elementary trading company has been rewritten to a more elaborate trading company that is more in line with the original value cycle from Figure 7.1. In the elementary trading company from Figure 7.3 all transactions are direct exchanges of money and goods. However, exchanges are not always direct and parties instead often agree on promises to deliver at a later moment. The extended model incorporates such rights and obligations. All arrows in the elementary trading company have been replaced by an additional state transition pair that models a right or obligation. *Payables* is a liability that represents an obligation to pay. It is a promise made in a *Buy* event together with an order that represents a promise of the delivery of a good by the other party. The situation on the sell side is the opposite with the asset *Receivables* and now an *Order* as liability. A *Sell* event produces the right to receive money and generates the obligation to deliver goods. These four extensions are a more systematic approach to the elements found in the original value cycle.

The elementary and extended value cycles are generic and still need to be instantiated for an actual enterprise by specifying an actual production process and inserting numbers on the edges. Besides the possible rights and obligations an actual process may also contain assembly steps. This gives a process graph that represents the relationship between the different actual goods flowing through a process. On the edges of this graph the characteristic production ratios for the enterprise are placed. Examples are the consumption of 750 ml of wine by the bottle wine event in the wine bottler case. Such numbers specify the enterprise's normal behavior. By choosing specific goods and putting numbers on the arrows the generic value net model is instantiated for a specific enterprise.

With the audit net formalization of the value cycle it is was shown to be possible to derive the audit equations as found in the Dutch audit literature from the structure of an audit net, but only for a limited class of value cycle models. Instead of a general set of equations as given by Frielink, a set of equations specific for an organization is derived. It was shown that such a set of equations is isomorphic to the audit net, and that consequently the equations are just a different representation of the value cycle [Els96]. An assumption in the derivation of the equations is however that

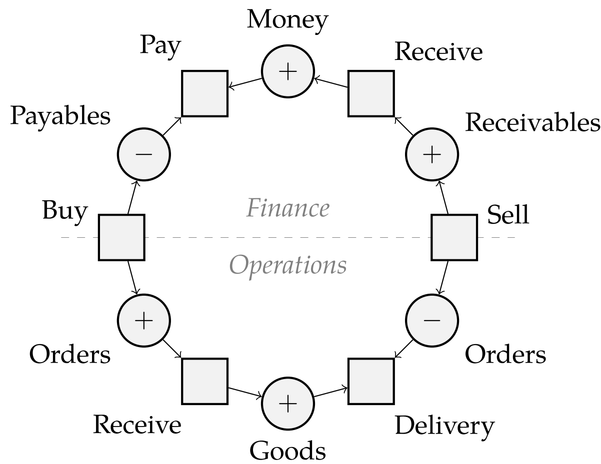


Figure 7.4: An extended value cycle. Extra events and states representing rights and obligations have been added to the elementary value net from Figure 7.3.

the production process has a tree structure. This requirement is guaranteed by the grammar rules, but it limits the application of the audit nets. Although the class of models is limited, deriving an organization specific set of equations automatically shows the benefits of the formal approach of computational auditing.

Another powerful result from computational auditing is the idea to generate a complete set of potential fraud scenarios for an audit net. When segregation of duties is insufficient, then opportunities for fraud arise. Fraud is the undetectable extraction of value from an enterprise and differs from theft. It is a special kind of theft that often involves clever manipulation of an enterprise's assets and liabilities for personal gain. Theft also extracts value, but that will be noticed if an inventory is taken. Fraud is constructed in such a way that the victim does not notice the missing value from the inventory. A good fraud can go on for long periods without being noticed and can form a permanent leak in an enterprise's value creation process. Since fraud is hard or (practically, cost-effectively) impossible to detect, it has to be prevented by for example segregation of duties. An important result from computational auditing is that the set of t-invariants in an audit net forms a basis from which all possible fraud scenarios can be generated. A t-invariant is a combination of actions that leaves no trace. Computing the t-invariants for the ist modality of the net gives potential fraud scenarios. This enables design time analysis of and reasoning about the absence of fraud classes and the effectiveness of the segregation of duties.

7.2 VALUE NETS

Although the value cycle was developed in the auditing field, its ideas are applicable more widely. The focus on key quantities in an enterprise's value creation process is not only useful for external audits, but also for tasks like costing, performance management, budgeting, internal control, etc. [GEvdR00]. Many aspects of an organization are concerned with norms and normal or expected behavior. This can be analyzing past behavior or predicting future behavior. All these application areas besides auditing could benefit from value cycles and its business process modeling concepts.

To open the ideas from the value cycle to other application areas we generalized audit nets into value nets. A value net is a Petri net that models an enterprise's value creation process just like an audit net, except that enrichments like registrations and capacity are left out until they are needed. The elementary and extended trading companies from Section 7.1.2 are valid value nets and simply interpreted as the Petri net without the refinements from the audit nets. By sticking to the core structure of the value cycle and extending it when needed, value nets become applicable beyond audit. Constructs like registrations and capacity are specific for certain analytics and can always be added if needed. The core idea of the value net is that it provides a top level model of an enterprise's value creation process.

As explained in the introduction, a challenge when adding numbers to the value nets is the contradictory needs for the units of measurement of the model's numbers. Accounting systems record value using a single monetary unit. An advantage of a single unit is that everything is commensurable. Aristotle already observed that money is an 'equalizer' that makes it possible to compare different quantities [AR99].

> ... This is why all things that are exchanged must be somehow comparable. It is for this end that money has been introduced, and it becomes in a sense an intermediate; for it measures all things, ...
>
> (Aristotle, Nicomachean Ethics, Book V, Chapter 5)

It is even a requirement for the balancing of events in double bookkeeping [Ell82; Ell86; Iji65; Iji67; RPN10]. However, a value net in monetary units is unsuitable for conservation laws. Usually it's an enterprise's goal to generate value and the margin that is generated by the process is an increase in value. This value jump shows that value is not a conserved quantity. Conservation of value would be in contradiction with the goal of profit making. The unsuitability of monetary units for conservation laws can be illustrated with the running example of the wine bottler. Remember that the wine bottler has to buy two batches of 500 bottles for each purchase of 750 liter of wine. This gives a total cost of 1500 € + 2×400 € = 2300 €. The bottles of wine generate a revenue of 1000×3.50 € = 3500 €. So this process has no conservation of

value but instead generates 1200 €. This shows that exact connections in a value net are only feasible in non-financial units.

A value net's numbers are explicitly in non-financial units of measurement with the valuation kept separately. The valuation of products relates the process in monetary units with the process in product units. By factoring value into the product of a valuation and a quantity, the value net can accommodate conservation laws as well as relate to accounting systems. When the value of the products is considered it is expected that the value increases since it is a value creation process. In accounting this value jump leads to the notion of profit and loss accounts as we will see later. By modeling in non-financial units there is no need to incorporate profit into value nets.

A value net's behavior is the usual behavior of Petri nets and can conveniently be described with the net's incidence matrices. Let P be the collection of places and let T be the collection of transitions. Incidence matrices *Prod* and *Cons* of the form $P \times T \to \mathbb{N}$ contain the numbers on value nets' edges.

$$\begin{aligned} Cons_{ij} &= \text{the number on the arrow from place } i \text{ to transition } j \\ Prod_{ij} &= \text{the number on the arrow from transition } j \text{ to place } i \end{aligned}$$

For any transition vector t with t_j the number of times transaction j occurs, $Cons \cdot t$ is the consumed value, and $Prod \cdot t$ is the produced value. The overall effect is $Prod \cdot t - Cons \cdot t = (Prod - Cons) \cdot t$. If we define

$$Flow = Prod - Cons$$

then this can simply be written as $Flow \cdot t$.

With the incidence matrices the bottler's behavior discussed above can be written in linear algebra notation. Let vector t contain the transition count. The numbers are as follows.

Event	t
Buy wine	1
Buy bottles	2
Bottle wine	1000
Sell bottle of wine	1000

Vector t has shape $T \to \mathbb{N}$ and contains the amount of times each event occurs. The effect of this behavior can be computed with the incidence matrices. The result is a

$P \to \mathbb{N}$ place vector with the profit 1200 € in the money place and zero for all other places.

$$Flow \cdot t = \begin{bmatrix} \text{-150,000 ¢} & \text{-40,000 ¢} & \text{0 ¢} & \text{350 ¢} \\ \text{0 btl} & \text{0 btl} & \text{1 btl} & \text{-1 btl} \\ \text{0 btl} & \text{500 btl} & \text{-1 btl} & \text{0 btl} \\ \text{750,000 ml} & \text{0 ml} & \text{-750 ml} & \text{0 ml} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 1{,}000 \\ 1{,}000 \end{bmatrix}$$

$$= \begin{bmatrix} \text{350,000 ¢} \\ \text{1,000 btl} \\ \text{1,000 btl} \\ \text{750,000 ml} \end{bmatrix} - \begin{bmatrix} \text{230,000 ¢} \\ \text{1,000 btl} \\ \text{1,000 btl} \\ \text{750,000 ml} \end{bmatrix} = \begin{bmatrix} \text{120,000 ¢} \\ \text{0 btl} \\ \text{0 btl} \\ \text{0 ml} \end{bmatrix}$$

With the incidence matrices the consumed and produced values can be conveniently computed from the event vectors.

Note how all the produced and consumed products cancel in the subtraction $Prod \cdot t - Cons \cdot t$ in the second equality. This is characteristic of normal behavior, but before we discuss this some examples are given.

7.2.1 *The Miller and the Baker*

The milling company from Figure 7.5 and the bakery from Figure 7.6 will be used as a case study. These models have been created in cooperation with field experts and have also been used in education in the auditing domain. The models were chosen to illustrate some common patterns in value nets. They appear similar at first glance, but some key differences between the two production processes lead to different behavior. Other reasons these two cases were selected were that they have transformations with interesting units of measurement and that they can be connected to create a supply chain. The output of the miller is the input for the baker.

As before the value net's places are drawn as circles, but for readability the transitions are drawn as rectangles with the label inside the box. The sign of a place is drawn in the circle's center. The edges are labeled with the consumed and produced quantities. Although the units of measurement on each edge should equal the unit of the connected node, it can be convenient to use a larger unit. For example 1 t instead of 1000 kg. Convention is that the unit of a place is the smallest unit of the edges directly connected to it.

The main characteristic of the milling company is that it grinds grain into flour, bran and germ in reasonably fixed proportions. The main business of the milling company is to buy grain, grind it, and sell the result. The grind transition in the bottom left grinds grain in batches of 1 tonne. A tonne of grain is purchased for 250 dollar. Grinding the grain result in flour, germ and bran. It is a property of grain that the ratios of these three products is 830:140:30. One tonne of grain yields more

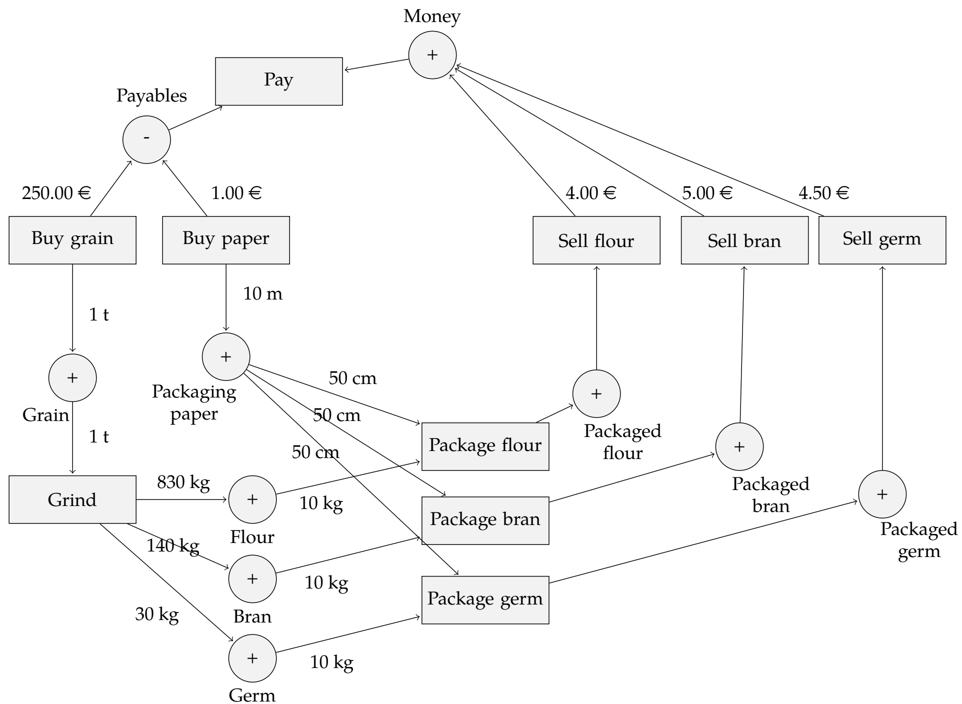


Figure 7.5: The Value Cycle of a miller.

or less 830 kg of flour, 140 kg of bran and 30 kg of germ. This is an example of a very strong connection between the buy side and the sell side. Unless the products leave the process in some other way, the ratio of pack transitions and also of the sales transactions must be the same 830:140:30. This strong numerical relationship in the process of grinding grain is a key characteristic of the milling company.

Another strong relationship in the milling process is the connection between paper and the rest of the process. This connection is also an example of redundancy that might be exploited to check the validity of behavior. Besides grain, the miller also buys paper to package the flour, bran and germ. According to the process model buying paper can occur independently, but normally it must occur in a certain ratio with the other transitions, just as the packing and sales. Although economically a minor part of the process, measuring such minor flows provides an opportunity to reveal anomalies in recordings. In this example it gives redundancy for the buy grain transition, just as the sales transitions do. In principle it would be sufficient to just record the buying of paper, or any other transition, because all other numbers follow from the amount of paper used.

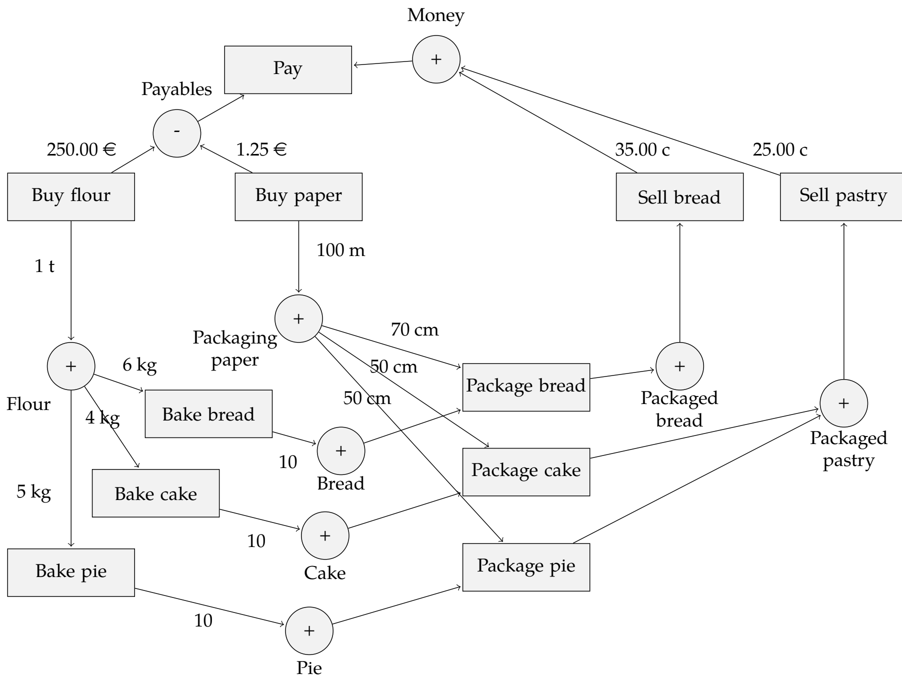


Figure 7.6: The Value Cycle of a bakery.

The bakery is similar to the milling company, but some small differences illustrate some more aspects of behavior in value nets. The baker also buys paper and a product, flour in this case, and it also produces three end products. The baker however, has a choice between baking bread, baking cake or baking pie. Another difference is that there are just two sales transitions. This models some information loss in the process. These differences give different behavior.

The two different constructions in the models are related to and-splits and or-splits and result in a larger space for behavior. The grind transition in the milling case is similar to an and-split [vdAvH02]. It produces tokens in three different places that each enable a branch in the rest of the process. The baking transitions in the baker are similar to an or-split. The tokens in the flour place enable all three baking transitions and the baker has a choice for the rest of the process. There is still a constraint, because if the baker chooses to bake a certain amount of one product it can no longer use the flower for other products. But compared to the miller the space of possibilities is much larger, and measuring the occurrences of one transition is no longer sufficient to determine the process history. This is one fundamental difference in the normal behavior of the two processes.

The information loss caused by the combination of cake and pie into one product is a second difference between the baker and the miller. The construction models information loss during the process. Once the cake and pie are packed they are recorded as pastry and are no longer distinguished as separate products. This shows that it is for example insufficient to log just the sales transitions to record behavior. The space of possible behavior is larger, and to record behavior requires at least three measuring points. In the remainder we show how such a space can be computed.

7.3 NORMAL BEHAVIOR: TOUR SPACES

A key question is what a value net's normal behavior is, given the possible behavior specified by the net's structure. The audit equations capture a value cycle's norms, but as explained in the previous section the equations derived in computational auditing are just a different representation of an audit net and do not give the normal behavior directly. Normal behavior follows from a value cycle's norms, but requires a solution to the equations. The bottler's behavior from the previous section is an example. The net's structure describes what is possible, but obtaining a net's normal behavior involves moving from the problem space to the solution space. This is however only possible in non-financial units of measurement, so the audit equations have to be reformulated without valuation.

The audit equations formulated by Starreveld are of two kinds, laws of proportionality and the so-called BETA equations, but only the BETA equations concern normal behavior. The law of proportionality states that events consume and produce items in fixed proportions. This law corresponds with the definition of behavior in a value net explained in the previous section. In the multiplication with the incidence matrices all consumption and production is in fixed proportions. This means that this law is guaranteed by the Petri net formalism and requires no further equations. The BETA equations are the laws related to normal behavior from which the valuation has to be eliminated in order to get proper conservation properties.

The BETA equations are a kind of conservation laws for an enterprise's accounting system [FdH89; SdMJ88]. A single BETA equation states that the sum of all increases and decreases of an asset or liability in a period of time must equal the difference between the begin and end position. It can be written as

$$B - E + T - A = 0$$

where B is the begin amount, E is the end amount, T (toename) is the increase and A (afname) is the decrease of the amount. The set of BETA equations for all assets and liabilities together forms a system of equations. These equations state a fundamental relationship between production and consumption of assets and liabilities.

An alternative for the BETA equations that is free of any valuation can be stated in the form of a steady state equation. Generally, a steady state equation means that for a transition vector t the produced value $Prod \cdot t$ equals the consumed value $Cons \cdot t$

$$Prod \cdot t \;=\; Cons \cdot t$$

Using the flow matrix this can be written as $Flow \cdot t = 0$. We saw this canceling of production and consumption in the wine bottler's normal behavior from Section 7.2 except that it only holds for the production part. All purchased wine and bottles in the production part are consumed, but at the end the amount of money has increased. If this margin is incorporated into the steady state equation then the normal behavior in a value net is characterized as follows.

$$Flow \cdot t \;=\; \begin{cases} \text{margin} & \text{for places in the finance part} \\ 0 & \text{for places in the operations part} \end{cases}$$

This single matrix equation is equivalent to the system of BETA equations, except it is in non-financial units. It describes that production equals consumption in the operations part of a value net.

A value net's normal behavior is the solution to the steady state equation. A solution to the steady state equation is a constellation of events where the production and consumption in the operations part cancel and whose total effect is thus on money only. Such a solution is called a tour. Since any linear combination of a set of tours is again a tour, the tours form a space. This space can be formulated as the zero space of the value net as follows.

$$\text{TourSpace} = \{t \in T \mid Flow \cdot t = 0 \text{ for the places in the operations part}\}$$

A basis for the tour space is a set of tours that spans the space. That means that any tour is a linear combination of the base tours. The bases can be obtained by computing the net's null space. Such a tour basis characterizes an enterprise's normal behavior as specified by the value net.

Event	t_0
Buy grain	1
Buy paper	10
Grind	1
Package flour	166
Package bran	28
Package germ	6
Sell flour	166
Sell bran	28
Sell germ	6

Event	t_1	t_2	t_3
Sell pastry	0	2,000	5,000
Bake pie	0	200	0
Sell bread	2,500	0	0
Package bread	2,500	0	0
Package cake	0	0	5,000
Package pie	0	2,000	0
Buy paper	20	10	25
Buy flour	1	1	2
Bake bread	250	0	0
Bake cake	0	0	500

Figure 7.7: Base tours for the milling case on the left and the bakery case on the right. The events in the tour produce and consume an equal amounts of goods to create an increase in money. The normal behavior is a linear combination of these basis vectors.

A tour space gives much useful information about a value net's behavior. Figure 7.7 shows the base tours for the miller and the baker. With the bases the normal behavior of the miller and the baker can be described as

$$\lambda_0 \begin{bmatrix} 1 \\ 10 \\ 1 \\ 166 \\ 28 \\ 6 \\ 166 \\ 28 \\ 6 \end{bmatrix}, \quad \text{and} \quad \lambda_0 \begin{bmatrix} 1 \\ 20 \\ 250 \\ 0 \\ 0 \\ 2500 \\ 0 \\ 0 \\ 2500 \\ 0 \end{bmatrix} + \lambda_1 \begin{bmatrix} 2 \\ 25 \\ 0 \\ 500 \\ 0 \\ 0 \\ 5000 \\ 0 \\ 0 \\ 5000 \end{bmatrix} + \lambda_2 \begin{bmatrix} 1 \\ 10 \\ 0 \\ 0 \\ 200 \\ 0 \\ 0 \\ 2000 \\ 0 \\ 2000 \end{bmatrix}$$

with $\lambda_i \in \mathbb{N}$. The normal behavior for each of the two cases is a linear combination of the bases from Figure 7.7. The rank or the size of the basis tells the degree of freedom in the process. In line with the case description from the previous section the miller has one tour and the baker has three. This tells for example that the baker requires at least three points of measurement or recording, but that for the miller only one is minimally required and the rest would be redundant. The event count in the tours reflects the proportionality from the value net. The tour spaces show that switching to non-financial units allows solving the equations. The lambda equations from the previous paragraph are the solution while Frielink's equations from Figure 7.2 state the problem. All information about the net's behavior is hidden in the BETA equations, but revealed by the tour space.

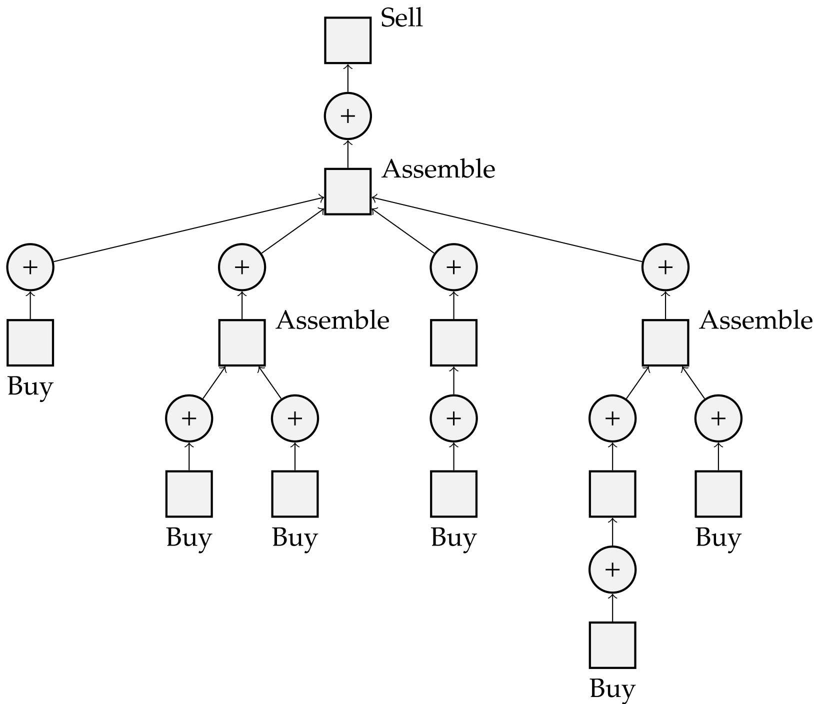


Figure 7.8: Example of the production part of an audit net instance. The audit net grammar limits the production part to a tree with branching at assembly steps and buy transitions at the leaves. The tour space will always contain one tour.

The characterization of normal behavior as a tour space helps explain the difference between the audit net and the value net formalisms. The structure of an audit net's operations part is formed by the grammar rule to add assembly steps to the process. The limited expressiveness of this single rule leads to a tree structure as is illustrated in Figure 7.8. Furthermore, in the audit net formalism the margin is part of the start configuration and maintained by the rules. This makes the margin a structural invariant of the audit net and an explicit part of the model. Viewed from the tour space perspective we see that this structure always gives exactly one tour and that there is thus no need to solve the steady state equation. The audit net grammar limits the models to processes with a tour space of one element. The underlying cause for this limitations is that the audit net grammar cannot express splits in a process. Two examples of value cycle models that cannot be modeled as an audit net are the miller and the baker. Models like these with splits in the process fall outside the class of audit nets as defined by the existing grammar and would require a non-trivial

extension of the formalism. The grammar approach certainly has advantages, but it is unclear how the required grammar extension could be done. With the current grammar the tour space of an audit net always contains exactly one tour.

The following table summarizes the key differences between the audit net and the value net formalization of the value cycle.

Value cycle	Audit net	Value Net
Conceptual model	Formal model	Formal model
Any process structure	Process limited to tree structure	Any process structure
Equations in financial units	Equations in non-financial units	Equations in non-financial units
No process behavior defined	Normal behavior restricted to one tour	Tour space gives normal behavior as solution to equations
Margin is problematic	Margin modeled as structural invariant	Margin computed for each tour

The value cycle is a conceptual model without any formally defined structure or behavior. The equations relate to accounting systems and are subject to valuation and therefore the margin is problematic. The equations in the audit net formalism are in non-financial units of measurement and explicitly model the margin. This avoids issues with the valuation, but assumes the process has a tree structure. The value net formalism reformulates the audit equations to a steady state equation in non-financial units of measurement. This allows the definition of normal behavior as a tour space without any limitation on the process structure.

7.4 TOUR FACTORING AND CAUSAL CHAINS OF EVENTS

A tour gives just the number of occurrences for each event, but if various parts are factored then an ordering of events can be identified similar to Ijiri's causal chains of events. Ijiri stresses that financial journals in an accounting system do not occur in isolation but form chains of events that follow from causality in enterprise behavior [Iji65; Iji67]. Similarly the events in a tour in a value net form a causal chain. Figure 7.9 shows the chain of events for the miller. It is a complete closed-loop chain from money to money that tells which products are bought and produced and in what amounts.

The tour computation from the previous section showed that the miller from Figure 7.5 has a tour space of one tour. The factoring of this tour is displayed in Figure 7.9. The parts the factored tour show the following stages in the causal chain:

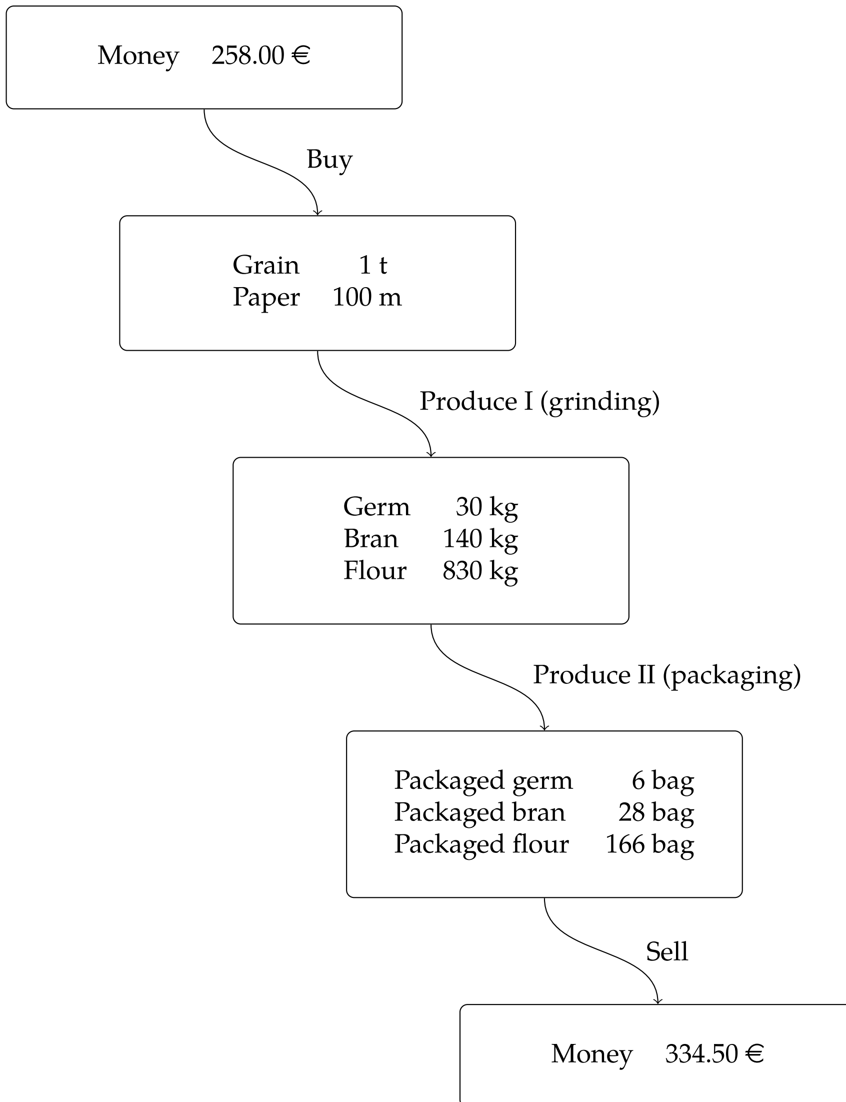


Figure 7.9: The factored tour for the milling case from Figure 7.7. From top to bottom it shows the sacrificed money, the purchased goods the intermediate goods, the sold goods, and the generated money.

1. Sacrificed monetary value
2. Purchased goods/materials
3. Intermediate products
4. Sold products or services
5. Generated monetary value

The sacrificed monetary value is 258 €. It is used in the buy event to purchase grain and paper. Grinding the grain gives the intermediate goods germ, bran and flour. The miller's intermediate goods germ, bran and flour are packaged into products to be sold. These final products together generate 334.50 € monetary value. All these parts form a chain of events that can be obtained by factoring the tour.

A tour can be factored into a causal chain by factoring the value net's flow matrix, but it requires some care with the place sign. The result of tour t is the matrix product $Flow \cdot t$. By definition this is a vector with zeros in the operations part, so that doesn't provide any insight. To see the effect of a tour the result is split into the produced and the consumed quantities for different parts of the value net. To achieve this the incidence matrix *Flow* is factored as

$$\begin{aligned} Flow &= Prod_{fin} + Prod_{buy} + Prod_{op} + Prod_{sell} - \\ &\quad Cons_{fin} - Cons_{buy} - Cons_{op} - Cons_{sell} \end{aligned} \tag{7.1}$$

Matrices *Prod* and *Cons* are the production and the consumption matrices, and the suffixes *fin*, *op*, *buy* and *sell* mean that the columns of the matrix are filtered for the transitions of that kind. For example matrix $Cons_{buy}$ is the consumption incidence matrix filtered for the buy transitions. The matrices are a convenient way to identify the effect of a tour in different parts of the value net. For example the sacrificed monetary value for some tour t is $Cons_{buy} \cdot t$. A minor technical point of attention is that for liabilities production and consumption are reversed. For example a buy event can consume value from the finance part by consuming money, but also by producing payables. For now, when we say a value is consumed we mean an asset is consumed or a liability produced, but we have to address that later on. With that proviso we can use the matrices to factorize a tour into a causal chain.

With the factored flow matrix the equation for the tour space from the previous section can be rewritten into the following two equations

$$Prod_{buy} \cdot t + Prod_{op} \cdot t - Cons_{op} \cdot t - Cons_{sell} \cdot t = 0 \tag{7.2}$$

$$Prod_{sell} \cdot t + Prod_{fin} \cdot t - Cons_{fin} \cdot t - Cons_{buy} \cdot t = Flow \cdot t \tag{7.3}$$

The first equation is the steady state equation for the operations part and the second equation gives the value jump or the margin. The first law holds because the four terms together are exactly the tour's effect in the operations part of the net and thus zero by the definition of a tour. The second equation contains the remaining parts and is exactly the financial part. Its effect is the margin.

The purchased and sold goods are the terms $Prod_{buy} \cdot t$ and $Cons_{sell} \cdot t$ in Equation (7.2). In the factored tour in Figure 7.9 the purchased goods are the grain and paper in the second rectangle. The produced products are $Prod_{op} \cdot t$ and the consumed ones are $Cons_{sell} \cdot t$. Some of these are intermediate products. Intermediate goods are produced and at some later point consumed again by the production process. Not every value net has intermediate goods. A trading company for example has no transformation process and no intermediate products. In that case the terms $Prod_{op} \cdot t$ and $Cons_{sell} \cdot t$ are zero. When intermediate products exist they can be computed by $Prod_{op} \cdot t - Cons_{sell} \cdot t$ or $Cons_{op} \cdot t - Prod_{buy} \cdot t$. The first is all products that are produced but not sold. The second is all consumed products that are not purchased. Both computations give the same result which is all products that are neither bought nor sold.

The sacrificed and generated values are the terms $Cons_{buy} \cdot t$ and $Prod_{sell} \cdot t$ in (7.3). The remainder $Prod_{fin} \cdot t - Cons_{fin} \cdot t$ consists of money movements only and is therefore zero. It is assumed that monetary transactions are always balanced and free from valuation so they are just trivial money movements. This leaves open how the transaction in the money part behaves, but in future research this could be refined. As long as all tokens end up in the monetary part there is no effect on the margin. The difference $Prod_{sell} \cdot t - Cons_{buy} \cdot t$ is the difference in money spent and received. The sum of this vector is is the margin or value jump.

7.5 ACCOUNTING

Accounting is the process of recording, classifying, and summarizing economic events in a logical manner for the purpose of providing financial information for decision making [McC82; Pac94; RS06]. Characteristic of the accounting systems is that it records the value of assets and liabilities in accounts. The quantity of each account shows the value present of that kind, and a transaction or event is recorded by a journal entry that describes how much each account changes. The chart of account determines the reporting capabilities of the accounting system. Various accounting methods exist, of which the most well-known and widely used is Pacioli's double-entry bookkeeping method. All methods vary in the procedures and recorded quantities but share the notion of an account.

The value cycle is closely related to the double-entry bookkeeping method, and the question is what the shift to physical units in value nets means for this relationship. The connection to accounting systems is for example evident by the payables and receivables in the extended net. Payables and receivables are typical accounting artifacts, and we would like to reason about accounts like these using the value net. However, in the operations part of the value net the equations can no longer be directly related to accounting information because the net is in units of measurement. Accounting systems record events in monetary units, and it was already discussed

that that is not compatible with conservation laws. A related issue is that accounting systems suffer from information loss and are therefore not suitable as first recording. The shift to quantities in units of measurement in value nets solves these issues, but requires a reconstruction of the mapping to accounting systems. This requires multiplication of quantities in units of measurement with a valuation. Doing this turns out to be equivalent to property accounting as suggested by Ijiri and Ellerman [Ell82; Ell86; Iji65; Iji67].

Given some valuation, a value net's behavior can be linked to accounting systems by associating accounts with the net's places and deriving journal entries from the net's transitions. A journal entry contains a credit journal line for each place from which the transition consumes, and a debit journal line for each place to which the transition produces. For example the *Buy flour* transition from the bakery in Figure 7.6 could be mapped to the following journal entry:

	Debit	Credit
Flour	250.00 €	
Payables		250.00 €

and the *Buy paper* transition to the following journal entry:

	Debit	Credit
Packaging paper	25.00 €	
Payables		25.00 €

Note that these journal entries do not record the quantities of the purchased products, just the value, and that this journal assumes a valuation of flour of 250 €/t and a valuation of paper of 1.25 ¢/m. The valuation chosen matches the purchase price exactly. This valuation is called a historic price and conveniently balances the journal entries.

Intermediate production steps can be mapped to journals just like buy events. In practice it might not be useful to set up an account for baked bread, but systematically following the value net gives journal entries for *Bake bread* and *Package bread*, for example:

	Debit	Credit
Bread	250.00 €	
Flour		250.00 €

and

	Debit	Credit
Packaged bread	275.00 €	
Packaging paper		25.00 €
Bread		250.00 €

Again the valuation has been chosen in such a way that the journal entry is balanced. The valuation 11 ¢ of the produced packaged bread is chosen to exactly match the sum of the consumed bread and paper value. Note that this requires knowledge about the amount of bread and paper that is not in the accounting system. Double bookkeeping keeps track of the value of goods, but the amount of goods has to be stored in some inventory system or derived in some other way.

For the sales events the journal entry can no longer be balanced by choosing a suitable valuation, which shows again that the principle of double entry bookkeeping is incompatible with a conservation law for the value cycle. Remember that from the intermediate products the valuation of the packaged bread is 11 ¢, but when this packaged bread is sold it gives an income of 35 ¢. If the journal entry uses the sales prices, just as we did for the purchase price, then it doesn't match the value of consumed intermediate goods and consequently the inventory and the accounting system would no longer match. But if it uses the intermediate goods' value, then the transaction doesn't balance. This is where profit and loss accounts come into play. The difference in value is placed on separate accounts that track profit and loss. In this way the transaction can always be balanced. A journal for *Sell bread* could look like:

	Debit	Credit
Money	875.00 €	
Packaged bread		275.00 €
Profit		600.00 €

The sales events generate 875.00 € income in total. The value of the sold bread is 275.00 €. The difference between the two is the margin and has to be recorded on a profit and loss account. The great feature of the zero-sum of journal entries is that all mutations in value are accounted for. In this way it guarantees that the margin is accounted for properly as a profit. However, by accounting for the value jump it loses a conservation property. In the context of equations the profit or loss account acts as an extra variable. While this is useful for accounting, it harms the conservation property and the usefulness of the equations.

The example shows that accounting systems are not suitable for auditing or accounting equations that are derived from the value cycle. The example shows that the valuation is completely arbitrary. Currently the purchased and intermediate products are valued against purchase price, but they could for example also be valued against sales price, or any other price. The only difference is the moment the value jump is recorded in the profit and loss accounts. The valuation has no effect on the margin at all. The example shows that the balancing of accounts in double-bookkeeping is a powerful feature that guarantees that value is accounted for, but the use of a monetary unit leads to loss of information that weakens the applicability of the equations. Switching the value net to units is better for the equations but requires mapping back to accounts.

When we combine account vectors from Rambaud et al [RPN10] and the units of Ijiri and Ellerman into dimensioned vectors, then the classification and reporting capabilities of accounting can be reconstructed. To accommodate quantities in units of measurement we want to model accounting information as dimensioned vectors. A formalization of the double-entry bookkeeping method using account vectors is given by Rambaud et al [RPN10]. The vectors are called balance vectors and capture the characteristic property of the double-entry method that the entire accounting system is always balanced. These vectors are however still monetary. Vectors are also used by Ijiri and Ellerman to enable a transition from value accounting to property accounting [Ell82; Ell86; Iji65; Iji67]. In contrast to value accounting where any number is in monetary units, property accounting takes care that quantities with different units are kept individually in a vector. To recover value requires multiplication with a price vector. Ellerman [Ell86] suggests:

> *... a model of double entry multidimensional accounting in 'physical terms' using vectors of property rights. Property accounting gives a valuation-free description of the property transactions underlying the value transactions of ordinary accounting. Thus it avoids the valuation controversies of value accounting. Given any vector of valuation coefficients (e.g. prices or costs), a system of value accounting can be derived from a valuation-free system of property accounting by multiplying the property vectors by the value vector.*

Remarkably Ijiri and Ellerman do not use account vectors but introduce a vector per account. The idea is to replace a single number by multiple numbers of varying dimensionality. When we combine account vectors and Ijiri and Ellerman's property accounting into dimensioned vectors then we get a representation that maps well on value nets.

The commuting diagram from Figure 7.10 shows the relationships between the various vector representations of data in the accounting process. The diagram's nodes are Pacioli types and the arrows represent linear transformations between vectors of the connected nodes' types. The type *Event!* in the top left stands for event vectors. The tour vectors used earlier are examples of event vectors. Each vector entry tells how often a certain event is performed. The diagram's bottom right node stands for the summarized accounting information corresponding to the events. It denotes a report where the information is grouped in a convenient form to show the effect of the events in a condensed form. The path via the bottom left represents traditional accounting and the path via the top right represents property accounting. The register arrow in the accounting path computes the effect of the event on the accounts and represents the construction of journal entries. This is a linear transformation from an event vector to a account vector. Concretely this linear transformation is a matrix that tells the effect of an event on the accounts. The summarize transformation aggregates an account vector into a group vector. The combined transformations on this path

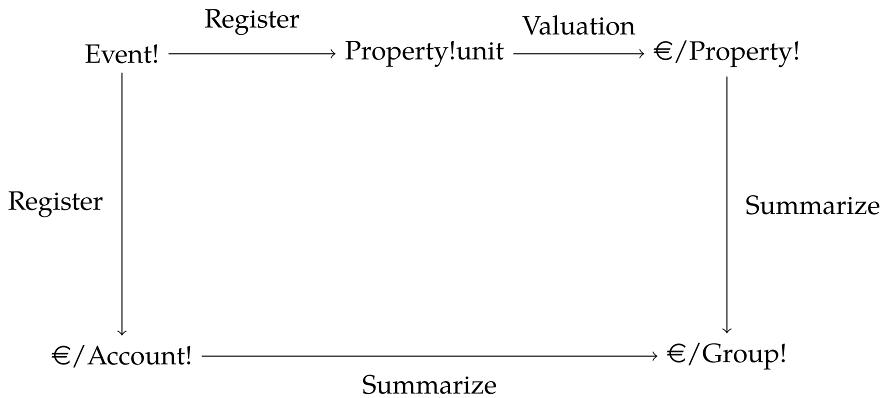


Figure 7.10: Commuting diagram that shows the difference between value accounting and property accounting. for a value accounting system. The nodes are types of the vectors and the arrows are linear transformations.

perform the recording, classification and summarizing of transaction as found a traditional accounting system. Instead of directly registering the event's effect in accounts, the property accounting path registers in quantities in the dimensioned vector *Property!unit*. The register arrow is similar to the other one, except it is in units of measurement instead of money. A separate valuation transforms the quantities into property value. These two steps together are essentially how the journal entries in value nets are formed. After that a similar summarization leads to the final summary.

The commuting diagram shows the relation between value nets and property accounting. Working with units of measurement as in value nets is more complex than working with the convenient single monetary currency from accounting. The commuting diagram shows how all property can be registered separately in the physical unit.

Dimensioned vectors conveniently express the relation between the value net and accounting systems. Handling the units of measurement prohibits aggregating data thus requiring distinct numbers. Vectors and matrices are convenient tools to calculate with such data. Dimensioned vectors allow handling of units of measurement in value nets with the convenient notation of linear algebra.

7.6 FRAUD ANALYSIS

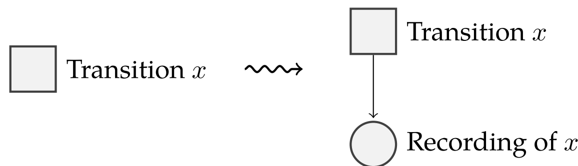
A key aspect of the theory of computational auditing is the computation of the spaces of potential fraud scenarios in a value cycle. As explained in Section 7.1.2, our definition of fraud excludes trivial cases that leave traces and are thus easy to

detect. It focuses on the more advanced cases that leave no traces and can only be prevented at design time. So far the value nets are free from illicit behavior. These normative models describe the 'soll' or 'should be' situation. Fraud analysis requires the extension from the 'soll' net to an 'ist' or 'as is' net. The 'ist' model adds illicit behavior that makes it possible to reason about fraud scenarios. A main result from computational auditing is the idea that from this 'ist' model a complete set of potential fraud scenarios can be computed.

The fraud analysis computation is computationally too expensive for practical purposes and the question is whether insights in the fraud space might give ideas for improvements. The fraud analysis computes the base of a value net's null space. This base grows exponentially with the number of nodes in the value net and computing it becomes quickly unfeasible. The high level value nets are generally not very large, but even small models already lead to excessive computation times. The question is whether it is necessary to compute these complete bases, or that there is a lot of less practical output whose computation can be skipped. In the latter case a practically feasible fraud analysis might be possible.

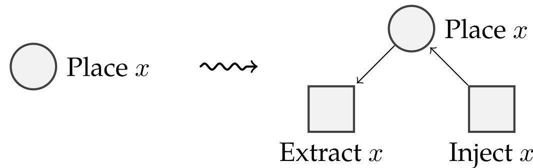
Ordering the base scenarios according to the number of licit events in them gives a picture that suggests that computing the complete base is not necessary, but that any claims about fraud absence depends on how accurate the value net describes the 'as is' situation. The fraud analysis from computational auditing generates the worst case 'as is' model. Combined with the fact that every event has a registration this leads to a large fraud base. If the worst case assumption is relaxed and instead the actual situation is modeled the computation is often greatly reduced. Experiments suggest that analysis per organization function like sales or production is a feasible strategy. The usefulness of any analysis depends however on accurate and detailed modeling. For example, fraud scenarios in the sales process involving discounts will never be found if discounts are not modeled. So, instead of computing scenarios for the overall process, it appears better to focus on modeling the 'as is' situation accurately for partial processes.

An 'as is' value net is constructed by extending the soll net with transaction recordings and illicit actions. The following transformation adds a recording to a transition.



The transaction recording is a counter that is increased every time an event occurs and represents a minimal registration that can be used to, for example, check the integrity of the inventory. Using this rule it is possible to model recording points in

the process and see if they provide sufficient segregation of duties. The fraud analysis also requires that illicit behavior is modeled. The following transformation adds illicit actions to manipulate value net places.



The added actions model the illicit insertion and extraction of a place's value. These two actions are sufficient to model any manipulation to the contents of a place.

The null space of the 'ist' value net is the net's space of fraud scenarios. The idea is that the null space contains all undetectable scenarios, because a scenario from the null space has zero effect. In the soll case events with zero effect are non-existing, but since the 'as is' net contains illicit actions, a constellation of event with total zero effect can be used to manipulate values. To see how this generates potential fraud scenarios, consider the fraud scenario as represented by the following event vector for the baker case.

Event	Count
Inject pie	1
Inject packaging paper	50
Package pie	1
Extract recording of package pie	1
Sell pastry	1
Extract recording of sell pastry	1
Extract money	25

The vector shows the sales of a pie that was not produced in the bakery, but brought in from somewhere else. Replaying the scenario shows that the total effect is indeed zero. The pie is sold, all transaction recordings are removed, and the profit is extracted. Figure 7.11 breaks down the effect of the scenario. The upper left numbers are the normal effect of the events, and the upper right the recording of them. The illicit actions cancel these numbers for every column and makes the total effect zero. This scenario could be a shop owner that tries to hide some business under the counter from for example the tax office. Or it could be an employee that runs an illicit shop in shop. This is a well known fraud where an employee or group of employees uses an employer's facilities and resources to earn extra income. This is a serious risk for example for franchises and receivers of royalties or taxes. The scenario shows how the illicit actions hide the effect of the licit actions. The null space gives all combinations of licit and illicit actions that are not detectable.

	Money	Packaging paper	Pie	Packaged pastry	Recording of package pie	Recording of sell pastry
Package pie	-	-50 cm	-1 pie	1 pastry	1	-
Sell pastry	25 c	-	-	-1 pastry	-	1
Inject pie	-	-	1 pie	-	-	-
Inject packaging paper	-	50 cm	-	-	-	-
Extract recording of package pie	-	-	-	-	-1	-
Extract recording of sell pastry	-	-	-	-	-	-1
Extract Money	-25 c	-	-	-	-	-
Total effect	0 c	0 cm	0 pie	0 pastry	0	0

Figure 7.11: A breakdown of the effect of the bakery fraud scenario. The events in the rows are split into the licit and illicit ones, and the places in the columns are split into regular places and transaction recordings.

A value net usually has an infinite number of fraud scenarios, but a set of representative scenarios can be obtained by computing the base of the net's null space. The fraud space is the set

$$\{t \in T \mid \text{effect of } t \text{ in the 'ist' value net is zero}\}$$

Vector t contains the event count just as in the tour vectors earlier. If fraud is possible in a value net then there will be an infinite number of fraud scenarios because any multiple of a fraud scenario is again in the null space and thus also a potential fraud. The infinite null space is a vector addition system that has a finite basis from which all scenarios in the space can be generated. A base itself cannot be constructed by combining other vectors, but given base vectors $t_0, \ldots, t_n$, any fraud scenario is a linear combination of these base vectors:

$$a_0 \cdot t_0 + a_1 \cdot t_1 + \cdots + a_n \cdot t_n$$

with $a_i \in \mathbb{N}$. Computing the bases for the examples gives a base of 237 scenarios for the baker and 635 scenarios for the miller. These are already a large number of base scenarios for such small models, so any reduction in the number of base vectors would be welcome for any practical application.

To gain insight into the structure of the fraud space we order the base scenarios according to the number of licit events involved. The trivial cases are the scenarios without licit events. Any 'as is' value net's base contains one such scenario for each place, for example for the money place:

Event	Count
Inject money	1
Extract money	1

This scenario inserts one unit in the money place and also extracts it. These scenarios are not meaningless if one considers that the extraction and insertion may occur at difference times. Value that is out of control of an organization, even if just temporarily, is a risk, especially from an auditing point of view, when inventory is taken but contents are shifted. Since there are no single events with a zero effect, these trivial scenarios are the smallest base vectors.

For each licit event there is a fraud scenario with just that licit event and no other licit events. When a licit event happens it generates a licit effect. This can be turned into a fraud scenario by adding the right insert and extract events that compensate the licit effect. For the running example it is illustrative to look at the scenarios for the two licit actions *Package pie* and *Sell pastry*. The fraud scenario discussed in detail earlier is the combination of these two actions. The fraud scenario that packages one pie and undoes its effect is

Event	Count
Inject pie	1
Inject packaging paper	50
Package pie	1
Extract recording of package pie	1
Extract packaged pastry	1

Just as in the combined scenario the event consumes a pie and 50 cm packaging paper so these have to be inserted. The packaged pie and the recording of the event have to be extracted. Similarly the fraud scenario for selling of pastry is constructed.

Event	Count
Inject packaged pastry	1
Sell pastry	1
Extract recording of sell pastry	1
Extract money	25

Pastry is sold and the effect is completely undone by illicit actions. Since these scenarios have just one licit event they cannot be created by combining other smaller

scenarios, so they must be base fraud scenarios. There is a base fraud scenario like these two examples for each soll transition.

The base fraud scenarios with two events are all pairs of events that are directly connected via some place. The scenario from the running example is an example of a base with two licit events. It was already shown that it is a fraud scenario because its effect is zero, but to see that it is a base vector we have to check that it cannot be constructed from other scenarios. The only two candidates to construct it from are the two scenarios from the previous paragraph. The combination of these two bases is identical to the fraud scenario, except it additionally has the *Inject packaged pastry* and *Extract packaged pastry* events. In the running example these two illicit actions are not needed because the produced packaged pastry is consumed by the sell event. If the two events were not connected, then the scenario could be constructed by combining two scenarios into one event. But in the connected case one of the events consumes the output from the other, and therefore the scenario is not a combination of other base vectors, but a base vector itself. Generally any producing and consuming pair of events forms a base vector. Such a pair cannot be constructed from single base vectors, and in the same way as for the single event case, the licit effect of the two events can be undone by the appropriate insert and extract actions.

Similar to the case with two licit events, base vectors in the case with three licit events are sets of connected transitions of length three, and so on for higher numbers, until the scenarios form a complete cycle. The reasoning is the same as in the case of two licit events. If a number of events are connected, then the corresponding fraud scenario cannot be constructed from smaller bases, so it must be a base scenario. When the bases eventually reach the complete cycle it becomes more complex and the combinations grow fast. The complete picture is that the computation of the fraud space generates connected components that grow larger and larger until they get so large that the computation becomes unfeasible.

The structure of the fraud space suggests a strategy that computes the null space for each authorization or capability profile instead of the worst case computation from computational auditing. In an audit net each transaction has a transaction recording by construction and when the 'as is' audit net is constructed for the fraud analysis each place is given the illicit insertion and extraction actions. In a next step each scenario is examined to see if it really is feasible for agents to perform the scenario's steps. This worst case strategy is attractive because the effect of changes in agents' authorization can be evaluated without recomputing the base set of scenarios. A disadvantage however is that many unnecessary scenarios are computed. A strategy that restricts the value net for a given authorization profile avoids these unnecessary scenarios.

The alternative strategy is viable when there is no need to compute the complete fraud space. As long as authorization profiles are limited the alternative strategy

performs better because the reduction in the exponential computation outweighs having to do the computing per authorization profile. This reduces the required computational effort to practically acceptable values.

Some first investigations with various models confirm the cost reduction of the computation, but suggest that a practical application would need more detailed modeling to be of interest. Analysis of the space showed that incorporation of agents' authorization is feasible and promising. For example limiting the computation to just the part of the value net that is accessible to sales personnel gave just a handful of scenarios. Although the scenarios were correct, they were also generic. Selling a product is not much different for different kinds of products. Interesting situations are constructions with discount, refunds, rebates, reimbursement, etc. Similarly with purchases there might be fraud with different qualities of products. If these different qualities are not modeled, then these frauds will not be found. The effectiveness of more detailed and accurate modeling of processes is an interesting topic for further research.

7.7 CONCLUSION

The value nets introduced in this chapter model an organization's value-creation process in non-financial units of measurement. The models are a formalization of the value cycle concept from auditing. Formalization of the value cycle started with the development of audit nets in Computational Auditing. Value nets generalize the audit net models to increase the possibilities for use in analysis and design of organizations.

The formulation of value nets in non-financial units eliminates the valuation from the value cycle and allows the characterization of normal behavior by tour spaces. The use of non-financial units allows the definition of normal behavior as the solution to a steady state equation. The solution to the equation is called the value net's tour space. Behavior in non-financial units of measurement is related to Ellerman's property accounting and can always be mapped back to an accounting system. Eventually the detail information is condensed and this requires a single units, just like financial accounting does.

The introduced aspects of value nets will be defined formally and implemented in the Pacioli language in Chapters 8 and 9. These two chapters define the value net data structure and give the details of the analysis of a value net's behavior. The second part closes with a case study in Chapter 10.

8 VALUE NET STRUCTURE

This chapter defines the structure of unit-aware value nets that was informally explained in the introduction and gives a unit-aware implementation in Pacioli. The behavior of value nets is discussed separately in the next chapter.

8.1 VALUE NET DEFINITION

A value net is a unit-aware Petri net extended with a valuation that is constructed according to the rules of a value cycle. The definition uses a linear algebra formulation of a Petri net. In that case the Petri net formalism becomes a vector addition system [Reu90].

Definition 8 *A* Value Net *is a tuple*

$$V = (P, T, Cons, Prod, val)$$

with P a finite collection of places, T a finite collection of transitions, $Cons$ and $Prod$ incidence matrices of the form $P \times T \rightarrow \mathbb{N}$ and val a vector of the form $P \rightarrow \mathbb{Z}$, where

- *$Cons_{ij}$ is the cardinality of the arrow from place i to transition j*
- *$Prod_{ij}$ is the cardinality of the arrow from transition j to place i*
- *val_i the valuation of place i*

Furthermore, the places are partitioned into disjunct sets P_{fin} and P_{op}

$$P = P_{fin} \cup P_{op} \tag{8.1}$$

with P_{fin} the places in the value net's finance part and P_{op} the places in the operations parts. The places are also partitioned into disjunct sets P_{asset}, P_{liab} and P_{rec}

$$P = P_{asset} \cup P_{liab} \cup P_{rec} \tag{8.2}$$

with P_{asset} the assets, P_{liab} the liabilities and P_{rec} the recordings. Value net transitions T are partitioned as licit and illicit. This will become relevant when the fraud scenarios are computed.

$$T = T_{lic} \cup T_{ill} \tag{8.3}$$

Another partitioning of transitions is derived from the place partitioning into finance and operations places. Transitions can be partitioned in the same way, except that buy and sell transitions do not fall exclusively in either category because they form

the border of finance and operations. They form their own category, resulting in four kinds of transactions.

$$T = T_{fin} \cup T_{op} \cup T_{buy} \cup T_{sell} \tag{8.4}$$

Transitions in T_{fin} are between places in the finance part, and transitions in T_{op} between places in the operations part. A T_{buy} transition consumes value from the finance part and produces value in the operations part. A T_{sell} transition consumes value from the operations part and produces value in the finance part. It is assumed that value nets are constructed in such a way that every transition is of one of these four types.

A value net's behavior is the usual firing of transitions from the underlying Petri net $(P, T, Cons, Prod)$. The places in a net can contain tokens. The state of a net is the amount of tokens in each place. A vector representing a net's state is called a marking.

Definition 9 *Let P be the places in some value net. A* marking *is a $P \longrightarrow \mathbb{N}$ vector. It denotes the number of tokens in each place.*

The transition between states is caused by firing of the net's transitions.

Definition 10 *Let $V = (P, T, Cons, Prod, val)$ be a value net with some marking m. A transition vector $t : T \longrightarrow \mathbb{N}$ is enabled when $Cons \cdot t \leq m$. When it fires it consumes $Cons \cdot t$ tokens and it produces $Prod \cdot t$ tokens.*

It is convenient to define flow matrix $Flow = Prod - Cons$. The effect of transition vector t is then the single matrix product $Flow \cdot t$, and the state changes from marking m to marking $m' = m + Flow \cdot t$. The notation $m \xrightarrow{t}_V m'$ is used to indicate that transition vector t changes the state of value net V from from marking m to marking m'.

8.2 UNIT-AWARE PETRI NETS

Before value nets are implemented, a separate implementation of unit-aware Petri nets is given. A unit-aware Petri net associates a unit of measurement with each place. The following type alias defines a unit-aware Petri net as a tuple with a consumption incidence matrix, a production incidence matrix and a vector containing the net's marking. The unit vector `Place!unit` contains the places' units.

```
deftype for_index Place, Trans: for_unit Place!unit:
  PetriNet(Place!unit, Trans!) =
    Tuple(Place!unit per Trans!, Place!unit per Trans!, Place!unit);
```

The actual type of the created tuple is quite large and it would be preferable to hide the details. The type alias is a simple mechanism to achieve this.

With the type alias the constructor `make_petri_net` can be defined. It creates a Petri net from its arguments. The arguments must be of the proper units.

```
declare make_petri_net :: for_index P,T: for_unit P!u:
  (P!u per T!, P!u per T!, P!u) -> PetriNet(P!u, T!);

define make_petri_net(pre, post, marking) =
  tuple(pre, post, marking);
```

Function `tuple` creates a tuple containing the Petri net parts. Matrix `pre` is the consumption incidence matrix and matrix `post` is the production incidence matrix.

The following functions give a net's incidence matrices and its marking. The implementations return the proper element from the tuple.

```
declare petri_pre, petri_post, petri_flow ::
  for_index P,T: for_unit P!u: (PetriNet(P!u, T!)) -> P!u per T!;

declare petri_marking :: for_index P,T: for_unit P!u:
  (PetriNet(P!u, T!)) -> P!u;

define petri_pre(net) =
  let (p, _, _) = net in p end;

define petri_post(net) =
  let (_, p, _) = net in p end;

define petri_marking(net) =
  let (_, _, m) = net in m end;

define petri_flow(net) =
  petri_post(net) - petri_pre(net);
```

The construction `let (x, y, ...) = t` is a Pacioli facility to destructure a tuple. The variables `x, y, ...` between the parenthesis get bound to the elements of tuple `t`. An underscore is a special variable that indicates that that tuple value is not used.

Given some transition vector a Petri net can be fired. Function `petri_net_enabled` tests if there are enough tokens in each place for the net to fire.

```
declare petri_net_enabled :: for_index P,T: for_unit P!u:
  (PetriNet(P!u, T!), T!) -> Boole();

declare fire_petri_net :: for_index P,T: for_unit P!u:
  (PetriNet(P!u, T!), T!) -> PetriNet(P!u, T!);

define petri_net_enabled(net, amount) =
  petri_pre(net) '*' amount <= petri_marking(net);

define fire_petri_net(net, amount) =
  let marking = petri_marking(net) + petri_flow(net) '*' amount in
    make_petri_net(petri_pre(net), petri_post(net), marking)
  end;
```

These functions implement a Petri net's behavior.

8.3 VALUE NET IMPLEMENTATION

The value net data structure is a tuple that is equivalent to the mathematical definition from the opening section, except that it uses bit vectors to classify the net's elements. Since the value net definition uses a linear algebra formulation of Petri nets it can directly be translated into a Pacioli tuple of vectors and matrices. For the classification of places as financial or operational we also want to use Pacioli's type system, but since there are no boolean vectors we use the common practice from mathematics to use bit vectors. A vector with a one if a place is from the monetary part and zero if it is from the production part encodes the monetary production classification. Using similar bit vectors for the other value net properties the definition can be implemented type correct as a Pacioli tuple.

The implementation of the value net structure defines a constructor and observers similar to the Petri net implementation. The following type definition for value nets `Net(mon, P!u, T!)` is an alias to enhance readability.

```
deftype for_index P,T: for_unit mon, P!u:
  Net(mon, P!u, T!) =
    Tuple(PetriNet(P!u, T!), mon/P!u, P!, P!, P!, T!);
```

The code in the next fragment uses it for the construction of a value net. It has a type declaration for constructor `make_net` and an implementation of that constructor.

```
declare make_net :: for_index P,T: for_unit mon, P!u:
  (P!u per T!, P!u per T!, mon/P!u, P!, P!, P!, T!)
    -> Net(mon, P!u, T!);

define make_net(pre, post, val, is_fin, is_asset, is_rec, is_lic) =
  let pn = make_petri_net(pre, post, 0 _* row_unit(pre)) in
    tuple(pn, val, is_fin, is_asset, is_rec, is_lic)
  end;
```

The type of the incidence matrices is `P!u per T!`, a place by transition matrix in proper units of measurement. From these matrices a Petri net is created. The initial marking for the Petri net is a zero place vector with the proper units. The valuation `val` has monetary type `mon` per place unit `P!u`. The last four arguments are three place bit vectors and one transition bit vector. The constructors puts all these elements in a tuple.

The next observers retrieve the incidence matrices and valuation that were used to construct a value net. The declarations and definitions are as follows.

```
declare net_pre, net_post, net_flow ::
  for_index P,T: for_unit mon, P!u:
    (Net(mon, P!u, T!)) -> P!u per T!;

declare valuation :: for_index P,T: for_unit mon, P!u:
  (Net(mon, P!u, T!)) -> mon/P!u;

define net_pre(net) =
  let (pn, _, _, _, _, _) = net in petri_pre(pn) end;

define net_post(net) =
  let (pn, _, _, _, _, _) = net in petri_post(pn) end;

define net_flow(net) =
  let (pn, _, _, _, _, _) = net in petri_flow(pn) end;

define valuation(net) =
  let (_, val, _, _, _, _) = net in val end;
```

The incidence matrices are fetched from the Petri net. The observers use the let construct to retrieve the proper tuple element.

The remaining observers retrieve the bit vectors that classify the places and the transitions. Their types are given in the next fragment.

```
declare is_finance, is_asset, is_recording ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!)) -> P!;

declare is_licit:: for_index P,T: for_unit mon, P!u:
  (Net(mon, P!u, T!)) -> T!;

declare is_buy_trans, is_sell_trans, is_fin_trans, is_op_trans ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!)) -> T!;
```

The type declarations state that the defined functions `is_finance`, `is_asset` and `is_recording` return a dimensionless place vector and that function `is_licit` returns a dimensionless transition vector. The implementations return the corresponding tuple element that was passed during construction, just like the other observers, and have been omitted for brevity. The implementation of the transition classification derives the vectors from the production classification. The implementation involves uninteresting manipulation of signs and bit vectors and has also been omitted for brevity's sake.

The basic data structure is completed with logical complements of the place and transition bit vectors. With the defined functions up to now all tuple elements can be obtained, but it is convenient to have names for the complements of the logical properties. The next code fragment contains the complements of the four bit vectors in the value net.

```
declare is_real, is_operations, is_liability ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!)) -> P!;

declare is_illicit ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!)) -> T!;

define is_real(net) =
  complement(is_recording(net));

define is_operations(net) =
  complement(is_finance(net));

define is_liability(net) =
  complement(is_asset(net)) * is_real(net);

define is_illicit(net) =
  complement(is_licit(net));
```

The functions satisfy the classification of places and transitions given in the value net definition and equations (8.2) and (8.3). Vector `is_real` is a bit vector that denotes all places that are not recordings. It is computed using the logical negation `complement` that was introduced in Section 6.3.3. Vector `is_operations` denotes the places in the operations part. The complement of the assets can besides the liabilities also be recordings so in the definition of `is_liability` they are excluded by multiplying with the `is_real` vector. Transition vector `is_illicit` is the complement of vector `is_licit`.

8.4 VALUE NET PROPERTIES

Besides the value net constructor and observers from the previous section, the data type contains various additional functions to access the value net structure.

8.4.1 *Index Sets*

The net's places and transitions are derived from the incidence matrices. The places and transitions are not explicitly passed as arguments to the constructor, but they follow from the type of the incidence matrices and the bit vectors. In a value net of type `Net(mon, P!u, T!)` the places are the index set `P` and the transitions the index set `T`. These sets can be accessed with Pacioli's standard functions. The following fragment defines the observers `net_places` and `net_transitions`.

```
declare net_places ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!)) -> List(P);

declare net_transitions ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!)) -> List(T);

define net_places(net) =
  row_domain(net_flow(net));

define net_transitions(net) =
  column_domain(net_flow(net));
```

Function `net_places` returns the row domain of the net flow. Type `List(P)` shows that this yields a list of places. Function `net_transitions` returns the column domain which gives a list of transitions.

8.4.2 *Units of Measurement*

All relevant information on units of measurement and vector spaces can be obtained from the value net data structure. In in type definition `Net(mon, P!u, T!)` unit `mon` is the currency in the monetary part and `P!u` is the unit of the places. Dimensionless `T!` is the unit of transitions.

Using standard Pacioli functions, all three arguments can be accessed. The functions in the following code give access to the unit information.

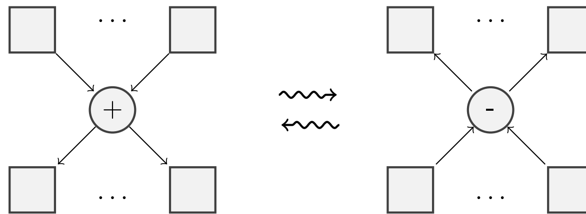
```
declare net_currency ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!)) -> mon;

declare net_unit ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!)) -> P!u per T!;

declare net_place_unit ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!)) -> P!u;

declare net_transition_unit ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!)) -> T!;

define net_currency(net) =
  unit_factor(valuation(net));

define net_unit(net) =
  unit(net_flow(net));

define net_place_unit(net) =
  row_unit(net_flow(net));

define net_transition_unit(net) =
  column_unit(net_flow(net));
```

Function `net_currency` gives the currency used in the value net's monetary part. In the `Net` type this is the `mon` factor. The implementation uses the standard function `unit_factor` to fetch the unit from the net's valuation. Function `net_unit` is redundant since it is the place unit per transition unit, but it is added for convenience. It gives a unit matrix of the same dimensioned vector space as the incidence matrices of the net. Function `net_place_unit` yields the net's place unit by returning the row unit of the net's flow matrix. Similarly, function `net_transition_unit` returns the column unit of the flow matrix.

8.4.3 *Place Sign*

Since places can be an asset or a liability it is useful to distinguish signed and unsigned value nets. The distinction between assets and liabilities requires careful handling of the sign of values. A liability has a negative value, and this has to be taken into account somewhere when it is valued. In some circumstances it is convenient to ignore the sign of the liabilities and switch to an unsigned value net. An example is the factorization of the flow matrix in Section 7.4. The value net data structure contains functions to change a value net into an unsigned equivalent.

The direction of flow in a value net is anti-clockwise, but for liabilities the direction reverses. The various value nets in the introduction illustrate that the convention that the buy side is on the left and the monetary part is at the top results in an anti-clockwise flow of value. But since the production of a liability equals the consumption of value and vice versa, the direction of the arrows attached to it reverses. The following transformation rule shows how an asset can be changed into a liability and back.



The place on the left is an asset as indicated by the plus sign. It can have any number of incoming and outgoing edges. When the place is changed to a liability the plus sign is changed into a minus sign and the direction of all edges is reversed. This construction leads to a clockwise flow of values for liabilities.

Mathematically the sign is conveniently handled with a sign matrix. The sign matrix is a place by place diagonal matrix with the place sign on the diagonal

Definition 11 *Let $V = (P, T, Cons, Prod, val)$ be a value net. Diagonal matrix S with*

$$S_{ii} = \begin{cases} -1 & \textit{if place } i \in P_{liab} \\ +1 & \textit{otherwise} \end{cases}$$

is the value net's sign matrix. Matrix S^+ is the positive part of S and S^- the negated negative part, i.e. $S = S^+ - S^-$.

Multiplying the sign matrix with a matrix or vector changes the sign of the places.

The sign matrices for a value net are implemented in the following code fragment.

```
declare sign_plus, sign_min, sign_matrix ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!)) -> P!u per P!u;

define sign_plus(net) =
  diagonal((is_asset(net) + is_recording(net)) * net_place_unit(net));

define sign_min(net) =
  diagonal(is_liability(net) * net_place_unit(net));

define sign_matrix(net) =
   sign_plus(net) - sign_min(net);
```

They follow the classification from equation (8.2).

The following definition uses sign matrix S to create an unsigned variant of a value net.

Definition 12 *Let value net $V = (P, T, Cons, Prod, val)$ have sign matrices S, S^+ and S^- and let*

$$\begin{aligned} Cons' &= S^+ \cdot Cons \ + \ S^- \cdot Prod \\ Prod' &= S^+ \cdot Prod \ + \ S^- \cdot Cons \end{aligned}$$

then value net $V' = (P, T, Cons', Prod', val)$ is the unsigned value net of V.

The definition uses sign matrix S to switch to the unsigned flow matrix $S \cdot Flow$. Matrix $Cons'$ contains the positive entries from $S \cdot Flow$ and matrix $Prod'$ contains the negated negative entries (the positive entries from $-S \cdot Flow$). The following proposition shows that an unsigned value net has the same result as the signed one, except that the sign of the initial marking and the outcome is changed for liabilities.

Proposition 1 *Let value net V have sign matrix S and corresponding unsigned value net U. For any marking m_0 and any transition vector t*

$$m_0 \xrightarrow{t}_V m \ \wedge \ S \cdot m_0 \xrightarrow{t}_U m' \ \Rightarrow \ m = S \cdot m'$$

Proof 2 *By definition of Petri net firing we get that $m = S \cdot m'$ follows from*

$$m_0 + (Cons - Prod) \cdot t = S \cdot (S \cdot m_0 + (Cons' - Prod') \cdot t)$$

Using that $S \cdot S = I$ this reduces to

$$Cons - Prod = S \cdot (Cons' - Prod')$$

Plugging in the unsigned incidence matrices gives

$$Cons - Prod = S \cdot (S^{+} \cdot Cons + S^{-} \cdot Prod - S^{+} \cdot Prod - S^{-} \cdot Cons)$$

and from $S = S^{+} - S^{-}$ follows

$$Cons - Prod = S \cdot (S \cdot Cons - S \cdot Prod)$$

which is again true by $S \cdot S = I$.

It is important to note that the outcome might be the same, but that the behavior of the underlying Petri net is not the same. Typically the outcome may not be reachable in the unsigned net.

With the unsigned incidence matrices an unsigned value net can be created.

```
declare unsigned_net ::
  for_index P,T:
    for_unit mon, P!u: (Net(mon, P!u, T!)) -> Net(mon, P!u, T!);

define unsigned_net(net) =
  let
    plus = sign_plus(net),
    min = sign_min(net),
    pre = net_pre(net),
    post = net_post(net)
  in
    make_net_basic(plus '*' pre + min '*' post,
                   plus '*' post + min '*' pre,
                   net_valuation(net),
                   is_finance(net),
                   is_real(net),   # Changes is_asset to is_real !
                   is_recording(net),
                   is_licit(net))
  end;
```

Function `unsigned_net` creates an unsigned equivalent of a value net. It calls the value net constructor with the unsigned incidence matrices. The other arguments are just the values in the given net, except that the `is_asset` is replaced by `is_real` to mark all places as asset.

8.4.4 *Valuation*

The value net's valuation allows switching from quantities in non-financial units to quantities in financial units. Value nets factor value into a product of valuation

and quantity. Multiplying physical quantities with the net's valuation brings back the financial value. Valuation only applies to the production part and not really to the monetary part because the valuation of one monetary unit is per definition one monetary unit.

Valuating a marking has to take the sign of liabilities into account in the multiplication of the valuation vectors and the quantity vector. Given valuation *val*, sign matrix S and marking m the inner product $\langle S \cdot val, m \rangle$ is the total value. Product $S \cdot val$ makes the valuation of the liabilities negative and the inner product multiplies the result with the marking vector.

The following fragment implements the valuation of a given marking for some value net.

```
declare valuate ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!), P!u) -> mon;

declare valuate_each ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!), P!u) -> mon/P!;

define valuate(net, marking) =
  inner(sign_matrix(net) '*' valuation(net), marking);

define valuate_each(net, marking) =
  sign_matrix(net) '*' valuation(net) * marking;
```

Functions `valuate` and `valuate_each` compute the monetary value for any marking. Function `valuate` gives the total value and function `valuate_each` give the monetary value per place.

8.5 CONCLUSION

The goal in this section was to create a type correct data abstraction for a value net that supports all features needed for the analysis of value net behavior in the next chapter. A value net was defined as an extended Petri net in which the transitions are business events and the places are assets and liabilities.

Each mathematical definition is accompanied by an implementation in Pacioli. The Pacioli data structure is shape and unit correct and capable of all required operations.

9 VALUE NET BEHAVIOR

The behavior of a value net follows from its structure. The question addressed in this chapter is how to compute and analyze the normal behavior that follows from the causality and the proportionality in the net. All information about a value net is available in the data structure from the previous chapter. Analyzing the behavior requires factoring the flow matrix, computing the tour space and splitting the tours into causal chains of events.

9.1 TOURS

A tour relates the behavior of an enterprise to the cyclic structure of its process model. The model's cyclic structure is not directly apparent in a net's behavior because any transition in a value net can fire at any point in time as long as its pre-condition is met. We want to group transactions that make up what Ijiri calls a causal chain of events [Iji67]. In general terms such a chain of events is a variation on buying products and resources, producing an end product or service, and selling the product or service. The cycle starts with the consumption of money and ends with the production of money. Such cyclic behavior follows from the cyclic structure of a business process.

The tours form a space that is the null space of the value net. As explained in Section 7.3 a value net's normal behavior is characterized by a steady state equation. Solving this steady equation results in a tour space. Informally the tour space was defined as

$$\text{TourSpace} = \{t \in T \mid \mathit{Flow} \cdot t = 0 \text{ for the places in the operations part}\}$$

A tour is a constellation of events whose total effect is on money only. All other produced tokens are at some point consumed by another step in the process. The tour space is spanned by a tour basis. This tour basis characterizes the normal behavior of a value net.

The following definition makes the informal definition precise. It filters the flow matrix for rows in the operations part only and defines the tour space as the null space of the remaining matrix.

Definition 13 *Let $V = (P, T, Cons, Prod, val)$ be a value net without illicit actions or recordings with flow matrix $Flow = Prod - Cons$ and let*

$$A_{ij} = \begin{cases} Flow_{ij} & \text{if } i \in P_{op} \\ 0 & \text{otherwise} \end{cases}$$

Any transition vector t with $A \cdot t = 0$ is a tour of V, and the set $\{t \in T \mid A \cdot t = 0\}$ is V's tour space.

Matrix A in the definition is the flow matrix of the operations part, including the buy and sell transitions. It is a requirement for the computation that the net is free from illicit actions or recordings, because these extensions have an effect on the tour space. The definition assumes they are absent, but if necessary they could be filtered. Vector t is the number of times each event has to occur per tour. Every time such a tour has occurred the production and the consumption in the production part cancel exactly.

The flow matrix for the operations part can be obtained from the flow matrix factorization from Section 7.4. Equations (7.2) and (7.3) in that section use the factoring of the flow matrix to reformulate the tour definition as two related equations. The first is the steady state equation for the value net's operations part and the second gives the margin as the result in the financial part. From the steady state equation follows that the matrix for the operations part A that is used in the tour definition can be derived from the factored matrices as follows

$$A = Prod_{buy} + Prod_{op} - Cons_{op} - Cons_{sell} \tag{9.1}$$

A caveat is that this computation assumes an unsigned value net, but in that case the sum of the four matrices is the operations part of a value net including the buy and sell transitions, and forms exactly matrix A from Definition 13. Since we need the flow matrix factorization anyway, we might as well use it to compute the operations part matrix.

The flow matrix factoring from equation (7.1) can be computed from the partitioning of a value net's transitions. The next code declares a function for each matrix.

```
declare cons_buy, cons_sell, cons_op, cons_fin,
        prod_buy, prod_sell, prod_op, prod_fin ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!)) -> P!u per T!;
```

Each function expects a value net as argument and returns a matrix of the same type as the flow matrix. The following implementations use primitive function `mask_cols` to create the matrices. This functions expects a matrix and a bit-vector as arguments and zeros the columns in the matrix that have a zero in the corresponding entry in the bit-vector.

```
define cons_buy(net)  = mask_cols(net_pre(net), is_buy_trans(net));
define cons_sell(net) = mask_cols(net_pre(net), is_sell_trans(net));
define cons_op(net)   = mask_cols(net_pre(net), is_op_trans(net));
define cons_fin(net)  = mask_cols(net_pre(net), is_fin_trans(net));
define prod_buy(net)  = mask_cols(net_post(net), is_buy_trans(net));
define prod_sell(net) = mask_cols(net_post(net), is_sell_trans(net));
define prod_op(net)   = mask_cols(net_post(net), is_op_trans(net));
define prod_fin(net)  = mask_cols(net_post(net), is_fin_trans(net));
```

The factoring splits the flow matrix into a production and a consumption matrix and splits these matrices along their columns by the transition partitioning. With these matrices a value net can be broken down into different parts.

The tour space computation creates the operational part of the value net from the flow matrix factoring and computes the space's basis with the Fourier-Motzkin algorithm. The Fourier-Motzkin algorithm computes the basis of a matrix' null space. A type correct implementation was given in Section 6.3.2. The following code fragment uses it to define function `tour_basis`. As the type declaration shows this function expects a value net as argument and returns a list of transition vectors. The computed vector list is the basis of the value net's tour space as defined in Definition 13.

```
declare tour_basis ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!)) -> List(T!);

define tour_basis(net) =
  let
      unsigned = unsigned_net(net),
      prod = prod_buy(unsigned) + prod_op(unsigned),
      flow = prod - cons_op(unsigned) - cons_sell(unsigned)
  in
    [x | x <- fourier_motzkin(flow), not(is_zero(prod '*' x))]
  end;
```

To avoid issues with the sign of liabilities the implementation first creates an unsigned value net from the value net argument. Next it binds the production side of the operations part to variable `prod` and the complete operations part to variable `flow`. The definition of `flow` is in accordance with equation (9.1). The function calls the Fourier-Motzkin function on matrix `flow` to get the null space. The non-zero test using matrix `prod` filters trivial nullifiers from the result. The Fourier-Motzkin function computes nullifiers for the entire column domain, including the filtered financial transitions. These transitions trivially give a zero result by firing once and must be filtered. The remainder is the base for the tour space.

The implementation was used to compute the tours for all value net examples in this thesis.

9.2 THE CAUSAL CHAIN

With the flow matrix factorization the various elements in the causal chain can be revealed. The causal chain splits a tour into several parts that follow from the causality in a value net. The elements of the chain were described in Section 7.4.

The causal chain in the production part starts with the purchased products, possibly creates intermediate products and produces end products ready for sale. The following functions use the factored flow matrices to compute the result of these three elements of the chain.

```
declare purchased_products, intermediate_products, sold_products ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!), T!) -> P!u;

define purchased_products(net, tour) =
  prod_buy(net) '*' tour;

define intermediate_products(net, tour) =
  (prod_op(net) - cons_sell(net)) '*' tour;

define sold_products(net, tour) =
  cons_sell(net) '*' tour;
```

Each function expects a value net and returns the partial tour result as a dimensioned place vector of type `P!u`. The result is computed by multiplying the appropriate incidence matrix with the tour vector. Function `purchased_products` computes $Prod_{buy} \cdot t$, the purchased products. Function `intermediate_products` computes the intermediate products. Intermediate products are products in the process that are neither bought nor sold and can be computed with incidence matrix $Cons_{op} - Prod_{buy}$ or $Prod_{op} - Cons_{sell}$. The implementation uses the latter matrix. Function `sold_products` computes the sold products with matrix $Cons_{sell}$.

The financial part handles payment and receipt of money. The revenue and expense of a tour chain are computed with the next functions. Both function return a dimensioned place vector of type `P!u` just as in the operations part.

```
declare expense, revenue ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!), T!) -> P!u;

define expense(net, tour) =
  cons_buy(net) '*' tour;

define revenue(net, tour) =
  prod_sell(net) '*' tour;
```

Function `expense` computes the total expense in a tour. It multiplies the tour vector with the $Cons_{buy}$ matrix to get the consumed value of the buy transitions. Function `revenue` performs a similar computation on the sell side. It returns the monetary value produced by the sell transitions. The margin is the difference between the tour's

revenue and the tour's expenses, multiplied by the net's valuation.

```
declare tour_margin ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!), T!) -> mon;

define tour_margin(net, tour) =
  inner(valuation(net), revenue(net, tour) - expense(net, tour))
```

The type is correctly derived as type `mon`, the monetary unit of the value net.

The functions defined in the previous paragraphs give all the elements of the causal chain. Figure 9.1 shows the causal chains for the miller and the baker tours from Figure 7.7. The numbers for the miller case were already shown in Figure 7.9 in a schematic way. For the baker case we see the three tours with intermediate products: bread, cake and pie. Each tour uses flour and packaging paper and produces a packaged product to be sold. Instead of one tour that produces three products as in in the miller case, there are three tours that each produce one product.

The causal chains characterize the behavior tours of the baker and the miller as we set out to do in Section 7.2.1. The single tour in the milling process is confirmed by the one dimensional tour space. The baker's space is three dimensional, which agrees with the observation that three points of recording are needed to record the behavior. Besides the process dimension, the spaces also correctly show the amounts of the produced and consumed products and resources. These numbers from the factored tours correspond with causal chains from the accounting field. And the difference in produced and consumed money equals the value jump, or gross margin. All these business concepts are captured by the tour spaces.

9.3 FRAUD ANALYSIS

The fraud analysis computes a complete basis of fraud scenarios for a value net. To model fraud scenarios the value net is extended with recordings and illicit actions. As explained in Section 7.1.2, such a value net is called an 'ist' net to distinguish it from the 'soll' value nets used so far that do not have these extensions. The fraud space is the null space of an ist value net. It contains all non-detectable scenarios that potentially extract value from the process [Els96]. The fraud analysis computes a base for the fraud space.

The construction of an 'ist' value net was explained in Section 7.6. The first extension adds recordings to the value net. To add a recording to a transition the following transformation is performed:

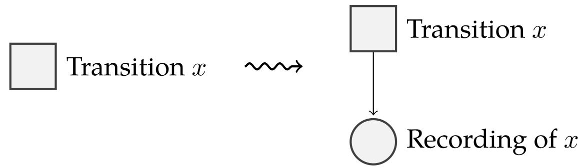
Place	Expense	Purchased	Intermediate	Sold	Revenue
Payables	258.00 €	-	-	-	-
Grain	-	1 t	-	-	-
Packaging paper	-	100 m	-	-	-
Germ	-	-	30 kg	-	-
Bran	-	-	140 kg	-	-
Flour	-	-	830 kg	-	-
Packaged germ	-	-	-	6 bag	-
Packaged flour	-	-	-	166 bag	-
Packaged bran	-	-	-	28 bag	-
Money	-	-	-	-	334.50 €

Place	Expense	Purchased	Intermediate	Sold	Revenue
Payables	264.58 €	-	-	-	-
Flour	-	1 t	-	-	-
Packaging paper	-	1167 m	-	-	-
Bread	-	-	1667 bread	-	-
Packaged bread	-	-	-	1667 bread	-
Money	-	-	-	-	583.33 €

Place	Expense	Purchased	Intermediate	Sold	Revenue
Payables	265.63 €	-	-	-	-
Flour	-	1 t	-	-	-
Packaging paper	-	1250 m	-	-	-
Cake	-	-	2500 cake	-	-
Packaged pastry	-	-	-	2500 pastry	-
Money	-	-	-	-	625.00 €

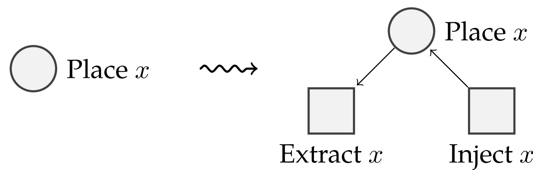
Place	Expense	Purchased	Intermediate	Sold	Revenue
Payables	262.50 €	-	-	-	-
Flour	-	1 t	-	-	-
Packaging paper	-	1000 m	-	-	-
Pie	-	-	2000 pie	-	-
Packaged pastry	-	-	-	2000 pastry	-
Money	-	-	-	-	500.00 €

Figure 9.1: The non-monetary part of a factored tour for the milling case from Figure 7.5. The columns show the chain of events for the tour. From left to right the table shows the sacrificed value, the purchased products, the intermediate products, the produced products and the received value.



Each time the transition occurs a token is produced in the recording place. In this way the recording place counts the number of occurrences of the connected transition.

Illicit behavior is modeled by transforming each place in the following way:



Actions Extract x and Inject x are illicit manipulations of the state of Place x. Note that this transformation must not be performed before adding the recordings. Illicit behavior is not recorded.

Informally the fraud space was defined as

$$\{t \in T \mid \text{effect of } t \text{ in the 'ist' value net is zero}\}$$

The following definition reformulates that in terms of a value net.

Definition 14 *Let $V = (P, T, Cons, Prod, val)$ be a value net with flow matrix $Flow = Prod - Cons$. Any transition vector t with $Flow \cdot t = 0$ is a potential fraud scenario of V and the set $\{t \in T \mid Flow \cdot t = 0\}$ is V's fraud space.*

The fraud space is simply the flow matrix's null space. The structure of this space completely depends on the way the 'ist' net is constructed.

Recordings reduce the fraud space and illicit actions increases it. When a transition is recorded it is impossible to perform it without being detected. In the extreme case that all transactions are recorded and there are no illicit actions the fraud space is therefore empty. When illicit actions are added the possibilities for fraud arise. More illicit actions means more fraud scenarios.

Function `fraud_scenarios` computes a basis for a value net's fraud space. It accepts a value net as argument and returns a list of transition vectors. Each vector is a base fraud scenario.

```
declare fraud_scenarios ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!)) -> List(T!);

define fraud_scenarios(net) =
  fourier_motzkin(net_flow(net));
```

The implementation calls `fourier_motzkin` to compute the null space of the flow matrix. The result is a list of dimensionless transitions vectors.

The effect of the fraud scenarios can be seen by splitting the effect into a licit and an illicit part. An example of a breakdown of a fraud scenario was given in Figure 7.11. The following functions split the fraud scenario to expose the extracted value.

```
declare illicit_effect, licit_effect ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!), T!) -> P!u;

declare illicit_margin ::
  for_index P,T: for_unit mon, P!u: (Net(mon, P!u, T!), T!) -> mon;

define illicit_effect(net, scenario) =
  net_flow(net) '*' (scenario  * is_illicit(net));

define licit_effect(net, scenario) =
  net_flow(net) '*' (scenario  * is_licit(net));

define illicit_margin(net, scenario) =
  valuate(net, illicit_effect(net, scenario));
```

Function `illicit_effect` filters the scenario for the illicit part by multiplying it with the `is_licit` bit vector. The filtered scenario is multiplied with the flow matrix to get the effect. Function `licit_effect` computes the licit effect in a similar way. The result of these functions is a place vector with units `P!u`. Function `illicit_margin` returns the valuation of the illicit effect. This is the illicitly extracted value. The type is the monetary type `mon` of the value net. All the types are derived correctly by the type system.

9.4 CONCLUSION

This chapter made the informal ideas from the introduction in Chapter 7 precise. It gives mathematical definitions for the concepts related to value net behavior, like the steady state equations and the value jump. The goal of this chapter is to provide Pacioli

definitions that correctly implement the mathematical definitions. A requirement is type correctness of the computations and the correct handling of the units of measurement.

The implementation shows that the Pacioli language handles the requirements well. An advantage of Pacioli's matrix support is that the code maps directly on the mathematical linear algebra definitions. The type system guarantees that vectors are from the correct spaces and that values have the right units of measurement.

10 CASE STUDY: PUBLIC TRANSPORT SERVICES

In this chapter we look at two applications of value net modeling. First the normative model of public transport services is used to analyze taxi companies' revenue models. Second the model is used in a dispute between the contract parties.

10.1 CASE DESCRIPTION

Dutch municipalities organize taxi-bus transport services for secondary school pupils that need to attend special schools. These pupils are not capable of traveling by themselves in other forms of transportation and depend on the services provided by municipalities. For the transport of this group of passengers, many rules and regulations apply. To fulfill this duty municipalities hire taxi companies that daily brings to pupils to and from the schools. Usually a contract for a certain period is agreed upon after a request for proposal. The municipality picks the taxi company that it deems most fit to perform the transport service it has to provide.

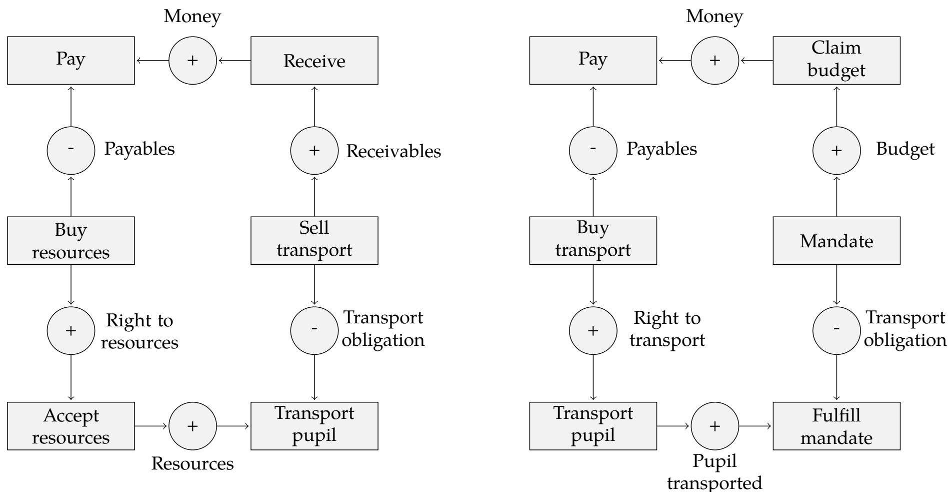


Figure 10.1: Value cycles of transport provider and municipalities

We use the two value cycles in Figure 10.1 that describe the situation. The value cycle on the left depicts the process of a taxi company that drives the pupils to and from school. It sells transportation and buys resources to perform the transportation. The value cycle on the right depicts the municipality's process. The sell side of a

municipality is somewhat unusual because money is not obtained by sales but by tax. The municipality gets a mandate that gives an obligation to transport the pupils and also the right to the necessary budget to do so. The buy side of the municipality mirrors the sell side of the taxi company. Selling transport and receiving money by the taxi company corresponds to buying and paying money by the municipality. The transportation of the pupil removes the rights and obligations between the two parties. This connection between the two value nets shows the business relationship in the case.

A main concern for the municipalities is how they can stay in control of the transport service. Besides all the regulations, the municipalities are also expected to be responsible with taxpayer's money and control the cost. What makes this difficult is that an invoice for transport is hard to judge. Taxi companies typically provide an invoice based on the amount of hours or mileage. So the request was transportation per person, but the invoice is transportation per hour or mile. This difference makes it difficult to judge the correctness of the invoice, and there is no natural incentive for the taxi company to strive for efficiency. How can a municipality check that billed hours or miles are not overstated?

10.2 REVENUE MODEL

In 2010 a transport provider recognized the municipalities' concerns and offered to transport the pupils using a revenue model that allowed for better control. By agreeing to a more transparent and objective normative model the taxi company provided an attractive alternative that helped the municipalities address their control issues.

The reasoning that led to the new revenue model can be illustrated by the value net in Figure 10.2. The value net is a concrete description of the taxi process that was introduced in the left value cycle in Figure 10.1. The taxi company differentiates two products; regular transport and accompanied transport. In the latter case an attendant travels along to look after the pupils. The value net has a sales transition for both services. On the buy side the various resources are purchased. Both services require a driver and a vehicle, and the accompanied transport additionally needs an attendant. The planning transitions take the bruto purchased resources and turns them into planned hours. These planned net hours are available to do the actual transport, and from historical data follows that for regular transport this is around 59 minutes per hour and for accompanied transport around 58 minutes. The actual transport is where the difference in units of measurement occurs. The transitions consume transport obligations that are per person and planned resources that are per hour. The values on the edges can directly be calculated for a given day or period from the total amount of transported pupils and the total amount of spent hours. Any further analysis however requires information on a more detailed level. From historical data

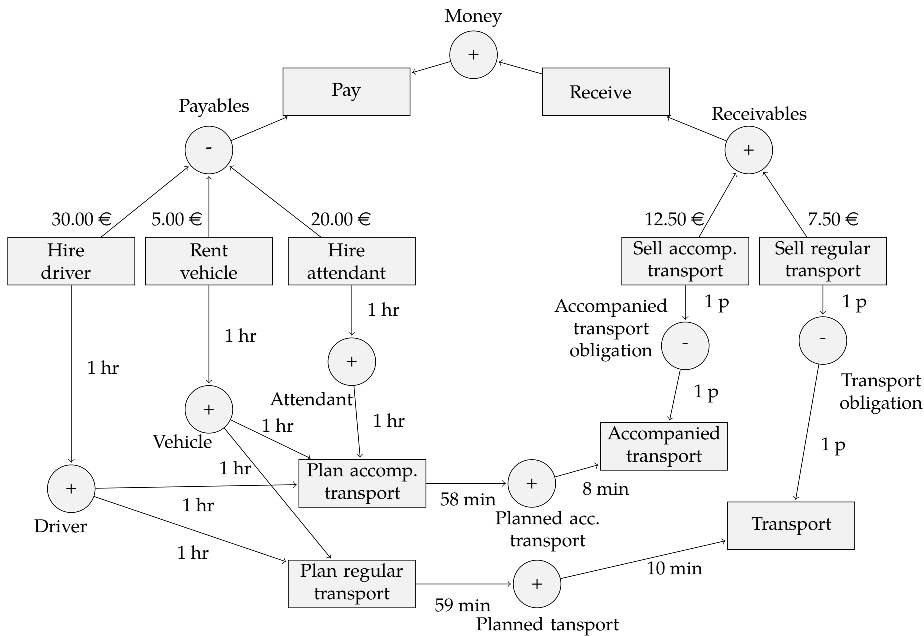


Figure 10.2: The value net of a transport provider

it turns out that on average the transport gives a resources consumption of around 10min/p for regular transport and around 8min/p for accompanied transport. These numbers that quantify the connection between the buy side and the sell side are central to the reasoning in the new revenue model.

The value net's tours gives insight into the relationship between the various quantities in the process. The factored tours for the two services are shown in Figure 10.3. The first tour is the regular transport service. The columns for the buy side shows that the cost of a vehicle and a driver hour is 35€. The intermediate product column shows that the planning efficiency is 98%. The value for the transport obligation shows an average occupancy of 5.9p/hr. These two values are key performance indicators for the regular transport service. In general the occupancy rate of a vehicle is a key performance indicator for analyzing operational planning effectiveness and profitability. It is the reciprocal of the resource consumption, converted to hours. For the accompanied transport service the the planning efficiency is 98% and the occupancy of 7.25p/hr. The causal chain of events reveals the ratios in the taxi process.

The ratios in the taxi process are unfortunately not constant and depend on many circumstances like traffic congestion, geographic region, etc, but using the output

Place	Expense	Purchased	Intermediate	Sold	Revenue
Payables	35 €	-	-	-	-
Vehicle	-	1 hr	-	-	-
Driver	-	1 hr	-	-	-
Planned reg. transport	-	-	0.98 hr	-	-
Reg. transport obligation	-	-		5.90 p	-
Receivables	-	-	-	-	44.25 €

Place	Expense	Purchased	Intermediate	Sold	Revenue
Payables	55 €	-	-	-	-
Vehicle	-	1 hr	-	-	-
Driver	-	1 hr	-	-	-
Attendant	-	1 hr	-	-	-
Planned acc. transport	-	-	0.97 hr	-	-
Acc. transport obligation	-	-		7.25 p	-
Receivables	-	-	-	-	90.63 €

Figure 10.3: Factored tour for the transport case. Columns show the chain of events.

of the routing software turns out to give a usable norm. Based on trip requests received from the municipality specifying travel specifications for a certain week day, the routing software package used by the transport provider calculates the daily routes at each time of day, entailing trip requests i.e. the number of school pupils per route. This prediction is based on standard speeds (100 km/h for motorways, 70 km/h for main roads, 40 km/h in town etc.). The parties agreed that for the sake of the contract the routing software package is used as norm for the best routes for the transport services. The software gives a usable estimate for the many varying circumstances that influence the efficiency. From the value net analysis it is obvious that the planning software fulfills all the needs. It gives an objective and transparent estimate for the performance indicators.

The revenue model proposed by the taxi company uses the normative hours calculated by the planning software as the basis for the invoice. For each route, the planning software calculates the occupied vehicle hours. These occupied vehicle hours are priced at a certain tariff, and the revenue per route is the tariff multiplied by the occupied vehicle hours. Within the context of the planning and control cycle of the transport provider the actuals i.e. the realized data are gathered on a daily basis for monitoring purposes. Deviations from the norm are very interesting from an operational perspective because operational management learns about the quality of planning operations and more over whether the norms used in calculating the price per occupied hour is correct. From a financial perspective the same information is very valuable in assessing whether the contract is profitable or not. Every month,

the transport provider sends an invoice with a data file, detailing the number of trips, passenger details, routes, departure and arrival times, departure and arrival addresses, ordered and canceled trips etc. This data file provides evidence of the services delivered for the invoice to be paid by the municipalities. The decision by management of the transport provider to offer their quote based on price per occupied hour and the planned routes improved the municipalities' trust in the validity of the invoice.

10.3 DISPUTE RESOLUTION

In addition to the design of the revenue model, the normative model described by the value net was used also to resolve a dispute between the contract parties. One of the effects of the alternative revenue model is that operational risk is transferred from the municipalities to the transport providers. Since the contract is based on planned hours, all fluctuations in traffic for example are now the risk for the taxi company. On the long run such fluctuations should average out, but there is still the possibility that the calculated norm is structurally unsound. The parties made agreements on such issues in the contract to share the risk in an acceptable way. However, at a certain point the taxi company felt that the norm calculated by the planning software was not fair. Thanks to the objective and transparent norm the issue could be resolved using statistical analysis.

All data about the disputed routes was available for analysis. The dispute eventually focused on a period of one month for a region of 37 routes. The data set used for analysis detailed the number of trips, passenger details, routes, departure and arrival times, departure and arrival addresses, ordered and canceled trips etc. The planned and realized data was available.

Figure 10.4 gives an overview of the data. The scatter plots display the trip duration against the number of passengers. The number of pupils in a vehicle varies between one and eight. The maximum trip time is around one and a half hour, the limit imposed by the contract between the transport provider and the municipality. The plots labeled AM are the trips to the school, and the PM plots are the return trips. The general pattern is that trips with more pupils take longer.

The totals of the quantities that flow in the value net can be derived from the data set. In the region under consideration 215 pupils were transported. The following table gives a count of the total number of trips and the total number of pupil movements.

	AM	PM	Total
Number of trips	442	355	797
Transported pupils	2436 p	1835 p	4271 p

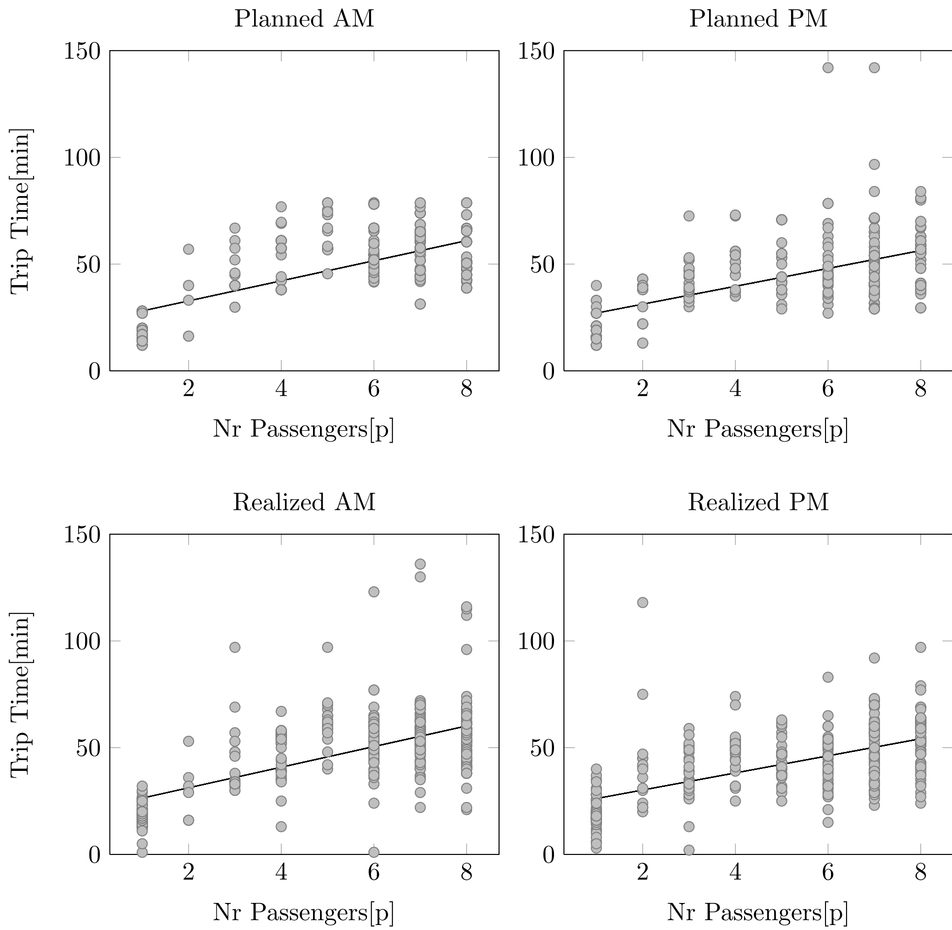


Figure 10.4: Scatter plots for the planned trip durations in the morning (AM) and in the afternoon (PM).

These numbers represent the requested transport for the pupils by the municipality. The spent hours by the taxi provider is the total of the trip durations. The following table shows the planned and the realized times.

	AM	PM	Total
Planned total time	357.01 hr	259.62 hr	616.63 hr
Realized total time	348.87 hr	250.17 hr	599.03 hr

These numbers show the totals on the buy and the sell side of the transport provider's value net.

The value net's normative parameters for the disputed period can now also be derived from the data set. The following table gives the key numbers:

	Planned AM	Realized AM	Planned PM	Realized PM
Mean	48.46 min	47.36 min	43.88 min	42.28 min
Std dev	17.57 min	19.06 min	18.00 min	16.83 min
Resource usage	8.79 min/p	8.59 min/p	8.49 min/p	8.18 min/p
Occupancy	6.82 p/hr	6.98 p/hr	7.07 p/hr	7.34 p/hr

The mean and the standard deviation describe the average trip duration. The resource usage is the total trip time divided by the number of pupils. These correspond to the 8min and 10min from Figure 10.2. The occupancy is the reciprocal of the resource usage converted to hours. These numbers correspond to the fourth column from Figure 10.3. The table shows the actual values for the normative numbers from the value net for the given period and region.

The data provided an objective norm to analyze the cause of the deviations and to resolve the dispute. After clarification of some issues with the disputed the deviations Detailed investigations revealed some data capture issues, but the data set contained enough information to analyze the dispute routes. After analyzing the data it turned out that traffic jams caused delays in the morning, but not outside acceptable statistical variations as the transport company claimed. Further analysis revealed that the planning department did not optimize routes for each route per day part. Instead the schedules were planned using weekly averages. After recalculation of a detailed planning the issue was resolved.

10.4 CONCLUSION

Besides helping analyzing the business case, the value net helped eliciting the key performance indicators in the design of the business model. The value net helped reasoning about contracts and norms from a control perspective. Confronting the value net of the municipality and the taxi company showed that units do not match

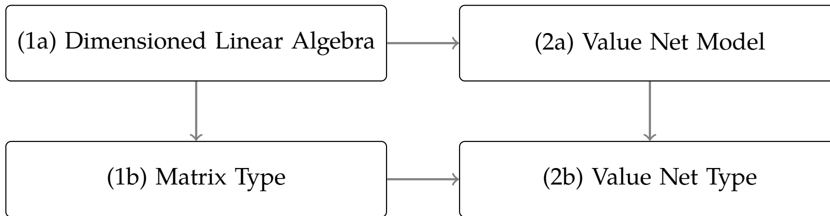
and that agreement on some norm is needed for a contract. With the analysis of the deviations of the actual data from the norms the process can be monitored. The value net is useful when reasoning about preventive, detective, and corrective measures for control.

The case was also a useful validation for the value net library. All computations were done in Pacioli and completely unit-aware and type correct. The modeling and analysis in units of measurement was crucial to the solution. Pacioli's type system supports computing in proper units and the value net library supported the required analysis. The process model in non-financial units provided the depth needed to analyze the issue.

CONCLUSION

The research in this thesis addresses the question how support for units of measurement can improve type systems for matrix programming, and how such support can increase the accuracy and quality of business process modeling. Process models in physical units are more accurate, but increase the complexity of computations because the convenience of a single monetary unit is lost. The investigation into unit-aware matrix programming in the first part of the thesis explores the feasibility of a parametric type for linear algebra that can handle the complexity of varying units of measurement. This increases the use case of units in programming languages. The value net formalism developed in the second part of the thesis uses unit-aware matrices to improve the accuracy of the value cycle models from the auditing field and generalize their use case to other application areas.

The research is divided into four related topics. The relations between the different research topics was summarized in Section 1.4 in the following figure.



Dimensioned linear algebra and the matrix type address the run time and compile time aspects of unit-aware matrices. The value net model and type address the corresponding modeling aspects.

The research shows that dimensioned linear algebra is feasible and a good reference for the matrix type. The matrix type gives principal types for linear algebra expressions with just two extensions of Kennedy's syntax for units of measurement. The resulting unit-aware matrices are expressive enough to model the physical units in business processes. The implementation of the value net data type is a practical validation of the value nets and the matrix type.

unit-aware matrix programming

The expressiveness of the matrix type shows that dimensioned vector spaces help unit-aware matrix programming and increases the use case for unification of units. The overall research question addressed the feasibility of a parametric matrix type based on dimensioned vector spaces.

Research question (1)

Can dimensioned vector spaces form the basis for a parametric matrix type for matrix programming?

The analysis of the expressiveness of the matrix type shows that the type rules infer the dimensioned vector space of any linear algebra expression. This increases safety to a much finer level than types based on shape checking.

The dimensioned matrices introduced in this first part of this research give a concrete definition of dimensioned linear algebra that is a suitable reference for the matrix type. The index-free evaluation rules for dimensioned matrices provide a baseline for the matrix type rules. The feasibility of an index-free formulation of the linear algebra operations was addressed in the first research question.

Research question (1a) Dimensioned Linear Algebra

How accurate can dimensioned linear algebra be expressed in index-free form using base unit vectors?

The dimensioned matrices developed in Section 4.3 show that the use of vectors as unit bases allows for a suitable index-free formulation of dimensioned linear algebra.

The matrix shape with a scalar, a co-variant and a contra-variant unit part is sufficient to type any vector, matrix or tensor. With the shape's three unit parts scalar all linear algebra operations can be expressed in index-free form. For each core linear algebra operation a rule for the units can be expressed with just unit vectors, except for the scaling operation. The scaling operation distributes a scalar unit over all entries of a vector or matrix and this is not an index-free operation. Instead of actually distributing the scalar unit over the vector, the matrix shape treats the units as an algebraic product between scalar and vector units. After factoring out the scalar unit like this all operations can be defined in index-free form.

Another issue with scaling is that it cannot be overloaded with the matrix product. In mathematics scaling and the matrix product are usually overloaded, but they are distinct operations with different evaluation rules. Besides the need to split scaling and the matrix product into two operators there were no issues to formulate index-free linear algebra, and the dimensioned matrices are a good reference for the matrix type.

The matrix type gives effective sound and complete unit-aware type inference for linear algebra expressions. With the index-free evaluation rules provided by dimensioned matrices, the matrix type has a reference relative to which the type rules are defined. Unification for the matrix type is straightforward so the remaining question is how expressive the type rules are.

Research question (1b) Matrix Type

How expressive are the type rules for the matrix type compared to the evaluation rules for dimensioned matrices?

The matrix type extends Kennedy's syntax for units of measurement with two cases. The first is a special naming scheme for unit vectors. The second case is the Kronecker product to add support for tensors. These two extensions make the type sufficiently rich for unit-aware type rules for linear algebra operators, as was explained in Section 5.4.

The matrix type can describe many different kinds of matrices, including matrices with heterogeneous units of measurement. The matrix type supports heterogeneous units of measurement because it uses unit vector names instead of actual unit vectors. The combination of an index set identifier and a unit identifier in the special naming scheme gives enough information to identify dimensioned vector spaces without any commitment to individual units. Besides homogeneous units, the notation also gives a notation for the dimensionless and homogeneous case. These different notations are combined in the type rules to build all kinds of vectors, matrices and tensors.

The matrix type depends on an index-free programming style, but primitives to access individual matrix elements can be defined and typed as well. Besides the linear algebra operators, primitive shape and unit operations are defined that can access matrix elements and units. These operations are by definition not index-free and have to fall back to homogeneous units. The primitives to access the unit from a matrix are however sufficient to split a matrix into a magnitude and a unit part. In this way the individual elements of a matrix with heterogeneous units can still be accessed. This shows that the expressiveness of the matrix type is not limited in any way.

The matrix type's use of index sets instead of natural numbers makes it more strict than dimensioned matrices, but also more safe. A dimensioned matrix stores the sizes of the index instead of the index sets itself. For matrices with different index sets that happen to be of the same size this means that they are considered to have equal shape. From a safety perspective this equality is accidental while the matrix type's equality is intentional. The matrix type is more conservative and disregards such accidental equality.

Conclusion and Future Research

A consequence of the matrix type is that cases where matrices are used as arrays and do not represent a linear transformation will be rejected and have to be changed to a different data structure. Mathematically matrices have a strict definition and specific properties, but in both mathematics and programming they are often used as arrays. The Fourier Motzkin algorithm from Section 6.3.2 is a typical case where shape analysis is complicated because the operations performed on matrices are array operations and not from linear algebra. The matrix type rejects this dual use and favors a more index-free and functional programming style. For the rewritten version the types of the algorithm can be inferred without any annotation. In matrix languages an index-free style is often associated with good programming practice

and it would be interesting to research the impact of the strict rules of the matrix type on the quality of mathematical software.

Another topic for further research is support for the multi-linear case. Unit-aware tensors have interesting applications in areas like data warehousing and analytics where multi-dimensional data is common. The matrix type supports multi-linear algebra via matricized tensors, but a consequence of the matricization is that type inference for tensor operations is not possible. The tensor operators rely on type declarations that describe the tensor structure. Another issue with the current tensor operators is that their practical utility is limited. They demonstrate that units-aware tensors are supported, but they were not chosen with practical applications in mind. A richer and improved set of operators for tensors would increase the practical value for unit-aware multi-linear algebra.

Another interesting question raised by the matrix type is how to do type erasure. A crucial property of the unit-aware matrices is minimal runtime overhead. Type erasure is however complicated because the units in the type are needed at runtime for I/O and conversions. Some initial investigations with the various runtime systems suggest that a unit computation completely decoupled from the number computation might be feasible. Other solutions are however conceivable and it is unclear what the best approach is.

VALUE NETS

The research on value nets addresses the question how the value cycle can be generalized to a process modeling technique that is applicable beyond the auditing field. The causality and proportionality that are central to the value cycle model are useful for other areas like decision making, budgeting, costing, etc. The problem is that the value cycle and the accompanying audit equations are formulated in accounting terms and therefore subject to valuation.

The overall question was how the useful properties of the value cycle can be generalized to other application areas.

Research question (2)

How can the analytical features based on the value cycle's causality and proportionality be generalized from auditing to other applications?

The reformulation in non-financial units enables the derivation of a value net's behavior as a separate entity without commitment to any specific application. The tour spaces are a useful general characterization of an enterprise's behavior. Dimensioned linear algebra successfully handled the different units of measurement.

The main question regarding the value cycle is how the valuation can be eliminated from the audit equations. This requires modeling and analysis in non-financial units of measurement.

Research question (2a) Value Net Model

Can the valuation be eliminated from the value cycle models by reformulating the audit equations in non-financial units of measurement?

The approach in this thesis formulates the audit equations as a steady state equation using dimensioned linear algebra. A value net's normal behavior is the behavior where for each place the input equals the output. This definition is in line with the BETA equations from the auditing field, but since it is expressed in non-financial units of measurement it is free of valuation.

The steady state equation in non-financial units of measurement is a viable valuation-free characterization of normal value net behavior. The audit equations are not suitable for the computation of normal behavior because they are not proper conservation laws. The steady state equation does not have this problem because there is no valuation involved. The steady state equation can be solved and gives a space of solutions. Each solution to the steady state equation is a constellation of transition occurrences that together perform one 'cycle' in the value net. This is called a tour, and the net's normal behavior is the complete tour space.

Tours allow analysis of the relationship between business events that is not possible with accounting systems. The structure of a value net provides all information necessary to factor a tour into what Ijiri calls a causal chain of events. Since a tour also contains the proportions this also gives the quantitative relationship between the events. The public transport case from Chapter 10 illustrates that analysis in non-financial units can be much more fine-grained than accounting information. In that case the normal behavior as well as the evidence is expressed in time units. Such an analysis is infeasible with accounting information. The tour's detailed normal behavior description based on the causality and proportionality in a value net extends the value cycle's analytical capabilities significantly.

Behavior in non-financial units of measurement is related to Ellerman's property accounting and can always be mapped back to an accounting system. Ellerman's property accounting prevents information loss by recording transactions in non-financial units. In that case the recordings are directly comparable with the the value net behavior in non-financial units. However, in many cases non-financial units cannot be maintained forever. For example, for financial reporting a single financial unit of measurement is required. Given some valuation, any recording in non-financial units can always be mapped back to an accounting system by multiplying the quantities with the desired valuation. Similarly, the value net behavior can always be mapped to financial units and an accounting system. The information is however only lost when necessary and not prematurely.

For the fourth research question a unit-aware value net implementation was developed as proof of concept for the value net.

Research question (2b) Value Net Type

Can a statically checked data type for value nets implement the value net models?

Implementing a unit-aware value net data type was a valuable test of the effect of modeling in non-financial units. Besides a proof of concept for the value net it is also a good test for the matrix type. The business analysis with value nets is a use case in which units of measurement play an important role.

Due to the mathematical nature of the matrix type the implementation can be very close to the definition. The data structure is a tuple just like the mathematical definition. All operations can be implemented with index-free operations and idiomatic constructions for units and are mostly one-liners. The computation of a value net's behavior is completely unit-aware and unit-correct. The computation of the null spaces needed in the analysis of the behavior in the value net's soll and ist modalities used the implementation of the Fourier-Motzkin algorithm from Section 6.3.2. The remainder of the computations use idiomatic linear algebra constructions and are all type correct.

Conclusion and Future Research

The value net implementation and its use in various cases are a successful proof of concept of value cycles in non-financial units of measurement. A more extensive validation of the possibilities of applications in management accounting and other areas would be a possible next step in the research. As the public transport case shows, the norms used in the value net models are closely related to performance indicators and are useful for planning and other applications. It would be interesting to validate this with more use cases.

Although the aim of the research is to generalize the value cycle, it would also be interesting for future work to see how the value net can benefit auditing. As the transport case shows, it helps to reason better about evidence for the quantitative aspects of auditing. With the recent developments in computation and communication technology the possibilities for quantitative analysis have increased enormously and more fine-grained modeling in non-financial units as the value nets provide has become viable.

Another topic for further research is the intra-organizational link between value nets. The buy and sell transactions in a value net are mirrored by the sell and buy transactions in the value nets of the other party in the transaction. The exact nature of these transactions is usually agreed upon in contracts. If this connection between contracts and value nets can be modeled then it might be possible to scale the analytical features of value nets up to supply chains.

Similarly, value nets can relate an organization's internal operational process, like the sales process or the production process. The top-level view of the value net is an aggregated model that abstracts from the operational processes. A more operational

process typically performs a part of the value cycle and has a start and end point. It would be interesting to see if these models can be related by integrating them into one unified value net.

A specific topic for further research is the integration of the bill of material analysis from Section 6.3.3 with value nets. The case is presented as a Pacioli example in part I, but is also relevant for the value nets from part II. The numbers in the bill of material are an example of the proportionality found in value nets. It would be interesting to see how the two concepts are related.

A final topic that remains open is the fraud analysis. The experiments with the fraud analysis method gave promising results but it also showed that more accurate modeling is required. A distinguishing feature of the method is that it allows reasoning about the absence of fraud instead of the presence of fraud. The computation gives a complete set of fraud scenarios for a given model. However, when steps in a process are not modeled the method will not find any scenarios involving these steps. Validation of the fraud analysis with more detailed cases would show the practical usefulness of the method.

APPENDIX

This section reviews relevant definitions from abstract algebra. See for example [Rom08].

A.1 GROUPS

An important mathematical concept for much of the theory in this thesis is the notion of a group. A group is a set with a single associative binary operation and an inverse.

Definition 15 *A group is a set G equipped with a binary operator $*$ that satisfies the following three properties:*

1. *for all $a, b, c \in G$*

$$a * (b * c) = (a * b) * c \qquad \textit{associativity}$$

2. *there is an element $e \in G$ such that for all $a \in G$*

$$e * a = a * e = a \qquad \textit{identity}$$

3. *for each $a \in G$ some $a^{-1} \in G$ exists satisfying*

$$a * a^{-1} = a^{-1} * a = e \qquad \textit{inverse}$$

Consider for example the two operations on the set {0, 1} given by the following two tables.

+	0	1
0	0	1
1	1	0

×	0	1
0	0	0
1	0	1

The operator on the left sums two bits with overflow (the exclusive or). This defines a group since it satisfies all rules. The multiplication on the right does not define a group since 0 does not have an inverse.

Another example of a group is the set formed by the three unary matrix operators from Section 4.3 together with the identity operator, and function composition as the group operator. It is easy to show that the operators satisfy the relationships given by the following table. For example, the entry for `^R` and `^T` is `^D` which means that $(A^T)^R = A^D$ for any matrix A, which is true.

∘	id	^T	^R	^D
id	id	^T	^R	^D
^T	^T	id	^D	^R
^R	^R	^D	id	^T
^D	^D	^R	^T	id

If we inspect the table with the rules from the previous section we see that it indeed forms a group. This is an example of a Klein four-group, a particular kind of group with four elements.

A.2 ABELIAN GROUPS

When the operation in a group is commutative the group is called abelian.

Definition 16 *A group G is abelian when for all $a, b \in G$*

$$a * b = b * a \qquad \textit{commutativity}$$

The group defined by the sum operator in the previous section is abelian. The multiplication operator is commutative, but it is not abelian because it doesn't define a group. Examples of multiplicative abelian groups are the sets $\mathbb{R}\backslash\{0\}$ and $\mathbb{Q}\backslash\{0\}$. The exclusion of 0 ensures that the inverse always exists.

A finitely generated abelian group is an abelian group that can be generated from a finite subset of its elements.

Definition 17 *A generating set for an abelian group G is a finite number of elements $a_1, \ldots, a_n \in G$ such that every $a \in G$ can be written as*

$$a = a_1^{i_1} * \cdots * a_n^{i_n}$$

where $a_x^{i_x}$ means the i_x-ary operation $a_x * \cdots * a_x$ for integer i_x. The abelian group is said to be finitely generated from the generating set.

An important example for us is the group formed by units of measurement. This is a group that is finitely generated from some finite set of base units.

A.3 FIELDS AND VECTOR SPACES

A field combines an additive and a multiplicative group. It is assumed that multiplication binds syntactically higher than addition.

Definition 18 *A field F is a combination of an abelian group F with identity 0 and an abelian group $F\backslash\{0\}$ satisfying for all $a, b, c \in F$*

$$a \cdot (b + c) = a \cdot b + a \cdot c \qquad \textit{distributivity over addition}$$

Well known fields are the real numbers $\mathbb{R}$ and the complex numbers $\mathbb{C}$.

A vector space is an abelian group extended with scalar multiplication.

Definition 19 *A vector space over a field of scalars F is an abelian group V extended with scalar multiplication $\cdot$, satisfying for all $a, b \in F$ and $v, w \in V$*

$$
\begin{aligned}
a \cdot (v + w) &= a \cdot v + a \cdot w && \textit{Distributivity over vector addition} \\
(a + b) \cdot v &= a \cdot v + b \cdot v && \textit{Distributivity over field addition} \\
a \cdot (b \cdot v) &= (a \cdot b) \cdot v && \textit{Compatibility of multiplication} \\
1 \cdot v &= v && \textit{Identity of multiplication}
\end{aligned}
$$

Transformations between vector spaces lead to the Lin spaces of linear maps.

Definition 20 *Let V and W be vector spaces over some field F. A map f from V to W is linear if for $a, b \in F$ and $v, w \in V$*

$$f(a \cdot v + b \cdot w) = a \cdot f(v) + b \cdot f(w) \qquad \textit{linearity}$$

The set of all linear maps from V to W is denoted by $\mathrm{Lin}(V, W)$.

a.4 THE REAL VECTOR SPACE $\mathbb{R}^n$

An $m \times n$ matrix A is a map $I \times J \to \mathbb{R}$ with $|I| = m$ and $|J| = n$. The matrix element at position i, j is written as A_{ij}. A column vector is a $n \times 1$ matrix and a row vector is a $1 \times n$ matrix. The real vector space of $m \times n$ matrices is denoted by $\mathbb{R}^{m \times n}$. Column vectors of size n are denoted by $\mathbb{R}^n$. The transpose A^T of a matrix, defined by$(A^T)_{ij} = A_{ji}$, swaps the indices.

The linear addition and scaling operations on matrices and vectors are defined as usual.

Definition 21 *Let $A, B \in \mathbb{R}^{m \times n}$ and let $c \in \mathbb{R}$. Addition and scaling are defined by*

$$
\begin{aligned}
(A + B)_{ij} &= A_{ij} + B_{ij} && \textit{Addition} \\
(c \cdot A)_{ij} &= c \cdot A_{ij} && \textit{Scaling}
\end{aligned}
$$

The matrix product linearly combines the rows and columns of two matrices.

Definition 22 *Let A be a $\mathbb{R}^{m \times k}$ matrix and let B be a $\mathbb{R}^{k \times n}$ matrix*

$$(AB)_{ij} = \sum_x A_{ix} \cdot B_{xj} \qquad \textit{Matrix product}$$

a.5 INNER AND OUTER PRODUCT

The inner product contracts two vectors of equal size into a single number. The outer product combines two vectors into a matrix. This is also called a rank one matrix.

Definition 23 *Let $v, w \in \mathbb{R}^n$. The inner product $\langle v, w \rangle$ is defined by*

$$\langle v, w \rangle \;=\; v^T w \qquad \textit{Inner product}$$

Definition 24 *Let $v \in \mathbb{R}^m$ and let $w \in \mathbb{R}^n$. The outer product $v \otimes w$ is defined by*

$$v \otimes w \;=\; v w^T \qquad \textit{Outer product}$$

Property 2 *From the definition of the matrix product follows*

$$\begin{aligned}\langle v, w \rangle &= \sum_x v_x \cdot w_x \\ (v \otimes w)_{ij} &= v_i \cdot w_j\end{aligned}$$

a.6 TENSOR PRODUCT

Multi-linearity is a generalization of bi-linearity. A binary operator is bi-linear if it is linear in each of its arguments.

Definition 25 *Let V, W and Z be vector spaces over some field F. A map $f : V \times W \to Z$ is bi-linear if for all $a, b \in F$, $v, v_1, v_2 \in V$ and $w, w_1, w_2 \in V$*

$$\begin{aligned}f(a \cdot v_1 + b \cdot v_2, w) = a \cdot f(v_1, w) + b \cdot f(v_2, w) \wedge \\ f(v, a \cdot w_1 + b \cdot w_2) = a \cdot f(v, w_1) + b \cdot f(v, w_2)\end{aligned}$$

The tensor product is the most general bi-linear product by definition.

The inner and outer product from the previous section are examples of bi-linear products. In a real vector space the tensor product is the outer product.

a.7 KRONECKER PRODUCT

The Kronecker product is a generalization of the outer product from vectors to matrices. It can be used to matricize tensors. A tensor is then turned into a matrix with compound indices.

Definition 26 *Let A be a $I_1 \times J_1 \to \mathbb{R}$ matrix, and let B be a $I_2 \times J_2 \to \mathbb{R}$ matrix. Matrix $A \otimes B$ is an $I \times J \to \mathbb{R}$ matrix with*

$$(A \otimes B)_{ij} = A_{i_1 j_1} \cdot B_{i_2 j_2}$$

where $I = I_1 \times I_2$, $J = J_1 \times J_2$, $i = (i_1, i_2)$ and $j = (j_1, j_2)$

REFERENCES

[ACR86] L. D. A., S. J. A. D. J. Cockburn, and C. J. Reiter. “An Assertion Based Approach to Auditing (Discussant’s Remarks)”. In: *Proceedings of the 1986 Touche Ross/University of Kansas Symposium on Auditing Problems*. 1986, pp. 31-67 (cit. on p. 95).

[AHW03] W. Aalst, A. Hofstede, and M. Weske. *Business Process Management: A Survey*. English. Ed. by W. Aalst and M. Weske. Vol. 2678. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 1–12 (cit. on p. 98).

[ADL05] A. Abdel-Maksoud, D. Dugdale, and R. Luther. “Non-financial performance measurement in manufacturing companies”. In: *The British Accounting Review* 37.3 (2005), pp. 261–297 (cit. on p. 13).

[AS15] A. Abe and E. Sumii. “A Simple and Practical Linear Algebra Library Interface with Static Size Checking”. In: *Proceedings ML Family/OCaml Users and Developers workshops, Gothenburg, Sweden, September 4-5, 2014*. Ed. by O. Kiselyov and J. Garrigue. Vol. 198. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2015, pp. 1–21 (cit. on p. 6).

[AE10] G. A. Akyuz and T. E. Erkan. “Supply chain performance measurement: a literature review”. In: *International Journal of Production Research* 48.17 (2010), pp. 5137–5155 (cit. on p. 13).

[ACL+06] E. E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, and G. L. S. Jr. “Object-oriented units of measurement.” In: *OOPSLA*. Ed. by J. M. Vlissides and D. C. Schmidt. ACM, Feb. 13, 2006, pp. 384–403 (cit. on p. 5).

[AP02] G. S. Almasi and D. A. Padua. “MaJIC: Compiling MATLAB for Speed and Responsiveness.” In: *PLDI*. Ed. by J. Knoop and L. J. Hendren. ACM, 2002, pp. 294–303 (cit. on p. 8).

[AL84] A. Arens and J. Loebbecke. *Auditing, an Integrated Approach*. Prentice-Hall series in accounting. Prentice-Hall, 1984 (cit. on pp. 11, 95, 97).

[AR99] Aristotle and W. Ross. *Nicomachean Ethics*. Batoche Books, Kitchener, 1999 (cit. on p. 102).

[AHK+10] G. Arnold, J. Hölzl, A. S. Köksal, R. Bodík, and M. Sagiv. “Specifying and Verifying Sparse Matrix Codes”. In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’10. Baltimore, Maryland, USA: ACM, 2010, pp. 249–260 (cit. on p. 6).

[Bar09] A. Bartels. *Smart Computing Drives The New Era of IT Growth*. Forrester. 2009 (cit. on p. 99).

[BQG05] P. Bientinesi, E. S. Quintana-Ortí, and R. A. v. d. Geijn. “Representing linear algebra algorithms in code: the FLAME application program interfaces”. In: *ACM Trans. Math. Softw.* 31.1 (Mar. 2005), pp. 27–59 (cit. on p. 7).

[BIP08] BIPM. *International Vocabulary Of Metrology - Basic And General Concepts*. 2008 (cit. on p. 45).

[Bir60] G. Birkhoff. *Hydrodynamics: a study in logic, fact, and similitude*. Princeton University Press, 1960 (cit. on p. 45).

[BDW95] H. Blokdijk, F. Drieënhuizen, and P. Wallage. *Reflections on Auditing Theory: A Contribution from the Netherlands*. Kluwer/Limperg Instituut-reeks. Kluwer Bedrijfswetenschappen, 1995 (cit. on pp. 11, 97).

[CGH15a] R. Christiaanse, P. Griffioen, and J. Hulstijn. "Adaptive Normative Modelling: A Case Study in the Public-Transport Domain". In: *Open and Big Data Management and Innovation : 14th IFIP WG 6.11 Conference on e-Business, e-Services, and e-Society, I3E 2015, Delft, The Netherlands, October 13-15, 2015, Proceedings*. Ed. by M. Janssen, M. Mäntymäki, J. Hidders, B. Klievink, W. Lamersdorf, B. van Loenen, and A. Zuiderwijk. Cham: Springer International Publishing, 2015, pp. 423–434 (cit. on p. 22).

[CGH15b] R. Christiaanse, P. Griffioen, and J. Hulstijn. "Reliability of Electronic Evidence: An Application for Model-based Auditing". In: *Proceedings of the 15th International Conference on Artificial Intelligence and Law*. ICAIL '15. San Diego, California: ACM, 2015, pp. 43–52 (cit. on p. 22).

[CS91] J. M. Colom and M. Silva. "Convex Geometry and Semiflows in P/T Nets: A Comparative Study of Algorithms for Computation of Minimal P-semiflows". In: *Proceedings on Advances in Petri Nets 1990*. APN 90. New York, NY, USA: Springer-Verlag New York, Inc., 1991, pp. 79–112 (cit. on p. 79).

[DL03] C. D Ittner and D. Larcker. "Coming Up Short on Nonfmancial Performance Measurement". In: 81 (Dec. 2003), pp. 88–95, 139 (cit. on p. 13).

[DM82] L. Damas and R. Milner. "Principal type-schemes for functional programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '82. Albuquerque, New Mexico: ACM, 1982, pp. 207–212 (cit. on p. 61).

[DGR+06] I. Davies, P. Green, M. Rosemann, M. Indulska, and S. Gallo. "How Do Practitioners Use Conceptual Modeling in Practice?" In: *Data Knowl. Eng.* 58.3 (Sept. 2006), pp. 358–380 (cit. on p. 98).

[dPoi06] B. I. des Poids et Mesures. *The International System of Units (SI)*. 8th ed. 2006 (cit. on p. 45).

[Eat06] F. Eaton. "Statically Typed Linear Algebra in Haskell". In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*. Haskell '06. Portland, Oregon, USA: ACM, 2006, pp. 120–121 (cit. on p. 6).

[Ell82] D. Ellerman. *Economics, accounting, and property theory*. Lexington Books, 1982 (cit. on pp. 14, 102, 115, 117).

[Ell86] D. P. Ellerman. "Double entry multidimensional accounting". In: *Omega* 14.1 (1986), pp. 13–22 (cit. on pp. 14, 102, 115, 117).

[Els96] P. Elsas. "Computational Auditing". PhD thesis. Vrije Universiteit Amsterdam, 1996 (cit. on pp. 1, 12, 14, 99, 100, 141).

[FMJ+07] M. Fletcher, C. McCosh, G. Jin, and K. Kennedy. "Compiling Parallel MATLAB for General Distributions using Telescoping Languages". In: *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*. Vol. 4. 2007, pp. IV-1193-IV-1196 (cit. on p. 8).

[For78] J. Forrester. *Industrial Dynamics*. MIT Press and John Wiley & Sons, 1978 (cit. on p. 12).

[Fou22] J. Fourier. *Théorie analytique de la chaleur*. Chez Firmin Didot, père et fils, 1822 (cit. on p. 45).

[FdH89] A. Frielink and H. de Heer. *Leerboek Accountantscontrole*. (in Dutch). Stenfert Kroese, Leiden/Antwerpen, 1989 (cit. on pp. 12, 14, 95–97, 107).

[GF87] E. Garcke and J. Fells. *Factory accounts : their principles and practice. A handbook for accountants and manufacturers*. Crosby Lockwood and Company, 1887 (cit. on p. 9).

[Gri15] P. R. Griffioen. "Type Inference for Array Programming with Dimensioned Vector Spaces". In: *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*. IFL '15. Koblenz, Germany: ACM, 2015, 4:1–4:12 (cit. on pp. 21, 22).

[GEvdR00] P. Griffioen, P. Elsas, and R. van de Riet. "Analyzing Enterprises: The Value Cycle Approach". In: *Database and Expert Systems Applications*. Springer Berlin / Heidelberg, 2000, pp. 685–697 (cit. on pp. 22, 98, 102).

[GCH18] P. Griffioen, R. Christiaanse, and J. Hulstijn. "Controlling Production Variances in Complex Business Processes". In: *Software Engineering and Formal Methods*. Ed. by A. Cerone and M. Roveri. Cham: Springer International Publishing, 2018, pp. 72–85 (cit. on pp. 21, 22).

[Har95] G. Hart. *Multidimensional analysis: algebras and systems for science and engineering*. Springer-Verlag, 1995 (cit. on p. 4).

[Har94] G. W. Hart. "The Theory of Dimensioned Matrices". In: *Proceedings of the 5th Society for Industrial and Applied Mathematics Conference on Applied Linear Algebra*. 1994 (cit. on p. 4).

[HBH+05] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. "An overview of the Trilinos project". In: *ACM Trans. Math. Softw.* 31.3 (Sept. 2005), pp. 397–423 (cit. on p. 7).

[Iji65] Y. Ijiri. *Management goals and accounting for control*. Studies in mathematical and managerial economics. North Holland Pub. Co., 1965 (cit. on pp. 13, 102, 111, 115, 117).

[Iji67] Y. Ijiri. *The foundations of accounting measurement: a mathematical, economic, and behavioral inquiry*. Prentice-Hall international series in management. Prentice-Hall, 1967 (cit. on pp. 13, 102, 111, 115, 117, 137).

[IL03] C. D. Ittner and D. Larcker. "Coming Up Short on Nonfmancial Performance Measurement". In: *Harvard business review* 81 (Dec. 2003), pp. 88–95, 139 (cit. on p. 13).

[JS06] L. Jiang and Z. Su. "Osprey: a practical type system for validating dimensional unit correctness of C programs". In: *Software Engineering, International Conference on* 0 (2006), pp. 262–271 (cit. on p. 5).

[JK87] H. Johnson and R. Kaplan. *Relevance Lost: The Rise and Fall of Management Accounting*. Harvard Business School Press, 1987 (cit. on p. 13).

[JB06] P. G. Joisha and P. Banerjee. "An algebraic array shape inference system for MATLAB&Reg;" in: *ACM Trans. Program. Lang. Syst.* 28.5 (Sept. 2006), pp. 848–907 (cit. on p. 8).

[Kap84] R. S. Kaplan. "The Evolution of Management Accounting". In: *The Accounting Review* 59.3 (1984), pp. 390–418 (cit. on pp. 10, 13).

[KCL+10] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. "Regular, Shape-polymorphic, Parallel Arrays in Haskell". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. Baltimore, Maryland, USA: ACM, 2010, pp. 261–272 (cit. on p. 6).

[Ken94] A. Kennedy. "Dimension Types". In: *ESOP*. Ed. by D. Sannella. Vol. 788. Lecture Notes in Computer Science. Springer, 1994, pp. 348–362 (cit. on p. 3).

[Ken09] A. Kennedy. "Types for Units-of-Measure: Theory and Practice". In: *CEFP*. Ed. by Z. Horváth, R. Plasmeijer, and V. Zsók. Vol. 6299. Lecture Notes in Computer Science. Springer, 2009, pp. 268–305 (cit. on pp. 3, 5).

[Ken96] A. J. Kennedy. "Programming Languages and Dimensions". PhD thesis. University of Cambridge, 1996 (cit. on pp. 3, 5, 45).

[LL86] W. Leontief and W. Leontief. *Input-output Economics*. Oxford University Press, 1986 (cit. on p. 84).

[LG64a] T. Limperg and G. Groeneveld. *Bedrijfseconomie: Algemene inleiding tot de bedrijfshuishoudkunde en leer van de waarde*. Bedrijfseconomie. (in Dutch). A. E. Kluwer, 1964 (cit. on p. 11).

[LG64b] T. Limperg and G. Groeneveld. *Bedrijfseconomie: Leer van de accountantscontrole en van de winstbepaling*. Bedrijfseconomie. (in Dutch). A. E. Kluwer, 1964 (cit. on p. 11).

[Lim85] T. Limperg. *The Social Responsibility of the Auditor*. Limperg Institute, Amsterdam. originally published in Dutch as Leer van het Gewekte Vertrouwen (Maandblad voor Accountancy en Bedrijfshuishoudkunde, 1932-1933). 1985 (cit. on p. 11).

[LCK+12] B. Lippmeier, M. M. Chakravarty, G. Keller, R. Leshchinskiy, and S. Peyton Jones. "Work Efficient Higher-order Vectorisation". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ICFP '12. Copenhagen, Denmark: ACM, 2012, pp. 259–270 (cit. on p. 6).

[McC82] W. E. McCarthy. "The REA Accounting Model: A Generalized Framework for Accounting Systems in a Shared Data Environment". In: *Accounting Review* 57.3 (1982), p. 554 (cit. on pp. 13, 114).

[ME14] T. Muranushi and R. A. Eisenberg. "Experience Report: Type-checking Polymorphic Units for Astrophysics Research in Haskell". In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell '14. Gothenburg, Sweden: ACM, 2014, pp. 31–38 (cit. on p. 6).

[Pac94] L. Pacioli. *Summa de arithmetica geometria proportioni et proportionalita*. Paganino de Paganini, 1494 (cit. on pp. 8, 114).

[PSS09] A. D. Padula, S. D. Scott, and W. W. Symes. "A software framework for abstract expression of coordinate-free linear algebra and optimization algorithms". In: *ACM Trans. Math. Softw.* 36.2 (Apr. 2009), 8:1–8:36 (cit. on p. 7).

[Por08] M. Porter. *Competitive Advantage: Creating and Sustaining Superior Performance*. Free Press, 2008 (cit. on p. 1).

[RR09] P. Ramachandran and M. Ramakrishna. "An Object-Oriented Design for Two-Dimensional Vortex Particle Methods". In: *ACM Trans. Math. Softw.* 36.4 (Aug. 2009), 18:1–18:28 (cit. on p. 7).

[RPN10] S. Rambaud, J. Pérez, and R. Nehmer. *Algebraic Models For Accounting Systems*. World Scientific, 2010 (cit. on pp. 102, 117).

[Reu90] C. Reutenauer. *The Mathematics of Petri Nets*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990 (cit. on p. 125).

[Rom08] S. Roman. *Advanced Linear Algebra*. 3rd. Springer New York, 2008 (cit. on p. 165).

[RS06] M. B. Romney and P. J. Steinbart. *Accounting Information Systems, 10e*. Prentice Hall, NJ, 2006 (cit. on pp. 97, 114).

[RGG+96] L. Rose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. "FALCON: A MATLAB interactive restructuring compiler". In: *Languages and Compilers for Parallel Computing*. Ed. by C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua. Vol. 1033. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, pp. 269–288 (cit. on p. 8).

[SSH08] M. Sala, K. S. Stanley, and M. A. Heroux. “On the design of interfaces to sparse direct solvers”. In: *ACM Trans. Math. Softw.* 34.2 (Mar. 2008), 9:1–9:22 (cit. on p. 7).

[Sch28] E. Schmalenbach. *Buchführung und Kalkulation im Fabrikgeschäft*. Gloeckner, 1928 (cit. on p. 10).

[Sch03] S.-B. Scholz. “Single Assignment C: Efficient Support for High-level Array Operations in a Functional Setting”. In: *J. Funct. Program.* 13.6 (Nov. 2003), pp. 1005–1059 (cit. on p. 6).

[Sim90] R. Simons. “The role of management controlsystems in creating competitive advantage: new perspectives”. In: *Accounting, Organizations and Society* 15.no 1/2 (1990), p. 16 (cit. on p. 97).

[Sim95] R. Simons. *Levers of Control: How Managers Use Innovative Control Systems to Drive Strategic Renewal*. Harvard Business School Press., 1995 (cit. on p. 97).

[Sim87] R. Simons. “Accounting control systems and business strategy: An empirical analysis”. In: *Accounting, Organizations and Society* 12.4 (1987), pp. 357–374 (cit. on p. 97).

[SdMJ88] R. Starreveld, H. de Mare, and E. Joëls. *Bestuurlijke informatieverzorging*. 2nd. Vol. deel 1: Algemene grondslagen. (in Dutch). Samson Uitgeverij, Aplhen aan den Rijn/Brussel, 1988 (cit. on pp. 12, 14, 95–98, 107).

[Str88] G. Strang. *Linear Algebra and Its Applications*. Brooks Cole, Feb. 1988 (cit. on p. 75).

[Tay11] F. W. Taylor. *The Principles of Scientific Management*. New York: Harper & Row, Publishers, Incorporated, 1911 (cit. on p. 10).

[vdAvH02] W. v. d. van der Aalst and K. v. van Hee. *Workflow Management: Models, Methods, and Systems (Cooperative Information Systems)*. The MIT Press, Jan. 2002 (cit. on p. 106).

[vdSch55] H. van der Schroeff. *Kwantitatieve verhoudingen en economische proportionaliteit = Proportion of factors and proportionality : a study on the application of the principle of substitution in production*. (in Dutch). Kosmos, Amsterdam, 1955 (cit. on pp. 10, 12).

[Vee72] R. Veenstra. *Handleiding Assistenten Accountantscontrole*. (in Dutch). Internal document of Deloitte & Touche, 1972 (cit. on p. 95).

[WPM+10] M. Weidlich, A. Polyvyanyy, J. Mendling, and M. Weske. “Efficient Computation of Causal Behavioural Profiles Using Structural Decomposition”. In: *Applications and Theory of Petri Nets*. Ed. by J. Lilius and W. Penczek. Vol. 6128. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 63–83 (cit. on p. 99).

[ZvdWC+92] S. Zeff, F. van der Wel, K. Camfferman, L. Instituut, and R. voor de Jaarverslaggeving (Netherlands). *Company financial reporting: a historical and comparative study of the Dutch regulatory process*. North-Holland, 1992 (cit. on p. 11).

SUMMARY

Support for units of measurement is the main topic of the research project described in this thesis. In this interdisciplinary project, knowledge from the fields of computing and auditing is combined to investigate the possibilities of unit-aware organization modeling. Such modeling also requires computations in proper units of measurement and thus a unit-aware programming language. The first part of this thesis is about unit-aware matrix programming. It describes an investigation into the possibilities of a matrix type based on unit inference with dimensioned vectors spaces. In the second part these ideas are applied to construct unit-aware value cycle models from the field of control and auditing.

Units of measurements are an indispensable part of numerical data and provide obvious opportunities to increase the safety of software, yet programming language support for units is not common. Most programming languages are well equipped to handle numbers, but support for units is mostly absent because it would create enormous overhead. A new technique that eliminates the overhead was introduced by Andrew Kennedy. His inference for units of measurement is part of a static type system and guarantees unit correctness at compile time. This eliminates the overhead, but because it is based on a parametric type it is unfortunately limited to data structures in which all numbers have the same unit. This restriction to homogeneous units makes it unsuitable for compound data structures like vectors or matrices where the units may vary.

The question addressed in the first part of the research is how to apply unit inference to matrix programming. If unit inference is lifted from single numbers to vectors and matrices then the restriction of homogeneous units can be eliminated.

The matrix type developed during the research extends unit inference from single numbers to vectors and matrices. The structure of the matrix type is based on Hart's dimensioned vector spaces. In such a dimensioned vector space a unit of measurement is associated with each vector entry. The matrix type infers the dimensioned vector space for any linear algebra expression. This generalizes unit inference from dimensioned numbers to dimensioned matrices resulting in complete type inference for dimensioned linear algebra.

Researching support for units of measurement is not just motivated by safety concerns, but also by the fundamental issue of valuation that underlies financial information. Financial information has the advantage that it is in a single currency. This means that any two numbers are always comparable and can be summed or subtracted without a problem. The caveat is however that information has been lost during the conversion of data to the single currency. Financial data always involves a valuation that hides the underlying non-financial facts. The second part of the

thesis studies the feasibility of reformulating the value cycle models from the fields of auditing and control in non-financial units of measurement. It uses the results from the first part to generalize the value cycle and make it available for other applications.

The value cycle is a key concept in Computational Auditing, an ongoing research effort to formalize the foundations of auditing. The value cycle gives a top-level view of the flow of values in an organization's core processes with a focus on the causality and proportionality between the different process steps. This top-level view helps auditors reason about normal behavior in an organization, but since such analysis is done with financial information it is limited due to the unavoidable loss of information. Value cycle models in non-financial units of measurement would allow more accurate modeling and increase the value cycle's applicability.

The question addressed in the second part of this research is how the analytical features based on the value cycle's causality and proportionality can be generalized from auditing to other applications. The concept of normal behavior that the value cycle provides is useful in areas besides auditing.

To generalize the value cycle concept the value net formalism was developed. Value nets are a generalization of the audit net formalism from Computational Auditing. It is a formalization of the value cycle concept, but in non-financial units of measurement. Value nets provide a valuation-free characterization of normal behavior in organizational processes. The use of non-financial units in value nets allows it to formulate normal behavior as a solution to a steady state equation. Each of these solutions forms a tour, an instantiation of the value cycle's cyclic structure. Together, the tours form a tour space that completely describes an organization's normal behavior. Since it is in non-financial units of measurement it is more accurate and free of valuation, which increase the potential applications.

As a proof of concept the value net formalism was implemented using the matrix type. The implementation is fully unit-aware and gives a more accurate foundation for the analysis of organizational processes.

To validate our approach we have modeled and analyzed three small cases: a miller, a bakery, and a wine bottler. Together with experts from the auditing domain we have applied it to a larger case regarding revenue models and dispute resolution in public transport services.

SAMENVATTING

De centrale vraag in het onderzoek uit dit proefschrift is hoe rekenen met meeteenheden verbeterd kan worden. In dit interdisciplinaire onderzoek is kennis uit de informatica en accountancy bij elkaar gebracht om te zien of modellen van organisatieprocessen in fysieke meeteenheden uitgedrukt kunnen worden. Om aan zulke procesmodellen te kunnen rekenen moet een programmeertaal ook met meeteenheden kunnen omgaan. In het eerste deel van dit proefschrift onderzoeken we daarom wat de mogelijkheden zijn om meeteenheden in matrix programmeertalen beter te ondersteunen. De uitkomst is een matrixtype waarmee automatisch de juiste vectorruimte afgeleid kan worden voor wiskundige expressies. In het tweede deel worden de resultaten gebruikt om procesmodellen in fysieke meeteenheden te ontwikkelen waarmee vraagstukken in de accountancy opgelost kunnen worden.

Meeteenheden zijn een belangrijk onderdeel van numerieke gegevens en bieden goede mogelijkheden om de kwaliteit van software te verbeteren, maar huidige programmeertalen hebben er doorgaans geen voorzieningen voor. De meeste programmeertalen zijn goed in het verwerken van getallen, maar rekenen met eenheden is te duur. Er is echter een nieuwe techniek ontwikkeld door Andrew Kennedy die de juiste eenheden in compilatietijd kan afleiden. Hiermee is het efficiency probleem opgelost, maar helaas is het beperkt tot datastructuren waarin alle getallen dezelfde eenheid hebben omdat het gebaseerd is op een polymorph type. Door deze beperking is het ongeschikt om datastructuren als vectoren en matrices te typeren omdat deze getallen met verschillende eenheden kunnen bevatten.

Het onderzoek uit het eerste deel van dit proefschrift gaat over de vraag hoe meeteenheden in matrix programmeertalen automatisch afgeleid kunnen worden. Als meeteenheden naar het niveau van vectoren en matrices getild kunnen worden, dan is het mogelijk om van de beperking tot een enkele eenheid af te komen.

Het matrixtype dat ontwikkeld is tijdens het onderzoek generaliseert het automatisch afleiden van meeteenheden van enkelvoudige getallen naar vectoren en matrices. Dit matrixtype is gebaseerd op de gedimensioneerde vectorruimtes van Hart. Deze gedimensioneerde vectorruimtes koppelen een meeteenheid aan elk getal in een vector. Het matrix type kan de gedimensioneerde vectorruimte van een willekeurige lineaire algebra expressie afleiden. Hiermee ontstaat een volledig type systeem voor gedimensioneerde lineaire algebra.

Het onderzoek naar ondersteuning van meeteenheden gaat niet alleen om het verbeteren van software, maar ook om het waarderingsvraagstuk dat financiële informatie met zich meebrengt. Financiële informatie heeft als voordeel dat het altijd om een enkele eenheid gaat. Hierdoor kunnen getallen altijd vergeleken en bij elkaar opgeteld worden. Het nadeel is echter dat er altijd informatieverlies optreedt

als alles in één geldeenheid wordt uitgedrukt. Bij financiële informatie is er altijd een waardering die de onderliggende niet-financiële feiten verbergt. In het tweede deel van dit proefschrift kijken we of het mogelijk is om de waardekringloop uit het vakgebied van de accountancy te herformuleren in fysieke meeteenheden. Hierbij gebruiken we de resultaten uit het eerste deel om de waardekringloop te generaliseren en geschikt te maken voor andere toepassingen.

De waardekringloop is een kernbegrip uit het Computational Auditing onderzoeksprogramma, een programma dat als doel heeft de fundamenten van accountancy te formaliseren. De waardekringloop geeft een omvattend overzicht van de waardestromen in de primaire processen van een organisatie, waarbij de causaliteit en proportionaliteit tussen de verschillende procesonderdelen centraal staat. Dit helpt een accountant om te redeneren over normatief gedrag van organisaties, maar dit wordt beperkt door het onvermijdelijk informatieverlies dat met financiële informatie gepaard gaat. Met waardekringlopen in fysieke meeteenheden worden de modellen accurater en nemen de mogelijke toepassingen toe.

Het onderzoek in deel twee van dit proefschrift gaat over de vraag hoe de analyses gebaseerd op de causaliteit en proportionaliteit uit de waardekringloop toegepast kunnen worden in andere gebieden. De waardekringloop is een normatief model dat ook bruikbaar is voor toepassingen buiten de accountancy.

Voor de generalisatie van de waardekringloop naar fysieke eenheden hebben we zogenaamde value netten ontwikkeld. Een value net is een generalisatie van het audit net formalisme uit Computational Auditing. Het is een formalisatie van de waardekringloop, maar in fysieke meeteenheden. Hiermee geeft een value net een normatieve beschrijving van organisatieprocessen die vrij is van een waardering. Door de herformulering in meeteenheden is het mogelijk om organisatiegedrag te beschrijven middels een vergelijking die een evenwichtstoestand karakteriseert. De oplossingen van deze vergelijking vormen zogenaamde tours. Een tour is een realisatie van de cyclische structuur in de waardekringloop. De oplossingsruimte beschreven door de tours vormen het normatieve gedrag van een organisatie. Omdat deze in fysieke eenheden is uitgedrukt, is de beschrijving accurater en speelt waardering geen rol, waardoor er meer mogelijke toepassingen zijn.

Als test hebben we een prototype voor value netten geïmplementeerd met het matrix type uit het eerste deel van dit proefschrift. De implementatie ondersteunt meeteenheden volledig automatisch en biedt een accurate basis voor de analyse van organisatieprocessen.

De aanpak is gevalideerd met drie kleine cases: een molenaar, een bakker en een wijnbottelbedrijf. Samen met experts uit de accountancy is een grotere studie gedaan waarbij de aanpak is toegepast bij het analyseren van het ontwerp van een bedrijfsmodel en het oplossen van een geschil in het publieke leerlingenvervoer.

Titles in the IPA Dissertation Series since 2016

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs*. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multi-*

processors. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod*. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages*. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences*. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code*. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems*. Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming*. Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical*

model checking. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi*. Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems*. Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces*. Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network*. Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16

M. Mehr. *Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems*. Faculty of Mathematics and Computer Science, TU/e. 2018-17

M. Alizadeh. *Auditing of User Behavior: Identification, Analysis and Understanding of Deviations*. Faculty of Mathematics and Computer Science, TU/e. 2018-18

P.A. Inostroza Valdera. *Structuring Languages as Object-Oriented Libraries*. Faculty of Science, UvA. 2018-19

M. Gerhold. *Choice and Chance - Model-Based Testing of Stochastic Behaviour*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-20

A. Serrano Mena. *Type Error Customization for Embedded Domain-Specific Languages*. Faculty of Science, UU. 2018-21

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow*. Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning*. Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management*. Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. *Practical General Top-down Parsers*. Faculty of Science, UvA. 2019-04

S. Kisfaludi-Bak. *ETH-Tight Algorithms for Geometric Network Problems*. Faculty of Mathematics and Computer Science, TU/e. 2019-05

J. Moerman. *Nominal Techniques and Black Box Testing for Automata Learning*. Faculty of Science, Mathematics and Computer Science, RU. 2019-06

V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-07

T.H.A. Castermans. *Algorithms for Visualization in Digital Humanities*. Faculty of Mathematics and Computer Science, TU/e. 2019-08

W.M. Sonke. *Algorithms for River Network Analysis*. Faculty of Mathematics and Computer Science, TU/e. 2019-09

J.J.G. Meijer. *Efficient Learning and Analysis of System Behavior*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-10

P.R. Griffioen. *A Unit-Aware Matrix Language and its Application in Control and Auditing*. Faculty of Science, UvA. 2019-11